インテル® 64 および IA-32 アーキテクチャー 最適化リファレンス・マニュアル参考訳

参照ドキュメント番号: 248966-045 February 2022

注意事項:

この日本語マニュアルは、インテル コーポレーションのウェブサイト https://software.intel.com/en-us/articles/intel-sdm (英語) で公開されている『Intel® 64 and IA-32 Architecture Optimization Reference Manual』の参考訳です。インテル社の許可を得て iSUS (IA Software User Society) が翻訳版を作成した iSUS の著作物です。

† 開発コード名

インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、またはサービスの有効化が必要となる場合があります。詳細については、各システムメーカーまたは販売店にお問い合わせいただくか、http://www.intel.co.jp/ を参照してください。

テストでは、特定のシステムでの個々のテストにおけるコンポーネントの性能を文書化しています。

ハードウェア、ソフトウェア、システム構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。購入を検討される場合は、ほかの情報も参考にして、パフォーマンスを総合的に評価することをお勧めします。性能やベンチマーク結果について、さらに詳しい情報をお知りになりたい場合は、http://www.intel.com/benchmarks/ (英語) を参照してください。

絶対的なセキュリティーを提供できるコンピューター・システムはありません。インテルは、データやシステムの紛失または盗難、およびそのような損失に起因するいかなる損害についても責任を負いません。

本資料に記載されているインテル製品に関する侵害行為または法的調査に関連して、本資料を使用または使用を促すことはできません。本資料を使用することにより、お客様は、インテルに対し、本資料で開示された内容を含む特許クレームで、その後に作成したものについて、非独占的かつロイヤルティー無料の実施権を許諾することに同意することになります。

本資料に含まれるコードは、0 条項 BSD オープンソース・ライセンス (OBSD) の下にライセンスが許可されることを除いて、本資料は、(明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず) いかなる知的財産権のライセンスも許諾するものではありません。

本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

本資料には、開発中の製品、サービスおよびプロセスについての情報が含まれています。本資料に記載されているすべての情報は、予告なく変更されることがあります。インテルの最新の製品仕様およびロードマップをご希望の方は、インテルの担当者までお問い合わせください。

結果はインテル社内での分析またはアーキテクチャーのシミュレーションあるいはモデリングに基づくものであり、情報提供のみを目的としています。システム・ハードウェア、ソフトウェア、構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。

本書で紹介されている注文番号付きのドキュメントや、インテルのその他の資料を入手するには、1-800-548-4725 (アメリカ合衆国) までご連絡いただくか、http://www.intel.com/design/literature.htm (英語) を参照してください。

Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

改版履歴

1. 2017 年 7 月 10 日: 036-JA
   2017 年 6 月のバージョン 036 をベースとして最初の翻訳版を作成。
   2 章の一部 (2-1、2-2、2-3 節)、および 11 章、12 章、14 章、15 章の全文を追加。

2. 2017 年 8 月 1 日: 037-JA_Rev1
   2017 年 7 月のバージョン 037 をベースとして、2-1 節 (Skylake† Server マイクロアーキテクチャー)、2.2.4 節 (Skylake† マイクロアーキテクチャーのポーズ・レイテンシー)、8 章 (サブ NUMA クラスタリング)、13 章 (インテル®AVX-512) が追加されました。
   既存の 1 章 (はじめに) と 7 章 (キャッシュ利用の最適化) の訳を追加しました。

3. 2017 年 9 月 1 日: 037-JA_Rev2
   2017 年 7 月バージョン 037 の Rev 1 をベースとして、付録 A (アプリケーション・パフォーマンス・ツール) と付録 B (パフォーマンス監視イベント) の訳を追加しました。

4. 2017 年 10 月 2 日: 037-JA_Rev3
   2017 年 7 月バージョン 037 の Rev 2 をベースとして、2.7 節 (Nehalem† マイクロアーキテクチャー) から 2.10 節 (SIMD 技術とアプリケーション・レベル拡張のまとめ)、4 章 (SIMD アーキテクチャー向けのコーディング)、6 章 (SIMD 浮動小数点アプリケーション向けの最適化)、9 章 (マルチコアとインテル® ハイパースレッディング・テクノロジー)、10 章 (64 ビット・モードのコーディング・ガイドライン)、付録 C (命令レイテンシーとスループット) の訳を追加しました。

5. 2017 年 11 月 7 日: 038-JA
   2017 年 10 月バージョン 037 の Rev 3 をベースとして、3 章 (一般的な最適化ガイドライン)、5 章 (SIMD 整数アプリケーション向けの最適化)、11 章 (テキスト処理/字句解析/構文解析向けのインテル® SSE4.2 と SIMD プログラミング)、14 章 (モバイル利用における電力の最適化)、付録 D (Intel Atom® マイクロアーキテクチャーとソフトウェアの最適化) の訳を追加しました。

   オリジナルの最新英語版 (248966-038 October 2017) の変更と追加を行いました。3.4.1.5 節、14.2 節、14.3 節、17 章序文、付録 A の一部に訂正が加えられました。17.26 節が追加されました。

6. 2018 年 1 月 10 日: 039-JA
   オリジナルの最新英語版 (248966-039 December 2017) の変更と追加を行いました。17.26 節の一部が変更され、16 章に Goldmont Plus† アーキテクチャーの説明が追加されています。

7. 2018 年 4 月 10 日: 040-JA
   2.3.5 節のアンラミネーションが追加されました。

8. 2019 年 5 月 041-JA
   オリジナルの最新英語版 (248966-041 April 2019) の変更と追加を行いました。7 章の INT8 ディープラーニング推論と、11 章のインテル® Optane™ DC パーシステント・メモリーが追加されました。

9. 2019 年 10 月 042b-JA
   オリジナルの最新英語版 (248966-042b September 2019) の変更と追加を行いました。「2-1 節 Ice Lake† マイクロアーキテクチャー」、「17-17 節のインテル® AVX-512 ベクトルバイト操作命令」、および「18 章 暗号化と有限体の算術演算の拡張」が追加されました。

10. 2020 年 8 月 043-JA
    オリジナルの最新英語版 (248966-043 May 2020) の変更と追加を行いました。4 章に「Tremont† マイクロアーキテクチャー」の説明、付録 C に「ランタイム・パフォーマンス最適化の考察: ラージ・コード・ページにおけるインテル® アーキテクチャーの最適化」の説明が追加されました。Skylake† 以前の古いアーキテクチャーの説

明と、Tremont† マイクロアーキテクチャー以前の古い Intel Atom® プロセッサー・アーキテクチャーの説明がそれぞれ付録 D と F に移動されました。

11. 2021 年 6 月 044-JA
オリジナルの最新英語版 (248966-044 June 2021) の変更と追加を行いました。

12. 2021 年 6 月 044b-JA
オリジナルの最新英語版 (248966-044b June 2021) の変更と追加を行いました。

13. 2022 年 3 月 045-JA
オリジナルの最新英語版 (248966-045 February 2022) の変更と追加を行いました。2 章に Alder Lake† パフォーマンス・ハイブリッド・アーキテクチャーと Golden Cove† マイクロアーキテクチャーの説明が、4 章に Gracemont† マイクロアーキテクチャーの説明が追加されています。

オリジナルの英語版 (248966-045 February 2022) で追加および削除された説明は、追加された緑色の文字と行の左に、緑の縦線でその場所を強調表示しています。既存の説明が変更された部分は緑色の文字で示します。

このドキュメントに含まれる章

このドキュメントに含まれる章

第 1 章
はじめに

本書では、IA-32 アーキテクチャー・ベースのプロセッサーとインテル® 64 アーキテクチャー・ベースのプロセッサーのパフォーマンス特性を利用して、ソフトウェアを最適化する方法について説明します。

本書の対象読者は、ソフトウェア・プログラマーとコンパイラー開発者です。本書の読者は、IA-32 アーキテクチャーの基礎知識をよく理解し、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』(全 5 巻) を参照する必要があります。多くを理解する上で、インテル® 64 プロセッサーと IA-32 プロセッサーについて熟知していることが求められます。また、基礎となるマイクロアーキテクチャーに関する知識が必要です。

本書で説明する高性能ソフトウェア開発向けの設計ガイドラインは、現在および将来の IA-32 プロセッサーとインテル® 64 プロセッサーのほとんどに適用できます。説明するコーディング規則とコード最適化手法は、インテル® Core® マイクロアーキテクチャー、Intel NetBurst® マイクロアーキテクチャー、インテル® Pentium® M プロセッサー・マイクロアーキテクチャーを対象としています。大部分のコーディング規則は、インテル® 64 アーキテクチャーの 64 ビット・モード、インテル® 64 アーキテクチャーの互換モード、IA-32 モード (IA-32 モードは IA-32 アーキテクチャーとインテル® 64 アーキテクチャーでサポートされる) で実行されるソフトウェアに適用されます。64 ビット・モードに固有のコーディング規則については、個別に記載しています。

注意

パブリック・リポジトリーでは、このマニュアルから選択されたオープンソース・サンプルを利用できます。これらのサンプルコードは、0-Clause BSD ライセンスの下でリリースされています。
インテルは、サンプルが作成および検証されると、追加のサンプルコードと更新をリポジトリーに提供していきます。
パブリック・リポジトリー: https://github.com/intel/optimization-manual (英語)
ライセンスへのリンク: https://github.com/intel/optimization-manual/blob/master/COPYING (英語)

1.1 アプリケーションをチューニングする

インテル® 64 プロセッサーと IA-32 プロセッサー上で最大限の性能を引き出すためアプリケーションをチューニングするには、以下の基礎知識が必要となります。

- インテル® 64 および IA-32 プロセッサー・アーキテクチャー
- C 言語とアセンブリー言語
- パフォーマンスに影響を与えるアプリケーションのホットスポット領域
- コンパイラーの最適化機能
- アプリケーションのパフォーマンスの評価手法

インテル® VTune™ プロファイラー (旧製品名: インテル® VTune™ パフォーマンス・アナライザー/インテル® VTune ™ Amplifier) は、アプリケーションのホットスポット領域の分析と特定を支援するツールです。このツールは、インテル® Core™ i7 プロセッサー、インテル® Core™2 Duo プロセッサー、インテル® Core™ Duo プロセッサー、インテル® Core™ Solo プロセッサー、インテル® Pentium® 4 プロセッサー、インテル® Xeon® プロセッサー、インテル® Pentium® M プロセッサー上で、一連のパフォーマンス監視イベントを使用してアプリケーションを監視し、コードの実行時に収集されたパフォーマンス・イベント・データを分析できます。随時最新のプロセッサーがサポートされますが詳細については、インテル® VTune™ プロファイラーのリリースノートを参照してください。

本書では、インテル®プロセッサーのパフォーマンス監視イベントによって、パフォーマンス・カウンターを使って収集される情報についても説明します。

1.2 本書について

インテル® Core™ i7 プロセッサー、インテル® Xeon® プロセッサー 3400/5500/7500 番台は、45nm Nehalem† マイクロアーキテクチャーをベースにしています。Westmere† マイクロアーキテクチャーは、Nehalem† マイクロアーキテクチャーの 32nm バージョンです。インテル® Xeon® プロセッサー 5600 番台、インテル® Xeon® プロセッサー E7 ファミリー、および各種のインテル® Core™ i7/i5/i3 プロセッサーは、Westmere† マイクロアーキテクチャーをベースにしています。

インテル® Xeon® プロセッサー E5 ファミリー、インテル® Xeon® プロセッサー E3-1200 製品ファミリー、インテル® Xeon® プロセッサー E7-8800/4800/2800 製品ファミリー、インテル® Core™ i7-3930K プロセッサーおよび第 2 世代インテル® Core™ i7-2xxx プロセッサー・シリーズ、インテル® Core™ i5-2xxx プロセッサー・シリーズ、インテル® Core™ i3-2xxx プロセッサー・シリーズは、Sandy Bridge† マイクロアーキテクチャーをベースにしており、インテル® 64 アーキテクチャーをサポートします。

インテル® Xeon® プロセッサー E7-8800/4800/2800 v2 製品ファミリー、インテル® Xeon® プロセッサー E3-1200 v2 製品ファミリー、および第 3 世代インテル® Core™ プロセッサーは、Ivy Bridge† マイクロアーキテクチャーをベースにしており、インテル® 64 アーキテクチャーをサポートします。

インテル® Xeon® プロセッサー E5-4600/2600/1600 v2 製品ファミリー、インテル® Xeon® プロセッサー E5-2400/1400 v2 製品ファミリーおよびインテル® Core™ i7-40xx プロセッサー・エクストリーム・エディションは、Ivy Bridge-E† マイクロアーキテクチャーをベースにしており、インテル® 64 アーキテクチャーをサポートします。

インテル® Xeon® プロセッサー E3-1200 v3 製品ファミリーと第 4 世代インテル® Core™ プロセッサーは、Haswell† マイクロアーキテクチャーをベースにし、インテル® 64 アーキテクチャーをサポートします。

インテル® Xeon® プロセッサー E5-2600/1600 v3 製品ファミリーとインテル® Core® i7-59xx プロセッサー・エクストリーム・エディションは、Haswell-E† マイクロアーキテクチャーをベースにし、インテル® 64 アーキテクチャーをサポートします。

Intel Atom® プロセッサー Z8000 シリーズは、Airmont† マイクロアーキテクチャーをベースにしています。

Intel Atom® プロセッサー Z3400 シリーズと Intel Atom® プロセッサー Z3500 シリーズは、Silvermont† マイクロアーキテクチャーをベースにしています。

インテル® Core™ M プロセッサー・ファミリーと第 5 世代インテル® Core™ プロセッサー、インテル® Xeon®プロセッサー D-1500 製品ファミリーおよびインテル® Xeon® プロセッサー E5 v4 製品ファミリーは、Broadwell† マイクロアーキテクチャーをベースとし、インテル® 64 アーキテクチャーをサポートします。

インテル® Xeon® スケーラブル・プロセッサー、インテル® Xeon® プロセッサー E3-1500m v5 製品ファミリー、および第 6 世代インテル® Core™ プロセッサーは、Skylake† マイクロアーキテクチャーをベースとし、インテル® 64 アーキテクチャーをサポートします。

第 7 世代インテル® Core™ プロセッサーは、Kaby Lake† マイクロアーキテクチャーをベースとに、インテル® 64 アーキテクチャーをサポートします。

Intel Atom® プロセッサー C シリーズ、Intel Atom® プロセッサー X シリーズ、およびインテル® Pentium® プロセッサー J シリーズ、インテル® Celeron® プロセッサー J シリーズ、およびインテル® Celeron® プロセッサー N シリーズは、Goldmont† マイクロアーキテクチャーをベースとします。

インテル® Xeon Phi™ プロセッサー 3200、5200、7200 シリーズは、Knights Landing† マイクロアーキテクチャーをベースとし、インテル® 64 アーキテクチャーをサポートします。

インテル® Pentium® Silver プロセッサー・シリーズ、インテル® Celeron® プロセッサー J シリーズ、インテル® Celeron® プロセッサー N シリーズは、Goldmont Plus† マイクロアーキテクチャーをベースとします。

第 8 世代インテル® Core™ プロセッサー、第 9 世代インテル® Core™ プロセッサー、およびインテル® Xeon® E プロセッサーは、Coffee Lake† マイクロアーキテクチャーをベースとに、インテル® 64 アーキテクチャーをサポートします。

インテル® Xeon Phi™ プロセッサー 7215、7285、7295 シリーズは、Knights Mill† マイクロアーキテクチャーをベースとし、インテル® 64 アーキテクチャーをサポートします。

第 2 世代インテル® Xeon® スケーラブル・プロセッサーは、Cascade Lake† マイクロアーキテクチャーをベースとし、インテル® 64 アーキテクチャーをサポートします。

第 10 世代インテル® Core™ プロセッサーは、Ice Lake† Client マイクロアーキテクチャーをベースとし、インテル® 64 アーキテクチャーをサポートします。

一部の第 10 世代インテル® Core™ プロセッサーは、Ice Lake† マイクロアーキテクチャーをベースとし、また一部は Comet Lake† マイクロアーキテクチャーをベースとしており、どちらもインテル® 64 アーキテクチャーをサポートします。

一部の第 11 世代インテル® Core™ プロセッサーは、Tiger Lake† マイクロアーキテクチャーをベースとし、また一部は Rocket Lake† マイクロアーキテクチャーをベースとしており、どちらもインテル® 64 アーキテクチャーをサポートします。

一部の第 3 世代インテル® Xeon® スケーラブル・プロセッサーは、Cooper Lake† 製品をベースとし、また一部は Ice Lake† マイクロアーキテクチャーをベースとしており、どちらもインテル® 64 アーキテクチャーをサポートします。

第 12 世代インテル® Core™ プロセッサーは、Alder Lake† パフォーマンス・ハイブリッド・アーキテクチャーをベースとし、インテル® 64 アーキテクチャーをサポートします。

以下に、本書に含まれる各章の概要を示します。

- **第 1 章「はじめに」**: 本書の目的と構成について説明します。
- **第 2 章「インテル® 64 および IA-32 プロセッサー・アーキテクチャー」**: IA-32 プロセッサー・ファミリーとインテル® 64 プロセッサー・ファミリーのマイクロアーキテクチャー、そしてソフトウェア最適化に関連したその他の機能について説明します。
- **第 3 章「一般的な最適化ガイドライン」**: 現在のインテル® プロセッサーの共通機能を活用するように設計されたすべてのアプリケーションに適用される、一般的なコード開発および最適化手法について説明します。
- **第 4 章「Intel Atom® プロセッサー・アーキテクチャー」**: 最近の Intel Atom® プロセッサー・ファミリーのマイクロアーキテクチャー、およびソフトウェアの最適化に関連する機能について説明します。
- **第 5 章「SIMD アーキテクチャー向けのコーディング」**: インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3)、インテル® ストリーミング SIMD 拡張命令 4.1 (インテル® SSE4.1)でサポートされる、SIMD 整数命令と SIMD 浮動小数点命令を使用するための手法と概念について説明します。
- **第 6 章「SIMD 整数アプリケーション向けの最適化」**: 128 ビット SIMD 整数命令を使用するアプリケーション向けの最適化のヒントと共通ビルディング・ブロックについて説明します。
- **第 7 章「SIMD 浮動小数点アプリケーション向けの最適化」**: 単精度および倍精度 SIMD 浮動小数点命令を使用するアプリケーション向けの最適化のヒントと共通ビルディング・ブロックについて説明します。
- **第 8 章「INT8 ディープラーニング推論」**: インテル® テクノロジー上でのディープラーニング向けの INT8 データ型について説明します。このドキュメントは、インテル® AVX-512 と新しいインテル® DL ブースト命令を使用した両方の実装をカバーします。
- **第 9 章「キャッシュ利用の最適化」**: PREFETCH 命令とキャッシュ制御命令を使用してキャッシュ利用とキャッシュ・パラメーターを最適化する方法について説明します。

- **第 10 章「サブ NUMA クラスタリングの概要」**: ラスト・レベル・キャッシュ (LLC) からローカルメモリーまでの平均レイテンシーを改善する、サブ NUMA クラスタリング (SNC) モードについて説明します。
- **第 11 章「マルチコアとインテル® ハイパースレッディング・テクノロジー」**: 最高の性能スケーリングが得られるようにマルチスレッド・アプリケーションを最適化するガイドラインや手法を説明します。これらのガイドラインや手法は、マルチコア・プロセッサー、インテル® ハイパースレッディング・テクノロジー対応プロセッサー、またはマルチプロセッサー (MP) システムが対象の場合に適用されます。
- **第 12 章「インテル® Optane™ DC パーシステント・メモリー」**: インテル® Optane™ DC パーシステント・メモリーを使用するアプリケーション向けの推奨される最適化を提供します。
- **第 13 章「64 ビット・モードのコーディング・ガイドライン」**: アプリケーション・ソフトウェアが 64 ビット・モードで動作するように作成するための、追加のコーディング・ガイドラインについて説明します。
- **第 14 章「テキスト処理/字句解析/構文解析向けのインテル® SSE4.2 と SIMD プログラミング」**: インテル® SSE4.2 とその他の拡張命令を使用してテキスト/文字列処理や字句/構文解析のアプリケーションを改善する SIMD 手法について説明します。
- **第 15 章「インテル® AVX、FMA およびインテル® AVX2 向けの最適化」**:インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX)、FMA およびインテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2) を使用するアプリケーションの最適化の提案と一般的な構成要素について説明します。
- **第 16 章「インテル® TSX の推奨事項」**: 競合するロックを持つマルチスレッド・ソフトウェアを最適化するため、インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) とロックの省略を使用するチューニングの推奨事項を説明します。
- **第 17 章「モバイル利用における電力の最適化」**: モバイル・プロセッサーの省電力手法に関する背景説明を行い、バッテリー持続時間の延長のために開発者が利用できる推奨事項を紹介します。
- ンション 512 を使用するアプリケーション向けの最適化の推奨事項と共通のビルディング・ブロックについて説明します。
- **第 19 章「暗号化と有限体の算術演算の強化」**: Ice Lake† Client マイクロアーキテクチャーで追加された暗号化フローと有限体演算の高速化のため追加された新しい命令セットについて説明します。
- **第 20 章「Knights Landing † マイクロアーキテクチャーとソフトウェアの最適化」**: Knights Landing† マイクロアーキテクチャー・ベースのプロセッサー・ファミリーのマイクロアーキテクチャーと、Knights Landing† マイクロアーキテクチャー・ベースのプロセッサー向けのソフトウェア最適化技法について説明します。
- **付録 A「アプリケーション・パフォーマンス・ツール」**: アセンブリー・コードを記述することなく、アプリケーション・パフォーマンスの分析と強化を行うツールについて説明します。
- **付録 B「パフォーマンス監視イベント」**: トップダウン解析でもたらされる情報と、インテル® Xeon® プロセッサー 5500 番台、Sandy Bridge† マイクロアーキテクチャー・ベースのプロセッサー、およびインテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサー固有のパフォーマンス監視イベントにより収集される情報の使用法について説明します。
- **付録 C「ラージ・コード・ページを使用するインテル® アーキテクチャーの最適化」**: ラージ・コード・ページを使用してランタイムのパフォーマンスを向上する方法に関連する情報を提供します。
- **付録 D「命令レイテンシーとスループット」**: IA-32 命令のレイテンシーとスループットのデータを示します。また最近のプロセッサー・ファミリー固有の命令タイミングデータについて説明します。
- **付録 E「旧世代のインテル® 64 および IA-32 プロセッサー・アーキテクチャー」**: 旧世代のインテル® 64 および IA-32 プロセッサー・ファミリーのマイクロアーキテクチャーと、ソフトウェアの最適化に関連する機能について説明します。
- **付録 F「旧世代の Intel Atom® マイクロアーキテクチャーとソフトウェアの最適化」**: 旧世代の Intel Atom® マイクロアーキテクチャー・ベースのプロセッサー・ファミリーのマイクロアーキテクチャーと、旧世代の Intel Atom® マイクロアーキテクチャー・ベースのプロセッサー向けのソフトウェア最適化技法について説明します。

1.3 関連情報

インテル® アーキテクチャー、手法、プロセッサー・アーキテクチャーの用語に関する詳細は、以下の資料を参照してください。

- 『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』(全 5 巻) (英語)
- 『Developing Multi-threaded Applications: A Platform Consistent Approach』(英語)
- インテル® C++ コンパイラーのドキュメントとオンラインヘルプ (英語)
- インテル® Fortran コンパイラーのドキュメントとオンラインヘルプ (英語)

- インテル® VTune™ プロファイラーのドキュメントとオンラインヘルプ (英語)
- 『Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor MP』(英語)

その他の関連リンクを以下に示します。

- インテル® デベロッパー・ゾーン:
  https://software.intel.com/en-us/all-dev-areas (英語)
- プロセッサー・サポートの全般リンク:
  https://www.intel.com/content/www/us/en/products/processors.html (英語)
- インテル® マルチコア・テクノロジー:
  https://software.intel.com/en-us/articles/multi-core-introduction (英語)
- インテル® ハイパースレッディング・テクノロジー (HT テクノロジー):
  http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyperthreading-technology.html (英語)
- インテル® SSE4.1 アプリケーション・ノート: インテル® ストリーミング SIMD 拡張命令 4 による動き予測:
  https://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4 (英語)
- インテル® SSE4 プログラミング・リファレンス:
  https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf (英語)
- インテル® 64 アーキテクチャー・プロセッサー・トポロジー:
  https://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration (英語)
- SIMD 拡張命令を使用した複数バッファリング技術:
  http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communicationsia-multi-buffer-paper.pdf (英語)
- 複数バッファリングを使用した並列ハッシュ処理:
  http://www.scirp.org/journal/PaperInformation.aspx?paperID=23995 (英語)
  http://eprint.iacr.org/2012/476.pdf (英語)
- インテル® AES-NI ライブラリーのサンプルコード:
  http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/ (英語)
- PCMMULQDQ に関するリソース:
  https://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usagefor-computing-the-gcm-mode (英語)
- 冗長表現とインテル® AVX2 を使用した冪剰余:
  http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3_9?LI=true (英語)

第 2 章
インテル® 64 および IA-32 プロセッサー・アーキテクチャー

本章では、最新世代のインテル® 64 プロセッサーと IA-32 プロセッサー[1]におけるソフトウェア最適化に関連するプロセッサーの機能について説明します。これらの機能には、以下のものが含まれます。

- 高クロックレートかつ高スループットでの命令実行が可能なマイクロアーキテクチャー、高速なキャッシュ階層、高速システムバス
- インテル® Core™ プロセッサー・ファミリーとインテル® Xeon® プロセッサー・ファミリーで利用可能なマルチコア・アーキテクチャー
- インテル® ハイパースレッディング・テクノロジー[2] (HT テクノロジー) のサポート
- インテル® 64 プロセッサーのインテル® 64 アーキテクチャー
- SIMD 拡張命令: インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3)、インテル® ストリーミング SIMD 拡張命令 4.1 (インテル® SSE4.1)、インテル® ストリーミング SIMD 拡張命令 4.2 (インテル® SSE4.2)
- インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX)
- 半精度浮動小数点変換と RDRAND
- 乗算加算融合(FMA)拡張
- インテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2)
- ADX と RDSEED
- インテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512)
- インテル® スレッド・ディレクター

2.1 Alder Lake† パフォーマンス・ハイブリッド・アーキテクチャー

Alder Lake† パフォーマンス・ハイブリッド・アーキテクチャーは、2 つのインテル® アーキテクチャーである Golden Cove† の Performance-core と Intel Atom® (Gracemont†) の Efficient-core を組み合わせて 1 つの SoC に統合しました。Golden Cove† マイクロアーキテクチャーの詳細については、2.2 節の「Golden Cove† マイクロアーキテクチャー」を参照してください。また、Gracemont† マイクロアーキテクチャーの詳細については、4.1 節の「Gracemont† マイクロアーキテクチャー」を参照してください。

2.1.1 パフォーマンス・ハイブリッド・アーキテクチャーをサポートする第 12 世代インテル® Core™ プロセッサー

第 12 世代インテル® Core™ プロセッサーは、最大 8 つの Performance-core (P-core) と 8 つの Efficient-core (E-core) から成るパフォーマンス・ハイブリッド・アーキテクチャーをサポートとしています。これらのプロセッサーには、IDI モジュールごとに 3MB の最終レベルキャッシュ (LLC) が含まれます。ここで、モジュールとは 1 つの P-core または 4 つの E- core を意味します。シンメトリックな ISA を備えており、さまざまな構成があります。

P-core はシングル、または制限されたスレッドのパフォーマンスをもたらし、E-core は改善されたスケーリングとマルチスレッド効率を提供します。これらのプロセッサー上の P-core では、インテル® ハイパースレッディング・テク

[1] 旧世代のインテル® 64 および IA-32 プロセッサーについては、付録 E「旧世代のインテル® 64 および IA-32 プロセッサー・アーキテクチャー」を参照してください。Intel Atom® プロセッサー・アーキテクチャーについては、第 4 章「Intel Atom® プロセッサー・アーキテクチャー」で説明しています。

[2] インテル® ハイパースレッディング・テクノロジーを利用するには、インテル® ハイパースレッディング・テクノロジーに対応したインテル® プロセッサーを搭載したコンピューター・システム、および同技術に対応したチップセットと BIOS、OS が必要です。性能は使用するハードウェアやソフトウェアによって異なります。

ノロジーを有効にすることもできます。オペレーティング・システム (OS) がすべてのプロセッサー上でのスケジュールを決定すると、それらのコアを同時にアクティブにすることもできます。

ハイブリッドを有効にする重要な OSV の要件は、パフォーマンス・ハイブリッド・アーキテクチャーにおける異なるコアタイプ間でのシンメトリックな ISA です。パフォーマンス・ハイブリッド・アーキテクチャーをサポートする第 12 世代インテル® Core™ プロセッサーでは、ISA は P-core と E-core 間で同じではありません。そのため共通のベースラインに収束されます。シンメトリックな ISA を維持するため、E-core はインテル® AVX-512、インテル® AVX-512 FP-16、およびインテル® TSX をサポートしていません。E-core はインテル® AVX2 とインテル® AVX-VNNI をサポートします。

2.1.2 ハイブリッドのスケジューリング

2.1.2.1 インテル® スレッド・ディレクター

インテル® スレッド・ディレクターは、ソフトウェアをリアルタイムで継続的に監視し、OS のスケジューラーにヒントを提供することで、インテリジェントでデータ駆動型のスレッドのスケジュールを可能にします。インテル® スレッド・ディレクターでは、ハードウェアは各種 IPC パフォーマンス特性に基づいて、次の形式でスレッドごとに OS へランタイム・フィードバックを提供します。

- 電力/熱制限に基づく P-core および E-core の動的なパフォーマンスと電力効率。
- 電力と熱が制限される場合のアイドリングのヒント。

インテル® スレッド・ディレクターは、Alder Lake† パフォーマンス・ハイブリッド・アーキテクチャーをベースとする第 12 世代インテル® Core™ プロセッサーのデスクトップおよびモバイル製品で最初に導入されました。

パフォーマンス特性が異なる P-core と E-core を持つプロセッサーは、オペレーティング・システムのスケジューラーに課題をもたらします。さらに、ソフトウェア・スレッドが異なると、P-core と E-core 間のパフォーマンスの比率も変化します。例えば、高度にベクトル化された浮動小数点コードの P-core と E-core 間のパフォーマンス比率は、スカラー整数コードのパフォーマンス比率よりも高くなります。したがって、オペレーティング・システムが最適なスケジューリングの決定を行う場合、スケジュールの候補となるソフトウェア・スレッドの特性を認識する必要があります。十分な数の P-core が利用できず、さまざまな特性を持つソフトウェア・スレッドが混在する場合、オペレーティング・システムは P-core から最も恩恵を受けられるスレッドを P-core にスケジュールし、ほかのスレッドを E-core にスケジュールする必要があります。

インテル® スレッド・ディレクターは、それぞれの論理プロセッサーで実行されるソフトウェア・スレッドの特性に関して、オペレーティング・システムが必要とするヒントを提供します。ヒントは動的であり、最後に監視されたスレッドの特性を反映しています。これは、スレッドの動的な命令の組み合わせに基づいて時間経過とともに変化する可能性があります。プロセッサーは、マイクロアーキテクチャーの要素も考慮して、この動的なソフトウェア・スレッドの特性を定義します。

スレッド固有のハードウェアによるサポートは CPUID 命令によって照会でき、MSR への書き込みによってオペレーティング・システムが有効にします。Alder Lake† パフォーマンス・ハイブリッド・アーキテクチャーをベースとするプロセッサーにおけるインテル® スレッド・ディレクターの実装では、次の 4 つのスレッドクラスが定義されています。

0. ベクトル化されていない整数、または浮動小数点コード
1. インテル® ディープラーニング・ブースト (インテル® DL ブースト) を除く、ベクトル化された整数または浮動小数点コード.
2. インテル® DL ブーストコード
3. ポーズ (スピン待機) が多いコード。

動的コードは、クラス定義に 100% 当てはまる必要はありません。クラスに属すると見なされる大きさがあれば十分です。また、消費されるメモリー帯域幅やキャッシュ帯域幅など、動的なマイクロアーキテクチャー・レベルのメトリックによって、ソフトウェア・スレッドをクラス間で移動することもあります。Alder Lake† パフォーマンス・ハイブリッド・

アーキテクチャーをベースとするプロセッサーで利用可能なインテル® スレッド・ディレクターの疑似コードシーケンスを、例 2-1 から 2-4 に示します。

また、インテル® スレッド・ディレクターは、クラスごとに P-core と E-core の性能比を定義するテーブルをシステムメモリー内に保持しています (オペレーティング・システムからのみアクセスできます)。これにより、オペレーティング・システムは、適切な論理プロセッサーに、適切なソフトウェア・スレッドをスケジュールすることができます。

P-core と E-core の性能比に加えて、インテル® スレッド・ディレクターは、これらのコア間の電力効率比も提供します。オペレーティング・システムがパフォーマンスよりも省電力を重視する場合、この情報を利用できます。例えば、インデックスの作成などのバックグラウンド・タスクは、パフォーマンスがそれほど重要でないため、最も電力効率が高いコアにスケジュールすることができます。

例 2-1 クラス 0 疑似コードの一部

```
while (1)
{
    asm("xor rax, rax;"
        "add rax, 5;"
        "inc rax;"
    );
}
```

例 2-2 クラス 1 疑似コードの一部

```
while (1)
{
    asm("vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
```

```
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
    );
}
```

例 2-3 クラス 2 疑似コードの一部

```
while (1)
{
    __asm(
        vpdpbusd ymm2, ymm0, ymm1
        vpdpbusd ymm3, ymm0, ymm1
        vpdpbusd ymm4, ymm0, ymm1
        vpdpbusd ymm5, ymm0, ymm1
        vpdpbusd ymm6, ymm0, ymm1
        vpdpbusd ymm7, ymm0, ymm1
        vpdpbusd ymm8, ymm0, ymm1
        vpdpbusd ymm9, ymm0, ymm1
        vpdpbusd ymm10, ymm0, ymm1
        vpdpbusd ymm11, ymm0, ymm1
        vpdpbusd ymm12, ymm0, ymm1
        vpdpbusd ymm13, ymm0, ymm1
    );
}
```

例 2-4 クラス 3 疑似コードの一部

```
while (1)
{
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
    );
}
```

このテクノロジーの詳細については、www.intel.com/sdm (英語) にある「Intel® 64 and IA-32 Architectures Software Developer Manuals」を参照してください。

2.1.2.2 x86 ハイブリッド・アーキテクチャーをサポートするプロセッサーでインテル® ハイパースレッディング・テクノロジーを有効にした場合のスケジューリング

E-core は、P-core でハイパースレッドがビジーの論理コアよりも優れたパフォーマンスを発揮するように設計されています。

2.1.2.3 複数の E-core モジュールのスケジューリング

アイドル状態のモジュール内の E-core は、ビジー状態のモジュール内の E-core よりも優れたパフォーマンスを発揮します。

2.1.2.4 x86 ハイブリッド・アーキテクチャーにおけるバックグラウンド・スレッドのスケジューリング

ほとんどのシナリオでは、バックグラウンド・スレッドは、E-core のスケーラビリティーとマルチスレッド効率を活用できます。

2.1.3 アプリケーション開発者向けの推奨事項

パフォーマンス・ハイブリッド・アーキテクチャーをサポートするプロセッサーを利用する場合、次のことが推奨されます。

- オペレーティング・システムや最適化されたライブラリーのアップデートを常に把握しておきます。
- ソフトウェアは、オペレーティング・システムがインテルのハイブリッド・プロセッサーで最適なコアの選択ができるよう、スレッドやプロセスにハード・アフィニティーの設定を避ける必要があります。
- ソフトウェアは、アクティブなスピン待機を、軽量な待機に置き換える必要があります。新しい UMWAIT/TPAUSE 命令や古い PAUSE 命令を使用します。これは、スケジューラーへのスピン時間に関するより良いヒントとなります。
- ソフトウェアは、プロセス情報とスレッド情報 API を使用して Windows* パワー・スロットリング情報を利用し、特定のスレッドやプロセスに必要とされるサービス品質 (QoS) に関するヒントをスケジューラーに提供して、パフォーマンスと電力効率の両方を向上できます。
- マルチメディア・アプリケーション開発に Windows* フレームワークとメディア API を活用します。Windows* メディア・ファンデーション・フレームワークは、ハイブリッド・アーキテクチャー向けに最適化されており、メディア・アプリケーションの誤作動を防ぎ、効率良く動作させます。
- Windows* の IrqPolicyMachineDefault ポリシーを適用すると、Windows* は割り込みを適切なコアに対応付けることを可能にします。これは、ハイブリッド・アーキテクチャーでも同様です。

パフォーマンス・ハイブリッド・アーキテクチャー向けの推奨事項と情報の詳細については、https://www.intel.com/content/www/us/en/developer/articles/technical/hybridarchitecture.html (英語) を参照してください。

2.2 Golden Cove† マイクロアーキテクチャー

Golden Cove† マイクロアーキテクチャーは、Ice Lake† マイクロアーキテクチャーの後継です。Golden Cove† マイクロアーキテクチャーには次の拡張が含まれます。

- よりワイドなマシン。5->6 アロケーション、10->12 実行ポート、および 4->8 リタイア。
- 主要な構造体のサイズを大幅に増やすことで、より深い OOO 実行とさらに多くの命令レベルの並列性が可能。

- 実行ポートごとの機能向上。例: 拡張機能と新しい高速な浮動小数点加算器を備えた 5 番目の整数 ALU 実行ポート。
- インテル® アドバンスト・マトリクス・エクステンション (インテル® AMX)[3]: ビルトインされた統合タイル行列乗算/マシンラーニング・アクセラレーター。
- 分岐予測の改善。
- 大きな分岐予測構造体、拡張されたコード・プリフェッチャー、拡大された TLB など、大規模なコード・フットプリントを持つワークロード向けの改善。
- より広いフェッチ: レガシー・デコード・パイプラインのフェッチ帯域幅が 32B/サイクルに増加、4→6 へのデコーダーの増加、マイクロオペレーション・キャッシュ・サイズの増加、そしてマイクロオペレーション・キャッシュの帯域幅の増加。
- 最大ロード帯域幅が、2 ロード/サイクルから 3 ロード/サイクルに増加。
- 4K ページ DTLB が大きくなり、未処理のページ・ミス・ハンドラーが増加。
- 未処理のミス数が増加 (16 FB、32→48 の深さの MLC ミスキュー)。
- メモリーの並列性を高めるデータ・プリフェッチャーの拡張。
- サーバー製品の中間レベルキャッシュが 2MB に増加。クライアント製品では 1.25MB のまま。

2.2.1 Golden Cove† マイクロアーキテクチャー概要

図 2-1 に Golden Cove† マイクロアーキテクチャーの基本パイプライン機能を示します。

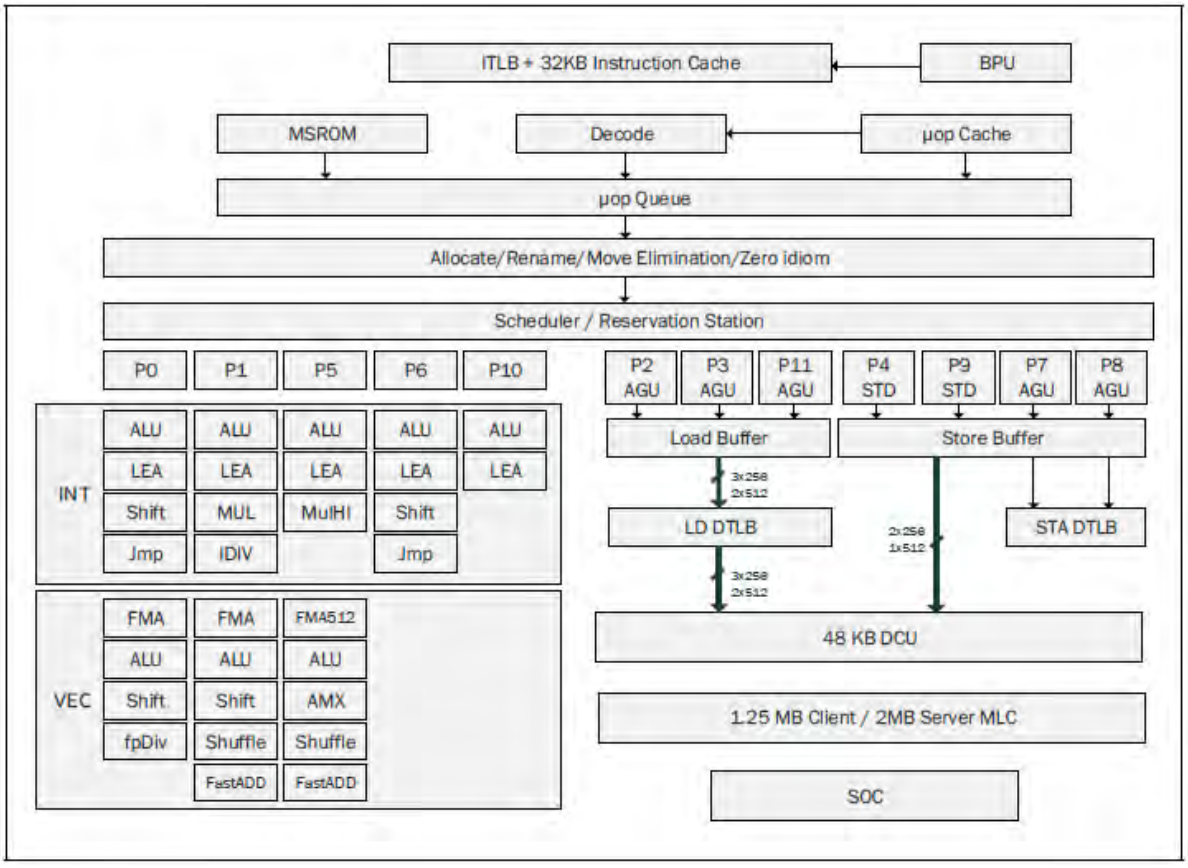


図 2-1 Golden Cove† マイクロアーキテクチャーのプロセッサー・コア・パイプラインの機能

[3] インテル® アドバンスト・マトリクス・エクステンションは、クライアント製品では利用できません。

2.2.1.1 フロントエンド

図 2-2 に Golden Cove† マイクロアーキテクチャーのフロントエンドを示します。フロントエンドは、より広くそしてより深いアウトオブオーダー・コアにマイクロオペレーションを供給するように構築されています。

- レガシー・デコード・パイプラインのフェッチ帯域幅が 16 から 32 バイト/サイクルに増えました。
- デコーダー数が 4 から 6 に増え、サイクルごとに最大 6 命令をデコードできるようになりました。
- マイクロオペレーション・キャッシュのサイズと帯域幅が増加して、サイクルあたり最大 8 つのマイクロオペレーションを供給できるようになりました。
- 分岐予測が改善しました。

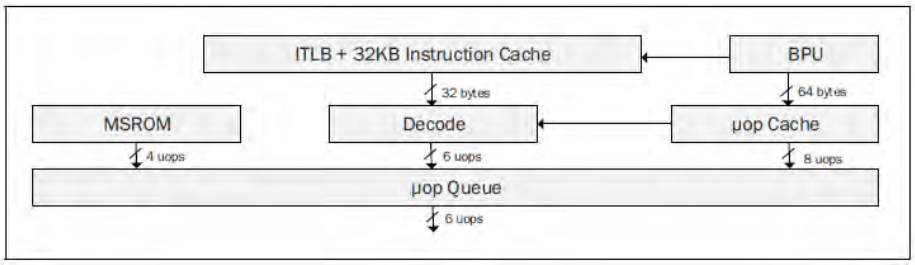


図 2-2 Golden Cove† マイクロアーキテクチャーのプロセッサー・フロントエンド

大規模なコード・フットプリントのワークロード向けの改善。

- 倍増された命令 TLB サイズ。4K ページでは 128→256 エントリー、2M/4M ページでは 16→32 エントリー。
- 大きくなった分岐予測構造体。
- 拡張されたコード・プリフェッチャー。
- LSD カバレージの改善。
- IDQ は、シングルスレッド・モードで論理プロセッサーごとに 144 個の uop を保持でき、SMT が有効な場合はスレッドごとに 72 個の uop を保持できます。

2.2.1.2 アウトオブオーダーと実行エンジン

Golden Cove† マイクロアーキテクチャーのアウトオブオーダーと実行エンジンには、次の変更があります。

- 主要なバッファー構造体のサイズを大幅に増やすことで、より深い OOO 実行とさらに多くの命令レベルの並列性が可能になります。
- よりワイドなマシン:
  - よりワイドな割り当て (サイクルあたり 5→6uop) とリタイアメント (サイクルあたり 4→8uop)。
  - 実行ポート数の増加 (10→12)。
  - 実行ポートごとの機能拡張。

表 2-1 は、異なる操作タイプをポートへディスパッチする OOO エンジンの役割をまとめています。

表 2-1 Golden Cove† マイクロアーキテクチャーのディスパッチ・ポートと実行スタック

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	INT ALU LEA	Load
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle*				FMA** Fast Adder Vec ALU Shuffle					

注意:

1. 表中の “*” で示される機能は 512 ビット・ベクトルでは利用できません。
2. 表中の “**” で示される機能は、クライアント製品の 512 ビット・ベクトルでは利用できません。

表 2-2 は、実行ユニットとこれらのユニットに関連する代表的な命令を示します。インテル® SSE、インテル® AVX、および汎用命令セット全体のスループット向上は、対応する命令向けのユニット数、および特定のユニットで実行されるさまざまな命令に関連しています。

表 2-2 Golden Cove† マイクロアーキテクチャーの実行ユニットと主要な命令[4]

Execution Unit	# of Unit	Instructions
ALU	5	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	3x256-bit 2x512-bit	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2x256-bit 1x512-bit	(v)psllv*, (v)psrlv*, vector shift count in imm8
VEC Add (in VEC FMA)	2x256-bit 1x512-bit	(v)add*, (v)cmp*, (v)max*, (v)min*, (v)sub*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
VEC Fast Add	2x256-bit 1x512-bit	(v)add*, (v)addsub*, (v)sub*
Shuffle	2x256-bit 1x512-bit	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw (new cross lane shuffle on both ports)

[4] 高速加算器ユニットは、クライアント製品の 512 ビット・ベクトルでは利用できません。

Vec Mul/FMA	2x256-bit (1 or 2)x512-bit	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

注意:

1. インテル® MMX® 命令にマッピングされる実行ユニットは、この表ではカバーされていません。15.16.5 節「インテル® MMX® 命令のスループットの制限によるインテル® AVX2 への変換方法」を参照してください。

表 2-3 は、生産と消費オペレーション間サイクルにおけるバイパス遅延を示しています。

表 2-3 生産と消費マイクロオペレーション間のバイパス遅延

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	SIMD/0,1/1	FMA/0,1/4	MUL/0,1/4	Fast Adder/1,5/3	SIMD/5/1,3	SHUF/ 1,5/1, 3	V2I/0/3
SIMD/0,1/1	0	1	1	1	0	0	0
FMA/0,1/4	1	0	1	0	0	0	0
MUL/0,1/4	1	0	1	0	0	0	0
Fast Adder/0,1/3	1	0	1	-1	0	0	0
SIMD/5/1,3	0	1	1	1	0	0	0
SHUF/1,5/1,3	0	0	1	0	0	0	0
V2I/0/3	0	0	1	0	0	0	0
I2V/5/1	0	1	1	0	0	0	0

バイパスにおける生産/消費の uop に関連する属性は、uop の略号/1 つ以上のポート番号/uop のレイテンシーサイクルの 3 項目です。次に例を示します。

- “SIMD/0,1/1” は、1 サイクルのベクトル SIMD uop が、ポート 0 または 1 のいずれかにディスパッチされる場合を示します。
- “SIMD/5/1,3” は、1 もしくは 3 サイクルのシャッフルしない uop が、ポート 5 にディスパッチされる場合を示します。
- “V2I/0/3” は、3 サイクルのベクトル-整数 uop が、ポート 0 にディスパッチされる場合を示します。
- “I2V/5/1” は、1 サイクルの整数-ベクトル uop が、ポート 5 にディスパッチされる場合を示します。
- “Fast Adder/1,5/3” は、3 サイクルの 256 ビット uop がポート 1 もしくは 5 に、または 512 ビット uop がポート 5 にディスパッチされる場合を示します。これより、ペアになって連続実行される高速加算 (Fast Adder) 演算は 2 サイクルでサポートされます。

新しい高速加算器[5]ユニットは、VEC スタックのポート 5 に 512 ビットとして、またはポート 1 と 5 に 256 ビットとして追加されています。高速加算器は、浮動小数点 ADD/SUB 操作を 3 サイクルで実行します。

高速加算器ユニットで実行される連続した ADD/SUB 操作は、2 サイクルで実行されます。

- 128/256 ビットでは、高速加算器ユニットで実行される連続した ADD/SUB 操作は、2 サイクルで実行されます。

[5] 高速加算器ユニットは、クライアント製品の 512 ビット・ベクトルでは利用できません。

- 512 ビットでは、ポート 5 の高速加算器ユニットを使用する場合、連続した ADD/SUB 操作は 2 サイクルで実行されます。

次の命令は、高速加算器ユニットで実行されます。

- (V)ADDSUBSS/SD/PS/PD
- (V)ADDSS/SD/PS/PD
- (V)SUBSS/SD/PS/PD

2.2.1.3 キャッシュ・サブシステムとメモリー・サブシステム

Golden Cove† マイクロアーキテクチャーのキャッシュ・サブシステムとメモリー・サブシステムの変更を以下に示します。

- 最大ロード帯域幅が、2 ロード/サイクルから 3 ロード/サイクルに増加しました。インテル® AVX-512 ロード、インテル® AMX ロード、およびインテル® MMX®/x87 ロードの帯域幅は、サイクルあたり最大 2 ロードのままです。
- バッファー数が増えたことにより、多くのロードとストアを同時に処理することが可能になりました。
- ロード DTLB の 4K ページのエントリー数が、64 から 96 に増えています。
- ページ・ミス・ハンドラーは、2 つではなく最大 4 つの D サイドのページウォークを並列に処理できます。
- 未処理の DCU および MLC ミスの数が増加しました。
- メモリーの並列性を高めるデータ・プリフェッチャーの拡張。
- パーシャル・ストア・フォワードにより、ロードの一部がストアとオーバーラップする場合でも、ストアからロードにデータをフォワードできます (ロードとストアのオフセットが一致する場合のみ)。

2.2.1.4 誤ったデスティネーションの依存関係の回避

一部の SIMD 命令では、デスティネーション・オペランドで誤った依存関係が発生します。次の命令が影響を受けます。

- VFMULCSH, VFMULCPH
- VFCMULCSH, VFCMULCPH
- VPERMD, VPERMQ, VPERMPS, VPERMPD
- VRANGE[SS,PS,SD,PD]
- VGETMANTSH, VGETMANTSS, VGETMANTSD
- VGETMANTPS, VGETMANTPD (memory versions only)
- VPMULLQ

推奨事項: 誤った依存関係による潜在的な速度低下を回避するため、影響を受ける命令の前で、デスティネーション・レジスターの依存関係を排除するゼロイディオムを使用してください。

例 2-5 ゼロイディオムによる誤った依存関係の排除

依存関係に誤った影響を与えるコード	回避: ゼロイディオムを使用して誤った依存関係を排除

`vaddps zmm3, zmm4, zmm5`
`vmovaps [rsi], zmm3`
`vfmulcph zmm3, zmm2, zmm1 ; zmm3 への誤った依存関係。vaddps が zmm3 へ書き込むまで、アウトオブオーダーで実行されません`
	`vaddps zmm3, zmm4, zmm5`
	`vmovaps [rsi], zmm3`
	`vpxord zmm3, zmm3, zmm3 ;依存関係を排除するゼロイディオム`
	`vfmulcph zmm3, zmm2, zmm1 ;vaddps への結果を待たずに、アウトオブオーダーで実行します`

2.3 Ice Lake† Client マイクロアーキテクチャー

Ice Lake† Client アーキテクチャーは、アプリケーションのパフォーマンスと電力消費を最適化するため、次の新機能を導入しています。

- ターゲットベクトルの高速化
- 暗号化の高速化
- インテル® ソフトウェア・ガード・エクステンションズ (インテル® SGX) の拡張
- キャッシュライン・ライトバック命令 (CLWB)

2.3.1 Ice Lake† Client マイクロアーキテクチャーの概要

Ice Lake† Client マイクロアーキテクチャーは、Skylake† Client マイクロアーキテクチャーでの成功をベースにして構築されています。図 2-3 に Ice Lake† Client マイクロアーキテクチャーの基本パイプライン機能を示します。

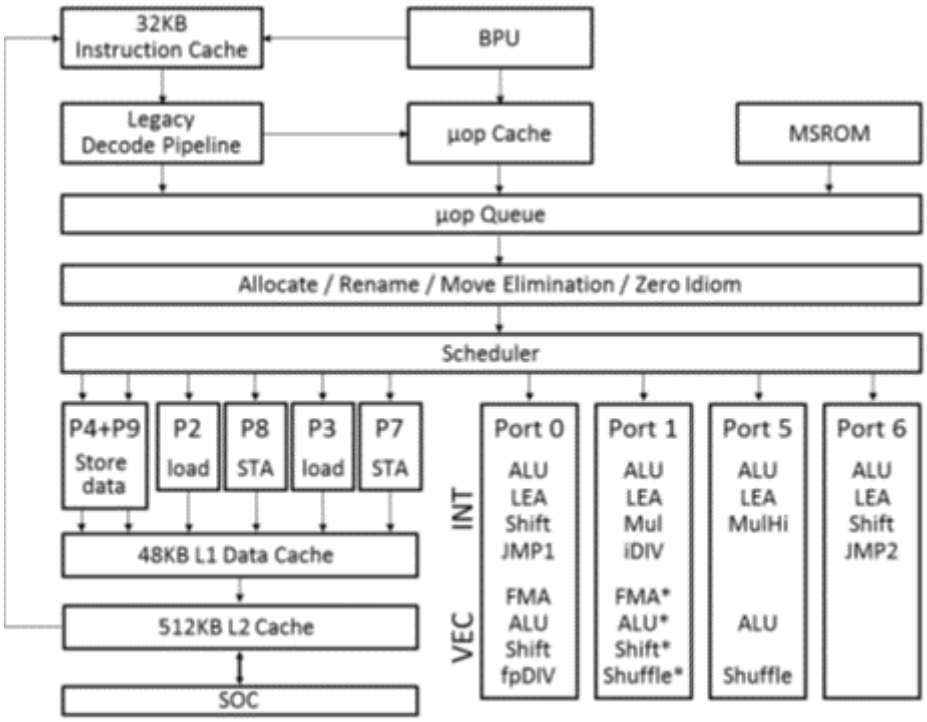


図 2-3 Ice Lake† Client アーキテクチャーのプロセッサー・コア・パイプラインの機能[1]

注意:

1. 図中の "*" で示される機能は、512 ビット・ベクトルでは利用できません。
2. ""INT" は GPR スカラー命令を表します。
3. "VEC" は浮動小数点と整数ベクトル命令を表します。
4. "MulHi" は、2 つの 64 ビット・レジスターを乗算し、結果を 2 つの 64 ビット・レジスターに配置する iMUL 操作の結果として上位 64 ビットを生成します。
5. ポート 1 での "Shuffle" は新しく追加され、同じ 128 ビット・サブベクトル内で動作するインラインシャッフルのみをサポートします。
6. ポート 1 の "iDIV" ユニットは新しく、低レイテンシーの整数除算操作を行います。

Ice Lake† Client マイクロアーキテクチャーは、次の新機能を提供します。

- 主要構造のサイズが大幅に増加したことにより、より深い OOO 実行が可能。
- よりワイドなマシン: 4 → 5 幅の割り当て、8 → 10 実行ポート。
- インテル® AVX-512 (クライアント向けの新命令): 512 ビット・ベクトル操作、512 ビットのメモリーロードとストア、および 32 個の新しい 512 ビット・レジスター。
- 実行ポートごとのギャザー操作 (例: SIMD シャッフル、LEA)、整数除算器のレイテンシー削減。
- 既存のバイナリーのインテル® AES-NI 命令のピーク・スループットを改善する 2xBW (マイクロアーキテクチャー)。
- 文字列リピート移動命令の加速。
- L1 データ・キャッシュ・サイズを 50% 増量。
- 実効ロード・レイテンシーの削減。
- 2x L1 ストア帯域幅: サイクルごとに 1 → 2 ストア。
- メモリーの並列性を高めるデータ・プリフェッチの拡張。
- 大きな L2 レベル TLB。
- 大きな uop キャッシュ。
- 分岐予測器の改善。
- シングル・スレッド・モードでのラージページ ITLB サイズが 2 倍に増加。
- 大きな L2 キャッシュ。

Ice Lake† Client マイクロアーキテクチャーは、L3 の複数スライスへのリング・インターコネクト、プロセッサー・グラフィックス、統合メモリー・コントローラー、インターコネクト・ファブリックなどを含む、多くのコンポ—ネントで構成される共有アンコア・サブシステムと、複数のプロセッサーコアによる柔軟性のある統合をサポートします。

2.3.1.1 フロントエンド

Ice Lake† Client マイクロアーキテクチャーのフロントエンドでは。次のような変更が行われています。

- 分岐予測器の改善。
- シングル・スレッド・モードのラージページ ITLB は、8 エントリーから 16 エントリーへ増加しました。
- 大きな uop キャッシュ。
- 同じコア上の 2 つの論理プロセッサーがアクティブである場合、IDQ は論理プロセッサーあたり 70 uop を保持できます。以前の世代では 64 uop しか保持できませんでした (コアごとに 2 x 70 vs. 2 x 64)。コア上で 1 つの論理プロセッサーのみがアクティブである場合、IDQ は 70 uop を保持できます (以前の世代では 64 uop)。
- IDQ 内の LSD (ループストリーム検出器) は、シングルスレッドまたはマルチスレッド操作にかかわらず論理プロセッサーあたり最大 70 uop を検出できます。

2.3.1.2 アウトオブオーダーと実行エンジン

Ice Lake† Client マイクロアーキテクチャーのアウトオブオーダーと実行エンジンには、次の変更が含まれます。

- リオーダーバッファー、ロードバッファー、ストアバッファー、およびリザベーション・ステーションのサイズが大幅に増加したことで、より深い OOO 実行と高いキャッシュ帯域幅が実現されました。
- よりワイドなマシン。4 → 5 幅の割り当て、8 → 10 実行ポート。.
- 実行ポートごとのギャザー操作 (例: SIMD シャッフル、LEA)。
- 整数除算器のレイテンシーを軽減しました。
- 新しく追加された iDIV ユニットは、レイテンシーを大幅に削減し、整数除算操作のスループットを改善します。

表 2-4 は、異なる操作タイプをポートへディスパッチする OOO エンジンの役割をまとめています。

表 2-4 Ice Lake† Client マイクロアーキテクチャーのディスパッチ・ポートと実行スタック

ポート 0	ポート 1[1]	ポート 2	ポート 3	ポート 4	ポート 5	ポート 6	ポート 7	ポート 8	ポート 9
INT ALU LEA INT Shift Jump1	INT ALU LEA INT MUL INT DIV	ロード	ロード	ストア データ	INT ALU LEA INT MULHi	INT ALU LEA INT Shift Jump2	ストア アドレス	ストア アドレス	ストア データ
FMA Vec ALU Vec Shift FP DIV	FMA* Vec ALU* Vec Shift* Vec Shuffle*				Vec ALU Vec Shuffle				

注意:

1. 表中の “*” で示される機能は、512 ビット・ベクトルでは利用できません。

表 2-2 は、実行ユニットとこれらのユニットに関連する代表的な命令を示します。

インテル® SSE、インテル® AVX、および汎用命令セット全体のスループット向上は、対応する命令向けのユニット数、および特定のユニットで実行される命令のバリエーションに関連しています。

表 2-5 Ice Lake† Client マイクロアーキテクチャーの実行ユニットと対応する命令[1]

実行ユニット	ユニット数	命令
ALU	4	add、and、cmp、or、test、xor、movzx、movsx、mov、(v)movdqu、(v)movdqa、(v)movap*、(v)movup*
SHFT	2	sal、shl、rol、adc、sarx、adcx、adox など
低速 INT	1	mul、imul、bsr、rcl、shld、mulx、pdep など
BM	2	andn、bextr、blsi、blsmsk、bzhi など
Vec ALU	3	(v)pand、(v)por、(v)pxor、(v)movq、(v)movq、(v)movap*、(v)movup*、(v)andp*、(v)orp*、(v)paddb/w/d/q、(v)blendv*、(v)blendp*、(v)pblendd
Vec_Shft	2	(v)psllv*、(v)psrlv*、imm8 のベクトルシフト数
Vec Add	2	(v)addp*、(v)cmpp*、(v)max*、(v)min*、(v)padds*、(v)paddus*、(v)psign、(v)pabs、(v)pavgb、 (v)pcmpeq*、(v)pmax、(v)cvtps2dq、(v)cvtdq2ps、(v)cvtsd2si、(v)cvtss2si
Shuffle	2	(v)shufp*、vperm*、(v)pack*、(v)unpck*、(v)punpck*、(v)pshuf*、(v)pslldq、(v)alignr、(v)pmovzx*、 vbroadcast*、(v)pslldq、(v)psrldq、(v)pblendw
ベクトル Mul	2	(v)mul*、(v)pmul*、(v)pmadd*
SIMD Misc	1	STTNI、(v)pclmulqdq、(v)psadw、xmm のベクトルシフト数
FP Mov	1	(v)movsd/ss、(v)movd gpr
DIVIDE	1	divp*、divs*、vdiv*、sqrt*、vsqrt*、rcp*、vrcp*、rsqrt*、idiv

注意:

1. インテル® MMX® 命令にマッピングされる実行ユニットは、この表ではカバーされていません。15.16.5 節の MMX 命令スループットの制限に対する AVX2 変換の対策を参照してください。

表 2-6 は、生産と消費操作間のサイクルにおけるバイパス遅延を示しています。

表 2-6 生産と消費 uop 間のバイパス遅延

	TO [EU/ポート/レイテンシー]						
FROM [EU/ポート/レイテンシー]	**SIMD/0、1/1**	**FMA/0、1/4**	**VIMUL/0、1/4**	**SIMD 5/1、3**	**SHUF 5/1、3**	**V2I/0/3**	**I2V/5/1**
SIMD/0、1/1	0	1	1	0	0	0	NA
FMA/0、4/1	1	0	1	0	0	0	NA
VIMUL/0、4/1	1	0	1	0	0	0	NA
SIMD 5/1、3	0	1	1	0	0	0	NA
SHUF 5/1、3	0	0	1	0	0	0	NA
V2I/0/3	0	0	1	0	0	0	NA
I2V/5/1	0	1	1	0	0	0	NA

バイパスにおける 生産/消費の uop に関連する属性は、省略形/1 つ以上のポート番号/uop のレイテンシー・サイクルの 3 項目です。次に例を示します。

- “SIMD/0、1/1” は、1 サイクルのベクトル SIMD uop が、ポート 0 または 1 のいずれかにディスパッチされることを示します。
- “SIMD/5/1、3” は、1 もしくは 3 サイクルの SIMD uop が、ポート 5 にディスパッチされることを示します。
- “V2I/0/3” は、3 サイクルのベクトル-整数 uop が、ポート 0 にディスパッチされることを示します。
- “I2V/5/1” は、1 サイクルの整数-ベクトル uop が、ポート 5 にディスパッチされることを示します。

2.3.1.3 キャッシュとメモリー・サブシステム

Ice Lake† Client マイクロアーキテクチャーのキャッシュ階層では、次のような変更が行われています。

- L1 データ・キャッシュ・サイズを 50% 増量。
- 2x L1 ストア帯域幅。3 → 4 AGU、1 → 2 ストアデータ。
- バッファー数が増えたことにより、多くのロードとストアを同時に処理することが可能になりました。
- 前世代に比べ高いキャッシュ帯域幅を実現しました。
- 大きな L2 レベル TLB: 1.5K エントリー → 2K エントリー。
- メモリーの並列性を高めるデータ・プリフェッチャーの拡張。
- L2 キャッシュサイズが、256KB から 512KB に増加しました。
- L2 キャッシュの連想性が 4 ウェイから 8 ウェイに変更されました。
- 実行ロード・レイテンシーが大幅に削減されました。

表 2-7 Ice Lake† Client マイクロアーキテクチャーのキャッシュ・パラメーター

レベル	**容量/連想性**	**ラインサイズ (バイト)**	**レイテンシー[1] (サイクル)**	**ピーク帯域幅 (バイト/サイクル)**	**持続帯域幅 (バイト/サイクル)**	**更新方式**
第 1 レベル (DCU)	48 KB/8	64	5	2×64B ロード + 1 x 64B または 2 x 32B ストア	ピークと同じ	ライトバック
第 2 レベル (MLC)	512KB/8	64	13	64	48	ライトバック
第 3 レベル (LLC)	コアごとに最大 2MB/最大 16 ウェイ	64	xx[2]	32	21	ライトバック

注意:

1. ソフトウェアから見えるレイテンシー/帯域幅は、アクセスパターンなどの要因によって異なります。

2. この数はコア数によって異なります。

TLB 階層は、L1 命令キャッシュ専用の TLB、L1D 用の TLB、4K および 4MB ページ共有の L2 TLB、および 1GB ページ専用の L2 TLB で構成されます。

表 2-8 Ice Lake† Client マイクロアーキテクチャーの TLB パラメーター

レベル	ページサイズ	エントリー (シングルスレッド)	スレッドごとのエントリーのレイテンシー (マルチスレッド)	連想性
命令	4KB	128	64	8
命令	2MB/4MB	16	8	8
第 1 レベルデータ (ロード)	4KB	64	64 競合する共有	4
第 1 レベルデータ (ロード)	2MB/4MB	32	32 競合する共有	4
第 1 レベルデータ (ロード)	1GB	8	8 競合する共有	8
第 1 レベルデータ (ストア)	すべてのページサイズを共有	16	16 競合する共有	16

注意:

1. 4K ページでは 2048 のエントリーをすべて使用できます。2/4MB ページでは 1024 エントリーを使用でき (8 ウェイ)、これらは 4K ページと共有されます。1GB ページでは 1024 エントリーを使用でき (8 ウェイ)、これらも 4K ページと共有されます。

ペアのストア

Ice Lake† Client マイクロアーキテクチャーにはコアに 2 つのストア・パイプラインが備わっており、次の機能がサポートされます。

- ポート 2 と 3 の LD 専用の 2 つの AGU。
- ポート 7 と 8 の STA 専用の 2 つの AGU。
- 2 つの完全な機能を持つ STA パイプライン。
- 2 つの 256 ビット幅 STD パイプライン (インテル® AVX-512 のストアデータの書き込みには 2 サイクルかかります)。
- ストアマージを介した DCU への第 2 の先行パイプライン。

Ice Lake† Client マイクロアーキテクチャーでは 2 つの先行するストアをペアにできる場合、1 サイクルで 2 つの先行するストアをキャッシュに書き込みできます。これには、次の条件を満たす必要があります。

- ストアは同一キャッシュラインへの書き込みである必要があります。
- 2 つのストアは、同じメモリー型 (WB または USWC) でなければなりません。
- キャッシュラインまたはページ境界をまたいではなりません。

2 番目のストアポートから最大限のパフォーマンスを得るため次のことを考慮してください:

- 可能な限りストア操作をアライメントします。
- 連続したストアを同一キャッシュラインに配置します (隣接する命令でなくてもかまいません)。

例 2-6 に見られるように、明示的であるかどうかにかかわらず、すべてのストアを考慮することが重要です。

例 2-6 ストアの考慮

ループ反復間でペアになっているストア	スタック間の更新によりペアとならないストア
Loop: reg を計算 … store [X], reg add X, 4 jmp Loop ; ループの異なる反復からのストアは、通常同じキャッシュラインに行われるため、すべてペアにすることができます。	Loop: reg を計算する関数を呼び出し … store [X], reg add X, 4 jmp Loop ; スタックへの呼び出しストアであるため、ループの異なる反復からのストアはペアにできません。 ; 呼び出しがペアになるのを妨げています。

状況によっては、ストアをペアにするためコードを再配置することができます。詳細は、下記の例を参照してください。

例 2-7.ストアをペアにするコードの再配置

異なるキャッシュラインへのストア - ペアにできない	アンロールにより問題を回避

Loop:
... ymm1 を計算 …
vmovaps [x], ymm1
... ymm2 を計算…
vmovaps [y], ymm2
add x, 32
add y, 32
jmp Loop ; 異なるキャッシュライン [x] と [y] へのストアが交互に切り替わるため、このループではストアをペアにできません。
	Loop:
	... ymm1 を計算…
	vmovaps [x], ymm1
	... 新しい ymm1 を計算…
	vmovaps [x+32], ymm1
	... ymm2 を計算…
	vmovaps [y], ymm2
	... 新しい ymm2 を計算…
	vmovaps [y+32], ymm2
	add x, 64
	add y, 64
	jmp Loop ; ループは 2 回アンロールされ、同一キャッシュラインへの 2 つのストアが連続するようにストアが再配置されています。アドレス [x] と [x + 32] へのストアは同一キャッシュラインにあり、ペアにされて同じサイクルで実行されます。

2.3.1.4 新しい命令

Ice Lake† Client マイクロアーキテクチャーの新しい命令とアーキテクチャーの変更を以下に示します。実際のサポートは製品によって異なります。

- 暗号化のアクセラレーション
  - SHA1 と SHA256 ハッシュ・アルゴリズムを高速化する SHA NI。
  - ビッグナンバー演算 (IFMA): VPMADD52 - ベクトル化された RSA ソフトウェアとその他の暗号化アルゴリズム (公開鍵) のパフォーマンスを高速化するため、ビッグナンバー乗算を行う 2 つの新命令。

  - 各種暗号化アルゴリズム、エラー訂正アルゴリズム、ビット行列乗算を高速化するガロア体新命令 (Galois Field New Instructions - GFNI)。
  - AES と AESGCM を高速化するベクトル AES とベクトル・キャリーレス乗算 (PCLMULQDQ) 命令。

- セキュリティー・テクノロジー
  - 利便性と適用性を改善するインテル® SGX の機能強化: EDMM、複数パッケージのサーバーサポート、VMM メモリーのオーバーサブスクリプションのサポート、パフォーマンス、より大きく安全なメモリー。

- セキュリティー VMM のパフォーマンスを高めるサブページ保護。
- ターゲットを絞った高速化
  - ベクトルビット操作命令: VBMI1 (置換、シフト) と VBMI2 (拡張、圧縮、シフト) - 列指向データベース・アクセス、辞書ベースの展開、離散数学、およびデータ・マイニング・ルーチン (ビット置換とビット行列乗算) に使用されます。
  - 8 ビットと 16 ビットの整数データ型のサポートと VNNI - CNN/ML/DL を高速化。
  - ビット代数 (POPCNT、ビット・シャッフル)。
  - キャッシュラインのライトバック命令 (CLWB) により、キャッシュラインにクリーンなコピーを保ちながら、メモリーへのキャッシュラインの高速更新が可能になります。

- さらに効率良いパフォーマンス・ソフトウェアのチューニングとデバッグを可能にするプラットフォーム解析機能。
  - AnyThread の排除。
  - 2 倍の汎用カウンター (スレッドごとに最大 8 個)。
  - 発行スロット向けの 3 つの固定カウンター。
  - レベル 1 のトップダウン方式の組込みサポート向けの新しいパフォーマンス・メトリック (フロントエンド依存、バックエンド依存、投機の問題、リタイアの発行スロットの割合)。ソフトウェアは、8 つの汎用カウンターで使用できます。

2.3.1.5 Ice Lake† Client マイクロアーキテクチャーの電力管理

Ice Lake† Client マイクロアーキテクチャーを採用するプロセッサーは、コアが互いに異なる周波数で実行される最初のクライアント・プロセッサーです。周波数は、特定の命令の組み合わせに基づいて選択されます。それぞれのコアで実行されるプログラムのベクトル命令のタイプ、幅そして数、各コアのアクティブ時間とアイドル時間の比率、および同じ特性を有するコアの数などの考慮事項があります。

Skylake† Server マイクロアーキテクチャー (2.4 節を参照) の電源管理機能の大部分は、Ice Lake† Client マイクロアーキテクチャーにも適用できます。次のような相違点があります。

- Ice Lake† Client マイクロアーキテクチャーでのインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) とインテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2) の一般的な P0n 最大周波数の差は、Skylake† Server マイクロアーキテクチャーよりもはるかに小さくなっています。そのため、アプリケーション全体のパフォーマンスへの負の影響は大幅に小さくなります。
- Ice Lake† Client マイクロアーキテクチャー・ベースのすべてのプロセッサーには、1 つの 512 ビット FMA ユニットが備わっていますが、Skylake† Server マイクロアーキテクチャー・ベースのプロセッサーには 2 つの FMA ユニットがあります。両方のアーキテクチャーのプロセッサーには、2 つの 256 ビット FMA ユニットがあります。Ice Lake† Client の FMA ユニットが消費する電力は 256 ビットと 512 ビット・ユニットで同じですが、Skylake† Server の 512 ビット・ユニットは 2 倍の電力を消費します。

複数の Ice Lake† Client コアで重いワークロードを計算すると、電力制限によりインテル® AVX-512 およびインテル® AVX2 命令セットは P0n より低い周波数で実行されます。このような状況では、同じタスクを実行するのにインテル® AVX-512 はインテル® AVX2 よりも少ない命令数で済むため、消費電力が少なくなり、より高い周波数で動作できます。最終的に、実行パスが短くなり、周波数が少し高くなるため、パフォーマンスが向上する可能性があります。

インテル® AVX-512 命令セットでのコーディングは、インテル® AVX2 命令セットでのコーディングよりもパフォーマンスが低くなることがあります。その原因は、長いベクトルのマイクロアーキテクチャーであることも、自然なベクトルでは長さが足りないこともあります。ほとんどのコンパイラーは、まだインテル® AVX-512 のサポートが十分ではなく、最適なコードを生成するにはまだ時間を要するでしょう。

Skylake† Server の電源管理の一般的な推奨事項 (2.4.3 節を参照) は、引き続き有効です。開発者は、インテル® AVX-512 命令セットを使用するコードを生成し、Ice Lake† Client マイクロアーキテクチャー上のインテル® AVX2 を使用するワークロードとパフォーマンスを比較してから、完全に移行するか決定する必要があります。

2.4 Skylake† Server マイクロアーキテクチャー

インテル® Xeon® スケーラブル・プロセッサーは、Skylake† Server マイクロアーキテクチャーをベースとしています。Skylake† マイクロアーキテクチャー・ベースのプロセッサーは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 4』の第 2 章の表 2-1 に記載される CPUID の DisplayFamily_DisplayModel シグネチャーを使用して検出することができます。

Skylake† Server マイクロアーキテクチャーは、アプリケーションのパフォーマンスと電力消費を最適化するため、次の新機能[6]を導入しています。

- Skylake† Server マイクロアーキテクチャー・ベースの新しいコアは、Kaby Lake† マイクロアーキテクチャーを基に改善を行っています。
- インテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) のサポート。
- ソケットあたりのコア数の増加 (最大 28 対最大 22)。
- Skylake† マイクロアーキテクチャーではソケットあたり 6 本のメモリーチャンネル (Broadwell† マイクロアーキテクチャーは 4 チャンネル)。
- より大きな L2 キャッシュと小さな非インクルーシブな L3 キャッシュ。
- インテル® Optane™ テクノロジーをサポート。
- インテル® Omni-Path アーキテクチャー (インテル® OPA)。
- サブ NUMA クラスタリング (SNC) をサポート。

図 2-4 に記される星印は、クライアント向け Skylake† マイクロアーキテクチャーに対する Skylake† Server マイクロアーキテクチャーでの新機能を示します: 1MB の L2 キャッシュとポート 5 に追加されたインテル® AVX-512 ユニット (製品によって利用可能)。

ポート 0 とポート 1 は 256 ビット幅であるため、インテル® AVX-512 操作はポート 0 にディスパッチされポート 0 とポート 1 の両方で実行されます。しかし、*lea* などのほかの操作はポート 1 で並列に実行できます。ポート 0 と 1 のフュージョンは図 2-1 の赤い囲みで行われます。クライアント版の Skylake† マイクロアーキテクチャーとは異なり、Skylake† Server マイクロアーキテクチャーでは、フロントエンドのループストリーム検出器 (LSD) が無効化されていることに注意してください。

[6] いくつかの機能はすべてのプロセッサー上で利用できるとは限りません。

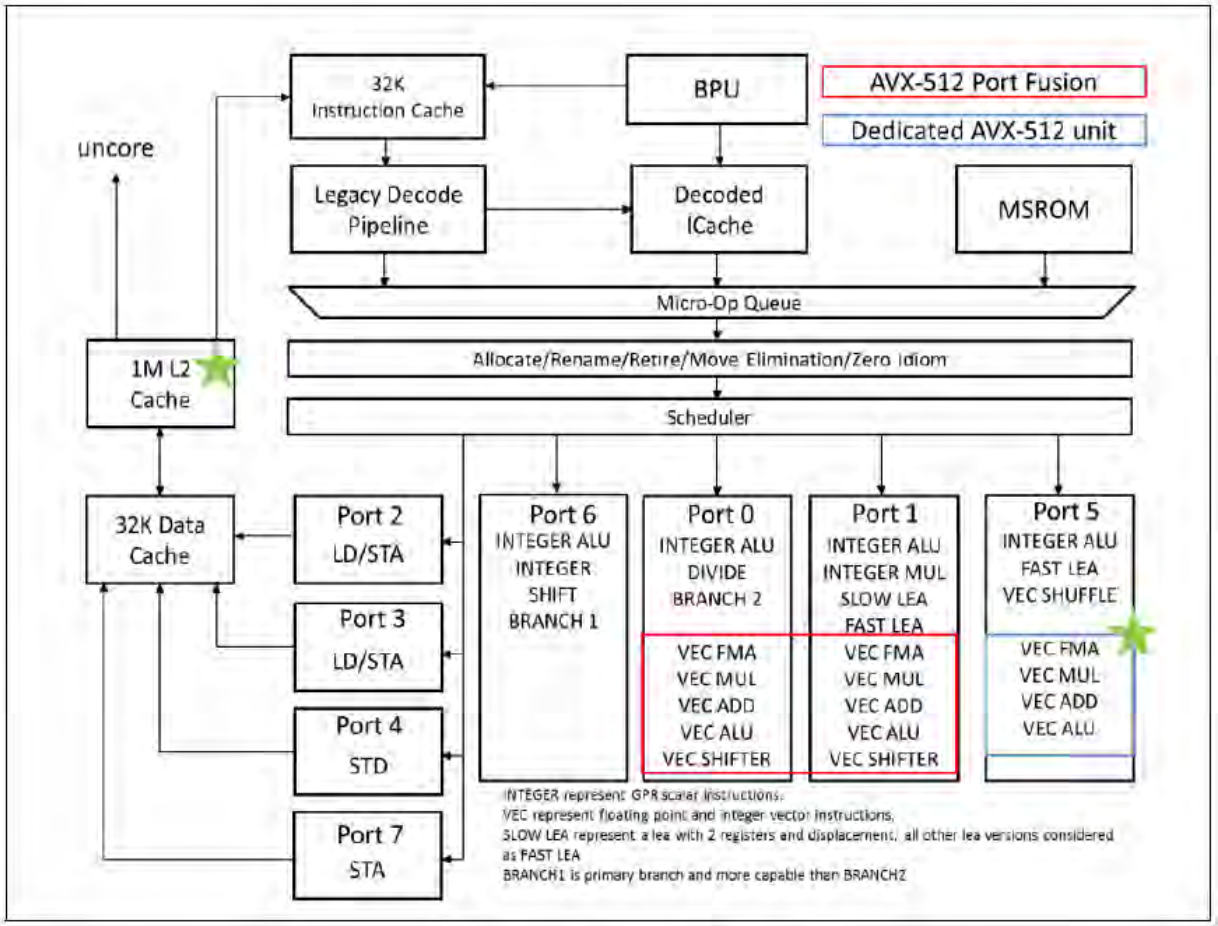


図 2-4 Skylake† Server マイクロアーキテクチャーの CPU コア・パイプラインの機能

2.4.1 Skylake† Server マイクロアーキテクチャーのキャッシュ

Skylake† Server マイクロアーキテクチャー・ベースのインテル® Xeon® スケーラブル・プロセッサーは、以前の世代の Broadwell† マイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサーと比較していくつかのコンポーネントのパフォーマンスとスケーラビリティーを改善するためコアとアンコア・アーキテクチャーが大幅に変更されています。

2.4.1.1 より大きな中間レベルキャッシュ

Skylake† Server マイクロアーキテクチャーは、1MB の容量を持つ中間レベル (L2) を実装し、読み出してから使用するまで最小 14 サイクルのレイテンシーを提供します。この中間レベルキャッシュは、以前の世代のインテル® Xeon® プロセッサーにおける実装より 4 倍の容量を持っています。中間レベルのキャッシュのラインサイズは 64 バイトで 16 ウェイの連想性を備えています。中間レベルのキャッシュはそれぞれのコアでプライベートです。

中間レベルのキャッシュにデータを収めるように最適化されているソフトウェアは、Skylake† Server マイクロアーキテクチャーの容量が増えた中間レベルキャッシュの利点を得るため修正する必要があるかもしれません。

2.4.1.2 非インクルーシブなラスト・レベル・キャッシュ (LLC)

Skylake† Server のラスト・レベル・キャッシュは、非インクルーシブな分散方式の共有キャッシュです。ラスト・レベル・キャッシュの各バンクサイズは、バンクごとに 1.375MB にシュリンクされています。ラスト・レベル・キャッシュが非インクルーシブであることから、あるコアの中間レベルキャッシュに送られたブロックは、ラスト・レベル・キャッシュのバンクにコピーを持たない可能性があります。アクセスパターン、アクセスされたコードとデータのサイズ、そしてコア間でのキャッシュブロック共有の挙動に基づいて、ラスト・レベル・キャッシュは中間レベルキャッシュのビクティム

キャッシュのように見え、コアごとの総キャッシュ容量はそれぞれのコアのプライベートな中間レベルキャッシュとラスト・レベル・キャッシュの部分的な組み合わせとして見えるかもしれません。

2.4.1.3 Skylake† Server マイクロアーキテクチャーにおけるキャッシュの推奨事項

Skylake† Server マイクロアーキテクチャーのキャッシュと以前の Broadwell† マイクロアーキテクチャー世代のキャッシュの簡単な比較を表に示します。

表 2-9 Skylake† マイクロアーキテクチャーと Broadwell† マイクロアーキテクチャーのキャッシュの比較

キャッシュレベル	**カテゴリー**	**Broadwell† マイクロアーキテクチャー**	**Skylake† Server マイクロアーキテクチャー**
L1 データ・キャッシュ・ユニット (DCU)	サイズ [KB]	32	32
	レイテンシー [サイクル]	4-6	4-6
	最大帯域幅 [バイト/サイクル]	96	192
	持続帯域幅 [バイト/サイクル]	93	133
	連想性 [ウェイ]	8	8
L2 中間キャッシュ (MLC)	サイズ [KB]	256	1024 (1MB)
	レイテンシー [サイクル]	12	14
	最大帯域幅 [バイト/サイクル]	32	64
	持続帯域幅 [バイト/サイクル]	25	52
	連想性 [ウェイ]	8	16
L3 ラスト・レベル・キャッシュ (LLC)	サイズ [MB]	コアあたり最大 2.5	コアあたり最大 1.375[1]
	レイテンシー [サイクル]	50-60	50-70
	最大帯域幅 [バイト/サイクル]	16	32
	持続帯域幅 [バイト/サイクル]	14	15

注意:

1. Skylake† Server 製品ではいくつかのコアが無効化されて、コアあたり 1.375MB 以上の L3 キャッシュを保持するものがあります。

次の表は、Skylake† Server マイクロアーキテクチャーのメモリーバランスが、高いレイテンシーの共有分散方式から低レイテンシーのプライベート・ローカル方式に移行していることを示しています。

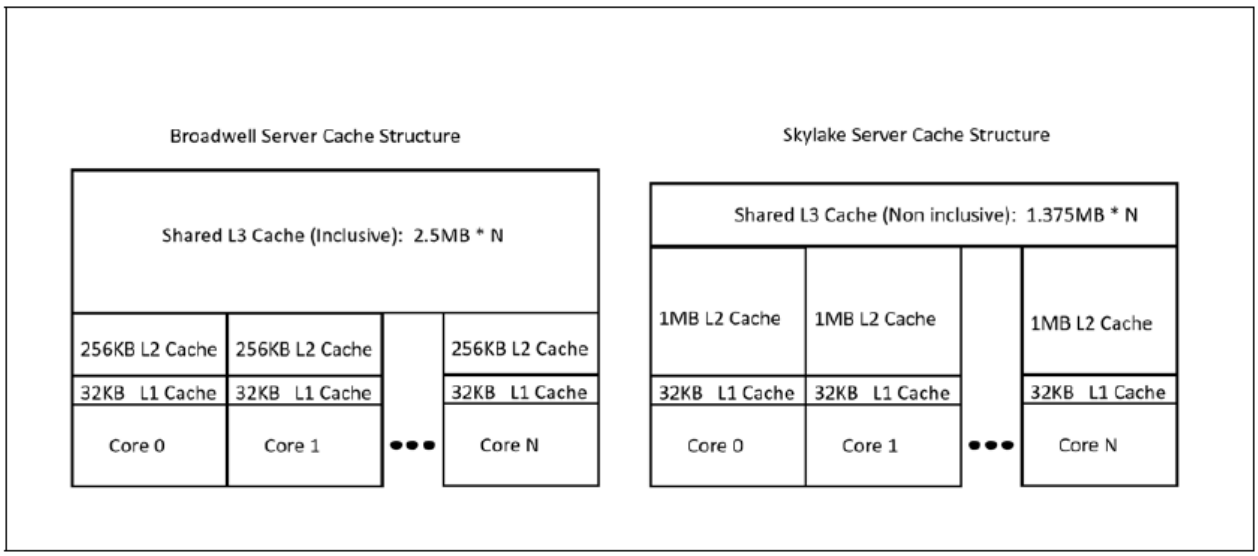


図 2-5 Broadwell† マイクロアーキテクチャーと Skylake† Server マイクロアーキテクチャーのキャッシュ構造

このキャッシュ・アーキテクチャーの変更から得られるパフォーマンス・ゲインの可能性は高く、ソフトウェアは新しいキャッシュサイズに適合するようにメモリーのタイル化を導入する必要があります。

推奨事項: アプリケーションの共有およびプライベート・データのサイズを、小さな非インクルーシブ L3 キャッシュと大きな L2 キャッシュに合わせて再度バランスを取ります。

キャッシュ・ブロッキングは、アプリケーションの帯域幅要件と、アプリケーション間の切り換えに基づいて選択する必要があります。4 倍の L2 キャッシュサイズと 2 倍の L2 キャッシュ帯域幅は、以前の Broadwell† マイクロアーキテクチャー世代と比較すると、アプリケーションによっては L1 に変わって L2 へのブロック化することを可能にし、それによりパフォーマンスを向上させます。

推奨事項: L2 キャッシュがアプリケーションの帯域幅要件を満たすことができるのであれば、Skylake† Server マイクロアーキテクチャーでは L2 へのブロック化を検討します。

ラスト・レベル・キャッシュがインクルーシブから非インクルーシブに変更されたことは、中間レベルキャッシュとラスト・レベル・キャッシュの容量を合計できることを意味します。実行時にコアごとのキャッシュ容量を求めるプログラムでは、有効なキャッシュサイズを算出するためコアごとに中間レベルキャシュとラスト・レベル・キャッシュのサイズを合計します。コアごとにラスト・レベル・キャッシュのサイズをだけを算出すると、利用可能なオンチップキャッシュを適切に利用できない可能性があります。詳細は、2.4.2 節を参照してください。

推奨事項: データを共有しない場合、アプリケーションは コアあたりのキャッシュ容量を L3 キャッシュだけでなく L2 と L3 キャッシュサイズの合計であると見なすべきです。

2.4.2 Skylake† Server マイクロアーキテクチャー上での非テンポラルなストア

Skylake† Server マイクロアーキテクチャーにおいてラスト・レベル・キャッシュの各バンクのサイズが変更されたことにより、アプリケーション、ライブラリー、またはドライバーがコアごとのオンチップキャッシュのサイズを決定するためラスト・レベル・キャッシュだけを考慮していると、Skylake† Server マイクロアーキテクチャーでは減少して見え、小さなブロックのメモリー書き込みとともに非テンポラルなストアを使用する可能性があります。非テンポラルなストアはキャッシュラインを追い出してメモリーに書き戻しを行うため、以前の世代のインテル® Xeon® プロセッサー・ファミリーと比べると、Skylake† Server マイクロアーキテクチャーでは以降のキャッシュミスとメモリー帯域幅の要求が増加するかもしれません。

また、Skylake† Server マイクロアーキテクチャーによる非テンポラルなストアによって生じるアクセスの扱いが変更されたことにより、以前の世代のインテル® Xeon® プロセッサー・ファミリーと比べると、同じようなアクセスを

行っても各コア内のリソースが長時間ビジーのままになります。その結果、そのような命令シーケンスが実行されると、プロセッサーはリソース不足からストールし、各コアからのメモリー書き込み帯域幅が制限される可能性があります。

非テンポラルなストアの多用からキャッシュミスが増加すると、非テンポラルなストアによるコアごとのメモリー書き込み帯域幅の制限からパフォーマンスが低下するアプリケーションもあります。

Skylake† Server マイクロアーキテクチャーで前述のパフォーマンスの問題を回避するには、アプリケーション、ライブラリー、およびドライバーが各コアで利用可能なオンチップキャッシュを調査する際に、それぞれのコアのラスト・レベル・キャッシュに加え中間キャッシュの容量を含めます。Skylake† Server マイクロアーキテクチャーで利用可能なオンチップキャッシュの容量をこのように求めるのは、非インクルーシブなラスト・レベル・キャッシュが実装されていることを考慮しているためです。

2.4.3 Skylake† Server の電力管理

この節では、Skylake† Server の電力管理とベクトル ISA の相互作用について説明します。

Skylake† Server マイクロアーキテクチャーは、それぞれのコアの実行周波数を動的に選択します。選択される周波数は、命令タイプ、命令幅、および一定時間に実行されるベクトル命令数などの組み合わせに依存します。プロセッサーはまた、同様の特性を持つコアの数を考慮します。

Broadwell† マイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサーも同じように振る舞いますが、256 ビット・ベクトル命令のみをサポートするため影響は軽度です。Skylake† Server マイクロアーキテクチャーがサポートするインテル® AVX-512 命令は、潜在的にインテル® AVX2 命令よりも多くの電流と電力を必要とします。

プロセッサーは、必要に応じて動的に最大周波数をより高い/低いレベルに調整します。そのため、プログラムが実行されている間の最大周波数は異なる可能性があります。

表 2-10 は、各命令タイプが実行される際のインテル® ターボ・ブースト・テクノロジーの最大コア周波数を示しています。最大周波数 (POn) は、カテゴリー内のコア数に応じた周波数の配列で表されます。多くのコアがある時点でカテゴリーに含まれると、最大周波数は低くなります。

表 2-10 インテル® ターボ・ブースト・テクノロジーの最大コア周波数レベル

レベル	カテゴリー	周波数レベル	最大周波数 (POn)	命令タイプ
0	軽量なインテル® AVX2 命令	最も高い	最大	スカラー、128 ビットのインテル® AVX、インテル® SSE、インテル® AVX2 (FP または INT MUL/FMA なし)
1	重いインテル® AVX 2 命令 + 軽量なインテル® AVX-512 命令	中間	最大インテル® AVX2	インテル® AVX2 FP + INT MUL/FMA、インテル® AVX-512 (FP または MUL/FMA なし)
2	重いインテル® AVX-512 命令	最低	最大インテル® AVX-512	インテル® AVX-512 FP + INT MUL/FMA

製品ごとの動作周波数の詳細 (図 1-15 を参照) については、インテル® Xeon® スケーラブル・プロセッサーの仕様アップデートを参照してください: https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-spec-update.html (英語)。

図 2-6 は、それぞれのコアの周波数がワークロードの要求に基づいて独立して決定される、特定のシステムのコア周波数の範囲を示しています。

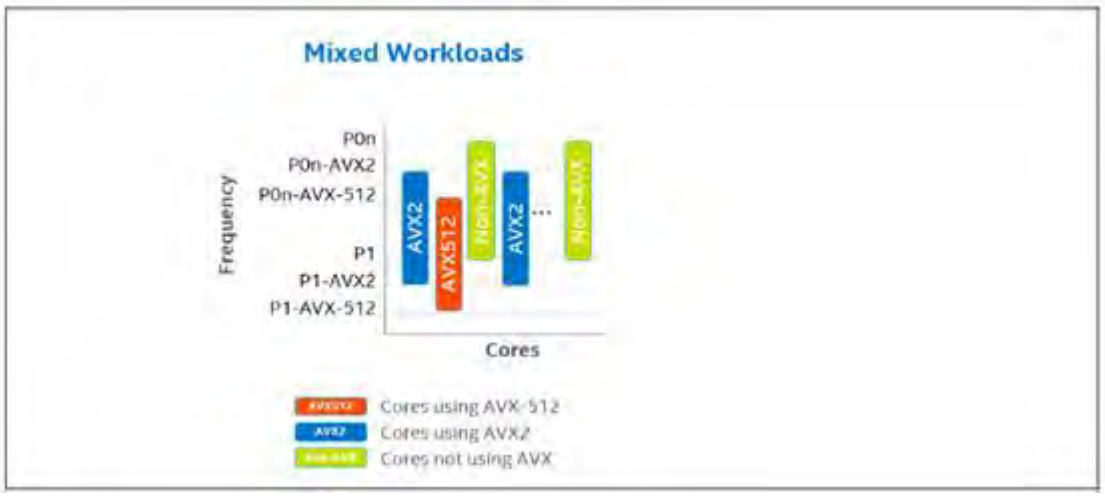


図 2-6 ワークロードの混在

次のパフォーマンス・モニタリング・イベントは、3 つの周波数レベルにおいて費やされるサイクル数を決定するために使用されます。

- CORE_POWER.LVL0_TURBO_LICENSE: 最大周波数が P0n でコアが動作したコアサイクル。
- CORE_POWER.LVL1_TURBO_LICENSE: 最大周波数が P0n-AVX2 でコアが動作したコアサイクル。
- CORE_POWER.LVL2_TURBO_LICENSE: 最大周波数が P0n-AVX-512 でコアが動作したコアサイクル。

コアが現在よりも高いライセンスレベルを要求すると、新たなライセンスに移行するのに最大 500 マイクロ秒かかります。それまでコアは低位のピーク周波数で動作します。この間、PCU は何個のコアが新しいライセンスレベルで動作するかを評価し、必要に応じてその周波数を調整します。周波数は下がる可能性があります。他のライセンスレベルで実行されるコアは、影響を受けません。

高い周波数レベルに戻る前には、およそ 2ms のタイマーが適用されます。新しいライセンスが要求されると、どのような状態でもタイマーはリセットされます。

注意

誤って予測されたパスで命令が実行されると、ライセンスの移行要求が発生する可能性があります。

軽量なインテル® AVX-512 命令と重いインテル® AVX2 命令の十分な大きさの混在は、通常ライセンス 1 に割り当てられているにも関わらず、ライセンス 2 を供給するようにコアを駆動します。同様に、軽量のインテル® AVX2 命令と重いインテル® SSE 命令の混在は、ライセンス 0 ではなくライセンス 1 でコアを駆動します。例えば、65 サイクルのウィンドウで 110 個の軽量なインテル® AVX-512 命令と 20 個の重い 256 ビット命令を混在して実行すると、インテル® Xeon® Platinum 8190 プロセッサーはライセンス 1 からライセンス 2 に移行します。

あるワークロードで、それらのワークロードが他の要因によって制限される場合、プロセッサーは最大周波数に達することはありません。例えば、LINPACK ベンチマークでは、電力が制限されプロセッサーは最大周波数に達しません。次のグラフはベクトル幅が広がるにつれて周波数が低下することを示していますが、周波数が低下しているにもかかわらずパフォーマンスは向上しています。このデータは、インテル® Xeon® Platinum 8190 プロセッサー上で測定されました。

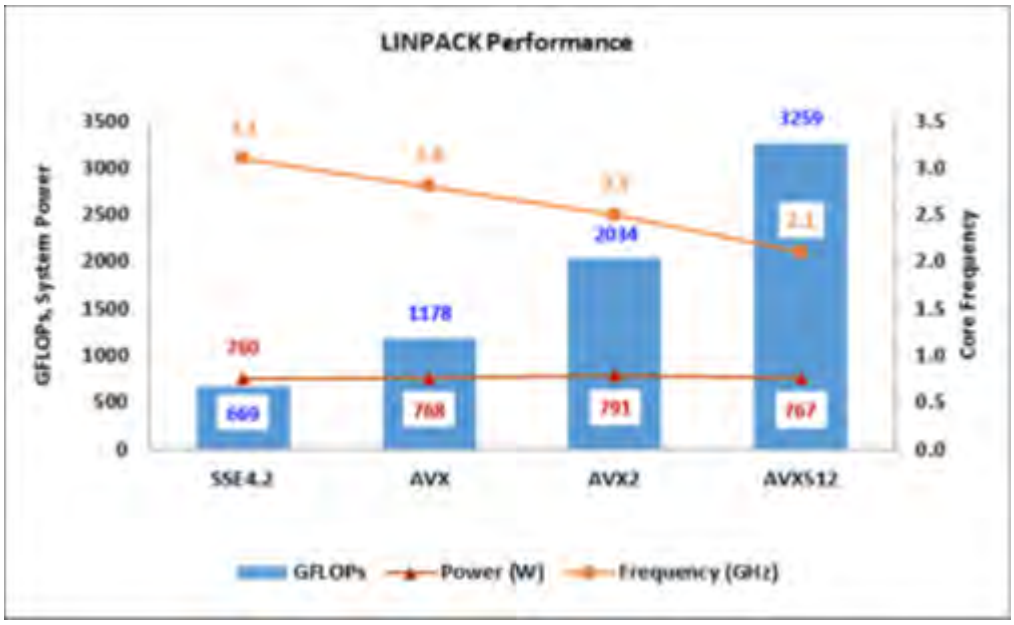


図 2-7 LINPACK のパフォーマンス

インテル® AVX-512 命令を実行するワークロードが、命令全体に占める割合が大きいと、インテル® AVX2 命令に比べパフォーマンス・ゲインは高くなりますが、動作周波数は低くなる可能性があります。例えば、高い比率で重いインテル® AVX-512 命令を最大周波数で要求するディープラーニングのワークロードは、インテル® AVX2 をターゲットとする同じワークロードに対し、1.3 倍から 1.5 倍のパフォーマンスを向上できます (両者とも Skylake† Server マイクロアーキテクチャー上で実行)。

しかし、インテル® AVX-512 命令をターゲットとしてビルドすることで、プログラムのパフォーマンスが向上するか予測することは常に容易ではありません。xmm や ymm レジスターを使用して高いパフォーマンスを得ているプログラムは、zmm レジスターに移行するすることでパフォーマンスがさらに向上することが期待されます。しかし、zmm レジスターを使用するプログラムのいくつかは、十分なパフォーマンスを得られないか、または低下する可能性があります。複数命令セットを使用するバイナリーを生成して、パフォーマンスを測定することを推奨します。

推奨事項: 最適なコンパイラー・オプションを特定するため、次に示す命令セットオプションでアプリケーションをビルドして、最も高いパフォーマンスを示す命令セットを選択します。

- `-xCORE-AVX2 -mtune=skylake-avx512` (Linux* および macOS*)
  `/QxCORE-AVX2 /tune=skylake-avx512` (Windows*)
- `-xCORE-AVX512 -qopt-zmm-usage=low` (Linux* および macOS*)
  `/QxCORE-AVX512 /Qopt-zmm-usage:low` (Windows*)
- `-xCORE-AVX512 -qopt-zmm-usage=high` (Linux* および macOS*)
  `/QxCORE-AVX512 /Qopt-zmm-usage:high` (Windows*)

これらのオプションの詳細については、18.26 節「コンパイラーを利用するヒント」を参照してください。

GCC コンパイラーは、ベクトル幅を選択するため -mprefer-vector-width=[none|128|256|512] オプションを提供します。-march=skylake-avx512 オプションは、Skylake† Server マイクロアーキテクチャーで最高のパフォーマンスを発揮するように設計されていますが、いくつかのプログラムでは異なるベクトル幅を指定することで利点が得られます。最適なコンパイラー・オプションを特定するため、次に示す命令セットオプションでアプリケーションをビルドして、最も高いパフォーマンスを示す命令セットを選択します。-mprefer-vector-width=256 は、skylake-avx512 のデフォルトです。

- `-march=skylake -mtune=skylake-avx512`
- `-march=skylake-avx512`
- `-march=skylake-avx512 -mprefer-vector-width=512`

Clang/LLVM にも、GCC と同様に -mprefer-vector-width=none|128|256|512 オプションが実装されました。最適なコンパイラー・オプションを特定するため、次に示す命令セットオプションでアプリケーションをビルドして、最も高いパフォーマンスを示す命令セットを選択します。

- `-march=skylake -mtune=skylake-avx512`
- `-march=skylake-avx512` (可能であれば、`-mprefer-vector-width=256` を追加)
- `-march=skylake-avx512` (可能であれば、`-mprefer-vector-width=512` を追加)

2.5 Skylake† Client マイクロアーキテクチャー

Skylake† Client マイクロアーキテクチャーは、Haswell† および Broadwell† マイクロアーキテクチャーの成功の上に構築されています。図 2-8 に Skylake† Client マイクロアーキテクチャーの基本パイプライン機能を示します。

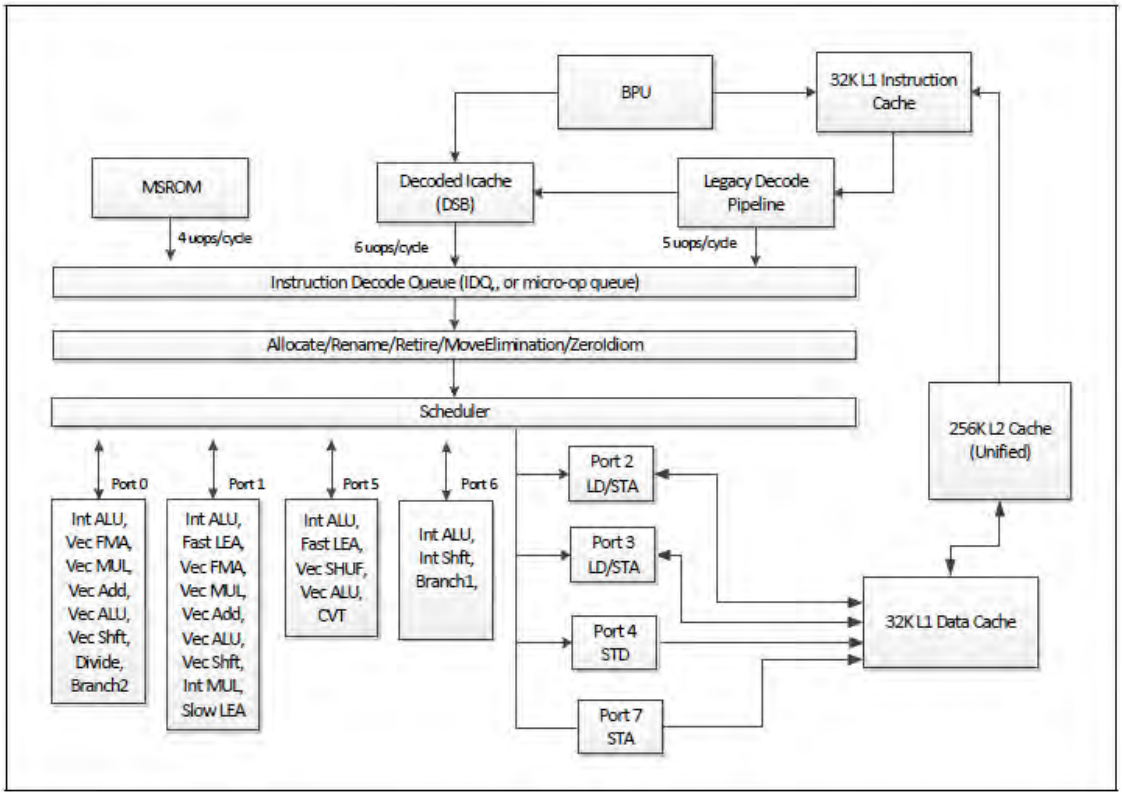


図 2-8 Skylake† Client マイクロアーキテクチャーの CPU コア・パイプラインの機能

Skylake† Client マイクロアーキテクチャーには次の拡張が含まれます。

- 深い OOO 実行と高いキャシュ帯域幅を可能にする大きな内部バッファー。
- フロントエンドのスループットを改善。
- 分岐予測器の改善。
- 除算器のスループットとレイテンシーの改善。
- 低消費電力化。
- インテル® ハイパースレッディング・テクノロジーと SMT パフォーマンスの改善。
- バランスの取れた浮動小数点 ADD、MUL、FMA 命令のスループットとレイテンシー。

マイクロアーキテクチャーは、リング接続された複数スライスの L3 (オフダイの L4 はオプション)、プロセッサー・グラフィックス、統合メモリー・コントローラー、インターコネクト・ファブリックなどから構成される、共有アンコア・サブシステムと複数プロセッサー・コアの柔軟な統合をサポートします。4 コア構成では、20.2.8.9 付録 E「旧世代の

インテル® 64 および IA-32 プロセッサー・アーキテクチャー」の 図 E-2 に示す配置と同様の構成がサポートされます。

2.5.1 フロントエンド

Skylake† Client マイクロアーキテクチャーのフロントエンドでは、1 世代前のマイクロアーキテクチャーに対し次のような改善が行われています。

- レガシー・デコード・パイプラインは、前世代の 4 uop に対してサイクルあたり IDQ (命令デコードキュー) へ 5 つの uop を供給します。
- DSB (デコード済み命令キャッシュ) は、前世代の 4 uop に対してサイクルあたり IDQ へ 6 つの uop を供給します。
- 同じコア上の 2 つの論理プロセッサーがアクティブである場合、IDQ は論理プロセッサーあたり 64 uop を保持できます。以前の世代では 28 uop しか保持できませんでした (コアごとに 2 x 64 vs. 2 x 28)。
- コア上で 1 つの論理プロセッサーのみがアクティブである場合、IDQ は 64 uop を保持できます (ST 操作において 64 vs. 56)。
- IDQ 内の LSD (ループストリーム検出器) は、ST または SMT 操作にかかわらず論理プロセッサーあたり最大 64 uop を検出できます。
- 分岐予測器の改善。

2.5.2 アウトオブオーダー実行エンジン

Skylake† Client マイクロアーキテクチャーのアウトオブオーダーと実行エンジンには、次の変更が含まれます。

- バッファーが拡大されたことで、前世代に比べ深い OOO 実行を可能にします。
- 除算/平方根および逆数近似のスループットとレイテンシーが改善されました。
- FMA ユニットで実行されるすべての操作のレイテンシーとスループットが同一になりました。
- 長いポーズ・レイテンシーは、さらに高い電力効率と SMT パフォーマンス・リソースの利用を可能にします。
- 表 2-8 は、異なる操作タイプを各種ポートへディスパッチする OOO エンジンの役割をまとめています。

表 2-11 Skylake† Client マイクロアーキテクチャーのディスパッチ・ポートと実行スタック

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STD	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

表 2-12 は、実行ユニットとこれらのユニットに関連する代表的な命令を示します。インテル® SSE、インテル® AVX、および汎用命令セット全体のスループット向上は、対応する命令向けのユニット数、および特定のユニットを使用して実行する命令のバリエーションに関連しています。

表 2-12 Skylake† Client マイクロアーキテクチャーの実行ユニットと主要な命令[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

注意:

1. インテル® MMX® 命令にマッピングされる実行ユニットは、この表ではカバーされていません。15.16.5 節のインテル® MMX® 命令スループットの制限に対するインテル® AVX2 変換の対策を参照してください。

インテル® SSE、インテル® AVX、および汎用命令の重要な部分では、レイテンシーも改善されています。付録 C に詳細を示します。ソフトウェアから見えるレイテンシーは、生産側のマイクロオペレーション (uop) のフローと消費側の uop のフロー間の関係に依存して、追加の遅延を含む可能性があります。例えば、VPMULLD などの 2 uop 命令は、それぞれの 2 uop VPMULLD から 1 サイクルのバイパス遅延を累積する可能性があります。

表 2-13 は、生産側の uop と消費側の uop 間の、サイクルにおけるバイパス遅延を示しています。最左の列は、生産側の uop のさまざまな状況における特性を示します。最上位行は、消費側の uop のさまざまな状況における特性を示します。

表 2-13 生産側と消費側の uop 間のバイパス遅延

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

バイパスにおける生産/消費の uop に関連する属性は、省略形/1 つ以上のポート数/uop のレイテンシー・サイクルの 3 項目です。次に例を示します。

- “SIMD/0,1/1” は、1 サイクルのベクトル SIMD uop が、ポート 0 または 1 のいずれかにディスパッチされることを示します。
- “VIMUL/0,1/4” は、4 サイクルのベクトル整数乗算 uop が、ポート 0 または 1 のいずれかにディスパッチされることを示します。
- “SIMD/5/1,3” は、1 もしくは 3 サイクルの非シャッフル uop が、ポート 5 にディスパッチされることを示します。

2.5.3 キャッシュとメモリー・サブシステム

Skylake† Client マイクロアーキテクチャーのキャッシュ階層では、次のような拡張が行われています。

- 前世代に比べ高いキャッシュ帯域幅。
- バッファー数が増えたことにより、多くのロードとストアを同時に処理することが可能になりました。
- Haswell† マイクロアーキテクチャーやそれ以前の世代と比べて、プロセッサーは並列に 2 つのページウォークができるようになりました。
- ページ分割ロードのペナルティーは、前世代の 100 サイクルから大幅に軽減され 5 サイクルになりました。
- L3 の書き込み帯域幅は、前世代の 1 ラインあたり 4 サイクルから、2 ラインあたり 4 サイクルに増加しています。
- CLFLUSHOPT 命令でキャッシュラインをフラッシュし、SFENCE 命令を使用してフラッシュされたデータのメモリー順序を管理することが可能になりました。
- NULL ポインターを指定するソフトウェア・プリフェッチのパフォーマンス・ペナルティーが軽減されました。
- L2 の連想性が 8 ウェイから 4 ウェイに変更されました。

表 2-14 Skylake† Client マイクロアーキテクチャーのキャッシュ・パラメーター

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

注意:

1. ソフトウェアから見えるレイテンシーは、アクセスパターンとその他の要因に依存するため異なります。

TLB の階層は、命令キャッシュ向けの TLB、L1D 向けの TLB、さらに L2 向けのユニファイド TLB から成ります。表 2-15 の Partition (パーティション) カラムは、インテル® ハイパースレッディング・テクノロジーが有効である際のリソース共有ポリシーを示します。

表 2-15 Skylake† Client マイクロアーキテクチャーの TLB パラメーター

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

2.5.4 Skylake† Client マイクロアーキテクチャーのポーズ・レイテンシー

PAUSE 命令は、一般に、同一プロセッサー・コアに配置される 2 つの論理プロセッサーで実行されるソフトウェア・スレッドが、ロックの開放を待機する際に使用されます。このような短いループは、数十から数百サイクルで終了する傾向があります。そのためパフォーマンスの観点からは、OS に任せるよりは短時間 CPU を占有する方が望ましいでしょう。待機ループが数千サイクル以上続くことが予測される場合、Windows* の WaitForSingleObject や Linux* の futex などの OS が提供する同期 API 関数を呼び出し、OS に任せことが推奨されます。

PAUSE 命令は次のことを意図します。

- 共有ハードウェア・リソースを競合する兄弟論理プロセッサー (スピンループを抜けて続行する準備ができている) を一時的に提供します。マイクロアーキテクチャー的にリソースを競合しながら共有する兄弟論理プロセッサーは、次に示す Skylake† Client マイクロアーキテクチャーを利用できます。
  - デコード済み命令キャッシュ、LSD、および IDQ のフロントエンド・スロット
  - RS の実行スロット

- 次の構成で等価なスピンループ命令シーケンスを実行するのと比べると、プロセッサー・コアで消費される電力を節約します。
  - 1 つの論理プロセサーがアクティブではない (例えば、C ステートに入っている)
  - 同じコアの両方の論理プロセッサーが PAUSE 命令を実行する
  - HT が無効化されている (BIOS オプションを使用して)

前の世代のマイクロアーキテクチャーの PAUSE 命令のレイテンシーは、およそ 10 サイクルでしたが、Skylake† Client マイクロアーキテクチャーでは最大 140 サイクルまで拡大しています。

レイテンシーが増加したことは、高度にスレッド化されたアプリケーションにおいてはわずかですが 1-2% のパフォーマンスが向上します (続行する準備が整っている論理プロセッサーに、競合しながら共有されるマイクロアーキテクチャー上のリソースをより効率良く利用することを可能にします)。進行が固定回数のループ内の PAUSE 命令によってブロックされていない場合、それほどスレッド化されていないアプリケーションではごくわずかな影響を受けると予想されます。2 コアや 4 コアのシステムでは消費電力の利点もわずかです。

PAUSE 命令のレイテンシーがかなり増加したことで、PAUSE のレイテンシーに影響を受けやすいワークロードはパフォーマンスの低下を被ります。

以下の例では、動的な反復カウントを持つループでの PAUSE 命令の使い方を示します。Skylake† Client マイクロアーキテクチャーでは、RDTSC 命令は現在のプロセッサー・クロック (INVARIANT TSC 特性を参照) とは独立してマシンの保証された P1 周波数をカウントします。そのため、インテル® ターボブーストが有効である場合、遅延は一定のままですが実行された可能性がある命令数は変化します。

ロックで PollDelay 関数を使用して、保証された P1 周波数サイクルだけ待機します (“clocks” 変数で指定される)。

例 2-8 動的なポーズループの例

```
#include <x86intrin.h>
#include <stdint.h>

/* ラップする可能性があるタイムスタンプを処理する便利な述語関数。
a は b の前か? タイムスタンプがラップされる可能性があるため、
a から b へ時計回りにすべきか、逆回りにすべきかを提起しています。
時計回りの方が反時計回りよりも時間が短い場合、
将来であり、その他は過去です。例えば、a= MAX-1、b = MAX +1 (=0)
の場合、a > b (真) は a が b に到達したことを意味しません。
signed(a) = -2、signed(b) = 0 は、実際の差を示します */

static inline bool before(uint64_t a, uint64_t b)
{
    return ((int64_t)b - (int64_t)a) > 0;
}
void pollDelay(uint32_t clocks)
{
    uint64_t endTime = _rdtsc()+ clocks;
    for (; before(_rdtsc(), endTime); )
      _mm_pause();
}
```

以下のベースラインの例で示す競合スピンロックでは、マシン上のスレッド間で引き起こされる競合によるパフォーマンスの低下を避けるため、ロックがビジーである場合、指数バックオフを推奨します。これは、マシン上のスレッド数を増やし、競合状態を悪化させる可能性があるアーキテクチャーを変更する場合さらに重要となります。共有メモリーを持つ複数ソケットのインテル® サーバー・プロセッサーでは、同じロックを使用するスレッド数が増えるに従って、スレッド間の競合を解決する時間はさらに長くなります。指数バックオフはパフォーマンス低下の可能性を避けるため、スレッド間の競合を回避するように設計されています。以下の例では、チューニングの対象である MAX_BACKOFF に到達するまで、PAUSE 命令の数は 2 倍に増加することに注意してください。

例 2-9 バックオフを増やした競合ロックの例

```
/********************/
/* ベースライン版      */
/********************/

// atomic {if (lock == free) ならロック状態をビジーに変更 }
while (cmpxchg(lock, free, busy) == fail)
{
   while (lock == busy)
   {
      __asm__ ("pause");
   }
}
```

```
/*******************/
/* 改善版          */
/*******************/

int mask = 1;
int const max = 64; //MAX_BACKOFF
while (cmpxchg(lock, free, busy) == fail)
{
   while (lock == busy)
   {
       for (int i=mask; i; --i){
          __asm__ ("pause");
       }
       mask = mask < max ? mask<<1 : max;
   }
}
```

2.6 インテル® ハイパースレッディング・テクノロジー

インテル® ハイパースレッディング・テクノロジー (HT テクノロジー) は、物理プロセッサー・パッケージ内や物理プロセッサー・パッケージの各プロセッサー・コア内で複数の論値プロセッサーを提供することで、ソフトウェアがタスクやスレッドレベルの並列処理の利点を得られることを可能にします。インテル® Xeon® プロセッサーにおけるインテル® ハイパースレッディング・テクノロジーの最初の実装では、単一の物理プロセッサー (プロセッサー・コア) で 2 つの論理プロセッサーを提供しました。Knights Landing† マイクロアーキテクチャー・ベースのインテル® Xeon Phi™ プロセッサーは、各プロセッサー・コアで 4 つの論理プロセッサーをサポートします。Knights Landing† マイクロアーキテクチャーにおけるインテル® ハイパースレッディング・テクノロジーの実装の詳細については第 20 章を参照してください。

ほとんどのインテル® アーキテクチャー・ベースのプロセッサー・ファミリーは、各プロセッサー・コアまたは初期の実装では物理プロセッサーで 2 つの論理プロセッサーのインテル® ハイパースレッディング・テクノロジーをサポートします。残りの節では、初期のインテル® ハイパースレッディング・テクノロジー実装の機能について説明します。大部分の説明は、2 論理プロセッサーをサポートする以降のインテル® ハイパースレッディング・テクノロジーの実装にも適用できます。この節のマイクロアーキテクチャーに関する説明では、個々のマイクロアーキテクチャーへ追加された詳細とインテル® ハイパースレッディング・テクノロジーの拡張を提供します。

2 つの論理プロセッサーは、それぞれが一連のアーキテクチャー・レジスターをすべて所有し、1 つの物理プロセッサーのリソースを共有します。HT テクノロジー対応のプロセッサーは、2 つのプロセッサー・アーキテクチャー・ステートを保持しているため、オペレーティング・システムやアプリケーションなどのソフトウェアからは、2 つのプロセッサーのように見えます。

ピーク時の要求を処理するために必要なリソースを 2 つの論理プロセッサー間で共有しているため、HT テクノロジーはマルチプロセッサー・システム適しており、従来の MP システムに比べてより高いスループット性能を発揮します。

図 2-9 は、HT テクノロジー対応のプロセッサーを基本とした一般的なバス接続による SMP (symmetric multiprocessor) を示します。それぞれの論理プロセッサーでソフトウェア・スレッドの実行が可能であり、1 つの物理プロセッサー上で最大 2 つのソフトウェア・スレッドを実行できます。2 つのソフトウェア・スレッドは同時に実行されますが、論理プロセッサー 0 からの "add" 演算、および論理プロセッサー 1 からの別の "add" 演算とロードを、同じクロックサイクルで実行エンジンによって同時に実行できます。

HT テクノロジーの最初の実装では、物理実行リソースが共有され論理プロセッサーごとにアーキテクチャー・ステートの複製を持ちます。これにより、HT テクノロジーの実装コストが最小限に抑えられ、マルチスレッド・アプリケーションまたはマルチタスク・ワークロードのパフォーマンスが向上します。

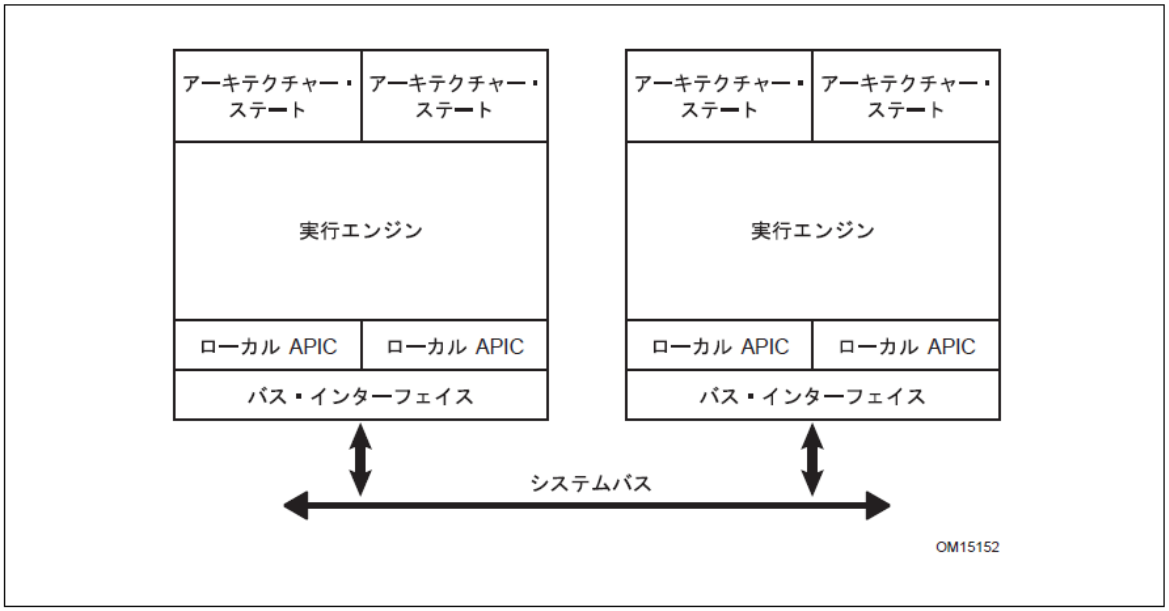


図 2-9 SMP 上のインテル® ハイパースレッディング・テクノロジー

インテル® ハイパースレッディング・テクノロジーを実装すると、次の理由によりパフォーマンスが向上する可能性があります。

- オペレーティング・システムおよびユーザープログラムでプロセスまたはスレッドをスケジュールして、各物理プロセッサー内の論理プロセッサー上で同時に実行できます
- シングルスレッドのみが実行リソースを消費しているときよりも、高いレベルでプロセッサーの実行リソースを活用できます。リソースをより高いレベルで活用すると、システム・スループットが向上します

2.6.1 プロセッサー・リソースと HT テクノロジー

物理プロセッサーのマイクロアーキテクチャー・リソースの大部分が、論理プロセッサー間で共有されます。論理プロセッサーごとに複製されるデータ構造はごくわずかです。この節では、リソースの共有、分割、および複製について説明します。

2.6.1.1 リソースの複製

アーキテクチャー・ステートは、論理プロセッサーごとに複製されます。アーキテクチャー・ステートは、プログラムの動作を制御したり、計算データを保存するレジスターから構成され、これらのレジスターは、オペレーティング・システムおよびアプリケーション・コードによって使用されます。アーキテクチャー・ステートには、8 つの汎用レジスター、制御レジスター、マシン・ステート・レジスター、デバッグレジスターなどが含まれます。メモリー・タイプ・レンジ・レジスター (MTRR) やパフォーマンス監視リソースなどいくつかの例外があります。アーキテクチャー・ステートと例外の詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A、3B、3C、3D』を参照してください。

命令ポインターやレジスター・リネーミング・テーブルなどのリソースが、2 つの論理プロセッサーの実行およびステートの変化を同時に追跡するために複製されます。また、リターン命令の分岐予測を改善するため、リターンスタック分岐予測機構も複製されます。

そのほか、煩雑な操作を低減するため、いくつかのバッファー (2 エントリー命令ストリーミング・バッファーなど) が複製されます。

2.6.1.2 リソースの分割

いくつかのバッファーは、各論理プロセッサーが使用するエントリーを半分に制限することで共有されます。これらは分割リソースと呼ばれます。バッファーを分割する理由は、以下のとおりです。

- 公平な操作を行うため
- 一方の論理プロセッサーからの操作が、ストールした可能性のあるもう一方の論理プロセッサーの操作により妨げられないようにする

例えば、キャッシュミス、分岐の予測ミス、または命令に依存性があると、数サイクルの間、論理プロセッサーは処理を進行できなくなる可能性があります。リソースを分割すると、ストールした論理プロセッサーは、処理の進行をブロックしなくなります。

一般に、主要なパイプステージ間で命令をステージ化するためバッファーは分割されます。こうしたバッファーの例として、実行トレースキャッシュ後のマイクロオペレーション (uop) キュー、レジスター・リネーム・ステージ後のキュー、リタイアメント用の命令をステージ化するリオーダーバッファー、およびロードバッファーやストアバッファーなどがあります。

ロードバッファーとストアバッファーは、各論理プロセッサーのメモリーアクセス順序を維持し、メモリーアクセス順序の違反を検出するために、すでに分割が実装されています。

2.6.1.3 リソースの共有

物理プロセッサー内のリソースの大部分は、リソース (キャッシュやすべての実行ユニットなど) の動的な利用を改善するため完全に共有されています。DTLB など、リニアに処理される共有リソースの中には、エントリーがどちらの論理プロセッサーに属しているのかを識別するため論理プロセッサー ID ビットを持つものがあります。

2.6.2 マイクロアーキテクチャー・パイプラインと HT テクノロジー

この節では、HT テクノロジーのマイクロアーキテクチャーについて説明します。また、2 つの論理プロセッサーからの命令が、パイプラインのフロントエンドとバックエンドでどのように処理されるのか説明します。

2 つのプログラムまたは 2 つのスレッドで実行される命令は同時に処理されますが (実行コアやメモリー階層内のプログラム順序に必ずしも従うわけではありません)、フロントエンドおよびバックエンドには、2 つの論理プロセッサーからの命令を選択するいくつかのポイントがあります。すべてのポイントでは、一方の論理プロセッサーがパイプライン・ステージを利用できない場合に、2 つの論理プロセッサー間で切り替わります。この場合、もう一方の論理プロセッサーは、パイプライン・ステージの全サイクルをすべて利用できます。論理プロセッサーがパイプライン・ステージを利用しない場合もありますが、その場合、キャッシュミス、分岐の予測ミス、命令依存性などによりストールしていることが考えられます。

2.6.3 実行コア

マイクロオペレーション (uop) の実行準備が整っている場合、実行コアは 1 サイクルあたり最大 6 uop をディスパッチできます。実行待ちのキューにマイクロオペレーション (uop) が配置されると、2 つの論理プロセッサーからの命令は区別されなくなります。実行コアとメモリー階層も、どの命令がどの論理プロセッサーのものであるかを記憶していません。

実行が完了すると、命令はリオーダーバッファーに格納されます。リオーダーバッファーは、リタイアメント・ステージから実行ステージを分離します。リオーダーバッファーは、各プロセッサーがエントリーの半分ずつを使用できるように分割されます。

2.6.4 リタイア

リタイアメント・ロジックは、2 つの論理プロセッサーからの命令がリタイアする準備が完了するのを追跡します。リタイアの準備が完了すると、2 つの論理プロセッサー間で切り替えを行うことにより、命令を論理プロセッサーごとにプログラムの順序でリタイアします。一方の論理プロセッサーで命令をリタイアする準備ができていない場合、もう一方の論理プロセッサーがすべてのリタイアメント帯域幅を使用できます。

ストアがリタイアした場合、プロセッサーは、そのストアデータを L1 データキャッシュに書き込む必要があります。選択ロジックは、ストアデータをキャッシュにコミットするため、2 つの論理プロセッサー間で切り替えを行います。

2.7 SIMD 技術

SIMD 計算 (図 2-17 を参照) は、インテル® MMX® テクノロジーでアーキテクチャーに導入されました。インテル® MMX® テクノロジーは、バイト、ワード、ダブルワードのパックド整数データ型の SIMD 操作を可能にします。この整数は、インテル® MMX® テクノロジー・レジスターと呼ばれる 8 つの 64 ビット・レジスターに格納されます (図 2-18 を参照)。

この SIMD 計算モデルは、インテル® Pentium® III プロセッサーでインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) が導入されたことによって拡張されました。インテル® SSE では、4 つのパックド単精度浮動小数点値データ要素を含むオペランドに対し、SIMD 計算を実行できます。オペランドは、メモリーまたは 8 つの 128 ビット XMM レジスター内に置かれます (図 2-11 を参照)。インテル® SSE では、SIMD 計算機能を拡張するために、64 ビットのインテル® MMX® 命令も追加されました。

図 2-10 に一般的な SIMD 計算を示します。4 つのパックドデータ要素 2 つ (X1、X2、X3、X4 と Y1、Y2、Y3、Y4) が並列処理され、対応するデータ要素ペア (X1 と Y1、X2 と Y3、X3 と Y3、X4 と Y4) ごとに同じ操作が行われます。4 つの並列処理の結果は、4 つのパックドデータ要素として格納されます。

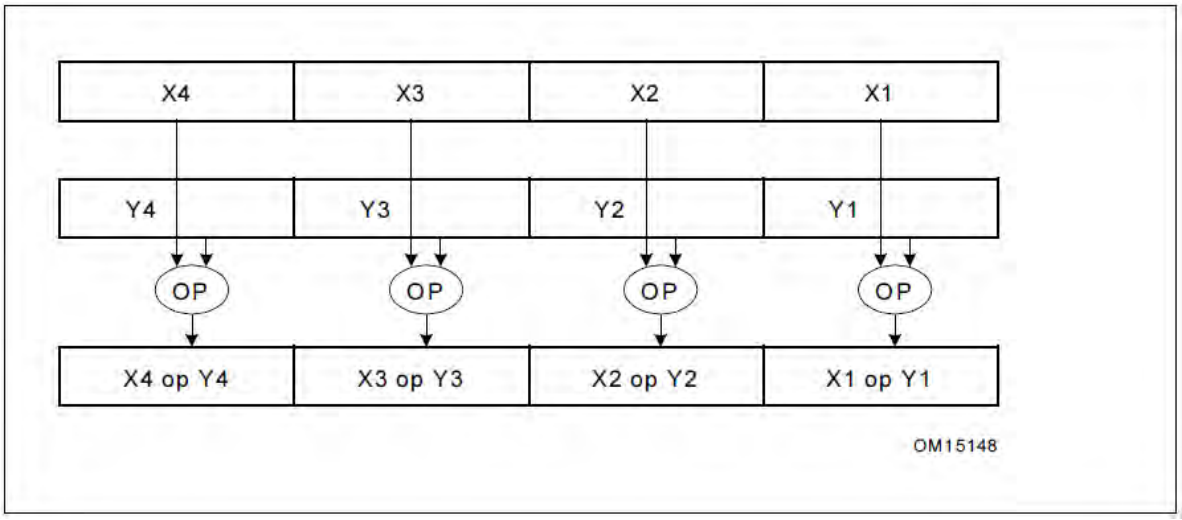


図 2-10 典型的な SIMD 操作

インテル® Pentium® 4 プロセッサーでは、さらにインテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3) の導入によって、SIMD 計算モデルが拡張されました。さらに、インテル® Xeon® プロセッサー 5100 番台では、インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3) が導入されました。

インテル® SSE2 は、メモリーまたは XMM レジスター内のオペランドを処理します。この技術により SIMD 計算が拡張され、パックド倍精度浮動小数点データ要素と 128 ビット・パックド整数を処理できるようになりました。インテル® SSE2 には 144 の命令が追加され、これらの命令により、2 つのパックド倍精度浮動小数点データ要素、

または 16 のパックドバイト整数、8 つのパックドワード整数、4 つのダブルワード整数、2 つのクワッドワード整数を操作できます。

インテル® SSE3 は、特定分野のアプリケーション・パフォーマンスを高める 13 の命令を提供することによって、x87、インテル® SSE、インテル® SSE2 を強化します。これらの分野には、ビデオ処理、複素数演算、スレッド同期などが含まれます。インテル® SSE3 は、SIMD データの非対称処理、水平計算の簡易化、キャッシュライン分割のロード防止を実行する命令によって、インテル® SSE とインテル® SSE2 を補完します。図 2-11 を参照してください。

インテル® SSSE3 では、デジタルビデオと信号処理に関連する 32 の命令によって SIMD 計算がさらに強化されています。

インテル® SSE4.1、インテル® SSE4.2、インテル® AES-NI は、メディア処理、テキスト/字句処理、ブロック暗号化/復号のアプリケーションの高速化を図る追加の SIMD 拡張命令です。

SIMD 拡張は、次の点を除いて、IA-32 アーキテクチャーとインテル® 64 アーキテクチャーで同じように動作します。

- 64 ビット・モードでは、XMM レジスターを参照する 128 ビット SIMD 命令が 16 個の XMM レジスターにアクセスできます。
- 64 ビット・モードでは、32 ビット汎用レジスターを参照する命令が 16 個の汎用レジスターにアクセスできます。

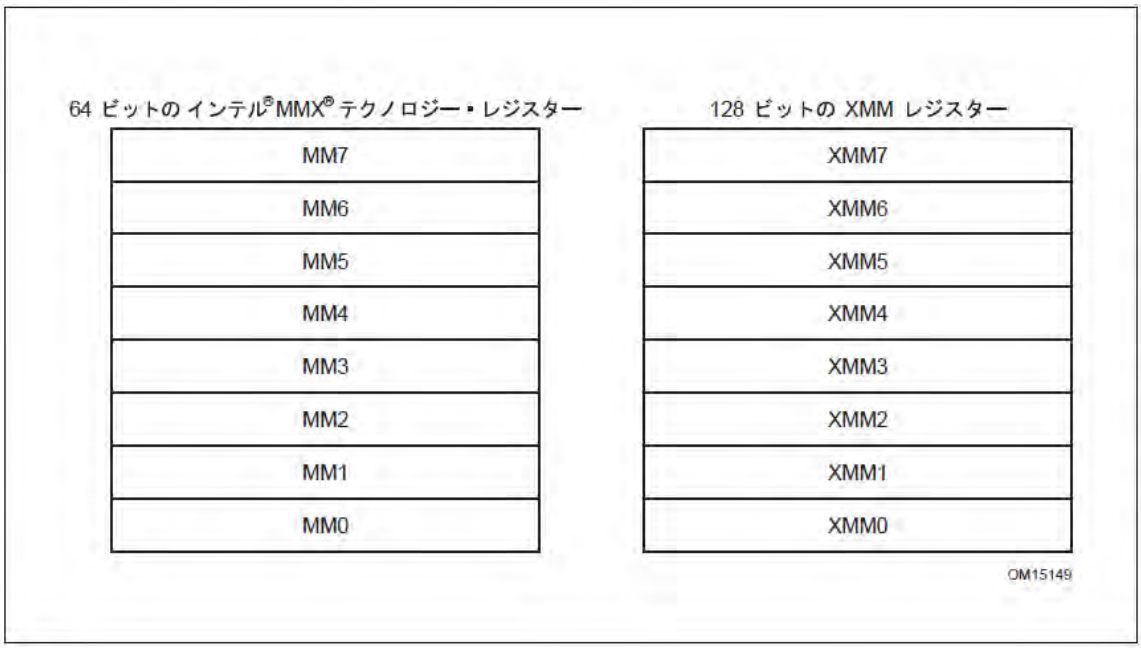


図 2-11 SIMD 命令のレジスターの利用

SIMD を活用して、3D グラフィックス、音声認識、画像処理、科学計算、および以下のような特性を持つアプリケーションのパフォーマンスを向上できます。

- 本質的にパラレルである処理
- 反復的なメモリー・アクセス・パターン
- データに対して局所的な反復操作が行われる
- 制御フローがデータに依存しない

2.8 SIMD 技術とアプリケーション・レベル拡張のまとめ

SIMD 浮動小数点命令は、IEEE 規格 754 のバイナリー浮動小数点算術演算を完全にサポートしています。SIMD 命令は、すべての IA-32 実行モード (プロテクトモード、実アドレスモード、仮想 8086 モード) で利用できます。

インテル® SSE、インテル® SSE2、インテル® MMX® テクノロジーはアーキテクチャーの拡張技術です。既存のソフトウェアは、修正することなく、これらの技術をサポートするインテル® マイクロプロセッサー上で正しく動作します。また、既存のソフトウェアは、SIMD 技術を組込んだアプリケーションと併用しても正常に動作します。

インテル® SSE 命令とインテル® SSE2 命令には、キャッシュ制御命令とメモリーアクセス順序命令も導入されており、これらの命令によってキャッシュ利用とアプリケーションのパフォーマンスを改善できます。

インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® MMX® テクノロジーに関する詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の以下の章を参照してください。

- 第 9 章「Programming with Intel® MMX™ Technology」
- 第 10 章「Programming with Streaming SIMD Extensions (SSE)」
- 第 11 章「Programming with Streaming SIMD Extensions 2 (SSE2)」
- 第 12 章「Programming with SSE3, SSSE3 and SSE4」
- 第 14 章「Programming with AVX, FMA and AVX2」
- 第 15 章「Programming with Intel® AVX-512」
- 第 16 章「Programming with Intel® Transactional Synchronization Extensions」

2.8.1 インテル® MMX® テクノロジー

インテル® MMX® テクノロジーでは以下が導入されました。

- 64 ビット MMX レジスター
- パックドバイト、パックドワード、パックド・ダブルワード整数の SIMD 演算のサポート

推奨事項: インテル® MMX® 命令を使用して記述された整数 SIMD コードは、インテル® SSE/インテル® AVX 命令を使用した効率良い実装を検討する必要があります。

2.8.2 インテル® ストリーミング SIMD 拡張命令

インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) では以下が導入されました。

- 128 ビット XMM レジスター
- 4 つのパックド単精度浮動小数点オペランドで構成される 128 ビット・データ型
- データ・プリフェッチ命令
- 非テンポラルなストア命令などのキャッシュ制御命令とメモリーアクセス順序命令
- 追加の 64 ビット SIMD 整数のサポート

インテル® SSE 命令は、3D ジオメトリー、3D レンダリング、音声認識、ビデオ・エンコーディング/デコーディングに効果的です。

2.8.3 インテル® ストリーミング SIMD 拡張命令 2

インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2) では以下が追加されました。

- 2 つのパックド倍精度浮動小数点オペランドで構成される 128 ビット・データ型

- 16 のバイト整数、8 つのワード整数、4 つのダブルワード整数、または 2 つのクワッドワード整数の SIMD 整数演算用の 128 ビット・データ型
- 64 ビット整数オペランドの SIMD 算術演算のサポート
- 新しいデータ型と既存のデータ型間での変換の命令
- データシャッフルのサポートを拡張
- キャッシュ制御操作とメモリーアクセス順序操作のサポートを拡張

インテル® SSE2 命令は、3D グラフィックス、ビデオ・デコーディング/エンコーディング、暗号化処理に効果的です。

2.8.4 インテル® ストリーミング SIMD 拡張命令 3

インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3) では以下が追加されました。

- 非対称計算および水平計算向けの SIMD 浮動小数点命令
- キャッシュライン分割を防止する専用 128 ビット・ロード命令
- 浮動小数点制御ワード (FCW) に依存しない整数への変換を行う x87 FPU 命令
- スレッド同期をサポートする命令

インテル® SSE3 命令は、科学、ビデオ、マルチスレッドのアプリケーションに効果的です。

2.8.5 インテル® ストリーミング SIMD 拡張命令 3 補足命令

インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3) では、8 種類のパックド整数演算を高速化する 32 の新しい命令が導入されました。これには以下のものがあります。

- 水平加算または減算を実行する 12 の命令
- 絶対値を評価する 6 つの命令
- 乗算と加算を実行して、ドット積の評価を高速化する 2 つの命令
- パックド整数の乗算を高速化して、スケーリングで整数値を生成する 2 つの命令
- 2 番目のシャッフル制御オペランドに応じてバイトごとのインプレース・シャッフルを実行する 2 つの命令
- ソースオペランド中の対応する要素の符号がマイナスの場合にデスティネーション・オペランド中のパックド整数を否定する 6 つの命令
- 2 つのオペランドの合成から得たデータをアライメントする 2 つの命令

2.8.6 インテル® ストリーミング SIMD 拡張命令 4.1

インテル® ストリーミング SIMD 拡張命令 4.1 (インテル® SSE4.1) では、ビデオ、画像処理、3D の各アプリケーションを高速化する 47 個の新しい命令が導入されました。また、インテル® SSE4.1 ではコンパイラーのベクトル化機能が改善され、パックド・ダブルワード演算のサポートが大幅に向上しました。これには以下のものがあります。

- パックド・ダブルワード乗算を実行する 2 つの命令
- 入出力を選択して浮動小数点ドット積を計算する 2 つの命令
- ストリーミング・ヒントを WC ロードに提供する 1 つの命令
- パックドブレンドを簡素化する 6 つの命令
- パックド整数の MIN/MAX のサポートを拡張する 8 つの命令
- 選択可能な丸めモードと精度例外のオーバーライドによって浮動小数点の丸めをサポートする 4 つの命令
- XMM レジスターからのデータ挿入/抽出を改善する 7 つの命令
- パックド整数の形式変換 (符号拡張とゼロ拡張) を改善する 12 の命令
- 小さなブロックサイズにおける絶対差の和 (SAD) の生成を改善する 1 つの命令
- ワード整数の水平検索操作を支援する 1 つの命令
- マスクされた比較を改善する 1 つの命令
- パックド・クワッドワードが等しいかどうかの比較を追加する 1 つの命令

- 符号なし飽和演算によるダブルワードのパックを追加する 1 つの命令

2.8.7 インテル® ストリーミング SIMD 拡張命令 4.2

インテル® ストリーミング SIMD 拡張命令 4.2 (インテル® SSE4.2) では、7 つの新しい命令が導入されました。これには以下のものがあります。

- 64 ビットの整数データ要素を比較する 1 個の 128 ビット SIMD 整数命令
- 豊富なプリミティブを提供する 4 つの文字列/テキスト処理命令。これらのプリミティブを使用すると、以下のものを高速化できます
  - strlen から strcmp、strcspn に至るまでの基本的および高度な文字列ライブラリー関数
  - テキストストリームを字句解析するための区切り文字の処理やトークンの抽出
  - XML 処理を含むパーサーやスキーマ検証

- 巡回冗長チェックサム・シグネチャーの計算を高速化する 1 個の汎用命令
- 整数値のビット・カウントを計算する 1 個の汎用命令

2.8.8 インテル® AES New Instructions と PCLMULQDQ

インテル® AES New Instructions (インテル® AES-NI) では、7 個の新しい命令が導入されました。そのうちの 6 つは AES 暗号化/復号規格 (AESN) に基づくアルゴリズムを高速化するためのプリミティブです。

PCLMULQDQ 命令は、汎用ブロックの暗号化を高速化し、最大 64 ビットの 2 進数のキャリーなし乗算を実行できます。

一般的に、AES 規格に基づくアルゴリズムでは、いくつかのプリミティブを介して、複数の反復でブロックデータを変換します。AES 規格では、128、192、および 256 ビットの暗号鍵をサポートしています。暗号鍵のサイズは、それぞれ 10、12、および 14 ラウンドの反復に対応しています。

AES 暗号化には、128 ビットの入力データ (プレーンテキスト) を 128 ビットの暗号化されたブロック (暗号文) にする、有限数の反復処理が含まれています。この処理を AES ラウンドと呼びます。暗号解読は、"逆暗号 (inverse cipher)" の代わりに "等価逆暗号 (Equivalent Inverse Cipher)" を使用して、この反復処理を逆方向に行います。

各ラウンドの暗号処理には、"状態" と "ラウンドキー" という 2 つの入力データがあります。"ラウンドキー" はラウンドごとに異なります。ラウンドキーは、"キー・スケジュール" アルゴリズムを使用して暗号鍵から生成されます。"キー・スケジュール" アルゴリズムは、暗号/復号のデータ処理から独立しており、暗号/復号フェーズとは別に実行することができます。

AES 拡張命令には、暗号化の AES ラウンドを高速化するため 2 つのプリミティブ、"等価逆暗号 (equivalent inverse cipher)" を使用して復号を行う AES ラウンド用の 2 つのプリミティブ、AES のキー・スケジュール生成プロシージャーをサポートするための 2 つの命令が含まれます。

2.8.9 インテル® アドバンスト・ベクトル・エクステンション

インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) は、これまでのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) よりもアーキテクチャー面で広範な拡張機能を提供します。インテル® AVX で導入されているアーキテクチャー上の拡張機能は以下のとおりです。

- 256 ビットのベクトルと SIMD レジスターセットのサポート
- 128 ビットのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) が 256 ビットの浮動小数点の命令セットに拡張され、最大 2 倍のパフォーマンスを実現

- 一般的な 3 オペランド構文の命令構文をサポートすることで、命令のプログラミングの柔軟性を向上させ、新しい拡張命令セットを効率良くエンコード
- 既存の 128 ビットのインテル® SIMD 拡張命令の拡張により、3 オペランド構文をサポートし、コンパイラーによる高級言語レベルでのベクトル化を簡略化します
- 256 ビットのインテル® AVX コード、128 ビットのインテル® AVX コード、128 ビットのレガシーコードおよびスカラーコード間の柔軟な導入をサポート

インテル® AVX 命令セットおよび 256 ビット・レジスターの状態管理の詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A、2B、2C、および 3D』を参照してください。インテル® AVX の最適化手法については、本書の第 15 章「インテル® AVX、FMA およびインテル® AVX2 向けの最適化」で説明します。

2.8.10 半精度浮動小数点変換 (F16C)

VCVTPH2PS と VCVTPS2PH は、単精度浮動小数点データ型と半精度浮動小数点データ型間のデータ変換をサポートする命令です。これら 2 つの命令は、インテル® AVX プログラミング・モデルとともに拡張されました。

2.8.11 RDRAND

RDRAND 命令は、暗号化による安全で、決定論的な乱数ビット生成器 (DBRG) によって供給される乱数を取得します。DBRG は NIST SP800-90A 規格を満たすように設計されています。

2.8.12 乗算-加算の融合 (FMA) 拡張

FMA 拡張は、融合乗算加算、融合乗算減算、融合乗算加算/減算インターリーブ、および融合乗算加算と融合乗算減算操作における符号付き反転乗算をカバーする高スループット、算術演算機能を備えることでインテル® AVX を強化します。FMA 拡張では、256 ビット・ベクトルと追加の 128 ビットを計算する 36 の 256 ビット浮動小数点命令とスカラー FMA 命令が提供されます。

2.8.13 インテル® アドバンスト・ベクトル・エクステンション 2

インテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2) は、インテル® AVX の 128 ビット SIMD 整数命令のほとんどを 256 ビットの数値処理に対応するように拡張します。インテル® AVX2 命令のプログラミング・モデルは、インテル® AVX 命令と同様です。

さらに、インテル® AVX2 では、データ要素のブロードキャスト/置換操作、データ要素ごとにシフトカウントの異なるベクトルシフト命令、連続していないデータをメモリーからフェッチする命令拡張機能が提供されます。

2.8.14 汎用ビット処理命令

汎用レジスター上でビット処理を行う命令群は、第 4 世代インテル® Core™ プロセッサー・ファミリーで導入されました。これらの命令のほとんどは、非破壊ソースオペランド構文を提供するため VEX プリフィクス・エンコード体系を使用します。

これらの命令は、CPUID によって示される 3 つの異なる機能フラグで列挙されます。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の 5.1 節と、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A、2B、2C、および 2D』の第 3、4 および 5 章を参照してください。

2.8.15 インテル® トランザクショナル・シンクロナイゼーション・エクステンション

インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) は、第 4 世代インテル® Core™ プロセッサー・ファミリーで導入され、ロックベースのプログラミング・モデルを持つマルチスレッド・アプリケーションのロックで保護されたクリティカル・セクションのパフォーマンスを向上させます。

インテル® TSX の背景と詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 16 章「Programming with Intel® Transactional Synchronization Extensions」を参照してください。

マルチスレッド・アプリケーションにおけるロックベースのクリティカル・セクションでインテル® TSX を使用するソフトウェア・チューニングの推奨事項は、同マニュアルの第 12 章「Intel® TSX Recommendations」を参照してください。

2.8.16 RDSEED

RDSEED 命令は、暗号化による安全で、決定論的な乱数ビット生成器拡張 (NRBG) によって供給される乱数を取得します。NRBG は、NIST SP 800-90B と NIST SP 800-90C 規格を満たすように設計されています。

2.8.17 ADCX と ADOX 命令

ADCX と ADOX 命令は、MULX 命令を連携して大きな整数値の計算をスピードアップします。詳細は、http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integer-squaring-ia-paper.pdf (英語) のドキュメントを参照してください。

第 3 章
一般的な最適化ガイドライン

本章では、インテルのプロセッサー上で実行するアプリケーションのパフォーマンスを向上する一般的な最適化手法について説明します。これらの手法は、第 2 章「インテル® 64 および IA-32 プロセッサー・アーキテクチャー」で説明したマイクロアーキテクチャーの利点を活用します。インテルのマルチコア・プロセッサー、インテル® ハイパースレッディング・テクノロジー、64 ビット・モード・アプリケーション向けの最適化ガイドラインについては、第 11 章「マルチコアとインテル® ハイパースレッディング・テクノロジー」と第 13 章「64 ビット・モードのコーディング・ガイドライン」で説明します。

パフォーマンスを最適化する上での基準は、以下の 3 分野に注目します。

- コードを生成するツールと手法
- ワークロードのパフォーマンス特性と、マイクロアーキテクチャー・サブシステムとの相互作用に関する分析
- ターゲット・マイクロアーキテクチャー (またはマイクロアーキテクチャー・ファミリー) 向けにチューニングされたコードによるパフォーマンスの向上

まず、最初の 2 つの作業を簡素化するため、ツールを利用する上でのヒントを紹介します。次に、ターゲット・マイクロアーキテクチャー向けにコードを生成し、コードをチューニングする際の推奨事項に注目します。

本章では、インテル® C++ コンパイラー、インテル® Fortran コンパイラー、およびその他のコンパイラーに関する最適化手法について説明します。

3.1 パフォーマンス・ツール

インテルでは、コンパイラー、パフォーマンス・アナライザー、マルチスレッド化ツールなど、アプリケーションのパフォーマンス最適化を支援するツールを用意しています。

3.1.1 インテル® C++ および Fortran コンパイラー

インテル® コンパイラーは、複数の OS (Windows*、Linux*、macOS*、組込み機器向け) をサポートしています。インテル® コンパイラーは、パフォーマンスを最適化し、次の高度な機能をアプリケーション開発者に提供します。

- 32 ビットまたは 64 ビットのインテル® プロセッサーを最適化の対象とする柔軟性
- さまざまな統合開発環境 (IDE) やサードパーティー製コンパイラーとの互換性
- ターゲット・プロセッサーのアーキテクチャーの利点を引き出す自動最適化機能
- プロセッサーごとに異なるコードを作成する必要性を減らすコンパイラーによる自動最適化
- Windows*、Linux*、macOS* でサポートされている以下の汎用的なコンパイラー機能
  - 一般的な最適化設定
  - キャッシュ管理機能
  - プロシージャー間の最適化 (IPO) 手法
  - プロファイルに基づく最適化 (PGO) 手法
  - マルチスレッド化のサポート:
  - 浮動小数点演算の精度と一貫性のサポート
  - コンパイラーの最適化とベクトル化のレポート

3.1.2 コンパイラーに関する一般的な推奨事項

一般的に、ターゲット・マイクロアーキテクチャー向けのチューニング能力を備えたコンパイラーを使用することで、手作業による最適化に相当するか、それを上回るパフォーマンスが得られるはずです。インテル® C++ コンパイラー

および Fortran コンパイラーなどの一部のコンパイラーでは、コンパイルされたコードにパフォーマンス上の問題が見つかった場合、コード開発者が組込み関数またはインライン・アセンブリーを挿入して、生成されるコードを制御することを可能にします。インライン・アセンブリーを使用する場合、開発者は生成されたコードの品質とパフォーマンスを検証する必要があります。

デフォルトのコンパイラー・オプションは一般的なアプリケーションを想定しており、ほとんどのプログラムに効果的な最適化手法が、コンパイラーのデフォルトに設定されています。パフォーマンス上の問題の原因が、コンパイラー・オプション選択の誤りである場合、異なるオプションを使用するか、または異なるコンパイラーでターゲットモジュールをコンパイルすることで問題が解決される可能性があります。プロセッサー固有のオプションを含むコンパイラーの最適化オプションに関する詳しい提案については、『インテル® C++ および Fortran コンパイラー最適化クイック・リファレンス・ガイド』を参照してください。

3.1.3 インテル® VTune™ プロファイラー

インテル® VTune™ プロファイラーは、パフォーマンス監視ハードウェアを使用して、アプリケーションや、アプリケーションとマイクロアーキテクチャーの相互作用について、統計情報とコーディング情報を収集します。これにより、ソフトウェア開発者はそれぞれのマイクロアーキテクチャーのワークロードのパフォーマンス特性を測定できます。インテル® VTune™ プロファイラーは、現在および過去のインテル® プロセッサー・ファミリーをサポートしています。

インテル® VTune™ プロファイラーは、次の 2 種類の情報をフィードバックします。

- 特定のコーディング上の推奨事項またはマイクロアーキテクチャー上の機能の使用によるパフォーマンスの向上のメトリック
- プログラムの変更が、特定のメトリックに対してパフォーマンスを向上させたか低下させたかを示す情報

インテル® VTune™ プロファイラーでは、以下のようなワークロード特性も測定できます。

- ワークロードにおける命令レベルの並列処理の度合いを示す、命令実行のリタイアメント・スループット
- キャッシュおよびメモリー階層のストレスポイントを示す、データ・トラフィックの局所性
- データ・アクセス・レイテンシーの軽減効果の度合いを示す、データ・トラフィックの並列性

注意

マシンの一部でパフォーマンスが向上しても、必ずしも全体のパフォーマンスが大きく向上するとは限りません。特定のメトリックについてパフォーマンスを改善すると、全体のパフォーマンスが低下する可能性もあります。

本章のコーディングの推奨事項には、必要に応じて、インテル® VTune™ プロファイラーのイベントの説明が記載されており、これらのイベントによって、その推奨事項に従えばどの程度パフォーマンスが向上するかを示す測定可能なデータを提供します。インテル® VTune™ プロファイラーの使用方法の詳細については、製品のオンラインヘルプを参照してください。

3.2 プロセッサーの相違

現在のマイクロアーキテクチャーまで、多くのマイクロアーキテクチャーにうまく適用できる多くのコーディングの推奨事項があります。ただし、推奨事項によっては、特定のマイクロアーキテクチャーでより有効性が高い場合があります。このような推奨事項の一部を以下に示します。

3.2.1 CPUID ディスパッチ手法と互換コード手法

すべての世代のプロセッサー上で最適なパフォーマンスを得るには、アプリケーションが CPUID 命令を使用してプロセッサーの世代を識別して、ソースコードにプロセッサー固有の命令を記述する必要があります。インテル® コンパイラーは、各プロセッサー向けの異なるバージョンのコードを統合する機能を備えています。実行されるコードは、

ランタイム時に CPUID に基づいて選択されます。異なる世代のプロセッサー向けのバイナリーコードは、プログラマーが制御して生成することも、コンパイラーに任せることもできます。CPU ディスパッチ (関数のマルチバージョン化) の詳細については、『インテル® C++ コンパイラー 19.0 デベロッパー・ガイドおよびリファレンス』の「cpu_dispatch」と「cpu_specific」の節を参照してください。

複数世代のマイクロアーキテクチャーでの実行を想定したアプリケーションでは、バイナリーコードのサイズを最小限に抑えて、単一のコードパスを生成する互換コードを使用すると良いでしょう。一般に、現在および将来の世代のインテル® 64 プロセッサーや IA-32 プロセッサーをベースにしたプロセッサー上でアプリケーションを実行するときは、インテル® Core™ マイクロアーキテクチャー向けに開発された手法を使用し、Nehalem† マイクロアーキテクチャー向けの手法と組み合わせてアプリケーションを最適化することで、コードの効率とスケーラビリティーが向上します。

3.2.2 透過的なキャッシュ・パラメーター手法

CPUID 命令が機能リーフ 4 (キャッシュ・パラメーター・リーフ) をサポートしている場合、このリーフは、インテル® 64 プロセッサー・ファミリーと IA-32 プロセッサー・ファミリー間で上位互換性のある形式で、キャッシュ・パラメーターをキャッシュ階層のレベルごとに報告します。

特定のキャッシュレベルのパラメーターに依存するコーディングを行う場合、キャッシュ・パラメーターを使用することで、コーディング手法は将来の世代のインテル® 64 プロセッサーや IA-32 プロセッサーとの上位互換性を確保し、キャッシュサイズが異なるプロセッサーとの相互互換性も維持できます。

3.2.3 スレッド化とハードウェアによるマルチスレッド・サポート

インテル® 64 と IA-32 プロセッサー・ファミリーは、デュアルコア・テクノロジーと HT テクノロジーという 2 つの方法でハードウェア・マルチスレッディングをサポートします。

現在および将来の世代のインテル® 64 プロセッサーと IA-32 プロセッサーでは、ハードウェア・マルチスレッディングが持つ潜在的なパフォーマンスを十分に引き出すには、アプリケーション設計時にスレッド化を導入する必要があります。さらに、マルチスレッド・ソフトウェアは、広範なコンピューターに対応できるように、ハードウェア・マルチスレッディングをサポートしていないシングル・プロセッサー上でも問題なく実行できる必要があります。また、単一の論理プロセッサー上では、スレッド化されていないソフトウェアに匹敵するパフォーマンスを発揮する必要があります (比較が可能な場合)。一般に、これを実現するには、スレッド同期のオーバーヘッドが最小限になるようにマルチスレッド・アプリケーションを設計する必要があります。マルチスレッド化の詳細なガイドラインについては、第 10 章「マルチコアとインテル® ハイパースレッディング・テクノロジー」で説明します。

3.3 コーディング規則、推奨事項、チューニングのヒント

この節では、コーディング規則、推奨事項、およびチューニングのヒントを紹介します。これらは以下のような技術者を対象としています。

- ソースコードを修正してパフォーマンスを改善するプログラマー (ユーザー/ソースの規則)
- アセンブラーまたはコンパイラーを作成するプログラマー (アセンブリー/コンパイラーの規則)
- きめ細かなパフォーマンス・チューニングを行うプログラマー (チューニングの推奨事項)

コーディングの推奨事項については、以下の 2 つの基準で重要度が示されます。

- **局所的な影響 (高、中、低)**: 特定のコード・インスタンスにおいて、この推奨事項がパフォーマンスに与える影響の度合を示します。
- **一般性 (高、中、低)**: アプリケーションのすべてのドメインでのこのようなインスタンスの発生頻度を示します。一般性は、「頻度」と考えてもよいでしょう。

これらの推奨事項は大まかなものであり、コーディングの形式やアプリケーションの構成などの要因で異なります。

高、中、低 (H、M、L) の優先度を記載した目的は、その推奨事項に従った場合に予想されるパフォーマンス向上の相対的なレベルを示すことにあります。

アプリケーション内で特定のコード・インスタンスがどの程度の頻度で発生するか予測できないため、優先度のヒントから、アプリケーション・レベルのパフォーマンスがどの程度向上するかを直接判断はできません。アプリケーション・レベルのパフォーマンスの向上が観察される場合は、説明を容易にするため定量的な特性評価を示しています。影響を判定できない場合は、優先度は示されません。

3.4 フロントエンドの最適化

フロントエンドの最適化では、以下の 2 つを対象とします。

- 実行エンジンへマイクロオペレーション (uop) の安定した供給を維持 — 分岐予測ミスが発生すると、マイクロオペレーション (uop) のストリームが中断されます。また、投機による実行のコードパス内でマイクロオペレーション (uop) のストリームが実行リソースを浪費する可能性があります。これら考慮したチューニングのほとんどは、分岐予測ユニットに対する作業が対象となります。一般的な手法については、3.4.1 節「分岐予測の最適化」を参照してください。
- 実行帯域幅とリタイアメント帯域幅をできるだけ多く利用できるようにマイクロオペレーション (uop) のストリームを供給 — インテル® Core™ マイクロアーキテクチャーとインテル® Core™ Duo プロセッサー・ファミリーでは、高いデコード・スループットを維持することが目的です。Sandy Bridge† マイクロアーキテクチャーでは、デコード済み命令キャッシュからホットコードの実行を維持することが目的です。インテル® Core™ マイクロアーキテクチャーのデコード・スループットを最大限に高める手法については、3.4.2 節「フェッチとデコードの最適化」を参照してください。

3.4.1 分岐予測の最適化

分岐予測は、パフォーマンスに極めて大きな影響を与えるます。分岐の流れを理解し、分岐予測の精度を上げることで、コードの処理速度は大幅に向上します。

分岐予測を支援する最適化手法には、以下のものがあります。

- コードとデータは別のページに配置します。これは非常に重要です。詳細については、3.6 節「メモリーアクセスの最適化」を参照してください。
- 可能な限り分岐を排除します。
- 静的分岐予測アルゴリズムに従ってコードを配置します。
- すべてのスピンウェイト・ループで PAUSE 命令を使用します。
- 関数をインライン展開し、コールとリターンを一致させます。
- 必要に応じてループをアンロールし、繰り返し実行されるループの反復回数が 16 回またはそれ以下になるようにします (この修正によって、コードのサイズが必要以上に大きくなることがない場合)。
- ループ中に、同じ分岐ターゲットアドレス持ち、同じ 16 バイト境界のコードブロックに配置される 2 つの条件分岐命令を配置するのを避けます。
- ターゲット IP の下位 6 ビットが同一である場合、同じ 8 バイト境界でアライメントされたコードブロック内に複数の条件分岐を配置しないでください (つまり、複数の分岐条件の最後のバイト・アドレスが同じ 8 バイト境界でアライメントコード内に配置されないようにします)。この制限は、Ice Lake† Client 以降のマイクロアーキテクチャーではなくなりました。

3.4.1.1 分岐の排除

分岐を排除すると、以下の理由によりパフォーマンスが向上します。

- 分岐の予測ミスの可能性が少なくなります。
- 分岐ターゲットバッファー (BTB) エントリーの消費が少なくなります。分岐しない条件分岐は、BTB リソースを消費しません。

分岐を排除するには、以下の 4 つの方法があります。

- 基本ブロックが隣接するようにコードを構成します。
- 3.4.1.6 節「ループアンロール」の説明に従ってループをアンロールします。
- CMOV 命令を使用します。
- SETCC 命令を使用します。

以下の規則は分岐の排除に適用されます。

アセンブリー/コンパイラー・コーディング規則 1 (影響 MH、一般性 M): 基本ブロックが隣接するようにコードを構成し、不要な分岐を排除します。

アセンブリー/コンパイラー・コーディング規則 2 (影響 M、一般性 ML): SETCC 命令と CMOV 命令を使用して、予測不可能な条件分岐をできるだけ排除します。予測可能な分岐については、この操作を行ってはなりません。これらの命令を使用して、すべての予測不可能な条件分岐を排除してはなりません (これらの命令を使用すると、条件分岐の両方のパスを実行する必要性から、実行のオーバーヘッドが発生します)。また、条件分岐を SETCC または CMOV に変更すると、データに依存する制御フローの依存関係がトレードオフされ、アウトオブオーダー・エンジンの機能が制限されます。チューニングの際は、すべてのインテル® 64 プロセッサーと IA-32 プロセッサーが、通常は高い分岐予測精度を持っていることを考慮すべきです。一貫した分岐予測ミスは一般的にまれです。これらの命令は、計算時間の増加分が、予想される分岐の予測ミスのペナルティーより小さい場合にのみ使用するべきです。

以下のように、C コードが、2 つの定数のうち 1 つに依存する条件を持つ場合を考えてみます。

```
X = (A < B) ? CONST1 : CONST2;
```

このコードは、2 つの値 A と B を条件付きで比較します。条件が真の場合 X は CONST1 に設定され、条件が偽の場合は CONST2 に設定されます。上記の C コードに相当するアセンブリー・コード・シーケンスは、2 つの値に相関関係がない場合、予測不可能な分岐を含むことがあります。

例 3-1 に、予測不可能な分岐を含むアセンブリー・コードを示します。予測不可能な分岐は、SETCC 命令を使って排除できます。例 3-2 に、分岐を含まない、最適化されたコードを示します。

例 3-1 予測不可能な分岐を含むアセンブリー・コード

```
    cmp a, b          ; 条件
    jbe L30           ; 条件分岐
    mov ebx const1    ; ebx の値は X
    jmp L31           ; 無条件分岐
L30:
    mov ebx, const2
L31:
```

例 3-2 コードの最適化による分岐の排除

```
xor ebx, ebx        ; ebx をクリア (C 言語では X)
cmp A, B
setge bl            ; ebx = 0 または 1
                    ; あるいは補間条件の時
sub ebx, 1          ; ebx=11…11 または 00…00
and ebx, CONST3     ; CONST3 = CONST1-CONST2
add ebx, CONST2     ; ebx=CONST1 または CONST2
```

例 3-2 の最適化されたコードは、EBX を 0 に設定した後、A と B を比較します。A が B より大きいか、または B に等しい場合は、EBX は 1 に設定されます。次に、EBX から 1 が引かれ、2 つの定数の差との AND 演算が行われます。これにより、EBX は、0 または定数の値の差に設定されます。EBX に CONST2 を加算することによって、正しい値が EBX に書き込まれます。CONST2 が 0 に等しい場合は、最後の命令は削除できます。

分岐を排除するもう 1 つの方法は、CMOV 命令と FCMOV 命令を使用することです。例 3-3 に、CMOV を使用して TEST と分岐命令のシーケンスを変更し、分岐を排除する方法を示します。TEST が「等しい」フラグを設定した場合、EBX の値は EAX に移されます。この分岐はデータに依存する分岐であり、予測不可能な分岐を置き換えます。

例 3-3 CMOV 命令による分岐の排除

```
   test ecx, ecx
   jne 1H
   mov eax, ebx
1H:
; コードの最適化のために、jne と mov を 1 つの cmovcc 命令
; (「等しい」フラグをチェックする) に結合します
   test ecx, ecx     ; フラグをテスト
   cmoveq eax, ebx   ;「等しい」フラグがセットされている場合、ebx を
                     ; eax に移します。1H: タグは必要なくなります
```

この概念の拡張は、インテル® AVX-512 マスク操作、およびデータ依存分岐を排除するために使用される VPCMP などの一部の命令で確認できます。18.4 節を参照してください。

3.4.1.2 静的予測

BTB (3.4.1 節「分岐予測の最適化」を参照) 内に履歴がない分岐は、静的予測アルゴリズムを使って予測されます。

- 前方分岐条件が分岐しないことを予測します。
- 後方分岐条件が分岐することを予測します。
- 間接分岐は分岐しないものと予測します。

次の規則が静的予測に適用されます。

アセンブリー/コンパイラー・コーディング規則 3 (影響 M、一般性 H): 静的分岐予測アルゴリズムに従ってコードを配置します。すなわち、条件分岐に後続するフォールスルー・コードが、前方に分岐する分岐のターゲットになるように調整します。また、条件分岐に後続するフォールスルー・コードが、後方に分岐する分岐のターゲットにならないようにします。

例 3-4 に、静的分岐予測アルゴリズムの例を示します。IF-THEN 条件分岐の本体が予測されます。

例 3-4 静的分岐予測アルゴリズム

```
// 前方に分岐する条件が合わない (フォールスルー)
   IF<condition> {....
   ↓
   }
   IF<condition> {...
   ↓
   }
// 後方に分岐する条件が採用される
   LOOP {...
   ↑ -- }<condition>
// 無条件分岐を採用
   JMP
   ------→
```

例 3-5、例 3-6 に、静的予測アルゴリズムの基本的な規則を示します。例 3-5 では、最初は後方分岐 (JC BEGIN) の履歴は BTB 内に存在しないため、BTB は予測を行いません。しかし、スタティック・プレディクターが分岐は行われると予測するため、予測ミスは発生しません。

例 3-5 分岐する静的予測

```
Begin: mov eax, mem32
       and eax, ebx
       imul eax, edx
       shld eax, 7
       jc Begin
```

例 3-6 の最初の分岐命令 (JC BEGIN) は、前方条件分岐です。この分岐の履歴は BTB 内に存在しませんが、スタティック・プレディクターはフォールスルーとして予測します。BTB 内に分岐履歴が作成される前であっても、静的予測アルゴリズムは、CALL CONVERT 命令が分岐するものと正しく予測します。

例 3-6 分岐しない静的予測

```
       mov eax, mem32
       and eax, ebx
       imul eax, edx
       shld eax, 7
       jc Begin
       mov eax, 0
Begin: call Convert
```

インテル® Core™ マイクロアーキテクチャーでは、静的予測のヒューリスティックは使用されません。ただし、インテル® 64 プロセッサーと IA-32 プロセッサーの一貫性を維持するため、ソフトウェアでは静的予測のヒューリスティックをデフォルトとすべきです。

3.4.1.3 インライン展開、コール、リターン

リターン・アドレス・スタック機構は、特にコールとリターンの静的予測と動的予測の精度を向上させます。リターン・アドレス・スタックは 16 個のエントリーを持ち、これによって、ほとんどのプログラムの呼び出し階層を十分にカバーできます。

17 以上のコールと 17 以上のリターンが入れ子になって連鎖している場合、パフォーマンスが低下する場合があります。

リターンスタック機構を使用するには、コールとリターンのペア数が一致していなければなりません。これを行うと、スタックのエントリー数超過によるパフォーマンス低下の可能性は非常に低くなります。

以下の規則は、インライン展開、コール、リターンに適用されます。

アセンブリー/コンパイラー・コーディング規則 4 (影響 MH、一般性 MH): near コールは near リターンに対応し、far コールは far リターンに対応していなければなりません。リターンアドレスをスタック上にプッシュして呼び出し先ルーチンにジャンプすると、コールとリターンの不一致が起こるため、この操作は推奨されません。

コールとリターンにはコストがかかるため、以下の理由からインライン展開を使用します。

- パラメーターの受け渡しのオーバーヘッドを排除できます。
- コンパイラーが、関数をインライン展開すると最適化の可能性が増えます。
- インライン展開されたルーチンに分岐が含まれる場合、呼び出し元のコンテキストを追加すれば、ルーチン内の分岐予測の精度が向上します。
- 小さい関数内で発生した分岐予測ミスは、関数がインライン展開された場合に発生するよりもペナルティーが大きくなります。

アセンブリー/コンパイラー・コーディング規則 5 (影響 MH、一般性 MH): 関数をインライン展開することでコードのサイズが小さくなる場合や、小さな関数が頻繁に呼び出される場合は、関数を選択した上でインライン展開を実施します。

アセンブリー/コンパイラー・コーディング規則 6 (影響 ML、一般性 ML): 17 以上のコールとリターンが連続してネストされている場合は、インライン展開を行いネストされているコールの数を減らします。

アセンブリー/コンパイラー・コーディング規則 7 (影響 ML、一般性 ML): 小さい関数内の分岐の予測精度が低い場合、関数をインライン展開すると予測の精度を改善できます。分岐予測ミスにより、リターンが誤って予測された場合は、ペナルティーが生じます。

アセンブリー/コンパイラー・コーディング規則 8 (影響 L、一般性 L): 関数内の最後の文が別の関数への呼び出しである場合、コールをジャンプに置き換えることを検討します。これにより、コール/リターンのオーバーヘッド、リターン・スタック・バッファー内のエントリーが消費されなくなります。

アセンブリー/コンパイラー・コーディング規則 9 (影響 M、一般性 L): 16 バイト・チャンク内に 5 つ以上の分岐を配置してはいけません。

アセンブリー/コンパイラー・コーディング規則 10 (影響 M、一般性 L): 16 バイト・チャンク内に 3 つ以上のループの終端分岐を配置してはいけません。

3.4.1.4 コードのアライメント

コードを注意深く構成することで、キャッシュとメモリーの局所性を向上できます。頻繁に実行される基本ブロックのシーケンスは、メモリー上で隣接するようにレイアウトします。その際に、まれにしか実行されないコード (エラー状態を処理するコードなど) を、そのシーケンスから削除することが推奨されます。命令プリフェッチの最適化については、3.7 節「プリフェッチ」を参照してください。

アセンブリー/コンパイラー・コーディング規則 11 (影響 M、一般性 H): コードがデコード済み命令キャッシュから実行される直接分岐は、64 B キャッシュラインにすべての命令バイトが格納され、キャッシュラインの終端近くにあるべきです。分岐ターゲットは 64B キャッシュラインの先頭または始まりにある必要があります。

レガシー・デコード・パイプラインから実行される直接分岐命令の場合、16B にアライメントされたメモリーチャンク内にすべての命令バイトが収まり、16B にアライメントされたチャンクの終端近くにある必要があります。分岐ターゲットは、16B にアライメントされたメモリーチャンクの先頭または始まりにある必要があります。

アセンブリー/コンパイラー・コーディング規則 12 (影響 M、一般性 H): 条件分岐の直後の命令が実行される可能性が低い場合、そのコードをプログラムの別の部分に配置します。また、条件分岐の直後の命令が実行される可能性が非常に低く、コードの局所性が重要である場合は、そのコードを異なるコードページ上に配置します。

3.4.1.5 分岐タイプの選択

間接分岐およびコールに対するデフォルトのターゲット予測は、フォールスルー・パス、つまり分岐しないです。該当する分岐に対してハードウェア予測が可能である場合、フォールスルー予測はオーバーライドされます。分岐予測ハードウェアが予測する間接分岐の分岐ターゲットは、以前に実行された分岐ターゲットです。

コードの局所性の低さや異常な分岐競合が原因で分岐予測できない場合、フォールスルー・パスへのデフォルト予測のみが重要な問題となります。間接コールの場合は、関連するリターンの後に実行が戻る確率が高いため、フォールスルー・パスの予測は通常は問題とはなりません。

間接分岐の直後にデータを配置すると、パフォーマンスの問題が生じる可能性があります。データがゼロのみで構成される場合、そのデータはメモリー・デスティネーションに対する加算の長いストリームのように見えるため、リソース競合が発生し、分岐のリカバリーが遅れる可能性があります。また、間接分岐の直後のデータは、分岐予測ハードウェアからは分岐として認識される場合があります。そのため、分岐によって他のデータページが実行されると、その後自己修正コードに関連する問題を引き起こす恐れがあります。

アセンブリー/コンパイラー・コーディング規則 13 (影響 M、一般性 L): 間接分岐を行う場合、実行頻度が最も高い分岐ターゲットを間接分岐の直後に配置します。あるいは、間接分岐が分岐予測ハードウェアによって予測できない場合、間接分岐の後に UD2 命令を配置します。これにより、プロセッサーによるフォールスルー・パスのデコードが停止します。

コードの構造から生じる間接分岐 (switch 文、計算による goto 文やポインターを介したコールなど) は、任意の数の位置にジャンプします。分岐先のアドレスがほぼ同じコードシーケンスでは、ほとんどが BTB によって正確に予測されます。BTB にストアできる分岐する (フォールスルーしない場合) ターゲットは 1 つだけであるため、複数の分岐するターゲットを持つ間接分岐は、予測精度が低くなることがあります。

ストアされるターゲットの有効数は、条件分岐を追加すると増やすことができます。

条件分岐をターゲットに追加するのは、次の場合に効果的です。

- 分岐の方向が、その分岐までの分岐履歴、つまり最後のターゲットだけではなく、その分岐にどのような経路で到達したかに関連します。
- ソースとターゲットのペアが十分一般的で、追加の分岐予測の使用を正当化できます。この場合、全体の分岐予測ミス数が増えることがありますが、間接分岐の予測ミスは改善されます。しかし、分岐の予測ミス数が非常に多い場合、その利点は少なくなります。

ユーザー/ソース・コーディング規則 1 (影響 M、一般性 L): 間接分岐が一般的な分岐するターゲットを 2 つ以上持っており、それらのターゲットの少なくとも 1 つがその分岐までの分岐履歴に関係している場合は、その間接分岐をツリーに変換し、1 つ以上の間接分岐の前にそれらのターゲットに対する条件分岐を配置します。この「ピーリング」手順は、分岐履歴に関係している間接分岐の一般的なターゲットに適用します。

この規則の目的は、分岐を追加するという犠牲を払ってでも分岐の予測精度を高めて、予測ミスの総数を減らすことです。これを可能にするには、追加する分岐の予測精度を高くしなければなりません。分岐予測精度が重要な理由の 1 つは、先行する分岐履歴との強い関係性にあります。つまり、先行する分岐が分岐する方向は、検討対象の分岐方向に対する指標となります。

例 3-7 は、先行する条件分岐のターゲットと間接分岐のターゲットの関係を示す簡単な例です。

例 3-7 2 つの優先ターゲットを持つ間接分岐

```
function ()
{
    int n = rand();              // 乱整数は 0 から RAND_MAX の間
      if ( ! (n & 0x01) ) {      // 半分は n = 0 になる
         n = 0;                  // 予測された分岐履歴を更新
      }
      // 複数のターゲットをとる間接的分岐は
      // さらに低い予測率になる
switch (n) {
      case 0: handle_0(); break; // 前方分岐が行われた分岐履歴に関係がある
                                 // 一般的なターゲット
      case 1: handle_1(); break; // 一般的でない
      case 3: handle_3(); break; // 一般的でない
      default: handle_other();    //一般的なターゲット
      }
}
```

コンパイラーやアセンブリー言語プログラマーにとって、この関係を分析して判別するのは困難なことがあります。コードから最高のパフォーマンスを得るため、ピーリングを使用した場合と使用しない場合とでパフォーマンスを評価することは有益です。

関係する分岐履歴を持つ間接分岐において、最も優先されたターゲットをピーリングした例を例 3-8 に示します。

例 3-8 間接分岐の予測ミスを軽減するピーリング手法

```
function ()
{
   int n = rand();                 // 乱整数は 0 と RAND_MAX の間
     if( ! (n & 0x01) ) THEN
       n = 0;                      // 半分は n = 0 になる
     if (!n) THEN
       handle_0();                 // 分岐履歴に関連した
                                   // 最も一般的なターゲットをピールする
     {
       switch (n) {
         case 1: handle_1(); break; //一般的でない
         case 3: handle_3(); break; //一般的でない
         default: handle_other();    // フォールスルー·パスにある
                                    // ターゲットを指定
     }
   }
}
```

3.4.1.6 ループアンロール

ループをアンロールすると、次のような利点があります。

- アンロールによって、分岐と、インダクション変数を管理するコードの一部が排除されるため、分岐のオーバーヘッドが軽減されます。
- アンロールによって、ループを積極的にスケジューリング (またはパイプライン化) して、レイテンシーを隠蔽できます。依存関係チェーンからクリティカル·パスを明確にする際に、変数をアクティブな状態に保つのに十分な空きレジスターがある場合は、この手法が効果的です。

- アンロールを行うと、冗長的なロードの削除、共通式の排除など、その他の最適化がコードに適用できるようになります。

ループをアンロールすると、以下のコストが生じる可能性があります。

- ループの本体に分岐が含まれる場合、それらのループをアンロールすると、余分な BTB を消費します。アンロールされたループの反復の回数が 16 以下の場合、分岐プレディクターは、ループ本体内の (分岐の向きを交互に変える) 分岐を正確に予測できます。

アセンブリー/コンパイラー・コーディング規則 14 (影響 H、一般性 M): 一般的に、分岐とインダクション変数のオーバーヘッドがループの実行時間の 10% より小さくなるまで、小さいループをアンロールします。

アセンブリー/コンパイラー・コーディング規則 15 (影響 M、一般性 M): 反復回数が予測可能な実行頻度の高いループをアンロールして、反復の回数を 16 以下に抑えます。コードのサイズが大きくなりすぎて、ワーキングセットがトレースキャッシュや命令キャッシュの容量を超える場合は、この操作は必要ありません。ループ本体に 2 つ以上の条件分岐が含まれる場合は、反復の回数が 16/(条件分岐の数) 回になるようにループをアンロールします。

例 3-9 に、アンロールによるその他の最適化の例を示します。

例 3-9 ループアンロール

```
アンロール前:
    do i = 1, 100
        if ( i mod 2 == 0 ) then a( i ) = x
            else a( i ) = y
    enddo
アンロール後:
    do i = 1, 100, 2
        a( i ) = y
        a( i+1 ) = x
    enddo
```

この例では、100 回実行されるループで、偶数番号の要素に X を代入し、奇数番号の要素に Y を代入しています。ループをアンロールすることで、ループ本体内の分岐を 1 つ排除し、代入を効率良く実行できます。

3.4.2 フェッチとデコードの最適化

インテル® Core™ マイクロアーキテクチャーには、フロントエンドのスループットを高める機能が複数搭載されています。以下では、そのような機能を活用する手法を説明します。

3.4.2.1 マイクロフュージョン向けの最適化

レジスターとメモリーオペランドを操作する命令は、対応するレジスター-レジスター形式よりも多くのマイクロオペレーション (uop) にデコードされます。前者の命令に代わってレジスター-レジスター形式を使用して同等の処理を行うには、2 つの命令からなるシーケンスが必要です。後者のシーケンスでは、フェッチ帯域幅が減少する可能性が高くなります。

アセンブリー/コンパイラー・コーディング規則 16 (影響 ML、一般性 M): フェッチ/デコードのスループットを高める場合、命令がマイクロフュージョンからメリットを得られるのであれば、レジスター方式の命令よりもメモリー方式の命令を優先します。

以下の例は、すべてのデコーダーによって処理可能なマイクロフュージョンです。

- 即値のストアを含めた、メモリーに対するすべてのストア。ストアはストア-アドレスとストア-データの 2 つの異なるマイクロオペレーション (uop) として実行されます。
- レジスターとメモリーとの間のすべての「読み出し-変更」(ロード + op) 命令。
  例:
  ```
  ADDPS XMM9, OWORD PTR [RSP+40]
  FADD DOUBLE PTR [RDI+RSI*8]
  XOR RAX, QWORD PTR [RBP+32]
  ```

- 「ロードとジャンプ」形式のすべての命令。
  例:
  ```
  JMP [RDI+200]
  RET
  ```

- 即値とメモリーオペランドを持つ CMP と TEST 命令。

RIP 相対アドレス指定を行うインテル® 64 命令は、以下の場合はマイクロフュージョンの対象になりません。

- 即値の追加が必要な場合。
  例:
  ```
  CMP [RIP+400], 27
  MOV [RIP+3000], 142
  ```

- 制御フローのため RIP が必要な場合。
  例:
  ```
  JMP [RIP+5000000]
  ```

これらの場合、インテル® Core™ マイクロアーキテクチャーおよび Sandy Bridge† マイクロアーキテクチャーでは、2 マイクロオペレーション (uop) からなるフローをデコーダー 0 から供給します。2 マイクロオペレーション (uop) のフローをアライメントの合ったデコーダーからデコーダー 0 に移さなければならないため、デコード帯域幅にわずかな損失が生じます。

グローバルデータへのアクセスでは、RIP アドレス指定が一般的です。これではマイクロフュージョンのメリットが得られないので、コンパイラーはその他のメモリーアドレス指定によってグローバルデータにアクセスすることを検討することがあります。

3.4.2.2 マクロフュージョン向けの最適化

マクロフュージョンでは、2 つの命令が単一のマイクロオペレーション (uop) にマージされます。インテル® Core™ マイクロアーキテクチャーでは、限られた状況下でこのハードウェアによる最適化が行われます。

マクロフュージョンされるペアの最初の命令は、CMP 命令または TEST 命令でなければなりません。この命令では、REG-REG (レジスター-レジスター)、REG-IMM (レジスター-即値)、またはマイクロフュージョンされた REG-MEM (レジスター-メモリー) の比較が可能です。2 番目の命令 (命令ストリーム内で隣接) は、条件分岐でなければなりません。

このような命令ペアは多くのアプリケーションに含まれる可能性があるため、再コンパイルされない既存のバイナリーでも、マクロフュージョンによってパフォーマンスが向上します。すべてのデコーダーは 1 サイクルあたり 1 つのマクロフュージョン済みペアとともに最大 3 命令をデコードできるので、デコード帯域幅は 1 サイクルあたり 5 命令になります。

マクロフュージョンされたそれぞれの命令は、一度のディスパッチで実行されます。このプロセスによってレイテンシーが排除され、分岐予測ミスのペナルティーが 1 サイクル減少します。ソフトウェアは、リネーム/リタイア帯域幅の向上、パイプライン中の仮想ストレージの拡大、少ないビット数で処理量を増加することによる省電力など、その他のメリットも得られます。

以下のリストは、マクロフュージョンが可能な状況の詳細を示します。

- 以下を比較する場合に CMP または TEST をフュージョンできます。
  レジスター-レジスター: 次に例を示します。CMP EAX,ECX; JZ label
  レジスター-即値: 次に例を示します。CMP EAX,0x80; JZ label
  レジスター-メモリー: 次に例を示します。CMP EAX,[ECX]; JZ label
  メモリー-レジスター: 次に例を示します。CMP [EAX],ECX; JZ label

- TEST はすべての条件分岐とフュージョンされます。
- インテル® Core™ マイクロアーキテクチャーでは、CMP は以下の条件分岐のみフュージョンできます。これらの条件分岐では、キャリーフラグ (CF) またはゼロフラグ (ZF) をチェックします。以下は、マクロフュージョン可能な条件分岐のリストです。
  JA または JNBE
  JAE または JNB または JNC
  JE または JZ
  JNA または JBE
  JNAE または JC または JB
  JNE または JNZ

MEM-IMM を比較する場合 (例えば、CMP [EAX],0x80; JZ label の場合)、CMP と TEST はフュージョンされません。インテル® Core™ マイクロアーキテクチャーの 64 ビット・モードでは、マクロフュージョンはサポートされていません。

- Nehalem† マイクロアーキテクチャーでは、以下のマクロフュージョン拡張機能をサポートしています。
  – CMP は、インテル® Core™ マイクロアーキテクチャーではサポートされていなかった以下の条件分岐とフュージョンできます。
    - JL または JNGE
    - JGE または JNL
    - JLE または JNG
    - JG または JNLE
  – 64 ビット・モードでマクロフュージョンがサポートされています。

- Sandy Bridge† マイクロアーキテクチャーにおけるマクロフュージョンの拡張サポートを表 3-1 に示します。詳細については、E.2.2.1 節と例 3-14 を参照してください。

表 3-1 Sandy Bridge† マイクロアーキテクチャーでマクロフュージョン可能な命令

命令	TEST	AND	CMP	ADD	SUB	INC	DEC
JO/JNO	Y	Y	N	N	N	N	N
JC/JB/JAE/JNB	Y	Y	Y	Y	Y	N	N
JE/JZ/JNE/JNZ	Y	Y	Y	Y	Y	Y	Y
JNA/JBE/JA/JNBE	Y	Y	Y	Y	Y	N	N
JS/JNS/JP/JPE/JNP/JPO	Y	Y	N	N	N	N	N
JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE	Y	Y	Y	Y	Y	Y	Y

Haswell† マイクロアーキテクチャーの拡張マクロフュージョンのサポートを表 3-2 に示します。マクロフュージョンでは、reg-imm、reg-mem、および reg-reg のアドレス指定を持つ CMP / TEST / OP をサポートしますが、mem-mem アドレス指定はサポートしていません。

表 3-2 Haswell† マイクロアーキテクチャーのマクロフュージョン可能な命令

Opcode		JCC	ADD / SUB / CMP	INC / DEC	TEST / AND
70	0F 80	Jo	N	N	Y
71	0F 81	Jno	N	N	Y
72	0F 82	Jc / Jb	Y	N	Y
73	0F 83	Jae / Jnb	Y	N	Y
74	0F 84	Je / Jz	Y	Y	Y
75	0F 85	Jne / Jnz	Y	Y	Y
76	0F 86	Jna / Jbe	Y	N	Y
77	0F 87	Ja / Jnbe	Y	N	Y
78	0F 88	Js	N	N	Y
79	0F 89	Jns	N	N	Y
7A	0F 8A	Jp / Jpe	N	N	Y
7B	0F 8B	Jnp / Jpo	N	N	Y
7C	0F 8C	Jl / Jnge	Y	Y	Y
7D	0F 8D	Jge / Jnl	Y	Y	Y
7E	0F 8E	Jle / Jng	Y	Y	Y
7F	0F 8F	Jg / Jnle	Y	Y	Y

アセンブリー/コンパイラー・コーディング規則 17 (影響 M、一般性 ML): 可能な限り、マクロフュージョンをサポートする命令ペアを使用してマクロフュージョンを行います。CMP よりも TEST を優先します。可能な限り、符号なし変数と符号なしジャンプを使用します。比較時に変数が負でないことを論理的に確認します。可能な限り、MEM-IMM 方式の CMP や TEST は回避します。ただし、MEM-IMM 方式を回避するためほかの命令を追加してはなりません。

例 3-10 マクロフュージョン、符号なし反復カウント

	マクロフュージョン未使用	マクロフュージョン使用
C コード	`for (int[1] i = 0; i < 1000; i++)` `  a++;`	`for ( unsigned int[2] i = 0; i < 1000; i++)` `  a++;`

逆アセンブル	`for (int i = 0; i < 1000; i++)`
	`mov  dword ptr [ i ], 0`
	`jmp  First`
	`Loop:`
	`mov  eax, dword ptr [ i ]`
	`add  eax, 1`
	`mov  dword ptr [ i ], eax`
	
	`First:`
	`cmp  dword ptr [ i ], 3E8H3`
	`jge End`
	`  a++;`
	`mov  eax, dword ptr [ a ]`
	`addqq  eax,1`
	`mov  dword ptr [ a ], eax`
	`jmp  Loop`
	`End:`
		`for ( unsigned int i = 0; i < 1000; i++)`
		`xor  eax, eax`
		`mov  dword ptr [ i ], eax`
		`jmp  First`
		`Loop:`
		`mov  eax, dword ptr [ i ]`
		`add  eax, 1`
		`mov  dword ptr [ i ], eax`
		
		`First:`
		`cmp  eax, 3E8H 4`
		`jae End`
		`  a++;`
		`mov eax, dword ptr [ a ]`
		`add  eax, 1`
		`mov  dword ptr [ a ], eax`
		`jmp  Loop`
		`End:`

注意:

1. 符号付き反復カウントではマクロフュージョンが禁止されています。
2. 符号なし反復カウントはマクロフュージョンに対応しています。

3. CMP MEM-IMM, JGE ではマクロフュージョンが禁止されています。
4. CMP REG-IMM, JAE ではマクロフュージョンが許可されます。

例 3-11 マクロフュージョン、If 文

	マクロフュージョン未使用	マクロフュージョン使用

C コード	int[1] a = 7;
	if (a < 77)
	a++;
	else
	a--;
		unsigned int[2] a = 7;
		if (a < 77)
		a++;
		else
		a--;

逆アセンブル	int a = 7;
	mov dword ptr [a], 7
	if (a < 77)
	cmp dword ptr [a], 4DH 3
	jge Dec
	a++;
	mov eax, dword ptr [a]
	add eax, 1
	mov dword ptr [a], eax
	else
	jmp End
	a--;
	Dec:
	mov eax, dword ptr [a]
	sub eax, 1
	mov dword ptr [a], eax
	End::
		unsigned int a = 7;
		mov dword ptr [a], 7
		if (a < 77)
		mov eax, dword ptr [a]
		cmp eax, 4DH
		jae Dec
		a++;
		add eax,1
		mov dword ptr [a], eax
		else
		jmp End
		a--;
		Dec:
		sub eax, 1
		mov dword ptr [a], eax
		End::

注意:

1. 符号付き反復カウントではマクロフュージョンが禁止されています。
2. 符号なし反復カウントはマクロフュージョンに対応しています。
3. CMP MEM-IMM, JGE ではマクロフュージョンが禁止されています。

アセンブリー/コンパイラー・コーディング規則 18 (影響 M、一般性 ML): 比較時に変数が負でないことを論理的に確認できれば、ソフトウェアによってマクロフュージョンを有効にできます。マクロフュージョンを有効にするには、変数をゼロ比較する際に TEST 命令を適切に使用します。

例 3-12 マクロフュージョン、符号付き変数

マクロフュージョン未使用	マクロフュージョン使用
test ecx, ecx jle OutSideTheIF cmp ecx, 64H jge OutSideTheIF <IF コードブロック> OutSideTheIF:	test ecx, ecx jle OutSideTheIF cmp ecx, 64H jae OutSideTheIF <IF コードブロック> OutSideTheIF:

符号付きまたは符号なしの変数「a」では、「CMP a,0」と「TEST a,a」はフラグに関する限り同じ結果となります。TEST はマクロフュージョンの対象になることが多いので、ソフトウェアがマクロフュージョンを有効にする目的で「CMP a,0」の代わりに「TEST a,a」を使用できます。

例 3-13 マクロフュージョン、符号付き比較

C コード	マクロフュージョン未使用	マクロフュージョン使用
if (a == 0)	cmp a, 0 jne lbl ... lbl:	test a, a jne lbl ... lbl:
if (a >= 0)	cmp a, 0 jl lbl; ... lbl:	test a, a jl lbl ... lbl:

Sandy Bridge† マイクロアーキテクチャーでは、より多くの算術命令および論理命令が条件分岐とマクロフュージョンできます。ALU ポートの負荷が高いループでは、マクロフュージョンされた命令はポート 5 と ALU ポートの両方ではなく、ポート 5 しか使用しないため、マクロフュージョンを行うことで負担を軽減できます。

例 3-14 では、「add/cmp/jnz」ループに、ポート 0、ポート 1、ポート 5 いずれかを介してディスパッチできる ALU 命令が 2 つあります。このため、ポート 5 がいずれかの ALU 命令とバインドする確率が高く、JNZ は 1 サイクル待機することになります。「sub/jnz」ループでは、ポート 0、ポート 1、ポート 5 のいずれかとバインドできるのは SUB 命令だけであるため、ADD/SUB/JNZ を同じサイクルでディスパッチできる確率が高くなります。

例 3-14 インテル® マイクロアーキテクチャー Sandy Bridge† におけるマクロフュージョンのさらなる利点

add + cmp + jnz 代替	sub + jnz でのループ制御

lea rdx, buff
xor rcx, rcx
xor eax, eax
loop:
add eax, [rdx + 4 * rcx]
add rcx, 1
cmp rcx, LEN
jnz loop
	lea rdx, buff - 4
	xor rcx, LEN
	xor eax, eax
	loop:
	add eax, [rdx + 4 * rcx]
	sub rcx, 1
	jnz loop

3.4.2.3 レングス変更プリフィクス (LCP)

命令長は、最大で 15 バイトです。一部のプリフィクスを使用すると、デコーダーが認識しなければならない命令長が動的に変化します。プリデコード・ユニットは通常、LCP が存在しないと仮定してバイトストリームの命令の長さを推測します。プリデコーダーがフェッチラインで LCP に遭遇すると、低速の命令長デコード・アルゴリズムが使用されます。低速の命令長デコード・アルゴリズムでは、6 サイクルでフェッチがデコードされます (通常は 1 サイクル)。一般に、プロセッサー・パイプライン内の通常のキューイング・スループットでは、LCP のペナルティーを隠蔽できません。

命令長を動的に変更できるプリフィクスは、以下のとおりです。

- オペランド・サイズ・プリフィクス (0x66)
- アドレス・サイズ・プリフィクス (0x67)

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサー、インテル® Core™ Duo プロセッサー、インテル® Core™ Solo プロセッサーでは、MOV DX, 01234h 命令で LCP ストールが発生する場合があります。固定エンコードの一部として imm16 を含んでいても LCP によって即値サイズを変更する必要がない命令では、LCP ストールは発生しません。64 ビット・モードの REX プリフィクス (4xh) では、2 種類の命令のサイズを変更できますが、LCP ペナルティーは発生しません。

密なループ内で LCP ストールが発生した場合、パフォーマンスが大幅に低下することがあります。浮動小数点演算が多用されるコードのようにデコードがボトルネックでなければ、孤立した LCP ストールでは通常、パフォーマンスの低下は発生しません。

アセンブリー/コンパイラー・コーディング規則 19 (影響 MH、一般性 MH): imm16 値の代わりに imm8 や imm32 値を使用したコード生成を優先します。imm16 が必要な場合、代わりに、同等の imm32 値をレジスターにロードしてから、レジスター内のワード値を使用します。

ダブル LCP ストール

LCP ストールを発生させ、16 バイト・フェッチ・ライン境界をまたぐ命令は、LCP ストールが 2 回発生する場合があります。以下のアライメント条件では、LCP ストールが 2 回発生する可能性があります。

- 命令が mod R/M バイトと SIB バイトとともにエンコードされ、mod R/M バイトと SIB バイトの間でフェッチライン境界をまたがっている場合。
- フェッチラインのオフセット 13 で始まる命令が、レジスターと即値バイトのオフセットアドレス指定を使用してメモリー・ロケーションを参照している場合。

最初のストールは最初のフェッチラインに対して発生し、2 番目のストールは 2 番目のフェッチラインに対して発生します。ダブル LCP ストールが発生すると、11 サイクルのデコード・ペナルティーが生じます。

以下の例では、フェッチラインにおける命令の先頭バイトの場所にかかわらず、LCP ストールが 1 回だけ発生します。

```
ADD DX, 01234H
ADD word ptr [EDX], 01234H
ADD word ptr 012345678H[EDX], 01234H
ADD word ptr [012345678H], 01234H
```

以下の命令では、フェッチラインのオフセット 13 から始まる場合にダブル LCP ストールが発生します。

```
ADD word ptr [EDX+ESI], 01234H
ADD word ptr 012H[EDX], 01234H
ADD word ptr 012345678H[EDX+ESI], 01234H
```

ダブル LCP ストールを回避するには、SIB バイト・エンコーディングまたはアドレス指定をバイト・ディスプレースメントとともに使用することで、LCP ストールを発生させる命令を使用しないようにします。

誤った LCP ストール

誤った LCP ストールは LCP ストールと同じ特性を持ちますが、imm16 値を持たない命令で発生します。

誤った LCP ストールは、LCP を持つ命令が (a) F7 オペコードを使用してエンコードされた場合と、(b) フェッチラインのオフセット 14 に置かれた場合に発生します。これに該当する命令は、not、neg、div、idiv、mul、imul です。命令長デコーダーは、命令オペコードの mod R/M バイトが格納される次のフェッチラインの前に命令長を判断できないので、誤った LCP によって遅延が発生します。

以下の手法は、誤った LCP ストールの回避に役立ちます。

- F7 グループの命令すべての短い操作を長い 32 ビット操作に置き換えます。
- F7 オペコードがフェッチラインのオフセット 14 から開始しないようにします。

アセンブリー/コンパイラー・コーディング規則 20 (影響 M、一般性 ML): 0xF7 オペコード・バイトを使用する命令がフェッチラインのオフセット 14 から始まらないようにします。また、このような命令を使用して 16 ビット・データを操作するのを回避し、短いデータを 32 ビットに置き換えます。

例 3-15 0xF7 グループ命令による誤った LCP 遅延の回避

デコーダーで遅延が発生するシーケンス	遅延を回避する代替シーケンス

`neg word ptr a`	`movsx eax, word ptr a`
	`neg eax`
	`mov word ptr a, AX`

3.4.2.4 ループストリーム検出器 (LSD) の最適化

LSD は、反復が多く、uop キューに収まるループを検出します。uop キューは、分岐予測ミスによって必然的に終了するまでループをストリームします。

LSD はフェッチ帯域幅を改善し、シングルスレッド・モードでは、フロントエンドをスリープ状態にすることで電力を節約します。また、マルチスレッド・モードでは、フロントエンドが他のスレッドに効率良くサービスを提供できるようになります。

次の条件をすべて満たす場合、ループは LSD ストリームの対象となります。

- ループ本体サイズが最大 60uop、分岐が最大 15 個、および 64 バイト・フェッチラインが最大 15 個であること。
- CALL または RET を含まないこと。
- スタック操作の不一致がないこと (POP より PUSH が多いなど)。
- 20 回を超える反復であること。

多くの計算主体ループ、検索、およびソフトウェア文字列の移動はこの条件に一致します。これらのループが BPU の予測容量を超えると、常に分岐予測ミスで終了します。

アセンブリー/コンパイラー・コーディング規則 21 (影響 MH、一般性 MH): 長い命令シーケンスのループ本体を、LSD のサイズ以下の命令ブロックからなるループに分割します。

Ice Lake† Client アーキテクチャーの割り当て帯域幅は、サイクルあたり 4uop から 5uop に増加しました。

LSD に適合するループの本体が 23uop で構成される場合、ハードウェアはループをアンロールして、uop キューに収まるように調整します (この場合は 2 回アンロール)。従って、uop キューには 46uop が格納されます。

ループは、サイクルごとに 5uop が割り当てに転送されます。46 個の uop のうち 45 個が転送された後、次のサイクルで単一の uop のみが転送されます。つまり、そのサイクルでは 4 つの割り当てスロットが無駄になります。このパターンは、予測ミスによってループが終了するまで繰り返されます。ハードウェアによるループのアンロールにより、LSD 中の無駄なスロット数を最小限に抑えることができます。

3.4.2.5 デコード済み命令キャッシュの最適化

デコード済み命令キャッシュは、Sandy Bridge† マイクロアーキテクチャーの新しい機能です。デコード済み命令キャッシュからコードを実行することで 2 つの利点があります。

- より高帯域幅でマイクロオペレーション (uop) をアウトオブオーダー・エンジンに供給できます。
- デコード済み命令キャッシュ内にあるコードはフロントエンドでデコードする必要がありません。これによって、消費電力が削減されます。

デコード済み命令キャッシュとレガシー・デコード・パイプラインの切り替え時にオーバーヘッドが発生します。フロントエンドとデコード済み命令キャッシュをコードで頻繁に切り替えた場合、レガシー・パイプラインだけからコードを実行したときと比べ、ペナルティーが増す恐れがあります。

デコード済み命令キャッシュから「ホット」なコードを供給するには、次の規則に従います。

- 各ホット・コード・ブロックに含まれる命令を 750 未満にします。具体的には、ループ内の命令が 750 よりも多くなるようにアンロールしてはなりません。これによって、ハイパースレッディングが有効であっても、デコード済み命令キャッシュを有効に利用できます。
- ループ内部の計算ブロックが非常に大きいアプリケーションでは、ループ分割を検討します。つまり、オーバーフローが発生しないように、複数のループに分割して、デコード済み命令キャッシュに取り込むようにします。
- 1 コアあたり 1 つのスレッドだけでアプリケーションが実行されることが確実であれば、ホット・コード・ブロックのサイズをおよそ 1500 命令に増やすことが可能です。

密度の高い読み出し-変更-書き込みコード

デコード済み命令キャッシュは、32 バイトにアライメントされたメモリーチャンクあたり、最大で 18 のマイクロオペレーション (uop) しか保持できません。このため、少ないバイト数でエンコードされ、マイクロオペレーション (uop) の数が多い、命令を多く含む密度の高いコードの場合、18 のマイクロオペレーション (uop) という制約をオーバーフローし、デコード済み命令キャッシュに入らない可能性があります。このような命令の典型的な例が、読み出し-変更-書き込み (RMW) 命令です。

RMW 命令は 1 つのメモリー・ソース・オペランドと 1 つのレジスター・ソース・オペランドで構成され、ソース・メモリー・オペランドをデスティネーションとして使用します。これと同じ機能は 2 つまたは 3 つの命令で表現できます。つまり、最初の命令がメモリー・ソース・オペランドを読み出し、2 番目の命令がレジスター・ソース・オペランドで操作を実行して、最後の命令がこの結果を再度メモリーに書き込みます。このように複数の命令を使用した場合、通常、マイクロオペレーション (uop) の数は同じでありながら、より多くのバイト数で同じ機能がエンコードされます。

RMW 命令が使用されるケースとして、コンパイラーがコードサイズを積極的に最適化する場合があります。

デコード済み命令キャッシュにホットコードを格納する際に考えられる解決策をいくつか示します。

- RMW 命令を、同じ機能を持つ 2 つまたは 3 つの命令で置換します。例えば、「adc [rdi], rcx」の長さはわずか 3 バイトです。これに相当するシーケンス「adc rax, [rdi]」 + 「mov[rdi], rax」は、6 バイトになります。
- 密度の高いコードが 2 つの異なる 32 バイト・チャンクに分割されるように、コードをアライメントします。この解決策は、コードが変更されても影響なく、コードを自動的にアライメントするツールを使用する場合に有用です。
- ループに複数の NOP を追加することで、コードを大きくします。なお、この解決策では、実行されるマイクロオペレーション (uop) が増加します。

デコード済み命令キャッシュに対する無条件分岐のアライメント

デコード済み命令キャッシュに入るコードでは、各無条件分岐は、デコード済み命令キャッシュウェイを占有する最後のマイクロオペレーション (uop) となります。そのため、32 バイトにアライメントされたチャンクあたり 3 つの無条件分岐しか、デコード済み命令キャッシュに格納できません。

無条件分岐は、ジャンプテーブルや switch 文では頻繁に見られます。このような構造をデコード済み命令キャッシュに収まるようにする方法を以下に示します。

コンパイラーで、C++ 仮想クラスメソッド用のジャンプテーブルまたは DLL ディスパッチ・テーブルが作成されるような場合、それぞれの無条件分岐は 5 バイトを占有します。そのため、そのうちの最大 7 つを 32 バイト・チャンクと関連付けることができます。したがって、32 バイトでアライメントされた各チャンクで無条件分岐の密度があまりに高い場合、ジャンプテーブルがデコード済み命令キャッシュに収まらなくなります。その結果、分岐テーブルの前後で実行されるコードでは、パフォーマンスが低下する恐れがあります。

この解決策は、分岐テーブルの分岐の間に複数バイトの NOP 命令を追加することです。ただし、コードサイズが大きくなるので、注意する必要があります。しかし、こうした NOP は実行されないので、後続のパイプステージでペナルティーが発生することはありません。

Switch-Case 文でも似たような状況となります。Case 条件の各評価で、無条件分岐が発生します。複数の NOP を使用する同じ解決策を、32 バイトでアライメントされたチャンク内部に収まる 3 つの連続した無条件分岐ごとに適用できます。

デコード済み命令キャッシュウェイの 2 つの分岐

デコード済み命令キャッシュでは、1 つのウェイに最大 2 つの分岐を保持できます。32 バイトでアライメントされたチャンクで分岐の密度が高い場合、またはそのほかの命令との順序によって、チャンク内の命令のすべてのマイクロオペレーション (uop) がデコード済み命令キャッシュに入らないことがあります。これは、そうたびたび起きるものではありません。このような状況が発生した場合、適宜 NOP 命令を追加して、コードにスペースを挿入します。ただし、こうした NOP 命令はホットコードの一部ではないことを確認する必要があります。

アセンブリー/コンパイラー・コーディング規則 22 (影響 M、一般性 M): ESP への明示的な参照を一連のスタック操作 (POP、PUSH、CALL、RET) に含めることを避けます。

3.4.2.6 その他のデコード・ガイドライン

アセンブリー/コンパイラー・コーディング規則 23 (影響 ML、一般性 L): 8 バイト未満の単純な命令を使用します。

アセンブリー/コンパイラー・コーディング規則 24 (影響 M、一般性 MH): 即値およびディスプレースメントのサイズ変更にプリフィクスを使用するのを避けます。

長い命令 (7 バイトを超える) を使用すると、サイクルあたりにデコードされる命令数に制約が生じます。プリフィクスを 1 つ使用するたびに命令長が 1 バイト増え、それによってデコーダーのスループットが制限されます。さらに、複数のプリフィクスは、最初のデコーダーでのみしかデコードされません。これらのプリフィクスは、デコード時に遅延を生じさせます。複数のプリフィクスの使用または即値またはディスプレースメントのサイズの変更にプリフィクスの使用が避けられないときは、ストールする命令の後にそれらをスケジューリングするようにします。

3.5 実行コアの最適化

最近の世代のマイクロアーキテクチャーにおけるスーパースケーラー・アウトオブオーダー実行コアは、複数の実行ハードウェア・リソースを備えており、複数のマイクロオペレーション (uop) を並列実行できます。これらのリソースによって通常、マイクロオペレーション (uop) は、一定のレイテンシーで効率良く実行されます。この並列処理を利用するための一般的なガイドラインを以下に示します。

- 規則 (3.4 節を参照) に従って、有効なデコード帯域幅とフロントエンドのスループットを最大限に高めます。これらの規則には、単一マイクロオペレーション (uop) 命令の優先や、マイクロフュージョン、スタックポインター追跡、マクロフュージョンの利用が含まれます。
- リネーム帯域幅を最大限にします。この節で説明するガイドラインには、パーシャルレジスター、ROB 読み出しポート、フラグを変更する命令の適切な使い方が含まれます。
- リザベーション・ステーション (RS) で複数の依存関係チェーンが同時に有効になり、コードが並列処理を最大限に活用できるように、命令コードにスケジューリングの推奨事項を適用します。
- ハザードを回避し、実行コアで生じる遅延を最小限に抑えることによって、ディスパッチされたマイクロオペレーション (uop) の進行と迅速なリタイアメントを可能にします。

3.5.1 命令の選択

一部の実行ユニットはパイプライン化されていないため、マイクロオペレーション (uop) を連続するサイクルで処理できず、スループットは 1 サイクルあたり 1 マイクロオペレーション (uop) 未満になります。

一般的に、各命令を構成するマイクロオペレーション (uop) の数を考慮し、単一マイクロオペレーション (uop) の命令、4 マイクロオペレーション (uop) 未満の単純な命令、マイクロシーケンサー ROM が必要な命令の順に優先

し、命令を選択すると良いでしょう。マイクロシーケンサーから実行されるマイクロオペレーション (uop) には、追加のオーバーヘッドが発生します。

アセンブリー/コンパイラー・コーディング規則 25 (影響 M、一般性 H): 単一マイクロオペレーション (uop) の命令を優先します。また、レイテンシーが小さい命令を優先します。

コンパイラーが命令の選択をすでに十分に最適化している可能性があり、この場合は、プログラマーの介入は不要です。

アセンブリー/コンパイラー・コーディング規則 26 (影響 M、一般性 L): プリフィクス、特に 0F 以外のプリフィクスが付いた複数のオペコードを避けます。

アセンブリー/コンパイラー・コーディング規則 27 (影響 M、一般性 L): セグメントレジスターを多用しないようにします。

アセンブリー/コンパイラー・コーディング規則 28 (影響 M、一般性 M): 5 つ以上のマイクロオペレーション (uop) を含み、デコードが 1 サイクルで完了しない複雑な命令 (例えば、enter、leave、loop) の使用を避けます。代わりに、単純な命令のシーケンスを使用します。

アセンブリー/コンパイラー・コーディング規則 29 (影響 MH、一般性 M): 関数の呼び出しとリターンにおけるスタック空間とアドレス調整の管理には、enter/leave 命令の代わりに push/pop 命令を使用します。非ゼロの即値を持つ enter 命令を使用すると、予測ミスに加えて重大なパイプラインの遅延が生じます。

理論上 4-1-1-1 テンプレートに合うように命令シーケンスを配置することが、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーには推奨されます。ただし、フロントエンドのマクロフュージョン機能やマイクロフュージョン機能を利用する場合、4-1-1-1 テンプレートを使用して命令シーケンスのスケジューリングを試みると、効果が減少する可能性があります。

代わりに、ソフトウェアは以下の追加デコード・ガイドラインに従うことを推奨します。

- 複数のマイクロオペレーション (uop) からなる、マイクロシーケンスでない命令を使用する場合は、少数の単一マイクロオペレーション (uop) 命令に分割します。以下は、複数のマイクロオペレーション (uop) からなる、マイクロシーケンサーが不要な命令の例です。
  ADC/SBB
  CMOVcc
  読み出し-変更-書き込み命令

- 複数のマイクロオペレーション (uop) で構成される命令を分割できない場合、同等の異なる命令シーケンスに分割します。例えば、読み出し−変更−書き込み命令は、読み出し−変更命令 + ストア命令に分割すると高速化できます。この手法を利用すると、新しいコードシーケンスが元のコードシーケンスより大きくなった場合でも、パフォーマンスが向上します。

3.5.1.1 整数除算

通常は、整数除算には CWD 命令または CDQ 命令が先行します。除算命令は、オペランドサイズによって、DX:AX または EDX:EAX を被除数として使用します。CWD 命令は、AX を符号拡張して DX に格納し、CDQ 命令は、EAX を符号拡張して EDX に格納します。これらの命令は、シフトとムーブよりコードのサイズが縮小しますが、マイクロオペレーション (uop) の数は同じです。AX または EAX が正であることが分かっている場合は、これらの命令を以下の命令で置き換えます。

```
xor dx, dx
```

または

```
xor edx, edx
```

現在のコンパイラーは、通常、コンパイル時に除数が既知の整数定数となる整数除算を含む数式を、IMUL 命令を使用する高速なシーケンスに変換します。したがって、プログラマーは、コンパイル時に値が不明な除数を含む整数除算数式をできる限り減らす必要があります。

また、特定の既知の除数値が不明な範囲よりも優先される場合、この優先される少数の既知の除数値を特定し、定数除数数式に置換するよう、ソフトウェアで対応することもできます。

13.2.4 節で、MUL/IMUL を使用して整数除算を置換する詳細について説明しています。

3.5.1.2 LEA 命令の使用

Sandy Bridge† マイクロアーキテクチャーでは、LEA 命令のパフォーマンス特性に 2 つの大きな変更が行われています。

- LEA はほとんどの場合、ポート 1 とポート 5 を介してディスパッチできるため、以前の世代と比べスループットが 2 倍になります。ただし、これが適用されるのは、ソースオペランドが 1 つもしくは 2 つの LEA 命令のみです。

例 3-16 独立した 2 オペランド LEA の例

```
mov edx, N
mov eax, X
mov ecx, Y

loop:
lea ecx, [ecx + ecx]    // ecx = ecx*2
lea eax, [eax + eax *4] // eax = eax*5
and ecx, 0xff
and eax, 0xff
dec edx
jg loop
```

- ソースオペランドが 3 つで、以下のような特定の LEA 命令では、命令レイテンシーが 3 サイクルに増え、ポート 1 を介してディスパッチされる必要があります。
  - base、index、offset の 3 つすべてのソースオペランドを持つ LEA
  - ベースが EBP、RBP、R13 いずれかのベースレジスターとインデックス・レジスターを使用する LEA
  - RIP 相対アドレス指定モードを使用する LEA
  - 16 ビット・アドレス指定モードを使用する LEA

例 3-17 3 オペランド LEA の代替手法

3 オペランド LEA (低速)	2 オペランド LEA 代替手法	代替手法 2

#define K 1
uint32 an = 0;
uint32 N= mi_N;
mov ecx, N
xor esi, esi;
xor edx, edx;
cmp ecx, 2;
jb finished;
dec ecx;
 	
loop1:
mov edi, esi;
lea esi, [K+esi+edx];
and esi, 0xFF;
mov edx, edi;
dec ecx;
jnz loop1;
finished:
mov [an] ,esi;
	#define K 1
	uint32 an = 0;
	uint32 N= mi_N;
	mov ecx, N
	xor esi, esi;
	xor edx, edx;
	cmp ecx, 2;
	jb finished;
	dec ecx;
	
	loop1:
	mov edi, esi;
	lea esi, [K+edx];
	lea esi, [esi+edx];
	and esi, 0xFF;
	mov edx, edi;
	dec ecx;
	jnz loop1;
	finished:
	mov [an] ,esi;
		#define K 1
		uint32 an = 0;
		uint32 N= mi_N;
		mov ecx, N
		xor esi, esi;
		mov edx, K;
		cmp ecx, 2;
		jb finished;
		mov eax, 2
		dec ecx;
		loop1:
		mov edi, esi;
		lea esi, [esi+edx];
		and esi, 0xFF;
		lea edx, [edi +K];
		dec ecx;
		jnz loop1;
		finished:
		mov [an] ,esi;

Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサーでは、多くの場合、LEA 命令か、LEA、ADD、SUB、SHIFT 命令のシーケンスを使用して、定数乗算命令を置き換えます。LEA 命令は、次の例のように、複数のオペランドの加算命令としても使用できます。

```
LEA ECX, [EAX + EBX * 4 + A]
```

この方法で LEA を使用すると、算術命令のオペランドにレジスターを必要とせず、レジスターの利用率を軽減できます。この方法は、コードサイズの節約にもなります。

LEA 命令が一定量だけシフトする場合、シフトの代わりに ADD 命令を使用して、LEA 命令を適切なマイクロオペレーション (uop) のシーケンスに置き換えるとレイテンシーが小さくなります。しかし、これによりマイクロオペレーション (uop) の総数が増えるため、次のようなトレードオフが生じます。

アセンブリー/コンパイラー・コーディング規則 30 (影響 ML、一般性 L): スケール・インデックスを使用する LEA 命令がクリティカル・パス上にある場合は、ADD 命令のシーケンスの方が適しています。コードのサイズとトレースキャッシュの帯域幅が重要な要因である場合、LEA 命令を使用します。

3.5.1.3 Sandy Bridge† マイクロアーキテクチャーにおける ADC および SBB 命令

Sandy Bridge† マイクロアーキテクチャーにおける ADC および SBB のスループットは、以前の世代が 1.5 ~ 2 サイクルであったのに対し、1 サイクルに改善されています。この 2 つの命令は、ハードウェアがサポートする最大幅よりも幅が広い整数データ型の数値処理に有用です。

例 3-18 512 ビット加算の例

```
// 64 ビットを 512 数に加算
   lea rsi, gLongCounter
   lea rdi, gStepValue
   mov rax, [rdi]
   xor rcx, rcx
oop_start:
   mov r10, [rsi+rcx]
   add r10, rax
   mov [rsi+rcx], r10
   mov r10, [rsi+rcx+8]
   adc r10, 0
   mov [rsi+rcx+8], r10

   mov r10, [rsi+rcx+16]
   adc r10, 0
   mov [rsi+rcx+16], r10
   mov r10, [rsi+rcx+24]
   adc r10, 0
   mov [rsi+rcx+24], r10
   mov r10, [rsi+rcx+32]
   adc r10, 0
   mov [rsi+rcx+32], r10
   mov r10, [rsi+rcx+40]
   adc r10, 0
   mov [rsi+rcx+40], r10

   mov r10, [rsi+rcx+48]
   adc r10, 0
   mov [rsi+rcx+48], r10
   mov r10, [rsi+rcx+56]
   adc r10, 0
   mov [rsi+rcx+56], r10
   add rcx, 64
   cmp rcx, SIZE
   jnz loop_start
```

```
// 512 ビット加算
loop1:
   mov rax, [StepValue]
   add rax, [LongCounter]
   mov LongCounter, rax
   mov rax, [StepValue+8]
   adc rax, [LongCounter+8]
   mov LongCounter+8, rax
   mov rax, [StepValue+16]
   adc rax, [LongCounter+16]

   mov LongCounter+16, rax
   mov rax, [StepValue+24]
   adc rax, [LongCounter+24]
   mov LongCounter+24, rax
   mov rax, [StepValue+32]
   adc rax, [LongCounter+32]
   mov LongCounter+32, rax
   mov rax, [StepValue+40]
   adc rax, [LongCounter+40]
   mov LongCounter+40, rax
   mov rax, [StepValue+48]
   adc rax, [LongCounter+48]

   mov LongCounter+48, rax
   mov rax, [StepValue+56]
   adc rax, [LongCounter+56]
   mov LongCounter+56, rax
   dec rcx
   jnz loop1
```

3.5.1.4 ビット単位のローテーション

ビット単位のローテーションでは、CL レジスターで指定したカウントによる ROTATE 命令、即値で与えられた定数による ROTATE 命令、1 ビットによる ROTATE 命令を選択できます。通常、即値による ROTATE 命令とレジスターによる ROTATE 命令は、1 ビットによる ROTATE 命令よりも低速です。1 命令による ROTATE のレイテンシーは、SHIFT と同じです。

アセンブリー/コンパイラー・コーディング規則 31 (影響 ML、一般性 L): レジスターや即値を使用した ROTATE 命令の使用を避けて、できるだけ 1 ビットの ROTATE 命令で置き換えます。

Sandy Bridge† マイクロアーキテクチャーでは、即値による ROL/ROR のスループットは 1 サイクルで、ソースおよびデスティネーションと同じレジスターを使用する即値定数による SHLD/SHRD のレイテンシーは 1 サイクル、スループットは 0.5 サイクルです。"ROL/ROR reg, imm8" 命令は、ローテートレジスター向けに 1 サイクル、およびフラグ向けの 2 サイクル (使用される場合) のレイテンシーをともなう 2 つの uop で構成されます。

Ivy Bridge† マイクロアーキテクチャーでは、1 以上の即値を持つ "ROL/ROR reg, imm8" 命令は、オーバーフロー・フラグの結果が使用される場合、1 つの uop につき 1 サイクルのレイテンシーが生じます。即値が 1 である

場合、後続の命令による ROL/ROR のオーバーフロー・フラグの結果に対する依存性は、ROL/ROR 命令で 2 サイクルのレイテンシーとして現れます。

3.5.1.5 可変ビット・カウント・ローテーションとシフト

Sandy Bridge† マイクロアーキテクチャーでは、"ROL/ROR/SHL/SHR reg, cl" 命令は 3 つの uop で構成されます。フラグの結果を必要としない場合、3 つの uop の 1 つが必要なくなり、多くの状況でパフォーマンスが向上します。これらの命令が以降で使用されるフラグの結果を部分的に更新する場合、3 つの uop フローは完全に実行されリタイアする必要があるため、パフォーマンスは低下します。Ivy Bridge† マイクロアーキテクチャーでは、フラグの結果を部分的に更新する 3 つの uop フローには追加の遅延が発生します。次のループシーケンスを考えてみます。

```
loop:
      shl eax, cl
      add ebx, eax
      dec edx          ; DEC はキャリーを更新しません。そのため SHL は 3 つの uop を
                       ; 低速に実行する原因となります
      jnz loop
```

DEC 命令を実行しても、キャリーフラグの状態は変化しません。その結果、SHL EAX, CL 命令は、後続の反復で 3 つの uop フローを実行する必要が生じます。SUB 命令はすべてのフラグを更新します。そのため、DEC を SUB に置き換えることで、SHL EAX, CL は 2 つの uop フローで実行できます。

3.5.1.6 アドレス計算

アドレスの計算には、汎用計算ではなく、アドレス指定モードを使用します。メモリー参照命令は、内部で以下の 4 つのオペランドを使用できます。

- 再配置可能なロード時定数
- 即値定数
- ベースレジスター
- スケール・インデックス・レジスター

Sandy Bridge† マイクロアーキテクチャーでは、オペランドが 3 つ以上の場合、LEA のレイテンシーおよびスループットが低下します (3.5.1.2 節を参照)。ベースレジスターとインデックス・レジスターの両方を使用するアドレス指定モードは、実行エンジンで使用する読み出しポートリソースが多くなり、読み出しポートリソースの可用性によりストールが発生する可能性が高くなります。そのため、ソフトウェアはより迅速にアドレス計算を行う必要があります。

セグメント化されたモデルでは、セグメントレジスターによるリニアアドレス計算のオペランドが 1 つ増えることがあります。多くの場合、メモリー参照のオペランドをフルに利用することで、いくつかの整数命令を排除できます。

3.5.1.7 レジスターのクリアと依存関係解消イディオム

パーシャルレジスター (レジスターの一部分) を変更するコードシーケンスは、依存関係の連鎖によって遅延を生じる可能性がありますが、これは依存関係を解消することで回避できます。

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、いくつかの命令を使用して、ソフトウェアがレジスターの内容をゼロクリアすることで実行依存関係を解消できます。これには以下の命令が含まれます。

```
XOR REG, REG
SUB REG, REG
XORPS/PD XMMREG, XMMREG
PXOR XMMREG, XMMREG
SUBPS/PD XMMREG, XMMREG
PSUBB/W/D/Q XMMREG, XMMREG
```

Sandy Bridge† マイクロアーキテクチャー・ベースのプロセッサーでは、上記の命令に加え、対応するインテル® AVX 命令もゼロイディオムとなり、依存関係チェーンを解消するために使用できます。これらの命令は発行ポートや実行ユニットを使用しません。そのため、レジスターに「0」を移動するよりも、ゼロイディオムを使用する方が適切です。インテル® AVX 対応のゼロイディオム命令には次のものがあります。

```
VXORPS/PD XMMREG, XMMREG
VXORPS/PD YMMREG, YMMREG
VPXOR XMMREG, XMMREG
VSUBPS/PD XMMREG, XMMREG
VSUBPS/PD YMMREG, YMMREG
VPSUBB/W/D/Q XMMREG, XMMREG
```

インテル® AVX-512 をサポートするマイクロアーキテクチャーには、マスクされていないバージョンの命令を使用する 512 ビット・レジスターのゼロイディオムに相当する命令があります。

```
VXORPS/PD ZMMREG, ZMMREG
VPXOR ZMMREG, ZMMREG
VSUBPS/PD ZMMREG, ZMMREG
VPSUBB/W/D/Q ZMMREG, ZMMREG
```

XOR 命令と SUB を使用して、デスティネーション・レジスターのゼロ評価における実行時の依存関係を排除できます。

アセンブリー/コンパイラー・コーディング規則 32 (影響 M、一般性 ML): レジスターを 0 に設定するときや、レジスターの再利用によって発生した不正な依存関係チェーンを解消するときは、依存関係を解消する機能を持つ命令を使用します。条件コードを維持する必要がある場合は、これらの命令に代わって、レジスターに 0 を移動します。この操作は、XOR や SUB よりコード空間を必要としますが、条件コードは設定されません。

例 3-19 では、配列要素に対して減算を実行する際に、PXOR を使用して XMM レジスターの依存関係を解消しています。

```
int a[4096], b[4096], c[4096];
for ( int i = 0; i < 4096; i++ )
      c[i] = - ( a[i] + b[i] );
```

例 3-19 配列要素を減算する際にレジスターをクリアして依存関係を解消

依存関係を解消していない減算 (-x = (x XOR (-1)) - (-1))	**PXOR reg, reg を使用して依存関係を解消している減算 (-x = 0 -x) 依存性**
Lea eax, a	ea eax, a
lea ecx, b	lea ecx, b
lea edi, c	lea edi, c
xor edx, edx	xor edx, edx
movdqa xmm7, allone	lp:
lp:	
movdqa xmm0, [eax + edx]	movdqa xmm0, [eax + edx]
paddd xmm0, [ecx + edx]	paddd xmm0, [ecx + edx]
pxor xmm0, xmm7	pxor xmm7, xmm7
psubd xmm0, xmm7	psubd xmm7, xmm0
movdqa [edi + edx], xmm0	movdqa [edi + edx], xmm7
add edx, 16	add edx,16
cmp edx, 4096	cmp edx, 4096
jl lp	jl lp

アセンブリー/コンパイラー・コーディング規則 33 (影響 M、一般性 MH): パーシャルレジスターの代わりに 32 ビット・レジスターを操作すると、レジスターの一部分を操作する命令との依存関係を解消できます。移動を行う場合、32 ビット移動命令または MOVZX 命令を使用します。

符号拡張によるセマンティクスは、オペランドをゼロ拡張すると維持できることがあります。例えば以下の C コードには、符号拡張は不要であり、オペランドサイズをオーバーライドするプリフィクスも不要です。

```
static short INT a, b;
IF (a == b) {
...
}
```

この 16 ビット・オペランドを比較するコードは、以下のようになります。

```
MOVZW EAX, [a]
MOVZW EBX, [b]
CMP EAX, EBX
```

このような状況はしばしば見うけられます。ただし、「より大きい」、「より小さい」、「より大きいか等しい」などの比較の場合や、EAX または EBX 内の値が、符号拡張を必要とする操作では、この手法は無効です。

アセンブリー/コンパイラー・コーディング規則 34 (影響 M、一般性 M): 符号拡張付き移動命令を使用する代わりに、ゼロ拡張を使用するか、または 32 ビット・オペランドを操作します。

32 ビット値でしか表現できないオペランドを使用する命令が隣接していない場合は、トレースキャッシュをよりコンパクトにパックできます。

アセンブリー/コンパイラー・コーディング規則 35 (影響 ML、一般性 L): (符号拡張された 16 ビット即値としてコード化できない) 32 ビット即値を使用する命令を隣接して配置しないようにします。32 ビット即値を使用するマイクロオペレーション (uop) の直前または直後には、即値を持たないマイクロオペレーション (uop) を配置します。

3.5.1.8 比較

レジスター値を 0 と比較するときは、TEST 命令を使用します。TEST 命令は、基本的には、デスティネーション・レジスターへの書き込みを行わず、オペランド同士の AND (論理積) 演算を実行します。AND 命令を使用するとレジスターを 1 つ余分に消費するため、TEST 命令を使用する方が適切です。また、命令サイズが小さいため、TEST 命令は CMP ..., 0 命令より優れています。

レジスターが EAX の場合、AND (論理積) の結果と即値定数を比較するときは、TEST 命令を使用します。以下に例を示します。

```
IF (AVAR & 8) { }
```

TEST 命令を使用して、mod (2 の整数乗) のロールオーバーを検出することもできます。例えば、次の C コードは、

```
IF ( (AVAR % 16) == 0 ) { }
```

次の命令でコーディングできます。

```
TEST EAX, 0x0F
JNZ AfterIf
```

フラグレジスターの一部を変更する可能性がある命令とフラグレジスターを使用する命令との間で TEST 命令を用いると、パーシャル・フラグ・レジスター・ストールを回避できます。

アセンブリー/コンパイラー・コーディング規則 36 (影響 ML、一般性 M): 論理 AND の結果が使用されないときは、AND 命令に代わって TEST 命令を使用します。これにより、実行時のマイクロオペレーション (uop) 数が軽減されます。レジスターをゼロと CMP するのではなく、レジスター自身との TEST を行います。これにより、ゼロをエンコードする必要がなくなり、命令を小さくエンコードできます。定数をメモリーオペランドと比較することは避けます。メモリーオペランドをロードし、定数をレジスターと比較する方が適切です。

多くの処理では、求められた値がゼロと比較され、分岐に使用されます。ほとんどのインテル® アーキテクチャー命令は、実行の結果として条件コードを設定するため、比較命令を排除できます。したがって、操作は JCC 命令によって直接テストできます。例外は、MOV 命令と LEA 命令です。これらの場合、TEST 命令を使用する必要があります。

アセンブリー/コンパイラー・コーディング規則 37 (影響 ML、一般性 M): 先行する算術命令によってすでにフラグが設定されている場合、適切な条件分岐命令を使用して、ゼロとの比較命令を排除します。必要に応じて、比較の代わりに TEST 命令を使用します。コードの変換によって、オーバーフローの問題が起こらないように注意してください。

3.5.1.9 NOP の使用

コード・ジェネレーターは、ノー・オペレーション (NOP) を生成して命令のアライメントを合わせます。32 ビット・モードで長さが異なる NOP の例を表 3-3 に示します。

1 バイト:XCHG EAX, EAX
2 バイト: 66 NOP
3 バイト: LEA REG, 0 (REG) (8 ビット・ディスプレースメント)

表 3-3 推奨される複数バイトシーケンスの NOP 命令

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

これらはすべて真の NOP であり、EIP レジスターの値を増やす以外に、マシンの状態に影響しません。NOP のデコードと実行にはハードウェア・リソースが必要なため、できるだけ少数の NOP を使用して、パディングを行う必要があります。

1 バイト NOP (XCHG EAX,EAX) はハードウェアによって特別にサポートされます。この NOP は 1 つのマイクロオペレーション (uop) とそれに関連するリソースを使用しますが、EAX の元の値への依存関係は解消されます。このマイクロオペレーション (uop) は、パイプラインの早い段階で実行できるため、未処理の命令の数を減らすことができ、最もコストの小さい NOP と言えます。

その他の NOP には、ハードウェアの特別なサポートはありません。これらの NOP の入力レジスターと出力レジスターは、ハードウェアによって解釈されます。したがって、コード・ジェネレーターは、NOP ができるだけ早い時期に RS リソースをディスパッチして解放できるように、最も古い値を保持しているレジスターを入力としてコードを構成する必要があります。

次の推奨事項に従って、NOP 生成の方法を選択します。

- できるだけ少数の NOP と疑似 NOP を使用して、パディングを行います。
- 処理の遅い実行ユニットで実行される可能性の低い NOP を選択します。
- 依存関係を減らすように、NOP のレジスター引数を選択します。

3.5.1.10 SIMD データ型の混在

以前のマイクロアーキテクチャー (インテル® Core™ マイクロアーキテクチャーより前) では、XMM レジスター上で整数演算と浮動小数点 (FP) 演算を混在させることに明確な制限はありませんでした。インテル® Core™ マイクロアーキテクチャーでは、XMM レジスターに整数演算と浮動小数点演算を混在させると、パフォーマンスが低下することがあります。

ソフトウェアは、XMM レジスター上での整数/浮動小数点演算の混在使用を回避する必要があります。具体的には以下のようにします。

- SIMD 整数演算への供給には SIMD 整数演算を使用します。イディオムには PXOR を使用します。
- SIMD 浮動小数点演算への供給には SIMD 浮動小数点演算を使用します。イディオムには XORPS を使用します。
- 浮動小数点演算がビット単位で等価である場合、PD データ型の代わりに PS データ型を使用します。

MOVAPS 命令と MOVAPD 命令の機能は同じですが、MOVAPS の命令エンコードに必要なバイト数は 1 バイト少なくなります。

3.5.1.11 スピル・スケジューリング

コード・ジェネレーターが使用するスピル・スケジューリング・アルゴリズムは、メモリー・サブシステムの影響を受けます。スピル・スケジューリング・アルゴリズムは、アクティブな値が多すぎてレジスターが不足するときに、メモリーに退避させる値を選択するアルゴリズムです。例 3-22 を考えてみます。この例では、A、B、または C を退避する必要があります。

例 3-20 スピル・スケジューリングのコード例

```
LOOP
   C := ...
   B := ...
   A := A + ...
```

最近のマイクロアーキテクチャーでは、以前のプロセッサーより、スピル・スケジューリングに依存関係の深さに関する情報を使用することが重要となっています。上記の例では、A はループ伝搬依存を含むため、A を退避させないことは特に重要です。A を退避させると、ストア/ロードが依存関係チェーン内に置かれるだけでなく、データの準備ができていないためロードがストールして、さらに数サイクルのペナルティーが生じます。

アセンブリー/コンパイラー・コーディング規則 38 (影響 H、一般性 MH): 小さいループは、ループ伝搬依存を退避させるより、ループ不変コードをメモリー上に置く方が適切です。

直感的な認識とは逆に、このような状況では、ループ不変コードをレジスターに置くよりメモリーに置く方が適切です。ループ不変コードをメモリー上に置くことで、ストアデータの準備ができていないことによりロードがストールすることはなくなります。

3.5.1.12 ゼロ・レイテンシー MOV 命令

Ivy Bridge† マイクロアーキテクチャーでは、レジスター-レジスター移動命令の一部はフロントエンドで実行されます (3.5.1.7 節のゼロ化イディオムと同様)。これは、アウトオブオーダー・エンジンのスケジューリングと実行リソースを節約します。大部分のレジスター-レジスター形式の MOV 命令は、ゼロ・レイテンシー MOV の恩恵を得られます。例 3-23 にこれらに適合する形式の詳細と、適合しない例を示します。

例 3-21 ゼロ・レイテンシー MOV 命令

排除できる MOV 命令のレイテンシー	排除できない MOV 命令のレイテンシー
MOV reg32, reg32	MOV reg8, reg8
MOV reg64, reg64	MOV reg16, reg16
MOVUPD/MOVAPD xmm, xmm	MOVZX reg32, reg8 (AH/BH/CH/DH である場合)
MOVUPD/MOVAPD ymm, ymm	MOVZX reg64, reg8 (AH/BH/CH/DH である場合)
MOVUPS?MOVAPS xmm, xmm	MOVSX
MOVUPS/MOVAPS ymm, ymm	
MOVDQA/MOVDQU xmm, xmm	
MOVDQA/MOVDQU ymm, ymm	
MOVDQA/MOVDQU zmm, zmm	
MOVZX reg32, reg8 (AH/BH/CH/DH でない場合)	
MOVZX reg64, reg8 (AH/BH/CH/DH でない場合)	

例 3-22 は、ゼロ・レイテンシー MOV 拡張の利点を生かすため、MOVZX 命令を使用して 8 ビット整数を処理する方法を示しています。以下を考えてみます。

```
X = (X * 3^N ) MOD 256;
Y = (Y * 3^N ) MOD 256;
```

“MOD 256” が “AND 0xff” を使用して実装されると、レイテンシーは結果依存のチェーンとして現れます。入力バイトの切り捨てで MOVZX 形式を使用すると、ゼロ・レイテンシー MOV の恩恵が得られ、45% のスピードアップにつながります。

例 3-22 バイト粒度のデータ計算手法

AND Reg32, 0xff を使用	**MOVZX を使用**
mov rsi, N	mov rsi, N
mov rax, X	mov rax, X
mov rcx, Y	mov rcx, Y
loop:	loop:
lea rcx, [rcx+rcx*2]	lea rbx, [rcx+rcx*2]
lea rax, [rax+rax*4]	movzx, rcx, bl
and rcx, 0xff	lea rbx, [rcx+rcx*2]
and rax, 0xff	movzx, rcx, bl
lea rcx, [rcx+rcx*2]	lea rdx, [rax+rax*4]
lea rax, [rax+rax*4]	movzx, rax, dl
and rcx, 0xff	llea rdx, [rax+rax*4]
and rax, 0xff	movzx, rax, dl
sub rsi, 2	sub rsi, 2
jg loop	jg loop

ゼロ・レイテンシー MOV 命令に依存する密な命令シーケンスをコード化場合、マイクロアーキテクチャー内部のリソースの制約を考慮する必要があります。

例 3-23 ゼロ・レイテンシー MOV 命令の有効性を高めるようにシーケンスを再配置

さらに内部リソースを必要とするゼロ・レイテンシー MOV	**内部リソースを必要としないゼロ・レイテンシー MOV**
mov rsi, N	mov rsi, N
mov rax, X	mov rax, X
mov rcx, Y	mov rcx, Y
loop:	loop:
lea rbx, [rcx+rcx*2]	lea rbx, [rcx+rcx*2]
movzx, rcx, bl	movzx, rcx, bl
lea rdx, [rax+rax*4]	lea rbx, [rcx+rcx*2]
movzx, rax, dl	movzx, rcx, bl
lea rbx, [rcx+rcx*2]	lea rdx, [rax+rax*4]
movzx, rcx, bl	movzx, rax, dl
llea rdx, [rax+rax*4]	llea rdx, [rax+rax*4]
movzx, rax, dl	movzx, rax, dl
sub rsi, 2	sub rsi, 2
jg loop	jg loop

例 3-23 では、RBX/RCX と RDX/RAX は、共有され上書きされるレジスターペアです。右のコードシーケンスでは、レジスターは即座に新しい結果で上書きされ、マイクロアーキテクチャーで提供されるリソースをそれほど消費しません。その結果、ゼロ・レイテンシー MOV 命令を利用するため、内部リソースの 50% しかサポートされない左のコードシーケンスより、およそ 8% 高速となります。

3.5.2 実行コアでのストールの回避

実行コアは一般的なケースを迅速に実行できるように最適化されていますが、マイクロオペレーション (uop) はフロントエンドから ROB や RS への順方向に進行する際に各種のハザード、遅延、ストールに遭遇することがあります。顕著なケースを以下に示します。

- ROB 読み出しポートのストール
- パーシャルレジスター参照のストール
- XMM レジスターへのパーシャル更新のストール
- パーシャル･フラグ･レジスター参照のストール

3.5.2.1 ライトバック･バスの競合

実行エンジン内のライトバック･バスは、処理中にマイクロオペレーション (uop) を容易にアウトオブオーダー実行するのに必要とされるリソースです。実行ユニットの同じスタックで実行される 2 つのマイクロペレーション (uop) が、同時にライトバック･バスを必要とする場合 (付録 E「旧世代のインテル® 64 および IA-32 プロセッサー･アーキテクチャー」の 表 E-11 を参照)、後続のマイクロオペレーション (uop) はライトバック･バスが利用可能になるまで待機することになります。この状況が発生する可能性として、本来ならば実行エンジンへディスパッチできる状態であるのに、レイテンシーの小さい命令に遅延が生じる場合が考えられます。

単一サイクルの MOV と同じディスパッチ･ポートにバインドされる独立した浮動小数点 ADD の反復シーケンスを考えてみます。MOV が利用可能なディスパッチ･ポートを見つけると、ライトバック･バスは ADD によって占有されます。これによって、MOV 操作に遅延が生じます。

この問題が検出された場合、異なるディスパッチ･ポートを使用するように命令を変更して、ライトバックの競合を軽減できます。

3.5.2.2 実行ドメイン間のバイパス

浮動小数点 (FP) ロードを実行すると、追加のレイテンシー･サイクルが発生します。FP スタックと SIMD スタック間で移動を行うと、さらにレイテンシー･サイクルが追加されます。

例:

```
ADDPS XMM0, XMM1
PAND XMM0, XMM3
ADDPS XMM2, XMM0
```

上記の計算の合計レイテンシーは 9 サイクルです。

- 各 ADDPS 命令で 3 サイクル (計 6 サイクル)
- PAND 命令で 1 サイクル
- ADDPS 浮動小数点ドメインから PAND 整数ドメインへのバイパスで 1 サイクル
- PAND 整数ドメインから 2 番目の ADDPS 浮動小数点ドメインへのデータ移動で 1 サイクル

このペナルティーを回避するには、ドメインの変更を最小限に抑えるようにコードを構成すべきです。バイパスでは回避できない場合もあります。

コードのレイテンシー全体をカウントする際は、バイパスサイクル数も考慮します。計算のレイテンシーが大きい場合は、多くの命令を並行して実行するか、依存関係チェーンを解消することで、合計レイテンシーを削減できます。

多くのバイパスドメインが存在し、レイテンシーが極めて大きいコードでは、インテル® Core™ マイクロアーキテクチャーは以前のマイクロアーキテクチャーよりも動作が遅くなることがあります。

3.5.2.3 パーシャル･レジスター･ストール

汎用レジスターには、バイト、ワード、ダブルワードの単位でアクセスできます。64 ビット･モードでは、クワッドワード単位もサポートされています。レジスターの一部分を参照することを、パーシャルレジスター参照と呼びます。

ほかの命令によって部分的に変更されているレジスターを命令が参照したときに、パーシャル・レジスター・ストールが発生します。例えば、前の命令が AL と AH にストアしているのを AX で読み出したり、前の命令が AX を変更している場合に EAX を読み出すと、パーシャル・レジスター・ストールが発生します。

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサー、インテル® Pentium® M プロセッサー (CPUID シグネチャーがファミリー 6、モデル 13 のもの)、インテル® Core™ Solo プロセッサー、インテル® Core™ Duo プロセッサーでは、パーシャル・レジスター・ストールによる遅延は小さくなります。インテル® Pentium® M プロセッサー (CPUID シグネチャーがファミリー 6、モデル 9 のもの) とインテル® Pentium® Pro プロセッサー・ファミリーでは、大きなペナルティーが発生します。

インテル® 64 アーキテクチャーの場合、64 ビット整数レジスターの下位 32 ビットに対する更新では、上位 32 ビットをゼロ拡張するようにアーキテクチャー上で定義されていることに注意してください。この操作は論理的に 32 ビット更新とみなされますが、実際には 64 ビット更新です (したがってパーシャルストールは発生しません)。

パーシャルレジスターを頻繁に参照すると、偽りの依存関係または真の依存関係が生じます。例 3-18 に、パーシャルレジスター参照を原因とする、一連の偽りの依存関係と真の依存関係の例を示します。

命令 4 および 6 (例 3-18 を参照) を変更して、MOV 命令に代わって MOVZX 命令を使用するようにした場合、命令 2 (および、命令 2 の前の命令 1) に対する命令 4 の依存関係、命令 5 に対する命令 6 の依存関係は解消されます。1 つのシリアルな計算のチェーンの代わりに、2 つの独立した計算のチェーンが作成されます。

例 3-24 に、3 つのバイト値を 1 つのレジスターにパックした場合に MOVZX 命令を使ってパーシャル・レジスター・ストールを回避する方法を示します。

例 3-24 整数コードにおけるパーシャル・レジスター・ストールの回避

パーシャル・レジスター・ストールが発生するシーケンス	**MOVZX を使用して遅延を回避する代替シーケンス**
mov al, byte ptr a[2] shl eax,16 mov ax, word ptr a movd mm0, eax ret	movzx eax, byte ptr a[2] shl eax, 16 movzx ecx, word ptr a or eax,ecx movd mm0, eax ret

Sandy Bridge† マイクロアーキテクチャーとそれ以降のすべてのインテル® Core™ マイクロアーキテクチャー世代では、次の場合にパーシャルレジスターをフルレジスターにマージするマイクロオペレーション (uop) を挿入することで、パーシャル・レジスター・アクセスがハードウェアで処理されます。

- AH、BH、CH、DH のいずれかのレジスターへの書き込み後に、同じレジスターの 2 バイト、4 バイト、8 バイト形式を読み出すまでの間。このようなケースは、マージ・マイクロオペレーション (uop) が挿入されます。この挿入により、割り当てサイクル全体が使用され、ほかのマイクロオペレーション (uop) を割り当てることができなくなります。
- 命令のソース (またはレジスターの大きい形式) でない 1 バイトまたは 2 バイトのデスティネーション・レジスターのマイクロオペレーション (uop) 後に、同じレジスターの 2 バイト、4 バイト、8 バイト形式を読み出すまでの間。このようなケースでは、マージ・マイクロオペレーション (uop) はフローの一部となります。次に例を示します。
  - `MOV AX, [BX]`
    メモリーからパーシャルレジスターへロードする場合は、MOVZX または MOVSX を使用して、マージ・マイクロオペレーション (uop) による追加ペナルティーの発生を回避することを検討します。
  - `LEA AX, [BX+CX]`

最適なパフォーマンスを得るには、レジスターを使用する前に、ゼロイディオムを使用することで、パーシャルレジスターのマージ・マイクロオペレーション (uop) が不要になります。

3.5.2.4 パーシャル XMM レジスターストール

パーシャル・レジスター・ストールは、XMM レジスターでも発生します。以下のインテル® SSE 命令とインテル® SSE2 命令では、デスティネーション・レジスターの一部のみが更新されます。

```
MOVL/HPD XMM, MEM64
MOVL/HPS XMM, MEM32
レジスター間での MOVSS/SD
```

上記の命令を使用すると、レジスターの変更されていない部分と変更された部分の間に依存関係チェーンが生じます。この依存関係チェーンは、パフォーマンス低下の原因となります。

例 3-25 に、3 つのバイト値を 1 つのレジスターにパックした場合に MOVZX 命令を使ってパーシャル・レジスター・ストールを回避する方法を示します。

XMM レジスターの部分的な更新によるストールを回避するには、以下の推奨事項に従います。

- XMM レジスターの一部のみを更新する命令の使用は避けます。
- 64 ビット・ロードが必要な場合は、MOVSD 命令または MOVQ 命令を使用します。
- 連続していない場所から同一のレジスターに対して 2 つの 64 ビット・ロードが必要な場合、MOVLPD/MOVHPD 命令に代わって MOVSD/MOVHPD 命令を使用します。
- XMM レジスターをコピーする際は、ソース・レジスター・データの一部のみが必要でも、以下の命令を使用してレジスター全体をコピーします。

```
MOVAPS
MOVAPD
MOVDQA
```

例 3-25 SIMD コードにおけるパーシャル・レジスター・ストールの回避

メモリー・トランザクションに movlpd を使用し、レジスターコピー間に movsd を使用した結果、パーシャル・レジスター・ストールが発生	**メモリーに movsd を使用し、レジスター間コピーに movsd を使用した結果、遅延を回避**

mov edx, x
mov ecx, count
movlpd xmm3,_1_
movlpd xmm2,_1pt5_
align 16
lp:
movlpd xmm0, [edx]
addsd xmm0, xmm3
movsd xmm1, xmm2
subsd xmm1, [edx]
mulsd xmm0, xmm1
movsd [edx], xmm0
add edx, 8
dec ecx
jnz lp
	mov edx, x
	mov ecx, count
	movsd xmm3,_1_
	movsd xmm2, _1pt5_
	align 16
	lp:
	movsd xmm0, [edx]
	addsd xmm0, xmm3
	movapd xmm1, xmm2
	subsd xmm1, [edx]
	mulsd xmm0, xmm1
	movsd [edx], xmm0
	add edx, 8
	dec ecx
	jnz lp

3.5.2.5 パーシャル・フラグ・レジスター・ストール

命令によってフラグレジスターの一部が変更され、後続の命令がフラグの結果に依存する場合、「パーシャル・フラグ・レジスター・ストール」が発生します。これは、シフト命令 (SAR、SAL、SHR、SHL) で最も頻繁に発生します。ゼロ・

シフト・カウントの場合フラグは変更されませんが、シフトカウントは通常、実行時にのみ認識されます。フロントエンドは、命令がリタイアするまでストールします。

フラグレジスターの一部を変更する可能性がある命令として、CMPXCHG8B 命令、各種のローテート命令、STC 命令、STD 命令があります。例 3-26 に、パーシャル・フラグ・レジスター・ストールが発生するアセンブリーと、ストールが発生しない代替コードの例を示します。

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、即値を 1 シフトする処理が専用ハードウェアによって行われるので、パーシャル・フラグ・ストールは発生しません。

例 3-26 パーシャル・フラグ・レジスター・ストールの回避

パーシャル・フラグ・レジスター・ストール	パーシャル・フラグ・レジスター・ストールの回避
xor eax, eax mov ecx, a sar ecx, 2 setz al ;SAR はストールによるキャリーを更新可能	or eax, eax mov ecx, a sar ecx, 2 test ecx, ecx ; test 命令は毎回すべてのフラグをアップデート setz al ;パーシャル・フラグ・レジスター・ストールは発生しない

Sandy Bridge† マイクロアーキテクチャーでは、パーシャル・フラグ・アクセスのコストは、ストールではなく、マイクロオペレーション (uop) の挿入に置換されます。とは言え、フラグを条件付きで書き込むことができる命令 (SHIFT CL など) の前は、フラグの一部分だけに書き込みを行う命令 (INC、DEC、SET CL など) はあまり使用しないことを推奨します。

例 3-27 では、非常に大きな整数 (1024 ビットなど) の加算を実装する 2 つの手法を比較します。例 3-26 の右側の代替シーケンスは、Sandy Bridge† マイクロアーキテクチャーでは左側よりも高速ですが、それ以前のマイクロアーキテクチャーではパーシャル・フラグ・ストールが発生します。

例 3-27 Sandy Bridge† マイクロアーキテクチャーでのパーシャル・フラグ・レジスター・アクセス

ストールを回避するためのパーシャル・フラグ・レジスターを保存	簡易化されたコードシーケンス

lea rsi, [A]
lea rdi, [B]
xor rax, rax
mov rcx, 16 ; 16*64 =1024 bit
lp_64bit:
add rax, [rsi]
adc rax, [rdi]
mov [rdi], rax
setc al ;次の繰り返しのためにキャリーを格納
movzx rax, al
add rsi, 8
add rdi, 8
dec rcx
jnz lp_64bit
	lea rsi, [A]
	lea rdi, [B]
	xor rax, rax
	mov rcx, 16
	lp_64bit:
	add rax, [rsi]
	adc rax, [rdi]
	mov [rdi], rax
	lea rsi, [rsi+8]
	lea rdi, [rdi+8]
	dec rcx
	jnz lp_64bit

3.5.2.6 浮動小数点/SIMD オペランド

レジスターの一部に書き込む移動操作によって、不要な依存関係が生じることがあります。MOVSD REG, REG 命令は、レジスターの全 128 ビットではなく、下位 64 ビットだけに書き込みを行います。これにより (不要であるにもかかわらず) 上位 64 ビットを生成した先行する命令との間に依存関係が発生します。この依存関係によって、レジスター・リネーミングが禁止されるため、並列性が低下します。

代替手段として、128 ビット全体に書き込む MOVAPD 命令を使用します。この命令はレイテンシーが長くなりますが、MOVAPD のマイクロオペレーション (uop) は異なる実行ポートを使用します (このポートは空いている可能

性が高いです)。この変更はパフォーマンスに影響する可能性があります。ただし、依存関係や実行ポートよりレイテンシーが重要な場合もまれにあります。

アセンブリー/コンパイラー・コーディング規則 39 (影響 M、一般性 ML): 例えば、MOVSD XMMREG1、XMMREG2 命令によって、パーシャル浮動小数点レジスターへの書き込みによる依存関係が発生するのを避けます。MOVSD 命令に代わって、MOVAPD XMMREG1、XMMREG2 命令を使用します。

MOVSD XMMREG, MEM 命令は、128 ビット全体に書き込むため依存関係は解消されます。

3.5.3 ベクトル化

この節では、ベクトル化に関する最適化の問題について簡単にまとめています。詳細については、以降の章を参照してください。

ベクトル化とは、特殊なハードウェアが複数のデータ要素に対して同時に同じ操作を実行できるように、プログラムを最適化することです。各世代のプロセッサーは、インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、およびインテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3) によって、ベクトル化をサポートしています。

ベクトル化は、SIMD の特殊なケースです。SIMD とは、Flynn によるアーキテクチャー分類法の定義によれば、複数のデータ要素 (Multiple Data) を並列に操作できる単一命令 (Single Instruction) ストリームを指す用語です。並列に操作できる要素の数は、インテル® ストリーミング SIMD 拡張命令の 4 つの単精度浮動小数点データ要素およびインテル® ストリーミング SIMD 拡張命令 2 の 2 つの倍精度浮動小数点データ要素から、インテル® ストリーミング SIMD 拡張命令 2 の 128 ビット・レジスター内の 16 のバイト要素の操作までの範囲にわたります。したがって、ベクトル長の範囲は 2 から 16 までであり、この値は使用される拡張命令とデータ型によって異なります。

インテル® C++ コンパイラーでは、以下の 3 つの方法でベクトル化をサポートします。

- コンパイラーは、ユーザーの介入なしに SIMD コードを生成できます。
- ユーザーは、プラグマを挿入して、コンパイラーがコードをベクトル化するのを支援します。
- ユーザーは、組込み関数と C++ クラス・ライブラリーを使用して、SIMD コードを明示的に作成できます。

コンパイラーが SIMD コードを生成しやすくするには、グローバルポインターとグローバル変数の使用を避けます。すべてのモジュールを同時にコンパイルし、プログラム全体の最適化を行う場合は、これらの点はあまり問題となりません。

ユーザー/ソース・コーディング規則 2 (影響 H、一般性 M): 長い SIMD ベクトルを使用してより多くの並列処理が可能となるように、できるだけ小さい浮動小数点データ型または SIMD データ型を使用します。例えば、可能な限り倍精度の代わりに単精度を使用します。

ユーザー/ソース・コーディング規則 3 (影響 M、一般性 ML): 最内のネストレベルに反復間の依存関係が生じないように、ループのネストを構成します。特に、直前の反復で生成したデータのストアが、語彙的に直後の反復の同じデータのロードの後に生成されるのを避けます (これは語彙的な後方依存関係と呼ばれます)。

インテル® ストリーミング SIMD 拡張命令セットの整数部は、8 ビット、16 ビット、32 ビットのオペランドを対象としています。32 ビット・オペランドをサポートしていない SIMD 演算もあるため、一部のソースコードは、少ないビット数のオペランドを使用しない限り、ベクトル化できません。

ユーザー/ソース・コーディング規則 4 (影響 M、一般性 ML): ループ中では条件分岐を使用しないようにします。インテル® SSE 命令を使用して分岐を減らします。

ユーザー/ソース･コーディング規則 5 (影響 M、一般性 ML): 複雑なインダクション (ループ) 変数式を使用しないようにします。

3.5.4 部分的にベクトル化可能なコードの最適化

プログラムには、ベクトル化可能なコードとベクトル化できないルーチンが混在していることがあります。部分的にベクトル化可能なコードの例として、ベクトル化されたコードとベクトル化できないコードが混在したループ構造が挙げられます。この様子を図 3-1 に示します。

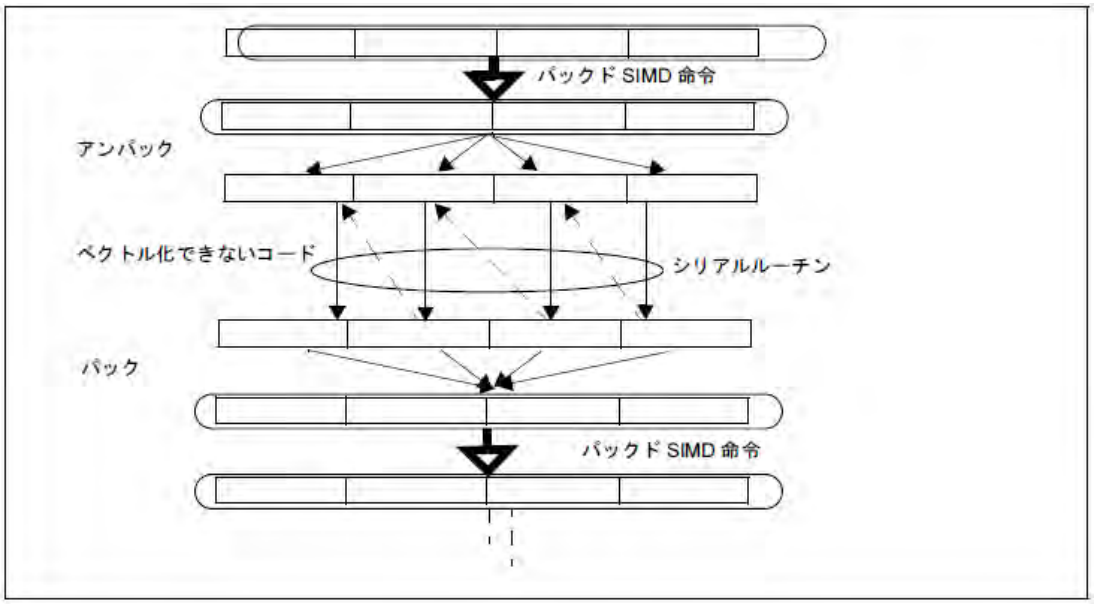


図 3-1 部分的にベクトル化されたコードのプログラムフロー

一般に、ループには以下の 5 つのステージで構成されます。

- プロローグ
- ベクトル化されたデータ構造を個別の要素にアンパック
- ベクトル化できないルーチンを呼び出して各要素をシリアルに処理
- 個別の結果をベクトル化されたデータ構造にパック
- エピローグ

ここでは、部分的にベクトル化可能なコードのパック/アンパックステージに関連したコストおよびボトルネックを削減する手法について説明します。

例 3-28 に示すサンプル･コード･テンプレートでは、部分的にベクトル化可能なコーディング例を示しており、このコードにはパフォーマンスの問題があります。ベクトル化できないコード部分は、「foo」というシリアル関数を複数回呼び出すシーケンスによって表現されています。この例は、「ストア･フォワーディング付きシャッフル」と呼ばれ、アンパックステージにおいてレジスターとメモリー間でデータ要素をシャッフルした後に、パックステージでストア･フォワーディングの問題が発生します。

シリアル化されたコード領域とパックステージとの間でのストア･フォワーディングのボトルネックを排除するには、いくつかの手法があります。以下の項では、パック、アンパック、シリアル化された関数呼び出しのパラメーターの受け渡しに関する代替手法を示します。

例 3-28 部分的にベクトル化可能なプログラムのサンプル･コード･テンプレート

```
// プロローグ ////////////////////////////////
push ebp
mov ebp, esp
// アンパック ////////////////////////////
sub ebp, 32
and ebp, 0xfffffff0
movaps [ebp], xmm0
// コンポーネントのシリアル処理 ///////
sub ebp, 4
mov eax, [ebp+4]
mov [ebp], eax
call foo
mov [ebp+16+4], eax
mov eax, [ebp+8]
mov [ebp], eax
call foo
mov [ebp+16+4+4], eax
mov eax, [ebp+12]
mov [ebp], eax
call foo
mov [ebp+16+8+4], eax
mov eax, [ebp+12+4]
mov [ebp], eax
call foo
mov [ebp+16+12+4], eax
// パック ////////////////////////////////
movaps xmm0, [ebp+16+4]
// エピローグ ////////////////////////////////
pop ebp
ret
```

3.5.4.1 代替パック手法

例 3-28 のサンプルコードで使用されているパック手法では、4 つのダブルワードの結果をメモリーから XMM レジスターにパックする際に、ストア･フォワーディングの制限によって遅延が発生します。

例 3-29 に示す 3 つの代替パック手法では、異なる SIMD 命令を使用してデータを XMM レジスターにパックしています。3 つの手法はいずれも、先行するストア操作と後続のロード操作との間でデータサイズの制限を満たすことにより、ストア･フォワーディングの遅延を回避します。

3-29 ストア･フォワーディングの問題を回避する 3 つの代替パック手法

パック手法 1	**パック手法 2**	**パック手法 3**

movd xmm0, [ebp+16+4]
movd xmm1, [ebp+16+8]
movd xmm2, [ebp+16+12]
movd xmm3, [ebp+12+16+4]
punpckldq xmm0, xmm1
punpckldq xmm2, xmm3
punpckldq xmm0, xmm2
	movd xmm0, [ebp+16+4]
	movd xmm1, [ebp+16+8]
	movd xmm2, [ebp+16+12]
	movd xmm3, [ebp+12+16+4]
	psllq xmm3, 32
	orps xmm2, xmm3
	psllq xmm1, 32
	orps xmm0, xmm1movlhps xmm0, xmm2
		movd xmm0, [ebp+16+4]
		movd xmm1, [ebp+16+8]
		movd xmm2, [ebp+16+12]
		movd xmm3, [ebp+12+16+4]
		movlhps xmm1,xmm3
		psllq xmm1, 32
		movlhps xmm0, xmm2
		orps xmm0, xmm1

3.5.4.2 結果の受け渡しの簡素化

例 3-30 では、隣接するメモリー・ロケーションにストアすることによって、個別の結果がパックステージに受け渡されています。メモリー退避により 4 つの結果を受け渡す代わりに、1 つまたは複数のレジスターを使用して結果を受け渡すこともできます。レジスターを使用して結果の受け渡しを簡素化し、メモリー退避を排除すると、実行時のレジスターへの負担に応じてパフォーマンスを高めることができます。

例 3-28 に示すコードシーケンスでは、4 つの XMM レジスターを使用して、結果を親ルーチンに戻す際に、メモリー退避を行いません。ただし、この手法を使用する場合、ソフトウェアは以下の条件を満たさなければなりません。

- レジスターが不足していない。
- ループ内にストアやロードがあまり存在しないが、多くの計算が含まれている場合、この手法ではパフォーマンスは向上しません。この手法では、ストアポートやロードポートがアイドル状態のときに、計算ユニットを使用する処理が行われます。

例 3-30 4 つのレジスターを使用した、メモリー退避の削減と結果の受け渡しの簡素化

```
mov eax, [ebp+4]
mov [ebp], eax
call foo
movd xmm0, eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
movd xmm1, eax

mov eax, [ebp+12]
mov [ebp], eax
call foo
movd xmm2, eax

mov eax, [ebp+12+4]
mov [ebp], eax
call foo
movd xmm3, eax
```

3.5.4.3 スタックの最適化

例 3-28 では、入力パラメーターはスタックにコピーされ、ベクトル化できないルーチンに渡されて処理されます。連続したメモリー・ロケーションからパラメーターを渡す場合、例 3-31 に示す手法によって簡素化できます。

例 3-31 パラメーターの受け渡しを簡素化するスタックの最適化手法

```
call foo
mov [ebp+16], eax

add ebp, 4
call foo
mov [ebp+16], eax

add ebp, 4
call foo
mov [ebp+16], eax

add ebp, 4
call foo
```

スタックの最適化は、以下の場合にのみ適用できます。

- シリアル操作が関数呼び出しである。関数「foo」が `int foo(int a)` として宣言されている。パラメーターがスタック上に渡される。
- 構成要素の操作の順序が末尾から先頭に向かっている。

ベクトル要素を末尾から先頭に 1 つずつ「foo」に渡す際は、「foo」の呼び出しと EDP の進行に注意してください。

3.5.4.4 チューニングに関する考慮事項

以下に、例 3-28 のループにおけるチューニングの考慮事項を示します。

- 次の組み合わせから 1 つ以上を適用します。
  - 代替パック手法を選択します。
  - 結果の受け渡しを簡素化する方法を検討します。
  - パラメーターの受け渡しを簡素化するスタックの最適化手法を検討します。

- ループの 1 回の反復を実行する平均サイクル数を減らします。
- アンパック操作とパック操作の反復ごとのコストを減らします。

ここで説明する手法によって得られる速度の向上は、選択する組み合わせや、ベクトル化できないルーチンの特性によって異なります。例えば、「foo」ルーチンが短い (密で短いループ表現の) 場合、パック/アンパックの反復ごとのコストは、ベクトル化できないコードに長い操作や多くの依存関係が含まれている状況と比べ、小さくなる傾向があります。これは、密で短いループの反復の多くは実行コア内でパイプライン中にあるので、パック/アンパックの反復ごとのコストは部分的に現れ、パフォーマンスの低下がほとんど発生しないためです。

パック/アンパックの反復ごとのコスト評価を、特定のテストケースに対し系統的な方法で行う必要があります。各テストケースでは、ここで説明している手法を組み合わせて実装します。反復ごとのコストは、以下の方法で評価できます。

- 1 回の反復の実行に必要な平均サイクル数を評価します。
- ベクトル化できないコードの基準ループシーケンスの 1 回の反復の実行に必要な平均サイクル数を評価します。

例 3-32 に示す基準コードシーケンスは、ベクトル化できないルーチンを実行するループの平均コストを評価するのに使用できます。

例 3-32 ループのオーバーヘッドを評価する基準コードシーケンス

```
push ebp
mov ebp, esp
sub ebp, 4

mov [ebp], edi
call foo

mov [ebp], edi
call foo

mov [ebp], edi
call foo

mov [ebp], edi
call foo

add ebp, 4
pop ebp
ret
```

パック/アンパックの反復ごとの平均コストは、以下のように多数の反復の実行時間を測定することによって計算できます。

((テストケースの実行サイクル数) - (同等の基準シーケンスの実行サイクル数)) / (反復数)

例えば、入力パラメーターを返す単純な関数 (密で短いループ表現の) を使用すると、パック/アンパックの反復ごとのコストは、7 サイクル強 (例 3-28 のストア・フォワーディングがあるシャッフルの場合) からおよそ 0.9 サイクル (複数のテストケースで達成) までさまざまです。27 のテストケース (代替パック手法のうちの 1 つで構成、1 つまたは 4 つの結果に対する簡素化あり/なし、スタックの最適化あり/なし) でのパック/アンパックの反復ごとの平均コストは約 1.7 サイクルです。

一般に、パック手法 2 と 3 (例 3-29 を参照) はパック手法 1 よりも堅牢です。1 つまたは 4 つの結果に対する簡素化の最適な選択は、実行時のレジスターへの負荷や、関連するマイクロアーキテクチャーの影響を受けます。

パック/アンパックの反復ごとのコストに関する数値は、単に例として記載されていることを留意してください。異なる基準コードシーケンスを使用したテストケースでは、コストの値も異なります。一般に、ベクトル化できないルーチンが実行に長い時間を必要とする場合、実行コア内のパイプライン中に存在可能なループの反復数が減少するため、コストは増大します。

3.6 メモリーアクセスの最適化

この節では、コードとデータのメモリーアクセスを最適化するガイドラインについて説明します。最も重要な推奨事項には次のものがあります。

- 利用可能な実行帯域幅の中でロード操作とストア操作を行います。
- スペキュレーティブ・エグゼキューションの順方向進行を有効にします。
- ストア・フォワーディングの利点を活用します。
- データのレイアウトとスタックのアライメントに注意して、データのアライメントを合わせます。
- コードとデータは別のページに配置します。
- データの局所性を高めます。
- プリフェッチ命令とキャッシュ制御命令を使用します。
- コードの局所性を向上させて、分岐ターゲットのアライメントを合わせます。

- ライト・コンバイニングを利用します。

3.6.1 ロード/ストア実行帯域幅

一般に、ロードとストアはワークロードで最も頻繁に行われる操作であり、ロードまたはストアを主体とするワークロードでは命令の最大 40% を占めることもまれではありません。各世代のマイクロアーキテクチャーでは、複数のバッファーによって、パイプライン中で実行される命令のロード操作やストア操作をサポートしています。これらのバッファーは、Sandy Bridge†および Ivy Bridge† マイクロアーキテクチャーでは、128 ビット幅のエントリーで構成されていました。Haswell†、Boradwell†、および Skylake† Client マイクロアーキテクチャーでは、サイズが 256 ビットに拡張されました。Skylake† Server、Cascade Lale†、Cascade Lale† Advanced Performance、Ice Lake† Client マイクロアーキテクチャーでは、512 ビットになっています。パフォーマンスを最大化するには、プラットフォームで利用可能な最大幅を使用するのが最適です。

3.6.1.1 Sandy Bridge† マイクロアーキテクチャーのロード帯域幅の利用

以前のマイクロアーキテクチャーはロードポートが 1 つ (ポート 2) であるのに対し、Sandy Bridge† マイクロアーキテクチャーではポート 2 とポート 3 でロードできます。したがって、1 サイクルあたり 2 つのロード操作を行うことができ、コードのロード・スループットは 2 倍になります。その結果、大量のデータを読み込み、メモリーに結果を頻繁に書き出す必要がないコードの性能が向上します (ポート 3 もストアアドレス操作を処理します)。この機能を活用するには、データが L1 データキャッシュにある必要があります。または、シーケンシャルにデータにアクセスすることで、ハードウェア・プリフェッチャーが L1 データキャッシュにデータを適宜取り込むことができるようにします。

配列のすべての要素を加算する以下の C コード 例を考えてみます。

```
int buff[BUFF_SIZE];
int sum = 0;

for (i=0;i<BUFF_SIZE;i++){
    sum+=buff[i];
}
```

代替手法 1 は、この C コードに対してインテル® コンパイラーを使用して、Nehalem† マイクロアーキテクチャー用の最適化オプションによってアセンブリー・コードを生成する方法です。コンパイラーは、インテル® SSE 命令を使用してベクトル化します。このコードは、各 ADD 操作は直前の ADD 操作の結果に依存しています。このため、スループットは 1 サイクルあたり 1 つのロードと ADD 操作に制限されます。代替手法 2 は Sandy Bridge† マイクロアーキテクチャーに合わせて最適化されたもので、追加のロード帯域幅を活用する方法です。このコードでは、配列値の累積合計を求めるため 2 つのレジスターを使用することで、ADD 操作間の依存関係を解消しています。そのため、1 サイクルあたり 2 つのロードと 2 つの ADD 操作を実行できます。

例 3-33 Sandy Bridge† マイクロアーキテクチャーでのロード帯域幅の最適化

レジスターの依存関係が PADD の実行を妨害	レジスターの依存関係を解消することで、2 つのロードポートで PADD の実行に対応

xor eax, eax
pxor xmm0, xmm0
lea rsi, buff
 	
loop_start:
paddd xmm0, [rsi+4*rax]
paddd xmm0, [rsi+4*rax+16]
paddd xmm0, [rsi+4*rax+32]
paddd xmm0, [rsi+4*rax+48]
paddd xmm0, [rsi+4*rax+64]
paddd xmm0, [rsi+4*rax+80]
paddd xmm0, [rsi+4*rax+96]
paddd xmm0, [rsi+4*rax+112]
add eax, 32
cmp eax, BUFF_SIZE
jl loop_start
sum_partials:
movdqa xmm1, xmm0
psrldq xmm1, 8
paddd xmm0, xmm1
movdqa xmm2, xmm0
psrldq xmm2, 4
paddd xmm0, xmm2
movd [sum], xmm0
	xor eax, eax
	pxor xmm0, xmm0
	pxor xmm1, xmm1
	lea rsi, buff
	
	loop_start:
	paddd xmm0, [rsi+4*rax]
	paddd xmm1, [rsi+4*rax+16]
	paddd xmm0, [rsi+4*rax+32]
	paddd xmm1, [rsi+4*rax+48]
	paddd xmm0, [rsi+4*rax+64]
	paddd xmm1, [rsi+4*rax+80]
	paddd xmm0, [rsi+4*rax+96]
	paddd xmm1, [rsi+4*rax+112]
	add eax, 32
	cmp eax, BUFF_SIZE
	jl loop_start
	sum_partials:
	paddd xmm0, xmm1
	movdqa xmm1, xmm0
	psrldq xmm1, 8
	paddd xmm0, xmm1
	movdqa xmm2, xmm0
	psrldq xmm2, 4
	paddd xmm0, xmm2
	movd [sum], xmm0

3.6.1.2 Sandy Bridge† マイクロアーキテクチャーにおける L1D キャッシュのレイテンシー

L1D キャッシュのロード・レイテンシーは変動する可能性があります (付録 E の表 E-15 を参照)。ベストケースは 4 サイクルで、これは以下のいずれかを使用した汎用レジスターに対するロード操作に該当します。

- 1 つのレジスター、または
- ベースレジスターと 2048 未満のオフセット

例 3-34 のポインター追跡のコード例を考えてみます。

例 3-34 ポインター追跡コードのインデックスとポインターの比較

インデックス・ベースのトラバース	ポインター・ベースのトラバース

// C コード例
index = buffer.m_buff[index].next_index;
// アセンブラーの例
loop:
shl rbx, 6
mov rbx, 0x20(rbx+rcx)
dec rax
cmp rax, -1
jne loop
	// C コード例
	node = node->pNext;
	// ASM の例
	loop:
	mov rdx, [rdx]
	dec rax
	cmp rax, -1
	jne loop

左のコードは、インデックス·ベースによるポインター追跡を実装しています。ベース + インデックスとオフセットを使用してメモリーアドレス指定し、コンパイラーはアセンブラーの例に示すコードを生成します。右のコードは、ポインター逆参照コードからベースレジスターのみを使用してコンパイラーが生成したコードです。

Sandy Bridge† マイクロアーキテクチャーおよびそれ以前のマイクロアーキテクチャーでは、右側のコードの方が高速です。ただし、インデックス·ベースの場合、Sandy Bridge† マイクロアーキテクチャーでは、それ以前のマイクロアーキテクチャーよりも速度が低下します。

3.6.1.3 L1D キャッシュバンクの競合の対処

Sandy Bridge† マイクロアーキテクチャーでは、L1D キャッシュの内部構成から 2 つのロード·マイクロオペレーション (uop) でバンクの競合が発生することがあります。2 つのロード操作の間でバンクの競合が発生した場合、競合が解消されるまで、後続のロード操作に遅延が生じます。バンクの競合は、同時に 2 つのロード操作がリニアアドレスの同じビット 2 ~ 5 を持つもので、キャッシュの同じセット (ビット 6 ~ 12) からではない場合に発生します。

バンクの競合については、ロード帯域幅によってコードが制約を受ける場合に限り対処します。バンクの競合が、他のパフォーマンスを制限する要因に隠れ、パフォーマンスの低下に直結しない場合もあります。このようなバンクの競合は排除しても、パフォーマンスは向上しません。

以下のバンクの競合例では、競合を解消するコードの変更方法を示します。サイズがキャッシュライン·サイズの倍数であるソース配列が 2 つ定義されています。同じインデックスを使用して A からの要素とそれに対応する B からの要素をロードした場合、両方の要素のキャッシュラインのオフセットが同じになり、バンクの競合が発生します。

Haswell† マイクロアーキテクチャーでは、この L1D キャッシュのバンク競合は発生しません。

例 3-35 L1D キャッシュのバンクの競合と解消の例

```
int A[128];
int B[128];
int C[128];
for (i=0;i<128;i+=4){
     C[i]=A[i]+B[i]; A[i] と B[i] からのロードが衝突
     C[i+1]=A[i+1]+B[i+1];
     C[i+2]=A[i+2]+B[i+2];
     C[i+3]=A[i+3]+B[i+3];
}
```

```
// バンク競合があるコード
   xor rcx, rcx
   lea r11, A
   lea r12, B
   lea r13, C
loop:
   lea esi, [rcx*4]
   movsxd rsi, esi
   mov edi, [r11+rsi*4]
   add edi, [r12+rsi*4]
   mov r8d, [r11+rsi*4+4]
   add r8d, [r12+rsi*4+4]
   mov r9d, [r11+rsi*4+8]
   add r9d, [r12+rsi*4+8]
   mov r10d, [r11+rsi*4+12]
   add r10d, [r12+rsi*4+12]

   mov [r13+rsi*4], edi
   inc ecx
   mov [r13+rsi*4+4], r8d
   mov [r13+rsi*4+8], r9d
   mov [r13+rsi*4+12], r10d
   cmp ecx, LEN
   jb loop
```

```
// バンク競合がないコード
   xor rcx, rcx
   lea r11, A
   lea r12, B
   lea r13, C
loop:
   lea esi, [rcx*4]
   movsxd rsi, esi
   mov edi, [r11+rsi*4]
   mov r8d, [r11+rsi*4+4]
   add edi, [r12+rsi*4]
   add r8d, [r12+rsi*4+4]
   mov r9d, [r11+rsi*4+8]
   mov r10d, [r11+rsi*4+12]
   add r9d, [r12+rsi*4+8]
   add r10d, [r12+rsi*4+12]

   inc ecx
   mov [r13+rsi*4], edi
   mov [r13+rsi*4+4], r8d
   mov [r13+rsi*4+8], r9d
   mov [r13+rsi*4+12], r10d
   cmp ecx, LEN
   jb loop
```

3.6.2 レジスタースピルを最小限に抑える

プロセッサーが汎用レジスターに保持できるよりも多くのアクティブな変数がコードの一部に含まれている場合、通常取られる方法は、変数の一部をメモリーに保持することです。この方法はレジスタースピルと呼ばれます。L1D キャッシュ・レイテンシーがこのコードのパフォーマンスに悪影響を及ぼすことがあります。この影響は、レジスタースピルのアドレスが、低速なアドレス指定モードを使用しているほど顕著になります。

スピルの影響を回避する 1 つのオプションは、XMM レジスターに汎用レジスターをスピルすることです。この方法は、これまでのプロセッサー世代でもパフォーマンスの向上につながる可能性が高くなります。以下に、レジスターをメモリーではなく、XMM レジスターにスピルする方法を示します。

例 3-36 メモリーではなく XMM レジスターを使用したレジスタースピル

メモリーへのレジスタースピル	XMM へのレジスタースピル

loop:
mov rdx, [rsp+0x18]
movdqa xmm0, [rdx]
movdqa xmm1, [rsp+0x20]
pcmpeqd xmm1, xmm0
pmovmskb eax, xmm1
test eax, eax
jne end_loop
movzx rcx, [rbx+0x60]
 	
add qword ptr[rsp+0x18], 0x10
add rdi, 0x4
movzx rdx, di
sub rcx, 0x4
add rsi, 0x1d0
cmp rdx, rcx
jle loop
	movq xmm4, [rsp+0x18]
	mov rcx, 0x10
	movq xmm5, rcx
	loop:
	movq rdx, xmm4
	movdqa xmm0, [rdx]
	movdqa xmm1, [rsp+0x20]
	pcmpeqd xmm1, xmm0
	pmovmskb eax, xmm1
	test eax, eax
	jne end_loop
	movzx rcx, [rbx+0x60]
	
	padd xmm4, xmm5
	add rdi, 0x4
	movzx rdx, di
	sub rcx, 0x4
	add rsi, 0x1d0
	cmp rdx, rcx
	jle loop

3.6.3 スペキュレーティブ・エグゼキューションとメモリー・ディスアンビゲーションの拡張

インテル® Core™ マイクロアーキテクチャーより前の世代では、コードにストアとロードの両方が含まれている場合、ストアのアドレスが解決されるまでロードを発行できません。この規則によって、先行するストアに対するロードの依存関係が適切に処理されます。

インテル® Core™ マイクロアーキテクチャーには、先行する未知のストアがある場合に一部のロードを投機的 (スペキュレーティブ) に発行できるメカニズムが搭載されています。プロセッサーは、ロードアドレスが、ロードが実行された時点でアドレスが不明な先行するストアとオーバーラップしていないかどうかを確認します。アドレスが重複する場合、ロード命令とそれに続くすべての命令を再実行します。

例 3-37 に示す状態では、ループ中で「Ptr->Array」が変化しないことをコンパイラーは認識できません。したがって、コンパイラーは「Ptr->Array」を不変なものとしてレジスター内に保持できず、反復ごとに読み出す必要があります。ポインターのアドレスが不変になるようにコードを書き直せばこの状態を回避できますが、メモリー・ディスアンビゲーションによって、コードを書き直さずにパフォーマンスを向上できます。

例 3-37 アドレスが不明なストアによってブロックされるコード

C コード	アセンブリー・シーケンス

struct AA {
AA ** array;
};
void nullify_array (AA *Ptr, DWORD
Index, AA *ThisPtr
)
{
while (Ptr->Array[--Index] != ThisPtr)
{
Ptr->Array[Index] = NULL ;
} ;
} ;
	nullify_loop:
	mov dword ptr [eax], 0
	mov edx, dword ptr [edi]
	sub ecx, 4
	cmp dword ptr [ecx+edx], esi
	lea eax, [ecx+edx]
	jne nullify_loop

IA32_SPEC_CTRL.SSBD MSR を使用して、投機的なストアバイパスを無効にすることもできます。このトピックに関する詳細は、https://software.intel.com/security-software-guidance/insights (英語) を参照してください。

3.6.4 ストア・フォワーディング

プロセッサーのメモリーシステムは、ストアのリタイア後にのみ、ストアされたデータをメモリー (キャッシュを含む) に送出します。しかし、同じアドレスからデータを読み込む次のロードに、ストアから直接ストアデータを転送すると、ストアとロードのレイテンシーを大幅に短縮できます。

ストア・フォワーディングを行うには、2 つの必要条件があります。これらの条件が満たされない場合、ストア・フォワーディングは行われず、ロードはキャッシュからデータを取得しなければなりません (したがって、その前にストア操作がキャッシュにデータを書き戻す必要があります)。これによって、基盤となるマイクロアーキテクチャーのパイプラインの段数に関連したペナルティーが発生します。

第 1 の必要条件は、ストア・フォワーディングされるデータのサイズとアライメントです。この制限は、アプリケーションの全体的なパフォーマンスに大きな影響を与える可能性が高くなります。通常、この制限の違反によるパフォーマンスのペナルティーは防止できます。ストア-ロード・フォワーディングの制限は、マイクロアーキテクチャーごとに異なります。ストア・フォワーディングのストールの原因になるコーディングの落とし穴と、それらの解決策の例は、3.6.4.1 節「ストア-ロード・フォワーディングのサイズとアライメントの制限」で詳しく説明しています。第 2 の必要条件は、データの可用性に関するものです。これについては、3.6.4.2 節「ストア・フォワーディングのデータの可用性の制限」で説明しています。冗長なロード操作を排除することが推奨されます。

一時的なスカラー変数をレジスター内に保持し、メモリーに書き込まないことが可能な場合があります。一般的に、このような変数は、間接ポインターを使用してアクセス可能であってはなりません。変数をレジスターに保持できれば、その変数のすべてのロードとストアが排除され、ストア・フォワーディングに関連する問題も発生しなくなります。ただし、この方法を使用すると、レジスターへの負荷が大きくなります。

ロード命令は計算チェーンの開始点に成りやすいことが判明しています。アウトオブオーダー・エンジンはデータの依存関係に基づいているため、ロード命令はエンジンが高いレートで実行する上で重要な役割を果たします。ロードの排除は、高い優先順位で行う必要があります。

ストアの時点と再び使用される時点で変数が変化しない場合、ストアされたレジスターをコピーすることも、直接使用することもできます。ただし、レジスターへの負荷が大きすぎる場合や、ストアと 2 番目のロードの前に関数呼び出しがある場合は、2 番目のロードを排除できないことがあります。

アセンブリー/コンパイラー・コーディング規則 40 (影響 H、一般性 M): 可能な限りスタックではなくレジスターでパラメーターを受け渡します。スタック上での引数の受け渡しは、ストアの後に再ロードを行う必要があります。ストア・フォワーディングの制限上可能であれば、データキャッシュにアクセスせずにメモリー・オーダー・バッファーから直

接ロードに値を供給することで、ハードウェア内でこのシーケンスを最適化できますが、浮動小数点値のフォワーディングには大きなレイテンシーが生じます。浮動小数点引数をレジスター (XMM が望ましい) に渡すと、この大きなレイテンシーの操作がなくなります。

パラメーター受け渡し規則によって、どのパラメーターがスタック上で受け渡され、どのパラメーターがレジスターで受け渡されるか制限される場合があります。ただし、コンパイラーが (プログラム全体を最適化して) バイナリーコード全体のコンパイルを制御できる場合、これらの制限は問題となりません。

3.6.4.1 ストア-ロード・フォワーディングのサイズとアライメントの制限

ストア・フォワーディング・データのサイズとアライメントの制限は、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサー、インテル® Core™2 Duo プロセッサー、インテル® Core™ Solo プロセッサー、インテル® Pentium® M プロセッサーに適用されます。ストア・フォワーディングの制限によるパフォーマンスの低下は、パイプラインが短いプロセッサーのほうが小さくなります。

ストア・フォワーディングの制限は、マイクロアーキテクチャーごとに異なります。以下の規則に従うと、ストア・フォワーディングのサイズとアライメントの制限を満たすことができます。

アセンブリー/コンパイラー・コーディング規則 41 (影響 H、一般性 M): ストアの転送先であるロードは、ストアデータと同じ開始アドレスおよび同じアライメントでなければなりません。

アセンブリー/コンパイラー・コーディング規則 42 (影響 H、一般性 M): ストアから転送されるロードのデータ全体が、ストアデータに含まれていなければなりません。

ストアからの転送先のロードは、ストアのデータがストアバッファーに書き込まれるまで処理を続行できませんが、関連しないロードはそれを待つ必要はありません。

アセンブリー/コンパイラー・コーディング規則 43 (影響 H、一般性 ML): ストアされるデータのうちアライメントされていない部分を抽出する必要がある場合、そのデータを含む、アライメントされた最小部分を読み出して、必要に応じてデータをシフト/マスクします。ストア・フォワーディングに失敗してペナルティーが生じるより、この操作を行ったほうが良いでしょう。

アセンブリー/コンパイラー・コーディング規則 44 (影響 MH、一般性 ML): 同じメモリー領域に対して、大きなデータ幅のストアの後に複数の小さなロードを続けないようにします。必要に応じて、1 回の大きな読み込みとレジスターのコピーを使用します。

例 3-38 に、大きなデータ幅のストアの後に小さなロードが続くストア・フォワーディングの例を示します。最初の 3 つのロード操作は、規則 44 で説明した状態を示しています。ただし、最後のロード操作は、ストア・フォワーディングによって問題なくデータを取得できます。

例 3-38 大きなデータ幅のストアの後に小さなロードが出現する例

```
mov [EBP],'abcd'
mov AL, [EBP]      ; ブロックされない – 同じアライメント
mov BL, [EBP + 1] ; ブロックされる
mov CL, [EBP + 2] ; ブロックされる
mov DL, [EBP + 3] ; ブロックされる
mov AL, [EBP]      ; ブロックされない – 同じアライメント
                   ; 注) 古いブロックされたロードを渡す
```

例 3-39 に、複数の小さなデータ幅のストアの後に大きなロードが続くストア・フォワーディングの例を示します。転送しなければならないすべてのデータがストアバッファーに格納されていないため、ロード操作が必要とするデータを転送できません。同じメモリー領域に対して、小さなデータ幅のストアの後に大きなロードを続けてはなりません。

例 3-39 小さなデータ幅のストアの後に大きなロードが続くためにストア･フォワーディングが失敗した例

```
mov [EBP], 'a'
mov [EBP + 1], 'b'
mov [EBP + 2], 'c'
mov [EBP + 3], 'd'
mov EAX, [EBP] ; ブロックされる
               ; 始めの 4 つの小さなストアは、フォワードしない状況を防ぐために、
               ; 1 つの DWORD ストアに集約されます。
```

例 3-40 に、コンパイラーが生成したコードで、ストア･フォワーディングがストールした例を示します。コンパイラーは、スタックに退避されたバイトを処理し、そのバイトを整数値に変換するために、例 3-40 のようなコードを生成することがあります。

例 3-40 コンパイラーが生成したコード内でのストア･フォワーディングの失敗

```
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
mov eax, DWORD PTR [esp+10h] ; ストール
and eax, 0xff                ; バイト値に変換
```

例 3-41 に、例 3-40 に示したストア･フォワーディングの失敗を避ける 2 つの代替手段を示します。

例 3-41 例 3-40 のストア･フォワーディングの失敗を防ぐ 2 つの方法

```
; A. スピルを無視できる時は、小さなデータのストアの後に大きなデータのロードを
; 避けるために、MOVZ 命令を使用
movz eax, bl                  ; 最近の 3 命令を置き換え
; B. MOVZ 命令を使用し、スタックへのスピルを処理
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
movz eax, BYTE PTR [esp+10h]  ; ブロックされない
```

64 ビットより小さいデータをメモリー･ロケーション間で移動する場合は、64 ビットまたは 128 ビット SIMD レジスター移動命令を使用する方が効率的です (アライメントされている場合)。これらの命令を使用して、アライメントされていないロードを避けることができます。浮動小数点レジスターは 64 ビットを一度に移動できますが、データが誤って変更される可能性があるため、この目的で浮動小数点命令を使用してはいけません。

もう 1 つの例として、例 3-42 の場合について考えます。

例 3-42 大きなデータ幅のロードのストールと小さなデータ幅のロードのストール

```
; A. 大きなロードのストール
mov mem, eax              ; アドレス MEM に DWORD をストア
mov mem, eax              ; アドレス MEM + 4 に DWORD をストア
fld mem ; アドレス MEM から QWORD をロード、ストール
; B. 小さなロードのストール
fstp mem                  ; アドレス MEM に QWORD をストア
mov bx, mem+2             ; アドレス MEM + 2 から WORD をロード、ストール
mov cx, mem+4             ; アドレス MEM + 4 から WORD をロード、ストール
```

最初の例 (A) では、(メモリーアドレス MEM を始点とする) 同じメモリー領域に対して、一連の小さなデータ幅のストアの後に大きなロードが続いています。このため、大きなロードがストールします。

FLD 命令は、ストアがメモリーに書き込むまで、必要なすべてのデータにアクセスできません。その他のデータ型の場合も同様にストールすることがあります。例えば、バイトやワードをいくつかストアしてから、ワードやダブルワードをいくつか同じメモリー領域から読み出す場合などです。

2 番目の例 (B) では、(メモリーアドレス MEM を始点とする) 同じメモリー領域に対して、大きなデータ幅のストアの後に一連の小さなロードが続いています。このため、小さなロードがストールします。

上記の例のワードのロード操作は、クワッドワードのストア操作によるメモリー書き込みが終了しない限り、必要なデータにアクセスできません。その他のデータ型の場合も同様にストールすることがあります。例えば、ダブルワードやワードをいくつかストアしてから、ワードやバイトをいくつか同じメモリー領域から読み出す場合です。この問題は、ロードからストアをできるだけ遠くに離すことで回避できます。

表 3-4 に、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでのストア・フォワーディングの制限を示します。

表 3-4 インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーにおけるストアフォワードの制限

ストア・アライメント	ストアの幅 (ビット)	ロード・アライメント (バイト)	ロードの幅 (ビット)	ストア・フォワーディングの制限
自然サイズ	16	word 単位でアライメント	8, 16	ストールが発生しない
自然サイズ	16	Word 単位でアライメントされない	8	ストールが発生する
自然サイズ	32	dword 単位でアライメント	8, 32	ストールが発生しない
自然サイズ	32	dword 単位でアライメントされない	8	ストールが発生する
自然サイズ	32	word 単位でアライメント	16	ストールが発生しない
自然サイズ	32	word 単位でアライメントされていない	16	ストールが発生する
自然サイズ	64	qword 単位でアライメント	8, 16, 64	ストールが発生しない
自然サイズ	64	qword 単位でアライメントされていない	8, 16	ストールが発生する
自然サイズ	64	dword 単位でアライメント	32	ストールが発生しない
自然サイズ	64	dword 単位でアライメントされていない	32	ストールが発生する
自然サイズ	128	dqword 単位でアライメント	8, 16, 128	ストールが発生しない
自然サイズ	128	dqword 単位でアライメントされていない	8, 16	ストールが発生する
自然サイズ	128	dword 単位でアライメント	32	ストールが発生しない
自然サイズ	128	dword 単位でアライメントされていない	32	ストールが発生する
自然サイズ	128	qword 単位でアライメント	64	ストールが発生しない
自然サイズ	128	qword 単位でアライメントされていない	64	ストールが発生する
アライメントされていないバイト 1 で開始	32	ストアのバイト 0	8, 16, 32	ストールが発生しない
アライメントされていないバイト 1 で開始	32	ストアのバイト 0 以外	8, 16	ストールが発生する
アライメントされていないバイト 1 で開始	64	ストアのバイト 0	8, 16, 32	ストールが発生しない
アライメントされていないバイト 1 で開始	64	ストアのバイト 0 以外	8, 16, 32	ストールが発生する
アライメントされていないバイト 1 で開始	64	ストアのバイト 0	64	ストールが発生する
アライメントされていないバイト 1 で開始	32	ストアのバイト 0	8	ストールが発生しない
アライメントされていないバイト 1 で開始	32	ストアのバイト 0 以外	8	ストールが発生しない
アライメントされていないバイト 1 で開始	32	考慮不要	16, 32	ストールが発生する
アライメントされていないバイト 1 で開始	64	考慮不要	16, 32, 64	ストールが発生する

3.6.4.2 ストア・フォワーディングのデータの可用性の制限

ストアされる値が使用可能になるまで、ロード操作は完了できません。この制限に違反すると、データが使用可能になるまでロード実行は延期されます。この遅延により、一部の実行リソースが必要以上に使用され、かなり大きな遅延が発生することがあります。ただし、これは非決定的な遅延です。この問題の全体への影響は、サイズとアライメントの必要条件に違反した場合よりも、はるかに小さくなります。

現代のマイクロアーキテクチャーでは、ハードウェアは、どの時点でロードが先行するストアに依存し、先行するストアから転送されたデータを取得するかを予測します。これらの予測は、パフォーマンスを大きく向上させます。しかし、ロードと先行するストアの間の時間間隔が短すぎる場合や、ストアされるデータの生成が遅れた場合、大きなペナルティーが生じる可能性があります。

以下の場合、データがメモリーを介して受け渡されるときに、ロードとストアを離す必要があります。

- スタックフレームへのレジスターのスピル、セーブ、リストア
- パラメーターの受け渡し
- グローバル変数と動的変数
- 整数と浮動小数点の間の型変換
- コンパイラーがインライン展開されたコードを分析しない場合、インライン展開されたコードとのインターフェイスに関連する変数が強制的にメモリー退避され、作成されるメモリー変数の数が増え、冗長なロードを排除できなくなります。

アセンブリー/コンパイラー・コーディング規則 45 (影響 H、一般性 MH): ストア・フォワーディングの問題が起こる可能性とその影響が最小限に抑えられるように、(レジスターの割り当てやパラメーターの受け渡しの際に) レジスターへの変数の割り当てに優先順位を付けます (これによって他のペナルティーが生じない場合)。レイテンシーが大きい命令 (例えば、MUL、DIV) によるデータ、ストアとロードの間隔が非常に小さい変数のデータ、多数の依存関係チェーンまたは長い依存関係チェーンを含む変数のデータには、ストア・フォワーディングを適用しません。特に、ループ伝搬依存関係チェーンにはストア・フォワーディングを適用しません。

例 3-43 に、ループ内の依存関係チェーンの例を示します。

例 3-43 ループ内の依存関係チェーン

```
for ( i = 0; i < MAX; i++ ) {
   a[i] = b[i] * foo;
   foo = a[i] / 3;
}                          //foo はループ内依存
```

アセンブリー/コンパイラー・コーディング規則 46 (影響 M、一般性 MH): ロードがストアの完了を待たずに進行するように、できるだけ早くストアアドレスを計算します。

3.6.5 データレイアウトの最適化

ユーザー/ソース・コーディング規則 6 (影響 H、一般性 M): ソースコード内で定義されるデータ構造をパディングして、すべてのデータ要素のアライメントをオペランドサイズの自然アドレス境界に合わせます。

オペランドを SIMD 命令にパックする場合、パックされた要素のサイズ (64 ビットまたは 128 ビット) にアライメントします。構造体と配列の中でパティングを使用して、データのアライメントを合わせます。プログラマーは、構造体と配列を再構成して、パディングによって浪費されるメモリーを最小限に抑えることができます。ただし、コンパイラーはこのような処理を自由に行えるとは限りません。例えば、C プログラミング言語では、構造体の要素がメモリー内に割り当てられる順序を指定します。詳細については、5.4 節「スタックとデータ・アライメント」を参照してください。

例 3-44 に、データ構造を再構成して、サイズを縮小する方法を示します。

例 3-44 データ構造の再構成

```
struct unpacked { /* パディングで 20 バイトに合わせる */
   int a;
   char b;
   int c;
   char d;
   int e;
};
struct packed { /* 16 バイトに合わせる */
   int a;
   int c;
   int e;
   char b;
   char d;
}
```

64 バイトのキャッシュラインのサイズは、ストリーミング・アプリケーション (例えば、マルチメディア) に影響する場合があります。このようなアプリケーションは、データを 1 回だけ参照して使用し、そのデータを廃棄します。データアクセスがキャッシュライン内のデータをまれにしか参照しないと、システムメモリー帯域幅の利用効率が低下します。例えば、例 3-45 に示すように、構造体配列を複数の配列に分解すると、パッキング効率をを改善できます。

例 3-45 配列の分解

```
struct { /* 1600 バイト */
   int a, c, e;
   char b, d;
} array_of_struct [100];

struct { /* 1400 バイト */
   int a[100], c[100], e[100];
   char b[100], d[100];
} struct_of_array;

struct { /* 1200 バイト */
   int a, c, e;
} hybrid_struct_of_array_ace[100];

struct { /* 200 バイト */
    char b, d;
} hybrid_struct_of_array_bd[100];
```

このような最適化の効率は、データの利用パターンによって異なります。構造体の要素がすべて同時にアクセスされ、配列内のアクセスパターンが無作為である場合、ARRAY_OF_STRUCT によって不要なプリフェッチが回避できます (ただし、メモリーが多少浪費されます)。

しかし、配列インデックスがスイープされる場合など、配列のアクセスパターンが局所性を示す場合は、構造体の要素が同時にアクセスされても、プロセッサーのハードウェア・プリフェッチャーは STRUCT_OF_ARRAY からデータをプリフェッチします。

要素 A がほかのエントリーの 10 倍の頻度でアクセスされる場合など、構造体の各要素のアクセス頻度が異なる場合、STRUCT_OF_ARRAY を使用すると、メモリーの節約になるだけでなく、不要なデータアイテム B、C、D、E のフェッチも避けられます。

また、STRUCT_OF_ARRAY を使用すると、プログラマーとコンパイラーが SIMD データ型を使用できます。

ただし、STRUCT_OF_ARRAY には、互いに独立したメモリーストリームの参照回数が増える欠点があります。そのため、プリフェッチの回数が多くなり、アドレス生成計算の量が増加します。また、DRAM ページアクセスの効率に影響することもあります。HYBRID_STRUCT_OF_ARRAY は、2 つの手法を組み合わせたものです。この場合、HYBRID_STRUCT_OF_ARRAY_ACE に 1 つ、HYBRID_STRUCT_OF_ARRAY_BD に 1 つ、2 つのアドレスストリームが別々に生成され、参照されます。また、2 番目の代替手法によって、不要なデータのフェッチを防げます。これは、(1) 変数 A、C、E が常に一緒に使用される場合であり、(2) 変数 B と D も常に同時に使用されるが、A、C、E と同時には使用されないことを前提としています。

ハイブリッド手法には、次のような利点があります。

- STRUCT_OF_ARRAY より、アドレス生成が簡単になり、回数も少なくなります。
- ストリーム数が少ないため、DRAM ページミスが減少する。
- ストリーム数が少ないため、プリフェッチの回数が少なくなる。
- 同時に使用されるデータ要素が、キャッシュラインに効率良く格納される。

アセンブリー/コンパイラー・コーディング規則 47 (影響 H、一般性 M): 連続してアクセスできるように、データ構造を配置します。

データが一連のストリームとして構成される場合、自動ハードウェア・プリフェッチャーは、アプリケーションが以降に必要とするデータをプリフェッチでき、実効メモリー・レイテンシーを軽減できます。しかし、データが非連続にアクセスされる場合、自動ハードウェア・プリフェッチはデータをプリフェッチできません。プリフェッチは、最大 8 つまでの同時ストリームを認識できます。ハードウェア・プリフェッチの詳細は、第 9 章「キャッシュ利用の最適化」を参照してください。

ユーザー/ソース・コーディング規則 7 (影響 M、一般性 L): キャッシュライン (64 バイト) 内のフォルス・シェアリングに注意します。

3.6.6 スタックのアライメント

スタックのアライメントの問題を回避する最も簡単な方法は、常にスタックのアライメントが合うようにすることです。これは、空間的に連続したアライメントされていない 8 回の quadword アクセスのうち 1 つは、4 つの連続したアライメントされていない double quadword アクセスのうちの 1 つと同様に常にペナルティーを被ることを意味します。

スタックのアライメントは、システムのデフォルト・スタック・アライメントを超えるデータ・オブジェクトには常に有効です。例えば、32/64 ビット Linux* と 64 ビット Windows* 上では、デフォルトのスタック・アライメントは 16 バイトです (32 ビット Windows* では 4 バイト)。

アセンブリー/コンパイラー・コーディング規則 48 (影響 H、一般性 M): 64 ビット・データがスタック上でパラメーターとして渡されるか、割り当てられる可能性がある場合は、常にスタックのアライメントを 8 バイト境界に合わせます。

これを行うには、汎用レジスター (EBP など) をフレームポインターとして使用する必要があります。(スタックがアライメントされていない場合) アライメントされていない 64 ビット参照が発生したり、(スタックがアライメントされている場合) 余分な汎用レジスターのスピルが発生するトレードオフが生じます。

動的なスタックのアライメントを実装するアセンブリー・レベルの手法は、コンパイラーや特定の OS 環境に依存する可能性があります。興味のある方は、コンパイラーが出力するアセンブリーを参照してみると良いでしょう。

例 3-46 動的なスタック·アライメントの例

```
// 32 ビット環境
   push ebp                      ; ebp を退避
   mov ebp, esp                  ; ebp は入力パラメーターを指す
   andl esp, $-<N>               ; esp を N バイト境界に整列
   sub esp, $<stack_size>        ; 新しいスタックフレーム向けに空間を確保
   .                             ; パラメーターは ebp を介して参照します
   mov esp, ebp                  ; esp をリストア
   pop ebp                       ;  ebp をリストア

// 64 ビット環境
    sub esp, $<stack_size +N>
    mov r13, $<offset_of_aligned_section_in_stack>
    andl r13, $-<N>       ; r13 はスタック中のアライメントされたセクションを指す
    .                     ; アライメントされたデータのベースとして r13 を使用
```

何らかの理由で、スタックのアライメントを 64 ビットに合わせられない場合は、ルーチンがパラメーターにアクセスして、レジスターか、またはアライメントされていることが判明しているメモリーに保存する必要があります。これによって、ペナルティーの発生は一度で済みます。

3.6.7 キャッシュの容量制限とエイリアシング

一定の間隔で複数のアドレスが、メモリー階層内のリソースを求めて競合する場合があります。

キャッシュは通常、複数のウェイのセット·アソシアティビティーを持つように実装されており、各ウェイは複数のセットのキャッシュライン (場合によってはセクター) で構成されています。複数のメモリー参照がキャッシュ内の各ウェイの同じセットを求めて競合すると、容量の問題が発生する可能性があります。また、特定のマイクロアーキテクチャーに適用されるエイリアシング条件があります。1 次キャッシュラインは 64 バイトであることに留意してください。エイリアスを比較する場合、最下位 6 ビットは考慮しません。

3.6.8 コードとデータの混在

インテル® プロセッサーが命令を積極的にプリフェッチし、プリデコーディングすると、以下の 2 つの影響があります。

- 自己修正コードは、インテル® アーキテクチャー·ベースのプロセッサーの必要条件に従って正常に動作しますが、パフォーマンスは大きく低下します。自己修正コードは極力使用しないようにします。
- コードセグメント内に配置された書き込み可能データは、自己修正コードと区別できない可能性があります。したがって、コードセグメント内に書き込み可能データを置くと、自己修正コードと同じペナルティーが生じることがあります。

アセンブリー/コンパイラー·コーディング規則 50 (影響 M、一般性 L): コードと同じページ上に (読み出し専用の) データを置かざるをえない場合は、間接ジャンプの直後にそのデータを置かないようにします。例えば、間接ジャンプの直後にはそのジャンプのターゲットになる可能性の高いコードを置き、無条件分岐の後にデータを配置します。

チューニングの推奨事項 1: まれに、コードページ上のデータが命令として実行され、パフォーマンスが低下することがあります。この問題が起こりやすいのは、トレースキャッシュ内に存在しない間接分岐の直後に実行処理が行われる場合です。パフォーマンスの低下の原因が明らかにこの問題である場合は、どこかほかの位置にデータを移動するか、間接分岐の直後に無効オペコード命令 (UD2) または PAUSE 命令を挿入します。状況によっては、無効オペコードや PAUSE 命令を挿入するとパフォーマンスが低下するので、注意が必要です。

アセンブリー/コンパイラー·コーディング規則 51 (影響 H、一般性 L): コードとデータは常に別のページに配置します。自己修正コードは極力使用しないようにします。コードが修正される場合は、すべて一度に修正するようにして、

修正を実行する側のコードと修正される側のコードを別の 4KB ページに置くか、別々のアライメントされた 1KB サブページに置きます。

3.6.8.1 自己修正コード

インテル® Pentium® III プロセッサーおよびそれ以前のプロセッサーで正常に動作する自己修正コード (SMC) は、それ以降のすべてのプロセッサーでも正常に動作します。高い性能が必要な場合は、SMC および相互修正コード (マルチプロセッサー・システム内の複数のプロセッサーがコードページに書き込みを行う場合) は、使用すべきではありません。

ソフトウェアは、実行されたコードと同じ 1KB サブページ内で、コードページへの書き込みを行うべきではありません。あるいは、書き込み中の内容の同じ 2KB サブページ内で、コードのフェッチを行ってはなりません。また、直接または投機的に実行されるコードを含むページを、データページとして他のプロセッサーと共有すると、SMC 条件の引き金となります。これは自己修正コード条件が原因となります。

データページにコードが書き込まれ後に、そのページがコードとしてアクセスされる場合、動的コードは SMC 条件とはなりません。動的修正コード (例えば、ターゲット・フィックスアップからの) は、SMC 条件の影響を受けやすいので可能な限り避けるべきです。間接分岐によりレジスター間接呼び出しを利用してデータページ (コードページではなく) 上のデータテーブルを使用するなどして、SMC 条件を避ける必要があります。

3.6.8.2 位置に依存しないコード

位置に依存しないコードは多くの場合、命令ポインターの値を取得する必要があります。例 3-48 (a) には、一致する RET がない CALL を発行して IP の値を ECX レジスターに格納する手法を示しています。例 3-48 (b) には、一致する CALL/RET ペアを使用して IP の値を ECX レジスターに格納する代替手法を示します。

例 3-48 命令ポインターの照会手法

```
a) IP を得る return なしの call は RSB に影響しない
   call _label          ; プッシュされた return アドレスは次の反復の IP
_label:
   pop ECX              ; この命令の IP は、ECX に代入されます
b) call/ret がマッチしたペアを使用
   call _lblcx          ;
   ...                  ; ECX はこの命令の IP を含む
   ...
_lblcx
   mov ecx, [esp]       ;
   ret
```

3.6.9 ライト・コンバイニング

ライト・コンバイニング (WC) は、次の 2 つの方法でパフォーマンスを向上させます。

- 1 次キャッシュへの書き込みミスの際に、キャッシュ/メモリー階層内の下位の階層からの所有権読み出し (RFO) が発生する前に、そのキャッシュラインに対する複数のストアを実行できます。その後、ラインの残りの部分が読み込まれ、書き込まれなかったバイトは、未修正のバイトと組み合わせてラインに返されます。
- ライト・コンバイニングは、複数の書き込みをまとめてキャッシュ階層内の下位の階層にユニットとして書き込むことができます。これは、ポートとバス・トラフィックの節約になります。トラフィックの節約は、キャッシュ不可メモリーへのパーシャル書き込みを避けるため特に重要です。

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーの各コアには、8 つのライト・コンバイニング・バッファーがあります。Nehalem† マイクロアーキテクチャー以降のプロセッサーには、10 個のライト・コンバイニング・バッファーがあります。

アセンブリー/コンパイラー・コーディング規則 52 (影響 H、一般性 L): 内部ループによって 5 つ以上の配列 (4 つの異なるキャッシュライン) にデータが書き込まれる場合、ループの本体を分割しそれぞれのループ反復で書き込まれる配列を 4 つ以内に抑えるようにします。

ライト・コンバイニング・バッファーは、すべてのメモリータイプのストアに適用されます。ライト・コンバイニング・バッファーは、キャッシュ不可メモリーへの書き込みには特に重要です。これらのバッファーは、同じキャッシュラインの異なる部分への書き込みを、複数のパーシャル書き込みとしてバス転送するのではなく、キャッシュライン全体を 1 つのバス・トランザクションにまとめることができます (キャッシュ不可メモリーへの書き込みはキャッシュされないため、バス転送される)。グラフィックス・バッファーなどキャッシュ不可メモリーである場合、パーシャル書き込みを避けると、バス帯域幅の制限を受けるグラフィックス・アプリケーションのパフォーマンスが大きく向上します。キャッシュ不可メモリーへの書き込みとライトバック・メモリーへの書き込みを別のフェーズに分けることで、ライト・コンバイニング・バッファーが一杯になるまで、他の書き込みトラフィックによって排出されないことが保証されます。いくつかのアプリケーションでは、パーシャル書き込みトランザクションが排除されると、パフォーマンスが約 20% 向上することが判明しています。キャッシュラインは 64 バイトであるため、63 バイトのバスへの書き込みにより、パーシャル・バス・トランザクションが発生します。

2 つのスレッドで同時に実行する関数をコーディングする場合、内部ループ内の書き込み回数を減らすことで、ライト・コンバイニング・ストアバッファーを最大限に活用できます。ライト・コンバイニング・バッファーの推奨事項とインテル® ハイパースレッディング・テクノロジーについては、第 11 章「マルチコアとインテル® ハイパースレッディング・テクノロジー」を参照してください。

ストア操作の順序と可視性も、ライト・コンバイニングの重要な問題です。まだ書き込まれていないキャッシュラインに対して、ライト・コンバイニング・バッファーへの書き込みが起こると、所有権読み出し (RFO) が発生します。他のライト・コンバイニング・バッファーに対して次の書き込みが起こると、そのキャッシュラインに対して別の RFO が発生します。2 番目のキャッシュラインの RFO が処理され、書き込みが正しい順序で参照可能なことが保証されるまで、最初のキャッシュラインおよびライト・コンバイニング・バッファーへの後続の書き込みは遅延されます。書き込みのメモリータイプがライト・コンバイニングである場合、ラインがキャッシュされないため、RFO は発生せず、このような遅延は生じません。ライト・コンバイニングの詳細については、第 9 章「キャッシュ利用の最適化」も参照してください。

3.6.10 局所性の改善

局所性を改善すると、キャッシュ/メモリー階層内の外側レベルのサブシステムで発生するデータ・トラフィックを削減できます。これは、サイクル数の観点から、アクセスコストが内側レベルよりも外側レベルの方が高くなるという事実に対処することが目的です。特定のキャッシュレベル (またはメモリーシステム) にアクセスするためのサイクルコストは通常、マイクロアーキテクチャー、プロセッサー、プラットフォーム・コンポーネントによって異なります。局所性やプロセッサー/プラットフォームごとに記載されたサイクルコストの数値表に従うのではなく、局所性に基づいて相対的なデータ・アクセス・コストの傾向を把握するだけで十分な場合があります。一般的な傾向としては、データアクセスの並列性の度合いが同等な場合、外側レベルのサブシステムのアクセスコストは、キャッシュ/メモリー階層内の最も内側のレベルのデータアクセスよりもおよそ 3 ~ 10 倍高くなります。

このように、局所性の改善は、主要なデータ・トラフィックの局所性を考慮した特性評価から始めます。付録 A「アプリケーション・パフォーマンス・ツール」では、ワークロードの主要なデータ・トラフィックの局所性を判別する手法について説明しています。

最終レベルキャッシュのキャッシュミス率がキャッシュ参照の数に比べて低い場合でも、プロセッサーはキャッシュミスの処理の待機に多くの時間を費やしています。プログラムの局所性を高めてキャッシュミスを減らすのは、重要な最適化手法です。これにはいくつかの手法があります。

- ブロッキングによってキャッシュに納まる配列の一部を反復します (データブロック［またはタイル］に対する以降の参照をキャッシュヒット参照にすることが目的)。
- ループ交換によってキャッシュラインやページ境界を越えることを避けます。
- ループ傾斜によってアクセスを隣接させます。

最終レベルキャッシュの局所性の改善は、ハードウェア・プリフェッチを利用できるようにデータ・アクセス・パターンの順序付けを行うことで達成できます。これにもいくつかの手法があります。

- 配置がまばらな多次元配列を 1 次元配列に変換して、間隔の狭い、ハードウェア・プリフェッチに適したシーケンシャルなパターンでメモリー参照を行います (20.2.8.9E.2.5.4 節「データ・プリフェッチ」を参照)。
- 最適なタイルサイズおよびパターンを選択すると、最終レベルキャッシュへのヒット率を高めることでテンポラルなデータ局所性がさらに改善され、ハードウェア・プリフェッチのメモリー・トラフィックを減らすことができます (9.5.11 節「ハードウェア・プリフェッチとキャッシュ・ブロッキング」を参照)。

局所性を改善する手法に反する操作を回避することも重要です。lock プリフィクスを過度に使用すると、データがキャッシュまたはシステムメモリーのどちらにあるのかにかかわらず、メモリーアクセスの遅延が大きくなることがあります。

ユーザー/ソース・コーディング規則 10 (影響 H、一般性 H): ブロッキング、ループ交換、ループ傾斜、パッキングなどの最適化手法はコンパイラーに任せる方が良いでしょう。コンパイラーのループ最適化を有効にして、1 次キャッシュの半分または 2 次キャッシュ全体に収まるように、データ構造を最適化し、ネストされたループの局所性を向上させます。

1 次キャッシュの半分のサイズに合わせて最適化すると、データアクセスごとのサイクルコストの点でパフォーマンスが大きく向上します。1 次キャッシュの半分で小さすぎる場合、2 次キャッシュに合わせて最適化します。両者の中間のサイズ (例えば、1 次キャッシュ全体) に合わせて最適化しても、パフォーマンス上のメリットは、2 次キャッシュに合わせて最適化した場合とほとんど変わりません。

3.6.11 非テンポラルなストア・バス・トラフィック

システムバスの最大帯域幅は、(メモリーからの) 読み込み、(キャッシュラインの) 所有権読み出し、書き込みなど、いくつかのタイプのバス・アクティビティーによって共有されます。一度に 64 バイトをバスに書き出す場合、バス書き込みトランザクションのデータ転送速度の方が高速です。

一般に、ライトバック (WB) メモリーへのバス書き込みでは、システムバスの帯域幅を所有権読み出し (RFO) トラフィックと共有する必要があります。非テンポラルなストアは RFO トラフィックを要求しませんが、一度に 64 バイトが排出されるようにするために (いくつかのチャンクではなく)、アクセスパターンを考慮する必要があります。

非テンポラルなストアが原因のフルサイズの 64 バイト・バス書き込みのデータ帯域幅は、WB メモリーへのバス書き込みの 2 倍のデータ帯域幅を持ちますが、いくつかのチャンクを転送するとバス要求帯域幅が浪費され、データ帯域幅が大幅に低下します。例 3-48 と例 3-49 にこの違いを示します。

例 3-48 非テンポラルなストアと 64 バイトのバス書き込みトランザクションの使用

```
#define STRIDESIZE 256
lea ecx, p64byte_Aligned
mov edx, ARRAY_LEN
xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0
movntps XMMWORD ptr [ecx + eax+48], xmm0
; 64 バイトは 1 つのバス・トランザクションに書かれる
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

例 3-49 非テンポラルなストアとパーシャルバス書き込みトランザクション

```
#define STRIDESIZE 256
Lea ecx, p64byte_Aligned
Mov edx, ARRAY_LEN
Xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0
; 48 バイトの格納は結果としていくつかのバスにパーシャル・トランザクションとなる
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

3.7 プリフェッチ

最近のインテル® プロセッサー・ファミリーでは、以下の複数のプリフェッチ機構を利用してデータ/コード移動の高速化とパフォーマンスの向上を図っています。

- ハードウェア命令プリフェッチ
- データのソフトウェア・プリフェッチ
- データまたは命令のキャッシュラインのハードウェア・プリフェッチ

3.7.1 ハードウェア命令フェッチとソフトウェア・プリフェッチ

ソフトウェア・プリフェッチを行うには、プログラマーが PREFETCH ヒント命令を使用して、キャッシュミスの適切なタイミングと場所を予想する必要があります。

ソフトウェア PREFETCH 操作は、メモリー操作からのロードと同様に機能しますが、以下の例外があります。

- ソフトウェア PREFETCH 命令は、仮想アドレスから物理アドレスへの変換が完了した後にリタイアします。
- ページフォールトなどの例外がデータをプリフェッチする必要がある場合、ソフトウェア・プリフェッチ命令はデータをプリフェッチせずにリタイアします。
- ソフトウェア・プリフェッチに NULL アドレスを使用するのを避けます。

3.7.2 1 次データキャッシュのハードウェア・プリフェッチ

インテル® Core™ マイクロアーキテクチャーの 1 次キャッシュのハードウェア・プリフェッチ機構については、E.3.4.2 節で説明しています。例 3-50 に、ハードウェア・プリフェッチをトリガーする手法を示します。このコードでは、リンクされたリストを横断して、2 つの異なるキャッシュラインに存在するそれぞれの要素の 2 つのメンバーに対して計算処理を行っています。各要素のサイズは 192 バイトです。すべての要素の合計サイズは、2 次キャッシュのサイズを上回ります。

例 3-50 DCU ハードウェア・プリフェッチを使用

オリジナルのコード	プリフェッチのメリットを得られる変更済みシーケンス

mov ebx, DWORD PTR [First]
xor eax, eax
scan_list:
mov eax, [ebx+4]
mov ecx, 60
 	
do_some_work_1:
add eax, eax
and eax, 6
sub ecx, 1
jnz do_some_work_1
mov eax, [ebx+64]
mov ecx, 30
do_some_work_2:
add eax, eax
and eax, 6
sub ecx, 1
jnz do_some_work_2
 	
mov ebx, [ebx]
test ebx, ebx
jnz scan_list
	mov ebx, DWORD PTR [First]
	xor eax, eax
	scan_list:
	mov eax, [ebx+4]
	mov eax, [ebx+4]
	mov eax, [ebx+4]
	mov ecx, 60
	do_some_work_1:
	add eax, eax
	and eax, 6
	sub ecx, 1
	jnz do_some_work_1
	mov eax, [ebx+64]
	mov ecx, 30
	do_some_work_2:
	add eax, eax
	and eax, 6
	sub ecx, 1
	jnz do_some_work_2
	
	mov ebx, [ebx]
	test ebx, ebx
	jnz scan_list

「変更済みシーケンス」では、一方のメンバーからデータをロードする命令を追加することによって、命令ポインターのスキップによる DCU ハードウェア・プリフェッチがトリガーされ、次のキャッシュライン中のデータがプリフェッチされます。これによって、2 番目のメンバーの処理を迅速に完了できます。

ソフトウェアは、以下の場合に 1 次データ・キャッシュ・プリフェッチからメリットを得られます。

- データが 2 次キャッシュに存在しない場合、1 次データ・キャッシュ・プリフェッチは 2 次キャッシュ・プリフェッチを早い段階でトリガーできます。
- データが 2 次キャッシュに存在し、1 次データキャッシュには存在しない場合、1 次データ・キャッシュ・プリフェッチは 1 次データキャッシュに対するシーケンシャル・キャッシュラインのデータの供給を早い段階でトリガーできます。

DCU ハードウェア・プリフェッチが不用意にトリガーされる可能性について、注意しなければいけない状況があります。多数のメンバーが多くのキャッシュラインをまたぐ大規模なデータ構造体で、メンバーの少数のみが参照され、同じキャッシュラインへのペアアクセスが複数発生する場合、DCU ハードウェア・プリフェッチャーによって不要なキャッシュラインのフェッチがトリガーされる可能性があります。例では、「Pts」配列と「AltPts」への参照によって DCU プリフェッチャーがトリガーされ、不要なキャッシュラインがフェッチされます。コードの一部分で DCU ハードウェア・

プリフェッチによるパフォーマンスへの影響が検出された場合、ソフトウェアは、このワーキングセットのサイズを削減して 2 次キャッシュの半分未満になるように試みることができます。

例 3-51 DCU ハードウェア・プリフェッチが原因の不要なラインのフェッチの回避

```
while ( CurrBond != NULL )
   {
   MyATOM *a1 = CurrBond->At1 ;
   MyATOM *a2 = CurrBond->At2 ;

   if ( a1->CurrStep <= a1->LastStep &&
       a2->CurrStep <= a2->LastStep
       )
           {
           a1->CurrStep++ ;
           a2->CurrStep++ ;

           double ux = a1->Pts[0].x - a2->Pts[0].x ;
           double uy = a1->Pts[0].y - a2->Pts[0].y ;
           double uz = a1->Pts[0].z - a2->Pts[0].z ;
           a1->AuxPts[0].x += ux ;
           a1->AuxPts[0].y += uy ;
           a1->AuxPts[0].z += uz ;

           a2->AuxPts[0].x += ux ;
           a2->AuxPts[0].y += uy ;
           a2->AuxPts[0].z += uz ;
           } ;
   CurrBond = CurrBond->Next ;
   } ;
```

このようなプリフェッチの利点を十分に活用するには、以下の手法によってデータの構成とアクセスを行います。

手法 1:

- 連続したアクセスが同じ 4KB ページ内で行われるようにデータを構成します。
- 前方または後方に一定の間隔の IP プリフェッチでデータにアクセスします。

手法 2:

- 連続したラインにデータを構成します。
- 昇順アドレスでシーケンシャル・キャッシュラインのデータにアクセスします。

例では、1 次キャッシュ・プリフェッチのメリットを得られるシーケンシャル・キャッシュラインへのアクセスを示しています。

例 3-52 1 次ハードウェア・プリフェッチを使用する手法

```
unsigned int *p1, j, a, b;
for (j = 0; j < num; j += 16)
{
   a = p1[j];
   b = p1[j+1];
   // これらの値を使用
}
```

メモリーからのロード操作を各反復の先頭に移すことによって、メモリーから 2 次キャッシュへのペア・キャッシュラインを転送するレイテンシーの大部分は、最初のキャッシュラインの転送と並行して発生する可能性が高くなります。

IP プリフェッチでは、アドレスの下位 8 ビットのみ使用してアドレスを識別します。ループのコードサイズが 256 バイトを超える場合、2 つのロードの最下位 8 ビットが同じに見えることがあるためで、IP プリフェッチが制限されます。そのため、256 バイトを超えるループで、IP プリフェッチを使用する場合、2 つのロードの最下位 8 ビットが同じでないことを確認します。

3.7.3 2 次キャッシュのハードウェア・プリフェッチ

インテル® Core™ マイクロアーキテクチャーには、2 つの 2 次キャッシュ・プリフェッチャーが用意されています。

- **ストリーマー** — データまたは命令をメモリーから 2 次キャッシュにロードします。ストリーマーを使用するには、128 バイトにアライメントされた 128 バイトのブロック単位でデータまたは命令を構成します。このブロックがメモリー内にあるときにブロック内の 2 つのキャッシュラインのいずれかに初めてアクセスすると、ストリーマーがトリガーされて、ペアラインがプリフェッチされます。ソフトウェアにとって 2 次キャッシュ・ストリーマーの機能は、Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサーの隣接キャッシュライン・プリフェッチ機構に似ています。
- **データ・プリフェッチ・ロジック (DPL)** — DPL と 2 次キャッシュ・ストリーマーは、ライトバック・メモリー・タイプでのみトリガーされます。これらはページ境界 (4KB) 内に対してのみプリフェッチを行います。いずれの 2 次キャッシュ・プリフェッチも、ソフトウェア・プリフェッチ命令と、DCU プリフェッチからのプリフェッチ要求によってトリガーされます。DPL は、所有権読み出し (RFO) 操作によってもトリガーされます。2 次キャッシュ・ストリーマーは、2 次キャッシュミスに対する DPL 要求によってもトリガーされます。

ソフトウェアでは、命令ポインターとライン間隔の両方に従ってデータを構成するとメリットが得られます。例えば行列演算の場合、列は IP ベース・プリフェッチによって、行は DPL と 2 次キャッシュ・ストリーマーによってプリフェッチできます。

3.7.4 キャッシュ制御命令

インテル® SSE2 では、インテル® SSE のキャッシュ制御命令を拡張する新しいキャッシュ制御命令をサポートしています。新しいキャッシュ制御命令には、以下のものがあります。

- 新しいストリーミング・ストア命令
- 新しいキャッシュライン・フラッシュ命令
- 新しいメモリーフェンス命令

詳細は、第 9 章「キャッシュ利用の最適化」を参照してください。

3.7.5 REP プリフィクスとデータ移動

REP プリフィクスは通常、MEMCPY (REP MOVSD を使用) や MEMSET (REP STOS を使用) などメモリー関連のライブラリー関数向けに、文字列移動命令とともに使用されます。REP プリフィクスに対応した STRING/MOV 命令は MS-ROM (マイクロコード・シーケンサー ROM) に実装され、パフォーマンスの異なる形式が複数存在します。

どの形式が使用されるかは、データレイアウト、アライメント、カウンター (ECX) の値に基づいて実行時に選択されます。例えば、最高のパフォーマンスを得るには、カウンター値が 3 以下の状態で REP プリフィクスとともに MOVSB/STOSB を使用する必要があります。

文字列 MOVE/STORE 命令には、複数のデータ型があります。効率良くデータを移動するには、データ型が大きい方が適しています。つまり、任意のカウンター値を複数のダブルワードとカウンター値が 3 以下の 1 バイト移動に分解することによって、効率化を達成できます。

ソフトウェアでは、SIMD データ移動命令を利用すると、一度に 16 バイトを移動できます。以下のセクションに MEMCPY()、MEMSET()、MEMMOVE() などの高性能ライブラリー関数を設計および実装する際の一般的なガイドラインを示します。検討すべき要素として以下の 4 つがあります。

- **反復ごとのスループット** — 2 組のコードのパス長がほぼ同じ場合、効率の面では、反復ごとに大量のデータを移動する命令の方が適しています。また、反復ごとのコードサイズが小さいと、一般的にはオーバーヘッドが減少し、スループットが向上します。ループ反復構造と REP プリフィクスを使用した反復との、相対的なオーバーヘッドの比較が必要になる場合があります。
- **アドレス・アライメント** — スループットが極めて高いデータ移動命令には、アライメントの制限があります。つまり、デスティネーション・アドレスが自然なデータサイズにアライメントされている方が効率的に処理できます。具体的には、16 バイトの移動では、デスティネーション・アドレスを 16 バイト境界にアライメントする必要があります。また、8 バイトの移動では、デスティネーション・アドレスを 8 バイト境界にアライメントした方が高いパフォーマンスを得られます。8 バイト境界にアライメントされたアドレスでは多くの場合、ダブルワード単位で移動すると、パフォーマンスが向上します。
- **REP 文字列移動と SIMD 移動** — SIMD 拡張命令を使用して汎用メモリー関数を実装する場合、通常 SIMD 命令を確実に利用することを可能にするプロローグコードや、実行時にアライメント済みデータ移動の条件を満たすためのプリアンブル・コードを追加する必要があります。REP 文字列と SIMD 手法について検討する際、スループットの比較では、プロローグのオーバーヘッドも考慮しなければいけません。
- **キャッシュの排出** — メモリールーチンによって処理されるデータの量が最終レベルキャッシュのサイズの半分に近づくと、キャッシュのテンポラルな局所性が影響を受けることがあります。ストリーミング・ストア命令 (MOVNTQ や MOVNTDQ など) を使用すれば、キャッシュをフラッシュする影響を最小限に抑えられます。ストリーミング・ストアを使用するしきい値は、最終レベルキャッシュのサイズに依存します。サイズの判断には、CPUID のキャッシュ・パラメーター・リーフを使用します。ストリーミング・ストアを使用して MEMSET() 形式のライブラリーを実装する場合、ターゲットアドレスをすぐに参照しない場合にのみ、アプリケーションがこの手法のメリットを得られる点も考慮する必要があります。この想定は、マイクロベンチマーク構成でストリーミング・ストアの実装をテストした場合には容易に確認できますが、フルスケールのアプリケーションには当てはまりません。

一般的なヒューリスティックを高性能の汎用ライブラリー・ルーチンの設計に適用するのであれば、カウンター値 N の任意のサイズとアドレス・アライメントを最適化する際に、以下のガイドラインが有用です。最適なパフォーマンスを得るには、N の大きさに応じて異なる手法があります。

- N が何らかのスモールカウントよりも小さい場合 (スモールカウントのしきい値は、マイクロアーキテクチャー間で異なります。経験則上、Intel NetBurst® マイクロアーキテクチャー向けに最適化する場合には 8 が適切な値です)、いずれの場合もループ構造のオーバーヘッドなしに直接コーディングできます。例えば、2 つの MOVSD 命令を明示的に使用して、REP カウンターが 3 に等しい MOVSB 命令を使用すれば、11 バイトを処理できます。
- N がそれほど小さくないが、何らかのしきい値よりも小さい場合 (このしきい値はマイクロアーキテクチャーによって異なりますが、経験則上で判断できます)、ランタイム CPUID とアライメント・プロローグを使用して SIMD を実装すると、プロローグのオーバーヘッドによってスループットが低下しやすくなります。REP 文字列を実装する場合は、ダブルワードの REP 文字列の使用を推奨します。アドレス・アライメントを改善するために、MOVSD/STOSD 命令の前に、カウントが 4 未満の MOVSB/STOSB 命令を使用するプロローグコードを用いて、アライメントされていないデータ移動を取り出すことができます。
- N が最終レベルキャッシュのサイズの半分よりも小さい場合、スループットに対する考慮事項としては次のいずれも適切です。
  - 最大のデータ単位で REP 文字列を使用する手法。REP 文字列では、ループの反復のオーバーヘッドがほとんどなく、アドレス・アライメントを処理するプロローグ/エピローグコードでの分岐予測ミスのオーバーヘッドは、多くの反復を実行する中で吸収されます。
  - 最大のデータ単位で命令を使用する反復手法。この手法では、SIMD 機能検出のオーバーヘッド、反復のオーバーヘッド、アライメント制御のためのプロローグ/エピローグを最小限に抑えられます。いずれの手法を選ぶべきであるかは、マイクロアーキテクチャーに依存します。

  32 ビット・モードでデスティネーション・アドレスがダブルワード境界にアライメントされている任意のカウンター値に STOSD 命令を使用して実装された MEMSET() の例を、例 3-54 に示します。

- N が最終レベルキャッシュのサイズの半分よりも大きい場合、デスティネーション・アドレスが直後に参照されないのであれば、アドレス・アライメント向けのプロローグ/ エピローグとともに 16 バイト単位のストリーミング・ストアを使用すると、効率化を期待できます。

例 3-53 任意のカウントサイズおよび 4 バイトにアライメントされたデスティネーションでの REP STOSD

C の MEMSET() の例	REP STOSD 使用した同等の実装

void memset(void *dst,int c,size_t
size)
{
char *d = (char *)dst;
size_t i;
for (i=0;i<size;i++)
*d++ = (char)c;
}
	push edi
	movzx eax, byte ptr [esp+12]
	mov ecx, eax
	shl ecx, 8
	or ecx, eax
	mov ecx, eax
	shl ecx, 16
	or eax, ecx
	mov edi, [esp+8] ; 4 バイト・アライメント
	mov ecx, [esp+16] ; バイト・アライメント
	shr ecx, 2 ; dword にする
	cmp ecx, 127
	jle _main
	test edi, 4
	jz _main
	stosd ; 1 つの DWORD を剥離
	dec ecx
	_main: ; 8 バイトでアライメント
	rep stosd
	mov ecx, [esp + 16]
	and ecx, 3 ; count 3 以下で行う
	rep stosb ; 3 以下で最適
	pop edi
	ret

インテル® コンパイラーが提供するランタイム・ライブラリーのメモリールーチンは、広範なアドレス・アライメント、カウンター値、マイクロアーキテクチャーに対して最適化されています。ほとんどの場合、アプリケーションは、インテル® コンパイラーで提供されるデフォルトのメモリールーチンを利用すべきです。

状況によっては、データのバイトカウントは、呼び出しで渡されたパラメーターによって認識されるのではなく、コンテキストによって認識されます。汎用ライブラリー・ルーチンに必要な手法と比べて、シンプルな手法を採用できます。例えば、バイトカウントが少ない場合、カウントが 4 未満の REP MOVSB/STOSB 命令を使用すれば、適切なアドレス・アライメントとループのアンロールによって残りのデータを完了することができます。MOVSD/STOSD 命令を使用する場合は、反復に関連したオーバーヘッドを削減できます。

REP プリフィクスを文字列移動命令とともに使用すると、上記の状況で高度なパフォーマンスを得られます。ただし、REP プリフィクスを文字列スキャン命令 (SCASB、SCASW、SCASD、SCASQ) または比較命令 (CMPSB、CMPSW、SMPSD、SMPSQ) とともに使用するのは、高いパフォーマンスを得る上では推奨されません。代わりに、SIMD 命令の使用を検討してください。

3.7.6 拡張 REP MOVSB と STOSB 操作

Ivy Bridge† マイクロアーキテクチャーから、MOVSB と STOSB を使用する REP 文字列操作は、メモリーコピーやセットのような一般的な操作を行うソフトウェアで柔軟性と高いパフォーマンスを提供します。拡張 MOVSB/STOSB 操作をサポートするプロセッサーは、次のように CPUID 機能フラグで検出できます。
CPUID:(EAX=7H, ECX=0H):EBX.[bit 9] = 1

3.7.6.1 高速な短い REP MOVSB

Ice Lake† Client マイクロアーキテクチャーからは、短い操作を行う REP MOVSB のパフォーマンスが向上しています。この機能強化は、1 から 128 バイトの文字列長に対して適用されます。高速な短い REP MOVSB のサポートは、CPUID 機能フラグ、CPUID [EAX=7H, ECX=0H).EDX.FAST_SHORT_REP_MOVSB[bit 4] = 1 で確認できます。これは REP STOS のパフォーマンスには影響しません。

3.7.6.2 Memcpy に関する考察

標準ライブラリー関数の memcpy のインターフェイスには、ライブラリー関数の実装におけるパフォーマンス特性を決定するマイクロアーキテクチャーに関連するいくつかの要因があります (レングス、ソースバッファーとデスティネーションのアライメントなど)。memcpy を実装する 2 つの一般的なアプローチでは、小さなコードサイズとスループットの最大化が追及されます。一般に前者は REP MOVSD + B (3.7.5 節を参照) を使用して、また後者は SIMD 命令セットを使用して、データ・アライメントの制限に対処しなければなりません。

拡張 REP MOVSB/STOSB をサポートするプロセッサーでは、REP MOVSB による memcpy は、コードサイズが小さくなり、REP MOVSD + B の組み合わせを使用するよりも優れたスループットが得られます。Ivy Bridge† マイクロアーキテクチャーベースのプロセッサーでは、拡張 REP MOVSB と STOSB を使用して実装された memcpy は、レングスとアライメント要件によっては 256 ビット、もしくは 128 ビットのインテル® AVX 代替コードが同じレベルのスループットを達成できない可能性があります。

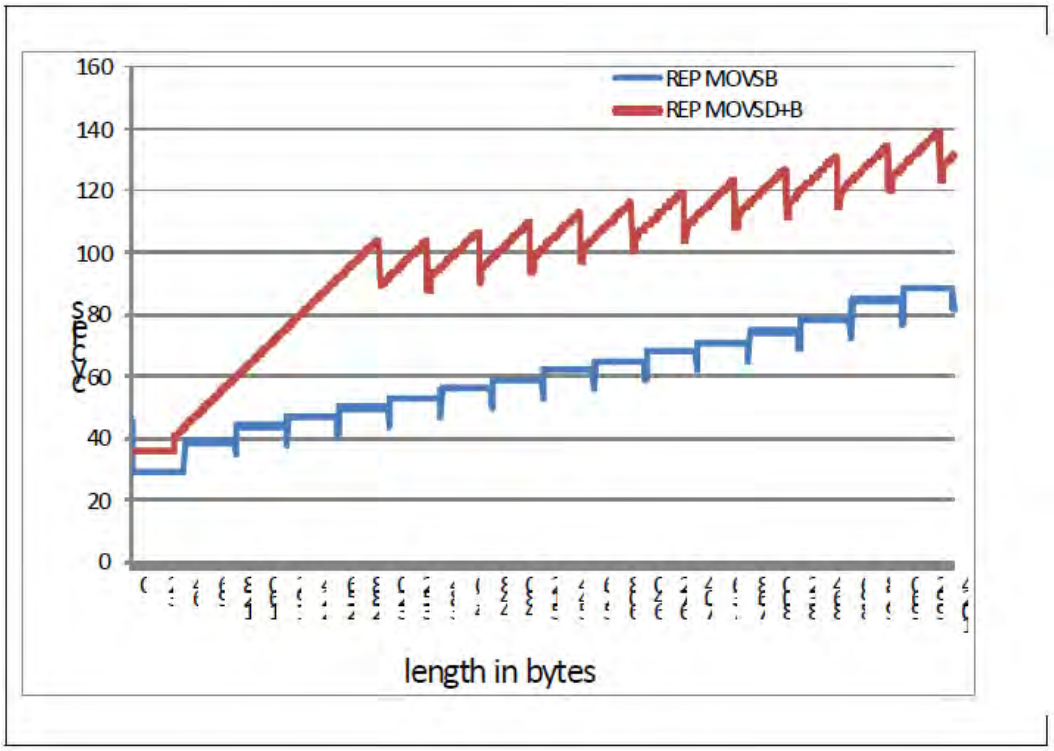


図 3-2 2KB までのレングスによる memcpy のパフォーマンス比較

図 3-2 には、拡張 REP MOVSB/STOSB と REP MOVSD + B を使用した第 3 世代インテル® Core™ プロセッサー上の memcpy 実装の相対パフォーマンスを示しています。ここでは、ソースとデスティネーションの両方が 16 バイト境界にアライメントされ、ソース領域はデスティネーション領域とオーバーラップしません。拡張 REP MOVSB と STOSB を使用すると、REP MOVSD + B を使用するよりも常に高いパフォーマンスが得られます。レングスが 64 の倍数であれば、さらに高いパフォーマンスが期待できます。例えば、65 から 128 バイトのコピーには 40 サイクルかかりますが、128 バイトであれば 35 サイクルで済みます。

アプリケーションがカスタム実装により標準 memcpy ライブラリーの実装をバイパスし、ソースとデスティネーション両方のバッファー長の割り当てを自由に管理しようとする場合、拡張 REP MOVSB と STOSB のコードサイズとパフォーマンスの利点を得るため、64 の倍数になるようにメモリーコピー操作のレングスを操作することは価値があるかもしれません。

SIMD レジスターを使用した汎用 memcpy ライブラリー関数実装のパフォーマンス特性は、汎用レジスターを使用する同等の実装よりも多様性があります。これは、レングスやインテル® SSE2、128 ビットのインテル® AVX、256 ビットのインテル® AVX 命令セットの選択、ソースとデスティネーションの相対的なアライメント、そしてメモリーアドレスのアライメントの粒度/境界などに依存します。

拡張 REP MOVSB/STOSB と SIMD 実装による memcpy のパフォーマンス特性の比較は、特定の SIMD 実装に依存します。このセクションの残りでは、拡張 REP MOVSB/STOSB を使用した memcpy と Ivy Bridge† マイクロアーキテクチャーのハードウェア機能を示す最適化された 128 ビットのインテル® AVX 実装の memcpy との相対的なパフォーマンスについて説明します。

表 3-5 拡張 REP MOVSB/STOSB と128 ビットのインテル® AVX を使用した memcpy() の相対パフォーマンス

レングス範囲 (バイト)	**<128**	**128 から 2048**	**2048 から 4096**
Memcpy_ERMSB/Memcpy_AVX128	0x7X	1X	1.02X

表 3-5 は、複数の memcpy レングス範囲での拡張 REP MOVSB/STOSB と128 ビットのインテル® AVX を使用した memcpy の相対パフォーマンスを示しています。ここでは、ソースとデスティネーションのアドレスは 16 バイトにアライメントされ、両者の領域はオーバーラップしないと仮定します。memcpy のレングスが 128 バイト未満である場合、拡張 REP MOVSB と STOSB を使用すると、REP 文字列内部のオーバーヘッドにより 128 ビットのインテル® AVX を使用可能な場合よりも低速になります。

一般に、アドレス・ミスアライメントが生じると、memcpy のパフォーマンスは 16 バイトにアライメントした場合と比べて相対的に低下します (表 3-6 を参照)。

表 3-6 memcpy() パフォーマンスへのアドレス・ミスアライメントの影響

アライメントされていないアドレス	**パフォーマンスの影響**
ソースバッファー	拡張 REP MOVSB/STOSB 実装と 128 ビットのインテル® AVX の影響を比較した場合も同様です。
デスティネーション・バッファー	128 ビットのインテル® AVX 実装の memcpy は 16 バイトにアライメントされた場合に比べ 5% のみの低下であるのに対し、拡張 REP MOVSB/STOSB 実装では 25% 低下します。

拡張 REP MOVSB/STOSB 実装による memcpy は、Haswell† マイクロアーキテクチャーの 256 ビット SIMD 整数データパスではさらに恩恵を得られます (15.16.3 節参照)。

3.7.6.3 Memmove の考察

ソースとデスティネーション領域間にオーバーラップがある場合、ソフトウェアは正当性を確実にするため memcpy の代わりに memmove を使用する必要があります。ソース領域の後半がデスティネーション領域の先頭とオーバーラップする場合、memmove の実装で方向フラグ (DF) と REP MOVSB を使用することができます。しかし、DF フラグを設定して REP MOVSB に上位から下位アドレスへのバイトコピーを強制すると、劇的にパフォーマンスが低下します。

拡張 REP MOVSB/STOSB を使用して memmove 関数を作成する場合、この状況を識別して最初にソース領域の後半のチャンクを処理します。これにより、REP MOVSB と DF=0 によりデスティネーション領域の一部として書き込みが可能となります (デスティネーション領域とはオーバーラップしません)。オーバーラップする後半のチャンクをコピーした後、残りのソースを DF=0 で通常のように処理できます。

3.7.6.4 Memset の考察

コードサイズとスループットに関する考察は、memset() の実装にも適用できます。拡張 REP MOVSB/STOSB をサポートするプロセッサーでは、REP STOSB を使用してコードサイズが小さく、3.7.5 節で示した STOSD + B 手法よりも高いパフォーマンスを実現できます。

デスティネーション・バッファーが 16 バイトにアライメントされている場合、拡張 REP MOVSB/STOSB を使用する memset() は、SIMD アプローチよりも適切です。Ivy Bridge† マイクロアーキテクチャーでは、デスティネーション・バッファーがアライメントされていない場合、拡張 REP MOVSB/STOSB を使用する memset() のパフォーマンスは、アライメントされている時と比べて 20% パフォーマンスが低下します。対称的に、memset() の SIMD 実装では、デスティネーションがアライメントされていない状況でもパフォーマンスの低下はわずかです。

拡張 REP MOVSB/STOSB 実装による memset() は、Haswell† マイクロアーキテクチャーの 256 ビット・データパスではさらに恩恵が得られます (15.16.3.3 節参照)。

3.8 REP ストリング処理

Golden Cove† マイクロアーキテクチャー・ベースのプロセッサーから、いくつかの REP ストリング処理のパフォーマンスが強化されました。

3.8.1 高速ゼロレングス REP MOVSB

ゼロレングス操作の REP MOVSB 命令のパフォーマンスが向上しました。ゼロレングスの REP MOVSB のレイテンシーが、レングス 1 から 128 バイトのレイテンシーと同じになりました。高速ショート REP MOVSB と高速ゼロレングス REP MOVSB 機能が有効になっている場合、REP MOVSB のパフォーマンスは、ソースとデスティネーション・オペランドがプロセッサーの L1 レベルキャッシュにある 0 から 128 バイトのすべての文字列に対して 9 サイクルで均一です。

高速ゼロレングス REP MOVSB のサポートは、CPUID の機能フラグで確認できます。
CPUID.07H.01H:EAX.FAST_ZERO_LENGTH_REP_MOVSB[bit 10] = 1

3.8.2 高速ショート REP STOSB

短い操作の REP STOSB のパフォーマンスが改善されました。この拡張機能は、0 から 128 バイト長の文字列に適用されます。高速ショート REP STOSB 機能が有効になっている場合、REP STOSB のパフォーマンスは、ソースとデスティネーション・オペランドがプロセッサーの L1 レベルキャッシュにある 0 から 128 バイトのすべての文字列に対して 12 サイクルで均一です。

高速ショート REP STOSB のサポートは、CPUID の機能フラグで確認できます。
CPUID.07H.01H:EAX.FAST_SHORT_REP_STOSB[bit 11] = 1

3.8.3 高速ショート REP CMPSB と SCASB

REP CMPSB と SCASB のパフォーマンスが改善されました。この拡張機能は、1 から 128 バイト長の文字列に適用されます。高速ショート REP CMPSB と SCASB 機能が有効になっている場合、REP CMPSB と REP SCASB のパフォーマンスは、ソースとデスティネーション・オペランドがプロセッサーの L1 レベルキャッシュにある 1 から 128 バイトのすべての文字列に対して 15 サイクルで均一です。

高速ショート REP CMPSB と SCASB のサポートは、CPUID の機能フラグで確認できます。
CPUID.07H.01H:EAX.FAST_SHORT_REP_CMPSB_SCASB[bit 12] = 1

3.9 浮動小数点に関する考察

浮動小数点アプリケーションをプログラミングする場合、C、C++、または Fortran などの高水準プログラミング言語から始めた方が良いでしょう。多くのコンパイラーは、可能であれば、浮動小数点演算のスケジューリングと最適化を自動的に行います。しかし、最適なコードを生成するには、プログラマーがコンパイラーを補助する必要があります。

3.9.1 浮動小数点コードの最適化のガイドライン

ユーザー/ソース・コーディング規則 9 (影響 M、一般性 M): インテル® SSE、インテル® SSE2、インテル® AVX、インテル® AVX2 またはより高度な SIMD 命令セット (インテル® AVX-512 など) 使用したコードを生成するように、コンパイラーの最適化オプションを利用します。x87 コードの代わりにスカラー SIMD コードを生成します。

浮動小数点アプリケーションのパフォーマンスをチューニングするには、以下の手順に従います。

- コンパイラーが浮動小数点コードをどのように扱うかを理解します。
- アセンブリー・リストを調べて、プログラム上行われている変換処理を確認します。
- 実行時間に大きな影響を与えている、アプリケーション内のネストされたループを調査します。
- コンパイラーが高速なコードを生成できなかった理由を判断します。
- 解決できる依存関係がないかどうか確認します。
- 問題のある領域 (バス帯域幅、キャッシュの局所性、トレースキャッシュ帯域幅、または命令のレイテンシー) を特定して、その最適化に焦点を合わせます。例えば、バスがすでに飽和している場合、PREFETCH 命令を追加しても効果がありません。また、トレースキャッシュ帯域幅が問題である場合、プリフェッチ・マイクロオペレーション (uop) を追加すると、パフォーマンスが低下することがあります。

また、通常は、本章で説明する以下の一般的なコーディングの推奨事項に従う必要があります。

- キャッシュをブロック化する
- プリフェッチを使用する
- ベクトル化を有効にする
- ループをアンロールする

ユーザー/ソース・コーディング規則 10 (影響 H、一般性 ML): アプリケーションが浮動小数点値の有効範囲を超えないようにし、デノーマル値、アンダーフローを避けます。

有効範囲外の値があると、非常に大きなオーバーヘッドが生じます。

切り捨てによって、浮動小数点値を 16 ビット、32 ビット、または 64 ビットの整数に変換する場合、x87 スタックをアクセスする命令よりも CVTTSS2SI 命令と CVTTSD2SI 命令の使用が推奨されます。これにより、丸めモードの変更も回避できます。

ユーザー/ソース・コーディング規則 11 (影響 M、一般性 ML): 通常は、数値演算ライブラリーは、初等関数を計算する際に超越関数命令 (例えば、FSIN) を利用します。超越関数を計算するとき、特に 80 ビットの拡張精度を使用する必要がない場合は、ソフトウェアに基づく別の手法 (補間法を使用するルックアップ・テーブル方式のアルゴリズムなど) を検討するべきです。これらの手法を使って、希望する数値精度とルックアップ・テーブルのサイズを選択し、インテル® SSE 命令とインテル® SSE2 命令の並列処理を利用すると、超越関数計算のパフォーマンスを向上できます。

3.9.2 浮動小数点モードと浮動小数点例外

浮動小数点数を操作するとき、高速マイクロプロセッサーは、ハードウェアまたはコードを必要とする例外を、頻繁に処理しなければなりません。

3.9.2.1 浮動小数点例外

パフォーマンス低下を招く主な原因は、次のマスクされた浮動小数点例外条件を使用する場合です。

- 算術オーバーフロー
- 算術アンダーフロー
- デノーマルオペランド

オーバーフロー例外、アンダーフロー例外、デノーマル例外の定義は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 4 章を参照してください。

デノーマル浮動小数点数は、以下の 2 つの場合パフォーマンスに影響します。

- 直接的：オペランドとして使用される場合
- 間接的：アンダーフロー状態の結果として生成される場合

アンダーフローを生成しない浮動小数点アプリケーションでは、デノーマルは常に浮動小数点定数によるものです。

ユーザー/ソース・コーディング規則 12 (影響 H、一般性 ML): できるだけデノーマル浮動小数点定数の使用を避けます。

デノーマル例外と算術アンダーフロー例外は、x87 命令の実行中に発生することも、インテル® SSE/インテル® SSE2/インテル® SSE3 命令の実行中に発生することもあります。Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサーは、インテル® SSE/インテル® SSE2/インテル® SSE3 命令を実行する場合と、IEEE 規格への準拠より処理速度を重視する場合に、これらの例外をより効率的に処理します。以下のセクションでは、浮動小数点例外によるパフォーマンスの低下を最小限に抑えるためコードを最適化する推奨事項について説明します。

3.9.2.2 x87 FPU コード内の浮動小数点例外の処理

3.9.2.1 節「浮動小数点例外」で示した特殊な状況は、すべてパフォーマンスに影響するペナルティーを発生させます。したがって、x87 FPU コードは、これらの状況を回避するように作成する必要があります。

基本的には、次の 3 つの方法で、x87 FPU コード内のオーバーフロー/アンダーフロー状態の影響を軽減できます。

- 十分に大きい浮動小数点データ型を選択して、算術オーバーフロー/アンダーフロー例外を発生せずに結果を表現できるようにします。
- オペランドと演算結果の範囲をスケーリングして、オーバーフロー/アンダーフロー状態の発生回数をできるだけ減らします。
- 最終結果が計算され、メモリーに格納されるまで、中間結果を x87 FPU レジスタースタック上に保持します。中間結果が x87 FPU スタック上に保持されている間は、オーバーフローやアンダーフローが発生する可能性は低くなります。これは、スタック上のデータは拡張倍精度形式で格納されるため、オーバーフロー/アンダーフロー状態もこの形式で検出されるためです。
- デノーマル浮動小数点定数 (これらは読み出し専用であるため、変更されない) の使用を避けて、できるだけ同じ符号のゼロで置き換えることが推奨されます。

3.9.2.3 インテル® SSE/インテル® SSE2/インテル® SSE3 コード内の浮動小数点例外

ハードウェアは、マスクされた浮動小数点例外を発生させる特殊な状況のほとんどを効率良く処理できます。インテル® SSE/インテル® SSE2/インテル® SSE3/インテル® AVX/インテル® AVX2/インテル® AVX-512 コードの実行中に、マスクされたオーバーフロー例外が発生した場合、プロセッサー・ハードウェアは、パフォーマンスを低下させることなくこの例外を処理します。

アンダーフロー例外とデノーマル・ソース・オペランドは、通常は IEEE754 仕様に従って処理されますが、大幅な遅延を招くことがあります。プログラマーが、IEEE 754 規格への準拠を犠牲にして処理速度を向上させたい場合は、2 つの非 IEEE 754 準拠モード (FTZ モードと DAZ モード) を使用できます。

FTZ モードが有効な場合、アンダーフロー結果は適切な符号のゼロに自動的に変換されます。この動作モードは、IEEE 754 に準拠していませんが、IEEE 754 への準拠よりパフォーマンスが重要なアプリケーションに使用されます。FTZ モードが有効な場合、デノーマル結果は生成されません。したがって、FTZ モードでデノーマル値が検出されるのは、読み出し専用のデノーマル浮動小数点定数の場合に限られます。

DAZ モードでは、SIMD 浮動小数点アプリケーションの実行時に、デノーマル・ソース・オペランドを効率的に処理できます。DAZ モードが有効になっている場合、入力デノーマルは同じ符号のゼロとして処理されます。パフォーマンスの向上が目標である場合、DAZ モードを有効にすれば、デノーマル浮動小数点定数を効率良く処理できます。

IEEE 754 仕様に完全に準拠しなくてもよい場合、パフォーマンスが重要であれば、FTZ モードと DAZ モードを有効にして、インテル® SSE/インテル® SSE2/インテル® SSE3/インテル® AVX/インテル® AVX2/インテル® AVX-512 アプリケーションを実行します。

注意

DAZ モードは、インテル® SSE 拡張命令とインテル® SSE2 拡張命令のいずれでも使用できます。ただし、DAZ モードによる処理速度の向上が十分に実現されるのはインテル® SSE 以降のコードだけです。

3.9.3 浮動小数点モード

x87 コードでは、FLDCW 命令を使用して浮動小数点モードを変更すると、ほとんどの場合コストの高い操作となります。

最近のプロセッサー世代では、プログラマーが 2 つの定数値を交互に効率良く使用できるように FLDCW のハードウェアが最適化されています。FLDCW 命令の最適化を有効にするため、FCW の 2 つの定数値は、FCW の次の 5 ビット以外は一致していなければなりません。

FCW[8-9] ; 精度制御
FCW[10-11] ; 丸め制御
FCW[12] ; 無限大制御

FCW の他のビット (例えば、マスクビット) を変更する場合は、FLDCW 命令によるペナルティーが生じます。

アプリケーションが 3 つ (またはそれ以上) の定数値を順番に切り替える場合、FLDCW 命令の最適化は適用されないため、FLDCW 命令を実行するたびにパフォーマンスが低下します。

この問題の 1 つの解決策は、FCW の 2 つの定数値を選択して、FLDCW 命令の最適化を利用してこれらの 2 つの定数値だけで FCW を切り替え、何らかの手段を工夫して、実際には FCW を 3 番目の定数値に切り替えずに、3 番目の FCW 値を必要とするタスクを実行することです。もう 1 つの解決策は、アプリケーションが最初は一定時間 2 つの定数値の間だけで FCW を切り替え、その後は別の 2 つの定数値の間で FCW を切り替えるように、コードを構成することです。アプリケーションが後で 2 つの FCW 値を切り替える場合、移行時にのみ、パフォーマンスが低下します。

SIMD アプリケーションが FTZ モードと DAZ モードの値を切り替える可能性は低いと予想されます。したがって、SIMD 制御ワードのレイテンシーは、浮動小数点制御レジスターのレイテンシーより大きくなります。MXCSR レジスターの読み出しには、相当大きなレイテンシーが生じます。MXCSR レジスターの書き込みはシリアル化されます。

単精度用と倍精度用に別々の制御ワードは存在しないため、単精度と倍精度の双方に同じ動作モードが適用されます。これは、FTZ モードと DAZ モードにも当てはまります。

アセンブリー/コンパイラー・コーディング規則 52 (影響 H、一般性 M): 浮動小数点制御ワードのビット 8 ～ 12 の変更を最小限にします。3 つ以上の値 (各値は次のビットの組み合わせ。精度制御、丸め制御、無限大制御、FCW の残りのビット) の間で切り替えると、パイプラインの段数にほぼ相当する遅延が発生します。

3.9.3.1 丸めモード

多くのライブラリーは、浮動小数点値を整数に変換する浮動小数点整数変換ライブラリー・ルーチンを備えています。これらのライブラリーの多くは、ANSI C コーディング基準に適合しています。ANSI C 基準では、丸めモードを切り捨てとして定義します。インテル® Pentium® 4 プロセッサーでは、CVTTSD2SI 命令と CVTTSS2SI 命令を使用して、丸めモードを変更せずに、切り捨てを使用してオペランドを変換できます。これらの命令を使用した場合のコスト削減効果は、x87 浮動小数点命令を使用するより十分に大きく、切り捨てを使用する場合は、できるだけインテル® SSE とインテル® SSE2 を使用することが推奨されます。

x87 浮動小数点では、FIST 命令は、浮動小数点制御ワード (FCW) で指定された丸めモードを使用します。この丸めモードは、通常は「直近値」への丸めとなっています。したがって、多くのコンパイラー開発者は、C および FORTRAN の基準に適合するため、プロセッサーの丸めモードの変更を実装します。この実装では、CW 命令を使用して、プロセッサーの制御ワードを変更する必要があります。丸めビット、精度ビット、無限大ビットを変更する場合、FSTCW 命令を使用して浮動小数点制御ワードをストアします。次に FLDCW 命令を使用して丸めモードを切り捨てに変更します。

FCW 内で丸めモードを変更する一般的なコードシーケンスでは、通常は FSTCW 命令の後にロード操作が続きます。このメモリーからのロード操作は、ストア・フォワーディングの問題を避けるため、16 ビット・オペランドを使用する必要があります。以前にストアされた FCW ワードに対するロード操作が 8 ビットまたは 32 ビットのオペランドを使用すると、ストア操作とロード操作の間でデータのサイズが一致しないため、ストア・フォワーディングの問題が発生します。

ストア・フォワーディングの問題が起きないように、FCW に対する書き込みと読み出しは、いずれも 16 ビット操作でなければなりません。

丸めビット、精度ビット、無限大ビットが 2 回以上変更され、結果的に丸めモードが重要でないときに、例 3-55 のアルゴリズムを使用して、同期の問題、FLDCW 命令のオーバーヘッドおよび丸めモード変更を回避します。この例では、パフォーマンスの低下につながるストア・フォワーディングの問題が発生することに注意してください。それでも、丸めビット、精度ビット、無限大ビットを 3 つ以上の値の間で変更するよりは効率が良くなります。

例 3-54 丸めモードの変更を回避するアルゴリズム

```
_fto132proc
   lea ecx, [esp-8]
   sub esp, 16           ; フレームの割り当て
   and ecx, -8           ; 境界 8 でポインターをアライン
   fld st(0)             ; FPU スタックの上端を複製
   fistp qword ptr[ecx]
   fild qword ptr[ecx]
   mov edx, [ecx+4]      ; 整数の上位 DWORD
   mov eax, [ecx]        ; 整数の下位 DWORD
   test eax, eax
   je integer_QnaN_or_zero
arg_is_not_integer_QnaN:
    fsubp st(1), st      ; TOS=d-round(d), { st(1) = st(1)-st & pop ST}
```

```
    test edx, edx       ; 整数の符号をチェック
    jns positive        ; 負の数
    fstp dword ptr[ecx] ; 減算の結果
    mov ecx, [ecx]      ; 差分の DWORD (単精度)
    add esp, 16
    xor ecx, 80000000h
    add ecx,7fffffffh   ; diff が 0 未満の場合、整数をデクリメント
    adc eax,0           ; INC EAX (CARRY フラグを加算)
    ret
positive:
    positive:
    fstp dword ptr[ecx] ; 17-18 減算の結果
    mov ecx, [ecx]      ; 差分の DWORD (単精度)
    add esp, 16
    add ecx, 7fffffffh  ; 差分が 0 未満の場合、整数をデクリメント
    sbb eax, 0          ; DEC EAX (CARRY フラグを減算)
    ret
integer_QnaN_or_zero:
    test edx, 7fffffffh
    jnz arg_is_not_integer_QnaN
    add esp, 16
ret
```

アセンブリー/コンパイラー・コーディング規則 53 (影響 H、一般性 L): 丸めモードの変更回数を最小限に抑えます。丸めビット、精度ビット、無限大ビットが合計 3 つ以上の値の間で切り替えられる場合は、丸めモードの変更を使ってフロアー関数とシーリング関数を実行してはなりません。

3.9.3.2 精度

単精度演算で十分な場合は、倍精度ではなく単精度を使用します。これは、以下の理由によります。

- 単精度演算の方が長い SIMD ベクトルを使用できます。これは、単精度データの方が、レジスターに保持できるデータ要素の数が多いためです。
- x87 FPU 制御ワードの精度制御 (PC) フィールドが単精度に設定されている場合、浮動小数点除算器は、倍精度計算や拡張倍精度計算よりはるかに高速に、単精度計算を実行できます。PC フィールドが倍精度に設定されている場合は、倍精度データに対する x87 FPU 演算は、拡張倍精度演算より早く完了します。これらの特性は、浮動小数点除算や平方根計算などに影響します。

アセンブリー/コンパイラー・コーディング規則 54 (影響 H、一般性 L): 精度モードの変更回数を最小限に抑えます。

3.9.4 x87 コードとスカラー SIMD 浮動小数点コードのトレードオフ

x87 浮動小数点コードとスカラー浮動小数点コード (インテル® SSE とインテル® SSE2 を使用) には、多くの違いがあります。これらの違いを考慮して、どのレジスターと命令を使用するか決定すべきです。

- SIMD 浮動小数点命令の入力オペランドに、そのデータ型で表現可能な範囲より小さい値が含まれている場合、デノーマル例外が発生します。これにより、パフォーマンスが大きく低下します。SIMD 浮動小数点操作には、ゼロ・フラッシュ・モード (FTZ) があり、結果のアンダーフローは発生しません。したがって、それ以降の計算に、デノーマル入力オペランドの処理によるパフォーマンスの低下は生じません。例えば、3D アプリケーションに低い照明レベルで多数のアンダーフローが発生する場合、ゼロ・フラッシュ・モードを使用すると、パフォーマンスは約 50% 向上します。
- 同等の x87 命令よりもスカラー浮動小数点 SIMD 命令の方がレイテンシーは小さくなります。スカラー SIMD 浮動小数点乗算命令はパイプライン化できますが、x87 乗算命令はパイプライン化できません。
- x87 は超越関数命令をサポートしていますが、ほとんどの場合ソフトウェア・ライブラリーの超越関数の方が高速です。
- x87 命令は、80 ビットの拡張倍精度浮動小数点をサポートしています。インテル® SSE は、最大 32 ビットの精度をサポートしています。インテル® SSE2 は、最大 64 ビットの精度をサポートしています。
- スカラー浮動小数点レジスターは、FXCH 命令とスタック上位の制限なしに、直接アクセスできます。
- Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサーでは、丸めモードの変更や例 3-58 のシーケンスを使用するより、インテル® ストリーミング SIMD 拡張命令 2 およびインテル® ストリーミング SIMD 拡張命令を使用する方が、切り捨てを使用して浮動小数点を整数に変換するコストは大幅に小さくなります。

アセンブリー/コンパイラー・コーディング規則 55 (影響 M、一般性 M): x87 命令の機能が特に必要な場合以外は、インテル® ストリーミング SIMD 拡張命令 2 またはインテル® ストリーミング SIMD 拡張命令を使用します。大部分のインテル® SSE2 算術演算は x87 よりもレイテンシーが小さく、x87 レジスタースタックの管理に関連するオーバーヘッドが生じません。

3.9.4.1 インテル® SSE/インテル® SSE2 スカラー命令

浮動小数点から整数への変換、単精度命令の除算、または精度の変更があるコードシーケンスでは通常、コンパイラーから x87 コードを生成すると、データが単精度でメモリーに書き込まれた後、再び読み出され、精度が低下します。x87 コードの代わりにインテル® SSE/インテル® SSE2 のスカラーコードを使った場合、Intel NetBurst® マイクロアーキテクチャー利用時にはパフォーマンス上の大きなメリットが得られ、インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサー利用時にはある程度のメリットが得られます。

推奨事項: コンパイラー・オプションを使用して、x87 コードではなく XMM を使用するインテル® SSE2 のスカラー浮動小数点コードを生成します。

インテル® SSE/インテル® SSE2 のスカラーコードを使用する場合は、XMM レジスターの未使用スロットの内容をクリアする必要性と、それに伴うパフォーマンスへの影響に注意します。例えば、MOVSS または MOVSD 命令を使ってメモリーからデータをロードすると、XMM レジスターの上位半分をゼロにするための追加のマイクロオペレーション (uop) が生じます。

3.9.4.2 超越関数

アプリケーションが、パフォーマンス上の理由などで、超越関数をソフトウェア上でエミュレートしなければならない場合 (3.9.1 節「浮動小数点コードの最適化のガイドライン」を参照)、数値演算ライブラリー呼び出しをインライン展開する方が適切です。これは、CALL 命令と呼び出しに付随するプロローグ/エピローグが、演算のレイテンシーに大きな影響を与える可能性があるためです。

3.10 PCIe* パフォーマンスの最大化

アップストリームの読み出しと書き込み (PCIe* エージェントからホストのメモリーに発行される読み出しおよび書き込みトランザクション) のサイズおよびアライメントによって、PCIe* パフォーマンスが大きな影響を受けることがあります。一般的な原則としては、帯域幅とレイテンシー両方の面から見た最高のパフォーマンスは、64 バイト境界にアップストリームの読み出しおよび書き込みの開始アドレスをアライメントし、要求のサイズが 64 バイトの倍数になるようにすることで実現できます。さらに、大きな倍数 (128 バイト、192 バイト、256 バイト) を使用した場

合、帯域幅はある程度向上します。特に、パーシャル書き込みの場合、後続の要求 (読み出しまたは書き込み) の遅延を招きます。

2 番目の規則としては、1 つのキャッシュラインへの複数の未処理の同時アクセスを回避することです。1 つのキャッシュラインに同時に複数の未処理のアクセスがあった場合、競合が発生し、本来ならばパイプライン化されるはずのアクセスがシリアル化され、その結果レイテンシーが長くなったり、帯域幅が低くなる可能性があります。この規則に違反するパターンとして、64 バイトの倍数でないシーケンシャル・アクセス (読み出しまたは書き込み)、同じキャッシュライン・アドレスへの明示的なアクセスが挙げられます。オーバーラップする要求 (開始アドレスは異なるものの、要求の長さによってオーバーラップを招く要求) も、同じ影響を及ぼす可能性があります。例えば、アドレス 0x00000200 の 96 バイトの読み出しに続いて、アドレス 0x00000240 の 64 バイトの読み出しを行うと、2 番目の読み出しで競合 (および遅延の可能性) が発生します。

64 バイトの倍数ではあるものの、アライメントされていないアップストリームの書き込みが行われると、パーシャル書き込みとフル・シーケンシャル書き込みが連続して発生します。例えば、アドレス 0x00000070 への長さ 128 バイトの書き込みは、アドレス 0x00000070、アドレス 0x00000080、アドレス 0x00000100 へのそれぞれ長さ 16 バイト、64 バイト、48 バイトの 3 つのシーケンシャル書き込みと同じです。

デュアルポートやクワッドポートのネットワーク・インターフェイス・カード (NIC) またはデュアル GPU グラフィックス・カードなど、マルチファンクション・デバイスを実装する PCIe* カードを使用する場合、そのうちのいずれかのデバイスの動作が最適でないと、そのカードのそれ以外のデバイスの帯域幅やレイテンシーに影響が生じる可能性があることに注意しなければなりません。ここで説明する動作に関しては、特定の PCIe* ポート上のすべてのトラフィックを、単一のデバイスおよびファンクションからのものであるものとして扱います。

最適な PCIe* 帯域幅を実現する方法を以下に示します。

1. アップストリームの読み出しおよび書き込みの開始アドレスを 64 バイト境界にアライメントします。
2. 64 バイトの倍数である読み出し要求および書き込み要求を使用します。
3. シーケンシャルなパーシャルライン・アップストリーム書き込みやランダムなパーシャルライン・アップストリーム書き込みは排除または避けます。
4. シーケンシャルなパーシャルライン読み出しなど、競合するアップストリーム読み出しは排除または避けます。

パフォーマンスの問題を回避する手法として、キャッシュラインですべてのディスクリプターとデータバッファーをアライメントする、アップストリームに書き込まれるディスクリプターを 64 バイト・アライメントにパディングする、受信したデータをバッファーに格納してより大きなアップストリーム書き込みペイロードにする、64 バイト (の倍数) 読み出しを使用できるように PCIe デバイスによるシーケンシャルな読み出し用のデータ構造体を割り当てる、といった方法が挙げられます。最適化されていない読み出しおよび書き込みの影響は、個別のワークロード、およびその製品がベースとするマイクロアーキテクチャーによって異なります。

3.10.1 コヒーレントなメモリーと MMIO 領域 (P2P) へのアクセスに対する PCIe* パフォーマンスの最適化

以下の表 3-7 に示すプロセッサーの PCIe* デバイスに対するパフォーマンスを最大化するため、ソフトウェアはアクセスがコヒーレントなメモリー (システムメモリー) であるか、MMIO 領域 (他のデバイスへの P2P アクセス) であるかを明確にする必要があります。アクセスが MMIO 領域である場合、ソフトウェアは TLP ヘッダーの RO ビットを設定するように HW に指示できます。これによりハードウェアは、このようなタイプのアクセスで最大限のスループットを達成できます。また、アクセスがコヒーレント・メモリーである場合、ソフトウェアは TLP ヘッダーの RO ビットをクリア (RO ではない) するように HW に指示できます。これによりハードウェアは、このようなタイプのアクセスで最大限のスループットを達成できます。

表 3-7 PCIe* パフォーマンスを最適化するインテル® プロセッサーの CPU RP デバイス ID

プロセッサー	CPU RP デバイス ID
Broadwell† マイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサー	6F01H-6F0EH
Haswell† マイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサー	2F01H-2F0EH

第 4 章
Intel Atom® プロセッサー・アーキテクチャー

この章では、最新世代の Intel Atom® プロセッサー[7]のソフトウェア最適化に関連する機能の概要を示します。

4.1 Gracemont† マイクロアーキテクチャー

Gracemont† マイクロアーキテクチャーは、Tremont† マイクロアーキテクチャーでの成功をベースに構築されています。Gracemont† マイクロアーキテクチャーで提供される拡張機能の一部を以下に示します。

- 拡張分岐予測ユニット。
- デュアル 32B リード (フェッチクラスターごとに 32B リード) の拡大された 64KB 命令キャッシュ。
- 共有 L2 プリデコード・キャッシュが、フェッチクラスターごとのオンデマンド命令デコーダーに置き換えられました。
- 2 つのフェッチクラスター間のダイナミック・ロード・バランス。
- よりワイドなアロケーションとリタイアメント幅。
- より大きなロードとストアバッファー。
- デュアルロードとデュアルストア実行パイプ。
- 機能拡張された 4 つの整数 ALU 実行ポート。
- 2 つの分岐実行ポート。
- デュアル整数乗算と整数除算ユニット。
- 暗号化のパフォーマンスを強化するため、インテル® SHA-NI とインテル® AES-NI レイテンシーを改善。
- 256 ビットのインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX とインテル® AVX2)。
- BMI1、BMI2、ADX、LZCNT ISA 拡張。
- VEX ベースの VNNI ISA 拡張。
- マルウェアに対する保護を強化するインテル® コントロールフロー・エンフォースメント・テクノロジー (インテル® CET)。

4.1.1 Gracemont† マイクロアーキテクチャー概要

図 4-1 に Gracemont† マイクロアーキテクチャーの基本パイプライン機能を示します。

[7] 旧世代の Intel Atom® プロセッサーについては、付録 F「旧世代の Intel Atom® マイクロアーキテクチャーとソフトウェアの最適化」を参照してください。

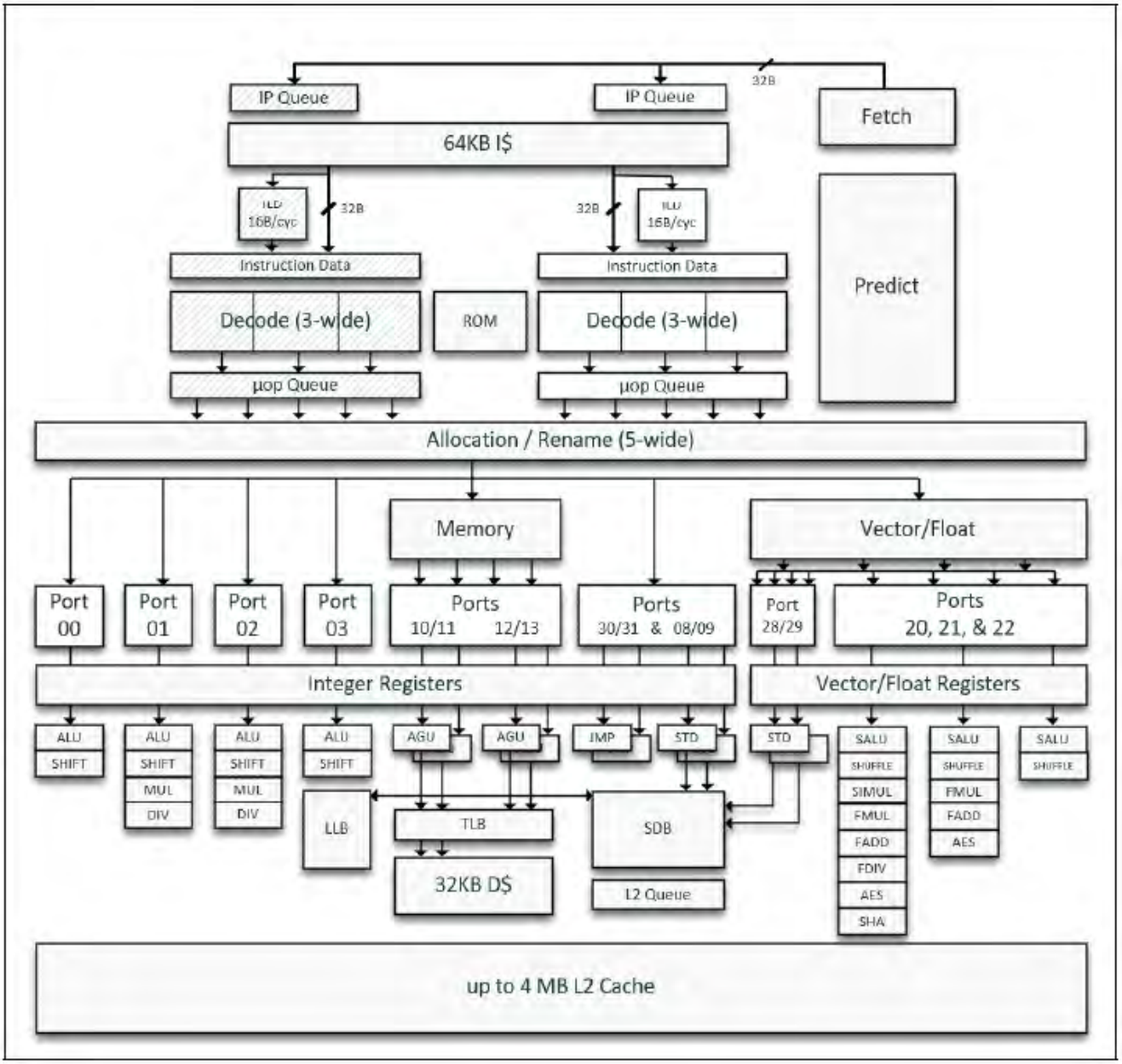


図 4-1 Gracemont† マイクロアーキテクチャーのプロセッサー・コア・パイプラインの機能

Gracemont† マイクロアーキテクチャーは、複数 L3 スライスへのリング・インターコネクト、プロセッサー・グラフィックス、統合メモリー・コントローラー、インターコネクト・ファブリックなどを含む、多くのコンポーネントで構成される共有アンコア・サブシステムと複数のプロセッサー・コアによる柔軟性のある統合をサポートします。

4.1.2 予測とフェッチ

Gracemont† マイクロアーキテクチャーは、32 バイトの予測 (プレディクション) 機能を備えたフロントエンドを備えています。最初の予測器は次のラインの予測器 (NLP - next-line predictor) であり、サイクルごとにジャンプする分岐を予測し、バブルなしで分岐先をフェッチできます。NLP は、パスベースの情報と組み合わせた 5K のエントリーターゲット配列により、3 サイクルで予測とターゲットアドレスを検証する 2 番目の予測器でバックアップされます。そして、命令デコードは、いずれの予測器にも存在しない分岐をデコードする際に、フロントエンドをリダイレクトできます。Gracemont† マイクロアーキテクチャーのフロントエンド・パイプラインの機能を図 4-2 に示します。

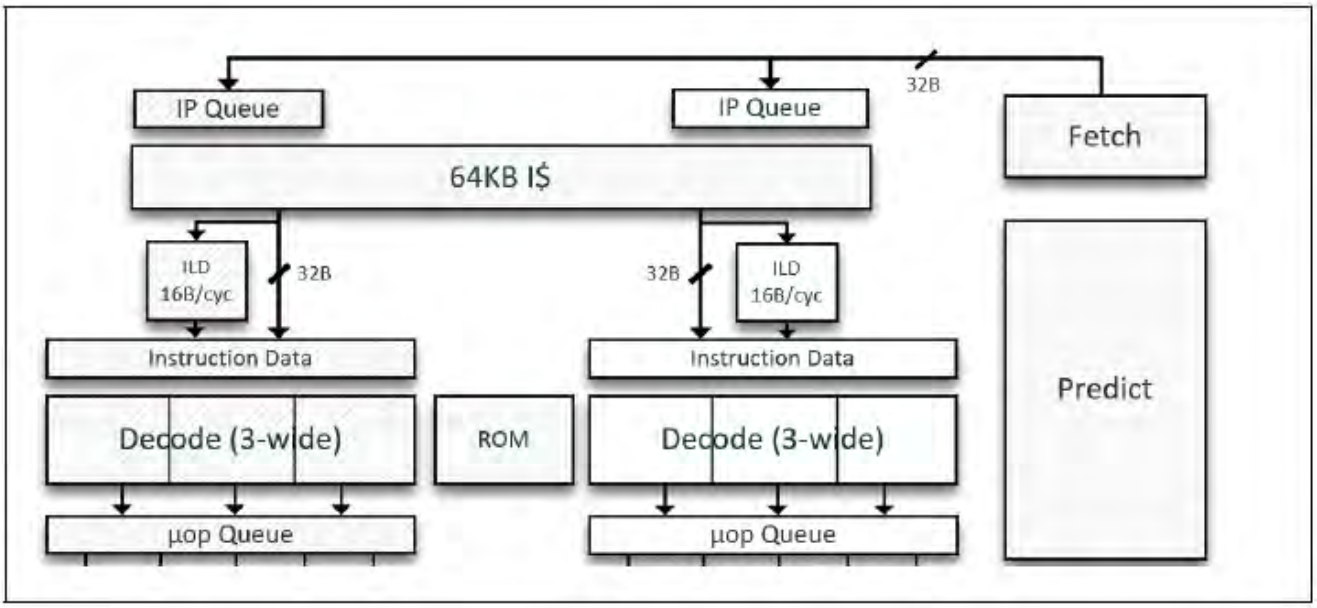


図 4-2 Gracemont† マイクロアーキテクチャーのフロントエンド・パイプラインの機能

サイクルごとに、予測された IP が命令フェッチ・パイプラインに送出されます。この予測では、命令 TLB (ITLB) と命令キャッシュタグを検索して、物理アドレスと命令キャッシュのヒットまたはミスを判定できます。変換に成功すると、リソースの空き状況に応じて、それらのアクセスは命令ポインター (IP) キューに格納されます。これにより、命令キャッシュのヒット/ミスを、実際の命令バイトをフロントエンドの残りの部分に供給することから分離できます。命令キャッシュミスが発生すると、IP キューはアドレスを保持しますが、データがメモリー・サブシステムから取得されるまで読み取れないことを通知します。フェッチで生成される IP ストリームは、最大で 8 つの命令キャッシュミスを同時に処理できます。独立した 2 つの IP キューは、それぞれ個別の命令データバッファーを持っています。これらは、関連するデコーダーと組み合わせてクラスターと呼ばれます。ジャンプした分岐や挿入されたトグルポイントごとに、予測は各 IP キュー間、つまりクラスター間で前後に切り替えられます。この切り替えにより、アウトオブオーダーでのデコードが可能になり、Gracemont† マイクロアーキテクチャーが最大 6 つの可変長 x86 命令のフェッチとデコードを行う重要な機能となっています。

予測やフェッチのパフォーマンス・デバッグは、https://perfmon-events.intel.com (英語) にあるパフォーマンス管理イベントのトップダウン・カテゴリーにあるフロントエンド依存イベントを利用して行うことができます。フロントエンド依存のイベントは、利用可能なスロットがあっても uop が存在しない場合にのみ、割り当て可能になったときにスロットをカウントします。例えば、3 サイクルの予測器によって発生したバブルが割り当てまで存在する場合、それらは TOPDOWN_FE_BOUND.BRANCH_RESTEER で表現されます。FRONTEND_RETIRED.BRANCH_RESTEER によって、バブル続く命令を正確にタグ付けできます。予測器が分岐ターゲットのキャッシュに失敗し、デコード中にリダイレクトが発生するとそれらのスロットは TOPDOWN_FE_BOUND.BRANCH_DETECT でカウントされます。命令キャッシュまたは命令 TLB のミスが原因で uop が供給されない場合、それらはそれぞれ TOPDOWN_FE_BOUND.ICACHE および TOPDOWN_FE_BOUND.ITLB として示されます。BRANCH_RESTEER と同様に、すべてのフロントエンド依存のスロットベースの計測は、対応する FRONTEND_RETIRED イベントセットにより正確に追跡できます。ほとんどの場合、命令コードを再配置することで、このようなボトルネックを最適化できます。複数のイベントクラスは、同じ汎用パフォーマンス・カウンターで、または複数のパフォーマンス・カウンター間で異なるイベントを使用して同時に追跡できます (例えば、ICACHE と ITLB の両方のイベントをマークします)。

コードのループが短すぎたり、キャッシュ内でアライメントが取れていないなど、マシンが高速にデコードできないことがあります。この場合、1 サイクルごとにフェッチすることでバブルが生じることはありませんが、バックエンドに供給し続けることはできません。これは、TOPDOWN_FE_BOUND.OTHER イベントクラスで検出できます。“その他”のイベントクラスは、ソースに関連付けることができないフロントエンド依存の動作を補足します。

4.1.3 ダイナミック・ロード・バランス

クラスター化されたデコーダーのマイクロアーキテクチャーでは、非常に大きな基本ブロックが実行されると、ユニークなパフォーマンスの問題が発生する可能性があります。コンパイラーは、並列性を高めオーバーヘッドを軽減するため、コードのループをアンロールして数百命令にもおよぶ命令ブロックを生成することがあります。これは、浮動小数点やベクトル処理向けの一部のコンパイラーでは一般的です。クラスタリングの手法はトグルポイントに依存するため、Tremont† マイクロアーキテクチャー向けの手動アセンブリーでは、無条件 JMP 命令が次のシーケンシャル命令のポインターとして挿入された可能性があります。Gracemont† マイクロアーキテクチャー以降では、このような挿入は必要ありません。Gracemont† マイクロアーキテクチャーは、ハードウェアによるロードバランサーを導入することで、このボトルネックの問題に対処しています。ハードウェアが大きな基本ブロックを検出すると、内部ヒューリスティックに基づいて追加のトグルポイントを作成できます。このトグルポイントは予測器に追加され、基本ブロック内でトグルするようにマシンをガイドします。

インテル® マイクロアーキテクチャーでは、ほぼすべての基本演算命令は 1uop です。インテル® CET が有効にされた CALL などの複雑な命令も 1uop にデコードされます。ロードバランサーの高度なアルゴリズムは、命令バイトのシーケンシャル・ストリームに存在する uop 数に基づいています。32uop 内に通常のトグルポイント (ジャンプした分岐など) が存在しない場合、ハードウェアはストリームの 24uop 以降、またはそれに相当する命令にトグルポイントを挿入します。挿入されるトグルポイントは予測器のリソースを消費するため、通常はすぐに挿入されずアドレステーブルで命令の位置をマークします。同じ位置に挿入されたトグルポイントが 2 回目にマークされると、その位置を予測器に割り当てます。

単一のトグルポイントに至るまでのシーケンシャル uop の数が動的に変動することもあります。例えば、ジャンプせずスルーした条件分岐が、その後ではジャンプするように変化するような場合です。そのような状況では、挿入されたトグルポイントの位置が長い uop シーケンスの終端ではなくなった場合、通常そのトグルポイントは削除されます。また、このアルゴリズムは uop ベースであるため、多くの uop で構成される長いマイクロコードのシーケンスとして実装される命令は、トグルポイントが挿入される可能性が高くなります。これは、そのような条件下でもデコードが継続されることを保証するため、有利です。

4.1.4 デコードとオンデマンド命令レングスデコーダー

Gracemont† マイクロアーキテクチャーは、命令キャッシュ内の各バイトに対し、命令境界を示す 1 ビットを格納します (プリデコード・ビットと呼ばれます)。このビットは、命令バイトをデコーダーレーンに誘導するために使用されます。通常の可変長エンコードでは、追加される命令を検出するには 1 つの命令をデコードして、その情報を次の命令のデコードに反映するような作業が必要であると考えられます。この機能がワイドになるほどコストは急激に上昇します。プリデコード・ビットを利用することで、このパスにある命令のデコードを排除できます。クラスター化されたデコードで 3 つのワイドデコーダーを実装すると、ハードウェアは 3 番目のシリアル命令の終端を検出する以外のことを行う必要はありません。その結果、非常に小さなダイ面積と低い消費電力で実装できる命令の多重化とデコード実現できます。

1 つ懸念があるとすれば、プリデコード・ビットを決定し、それを参照して命令境界をマークすることがあげられます。Tremont† マイクロアーキテクチャーからの別の変更点として、大容量 (128KB) の共有 L2 プリデコード・キャッシュを削除したことがあげられます。このキャッシュは、L1 命令キャッシュにミスした際に、L1 プリデコード・キャッシュのシードを補助します。これにより、ほとんどの場合はパフォーマンスが向上しましたが、フットプリントが 1MB を超えるクリティカルなコードのループでは、誤ったプリデコード・ビットによるデコード帯域幅の低下により、フロントエンドがボトルネックになることがありました。これは、TOPDOWN_FE_BOUND.PREDECODE イベントで検出できました。

L2 プリデコード・キャッシュに変わり、Gracemont† マイクロアーキテクチャーでは “オンデマンド” 命令レングスデコーダー (OD-ILD) が導入されました。この機能は、通常、新しい命令バイトがミスから命令キャッシュにフェッチされる場合にのみアクティブになります。この機能が動作すると、プリデコード・ビットをオンザフライで生成するため、フェッチ・パイプラインに 2 サイクル追加されます。これは、サイクルあたり 16 バイトを処理できます。Gracemont† マイクロアーキテクチャーは、クラスタリングにより 2 つの独立した OD-ILD でサイクルあたり 32 バイトを処理できることになります。多くのワークロードでは、Gracemont† と Tremont† マイクロアーキテクチャーの動作の違

いは感じられませんが、コードのフットプリントが大きなワークロードでは利点を感じられるかもしれません。このような x86 命令デコードのアプローチは、デコード後の命令をキャッシュすることなく、非常にワイドな設計への明確な利点を示します。

各命令デコーダーは単一の uop を生成しますが、x86 コード大部分の生成は動的な命令カウントで測定できます。ロード-オペレーション-ストア、複雑なアドレス指定形式、インテル® CET 命令などの多くが、単一の内部 uop 形式で生成されます。また、各デコーダーは、マイクロコードのエントリーポイントを検出できます。一般的なショート形式のマイクロコードはクラスター間でアウトオブオーダー実行でき、さらなるパフォーマンス向上を可能にします。すべての uop は 2 つの並列 uop キューに書き込まれ、コアのフロントエンドとバックエンドが独立して実行できるように設計されています。アロケーションとリネーム・パイプラインは、両方の uop キューを並列に読み取り、レジスターのリネームとリソース割り当てのため命令ストリームをインオーダーに戻します。

デコードクラスター内のマイクロアーキテクチャーの低レベルの特性は、Tremont† マイクロアーキテクチャーと同様です。例えば、命令は 4 バイトを超えるプリフィクスとエスケープを回避する必要があります。詳細については、20.2.8.9 付録 F の「旧世代の Intel Atom® マイクロアーキテクチャーとソフトウェアの最適化」を参照してください。

パフォーマンスのデバッグを行う際に、負荷バランスや他のデコードの制限が問題となる場合、それらは TOPDOWN_FE_BOUND.DECODE イベントで観察できます。デコーダーが適切なプリデコード・ビットを持たなかったり、命令にプレフィクスやエスケープが多すぎる場合は、TOPDOWN_FE_BOUND.PREDECODE に示されます。マシンが長いマイクロコード・シーケンスを待機している場合、それは TOPDOWN_FE_BOUND.CISC で検出できます。すべてのアロケーション・スロット・ベースの FE_BOUND イベントと同様に、指定されたイベントクラスが発生した後の命令をマークする FRONTEND_RETIRED イベントがあります。ただし、CISC イベントではこのレポートに違いがあります。長いマイクロコード命令の実行によるスロットベースのボトルネックは、通常、命令内で発生するため、FRONTEND_RETIRED.CISC イベントは CISC 命令の後ではなく、その命令自身にタグ付けすることがあります。また、外部割込みやフォールト、トラップなどのアシスト処理のためマイクロコードが呼び出されると、FRONTEND_RETIRED.CISC イベントはその命令の次の命令をマークします。

4.1.5 アロケーションとリタイアメント

Gracemont† マイクロアーキテクチャーは、サイクルあたり最大 5 つの uop をアロケーションできます。アロケーションでは、すべてのフロントエンド・クラスターの uop キューを同時に読み出し、必要に応じて同一サイクルでクラスタリングの境界を超えて連鎖するインオーダー・ストリームを生成します。uop キュー内の形式とマシンに割り当てられている形式に違い (拡張) が存在する可能性があります。例えば、256 ビットのインテル® AVX 命令は、フロントエンドは単一の uop として命令をデコードしますが、アロケーション時に 2 つの 128 ビット操作に分割されます。この場合、2 つの 128 ビット命令を割り当てるため、2 つのアロケーション・レーンが使用されます。256 ビットのインテル® AVX 以外の uop でこの方式が適用されるのは、整数乗算や除算など、複数の論理レジスター・デスティネーションを必要とする整数 uop が一般的です。別の例として、あるアドレスのメモリーから値をロードし、スタックポインターのメモリーに値をストアして、スタックポインターを更新する PUSH メモリーがあります。ある操作が 2 つのアロケーション・レーンを必要とし、それが最後の (5 番目の) アロケーション・レーンである場合、ハードウェアは最初のサイクルで最初の操作を割り当てて、次のサイクルで 2 番目の操作を割り当てます (最大 4 つの uop)。ムーブ除去、NOP 検出およびイディオム検出 (例えば、レジスター自信を XOR してゼロ値を生成)、およびメモリーのリネームは割り当て時に行われます。これにより、依存関係の連鎖が減少し、場合によっては uop を実行から除外することができます。

リタイアメント・バッファーは 256 エントリーで、サイクルあたり最大 8 命令をリタイアできます。リタイアメントをアロケーションよりも広くすることで、ストアの取り消しなどのパフォーマンスを、ほかのより一般的ではないフラッシュのような場合も含めて向上できます。これは、電力効率の向上につながります。リタイアメント拡大のコストは比較的小規模です。これにより、操作のライフタイムが短くなるため、コアの構造を小さく、そして浅くすることができます。

4.1.6 アウトオブオーダーと実行エンジン

Gracemont† マイクロアーキテクチャーのアウトオブオーダーと実行エンジンには、次の変更があります。

- リオーダーバッファー、ロードバッファー、ストアバッファー、およびリザベーション・ステーションのサイズが大幅に増加したことで、より深い OOO 実行と高いキャッシュ帯域幅を実現。
- よりワイドなマシン。10→17 実行ポート。
- 実行ポートごとの機能拡張。

図 4-3 に Gracemont† マイクロアーキテクチャーの実行パイプラインの機能を示します。

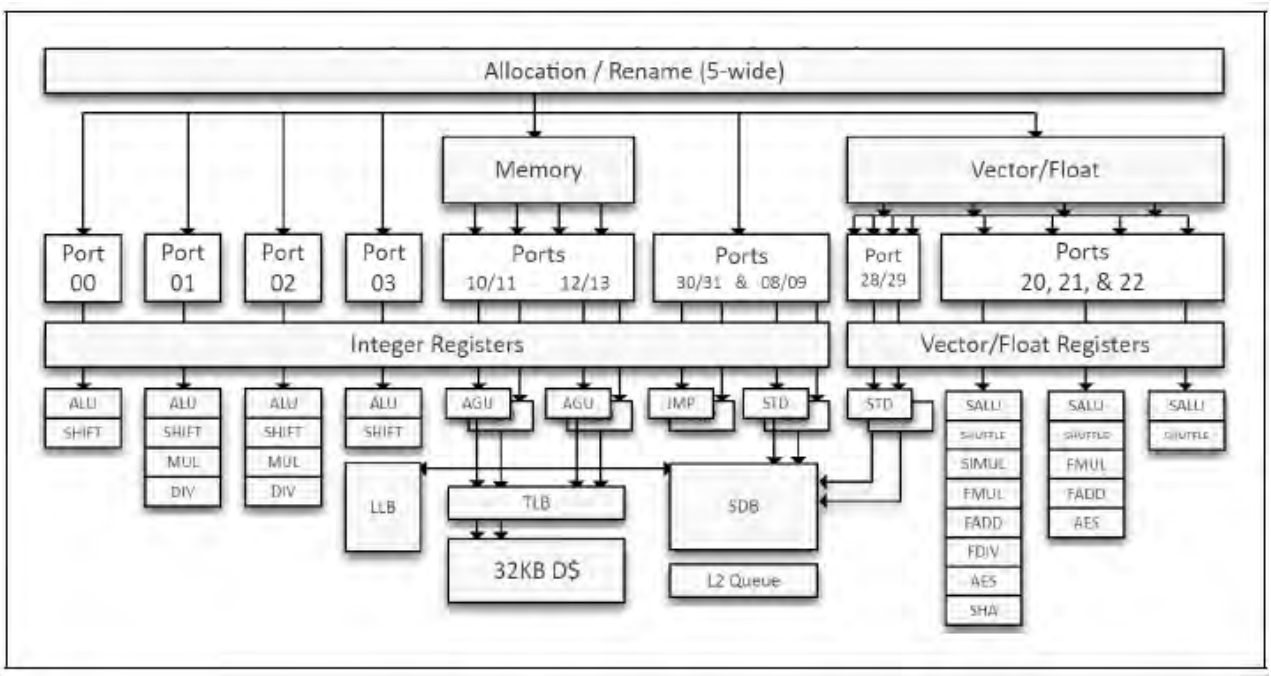


図 4-3 Gracemont† マイクロアーキテクチャーの実行パイプラインの機能

uop は 3 種類の構造体に割り当てられます。純粋な整数操作では、各 uop は 5 つのリザベーション・ステーション (RS) のうち 1 つ以上に書き込まれます。RS は命令を保持して依存関係を追尾し、実行のためスケジュールを行います。4 つのは ALU 操作用でポート 00 から 03 と表記されます。これらの実行ユニットは、単一サイクル操作でほとんどがシンメトリックです。4 つのポートのうち 2 つ (01 と 02) は、乗算や除算などレイテンシーの長い操作を実行します。5 番目のリザベーション・ステーションでは、分岐やストアデータ操作を実行します。この構造体はバンク化されており、ポート 08 と 09 に 2 つのストアデータ、ポート 30 と 31 に 2 つの分岐と、それぞれのタイプの 2 つの uop をサイクルごとにスケジュールできます。ソースとデスティネーションがメモリー上にある ADD のような複雑な命令は、フロントエンドでデコードされ、単一の uop として割り当てられます。Gracemont† マイクロアーキテクチャーは、サイクルごとにこのような 5 つの uop を割り当てできます。しかし、このように uop は、バックエンドに投入されると複数の断片に分割されてしまいます。この例では、単一の複合 uop で、ロード、加算、ストアアドレス操作、そしてストアデータ操作が生成されています。これらの断片化された実行を、アウトオブオーダー・マシンで独立して実行するには、4 つの異なるディスパッチ・ポートが必要になります。

LEA (ロード・エフェクティブ・アドレス) 操作は特殊な操作であり、注意が必要です。ALU ポートは標準的な 2 ソース/論理操作の実行に最適化されており、AGU は x86 の複雑なメモリーアドレス指定を処理するように最適化されています。2 つのソースを持つ LEA のうち、ベース、インデックス、およびディスプレースメント中にスケーリング・インデックスのないものは、いずれのポート (00 から 03) でも通常の ALU 操作として実行できます。3 つのソースを持つ LEA は 2 つの操作に分割されるため、追加のレイテンシー・サイクルが生じます。スケーリング・インデックスを持ち、ディスプレースメントがない LEA は、単一のオペレーションとして実行されますがポート 02 に静的にバインドされます。

また、アロケーションはメモリーキューに書き込むこともできます。これは、より深いバッファー処理のマイクロアーキテクチャーを低コストで実装可能にする FIFO キューです。そして、メモリーキューは、ロードとストアのアドレス生成オペレーションを保持するリザベーション・ステーションに書き込むことができます。このリザベーション・ステーションは、サイクルあたり 2 つのロード (ポート 10 と 11) と、2 つのストアアドレス (ポート 12 と 13) 計算を生成できます。また、メモリーキューは、メモリー・サブシステムにロードとストア uop を書き込むことで、アドレス変換とデータキャッシュのアクセスを行います。

最後に、アロケーションはベクトルキューに書き込みます。ベクトル SIMD と浮動小数点 ALU の操作はすべてここで処理されます。この FIFO キューは、3 つのスケジュール・パイプラインを持つ統合リザベーション・ステーション (ポート 20、21、22) 、またはサイクルあたり 2 つのストアデータをディスパッチできるストア・データ・リザベーション・ステーション (ポート 28、29) に書き込むことができます。ベクトルユニットは、2 つの浮動小数点値の乗算、加算、乗算加算を任意の組み合わせで実行できます。これにより、サイクルあたり単精度を 16 個、倍精度を 8 個の FLOPS ピーク値を達成します。また、最大 3 つの SIMD 整数 ALU やシャッフル操作、専用の AES および SHA ユニットを実行することができます。

4.1.7 キャッシュとメモリー・サブシステム

Gracemont† マイクロアーキテクチャーのキャッシュ階層では、次のような変更が行われています。

- ピーク時のロードとストア帯域幅の合計が 2 倍になりました。
  - 2 つの専用ロードポート。
  - 2 つの専用ストアポート。

- バッファー数が増えたことにより、多くのロードとストアを同時に処理することが可能になりました。
- 4 サイクルのロードから使用できるまでのレイテンシー。
- 4 つのページウォークを同時に処理可能なパイプライン化されたページ・ミス・ハンドラー。
- ページング階層全体が、大きなページ変換をできるように強化されました。
- 大きな L2 TLB。
- L2 キャッシュは、2MB から 4MB まで製品設計ごとに異なります。
  - Alder Lake† パフォーマンス・ハイブリッド・アーキテクチャーを採用するプロセッサーの L2 キャッシュサイズは 2MB です。

Gracemont† マイクロアーキテクチャーのメモリー・サブシステムは、サイクルあたり 16 バイトのロードと 16 バイトのストアをそれぞれ 2 つずつ処理し、サイクルあたり 32 バイトのリード帯域幅と 32 バイトのライト帯域幅を同時に提供できるように設計されています。ロードしたデータを利用できるまでのレイテンシーは、通常 4 サイクルです。計算中のアドレスが 1 つ前のロードの結果であり、+1023 以下の正のディスプレースメントでポインター追尾を行う場合、ローから使用までのレイテンシーを 3 サイクルに減らすことができます。L1 データキャッシュは、デュアルポート化され、バンク競合の可能性が排除されています。

メモリー・ディスアンビゲーションがサポートされるため、先行するストアのアドレスが未解決のままでもロードを実行することができます。ストアからフォワードされるロードは、ストアアドレスが判明していてストアデータが利用できる場合、キャッシュヒットと同じロード・レイテンシーでデータを取得できます。ストアアドレスやデータが直ちに利用できない場合は、ハードウェアが関連するアドレスである可能性が高いと判断すると、正確なブロック化とスケジュールが行われます。

アドレス変換は、フル・アソシアティブな L1 DTLB を介して行われます。Gracemont† マイクロアーキテクチャーでは、2MB の変換が L1 DTLB 内にネイティブにキャッシュされます。DTLB は、コード要求とデータ要求で共有される 2 つの L2 TLB (STLB) 構造によって支えられています。メインの STLB は 2048 エントリーの 4 ウェイ・セット・アソシアティブで、4KB と 2MB の変換をキャッシュします。さらに、Gracemont† マイクロアーキテクチャーでは、GB ページ変換用に 8 エントリーのフル・アソシアティブ構造を備えています。STLB ミスはページ・ミス・ハンドラー (PMH) に送られ、パイプライン化された PMH は、最大 4 つのページウォークを並列に実行できます。

表 4-1 Gracemont† マイクロアーキテクチャーのキャッシュ・ページング・パラメーター

レベル	エンティティー	アソシアティビティー	アーキテクチャー上のページサイズ	キャッシュされた変換サイズ
ITLB	64	フル・アソシアティブ	すべて	4KB、256KB
DTLB	32	フル・アソシアティブ	すべて	4KB、2MB
STLB	2048	4 ウェイ	4K/2M/4M	4KB、2MB
STLB	8	フル・アソシアティブ	1GB	1GB

3 つの独立した L1 プリフェッチャーがあります。1 つは、DL1 ロードミスが発生すると単純に次のラインのフェッチを行います。2 つ目は、各種サイズのストライド・アクセス・パターンを検出できる命令ポインターベースのプリフェッチャーです。このプリフェッチャーは、リニアアドレス空間で動作するため、ページ境界を越える TLB ミスに伴う変換を開始できます。3 番目のプリフェッチャーは、ページ境界を超える可能性があるアクセスを検出し、早期にアクセスを開始する次のページのプリフェッチャーです。これらのプリフェッチャーで発生した L1 データミスは、L2 プリフェッチャーに追加情報を伝達して、両者が協調して動作します。

L2 キャッシュはサイクルあたり 64 バイトのラインデータを 17 サイクルのレイテンシーで供給し、その帯域幅を 4 つのコアで共有します。L2 キャッシュ・サブシステムには、ストライド・アクセス・パターンを検出するストリーミング・プリフェッチャーを含む、複数のプリフェッチャーが搭載されています。追加の L2 プリフェッチャーは、より複雑なアクセスパターンを検出できます。また、これらのプリフェッチャーは、DRAM レイテンシーを低減するため、LLC にはデータを取り込みますが L2 に取り込まないようにすることができます。

単一の 4 コアモジュールの L2 キャッシュ・サブシステムでは、ファブリック上で 64 の要求と 16 の L2 データ排出 (エビクション) が発生する可能性があります。公平性を維持するため、コアごとのリザベーションにおいて競合的にコア間で共有されます。

4.1.8 インテル® AVX とインテル® AVX2 命令のサポート

Gracemont† マイクロアーキテクチャーは、インテル® AVX とインテル® AVX2 命令をサポートします。256 ビットのインテル® AVX およびインテル® AVX2 命令のほとんどは、単一の命令としてデコードされ、フロントエンド・パイプラインに単一 uop として格納されます。128 ビット・ベクトル・ユニットおよびロード・データ・パスで 256 ビット命令を実行するため、ほとんどの 256 ビット uop は MEC および FPC リザベーション・ステーションに送られる前に、アロケーション時に 2 つの独立した 128 ビット uop に分割されます。これら 2 つの独立した uop は、通常、並列に実行されるよう異なる実行ポートに割り当てられます。一般に、256 ビットの uop は、128 ビットの uop に比べて、アロケーション、実行、リタイアメントのリソースを 2 倍消費します。

ほとんどの 256 ビットのインテル® AVX2 命令は、2 つの独立した 128 ビットのマイクロオペレーションに分解できますが、クロスレーン・オペレーションと呼ばれるレーンにまたがるインテル® AVX2 命令のサブセットは、ほかの要素に属する 1 つ以上のソースを利用することでのみ要素の結果を計算できます。例えば、上位 128 ビットの結果 [255:128] の一部またはすべてが、下位要素 [127:0] の 1 つまたはすべてに依存するような場合です。これらの 256 ビットのクロスレーン命令は、同等の 256 ビットの非クロスレーン命令と比較して、長いレイテンシー、低いスループットで実行されます。

4.1.8.1 256 ビットのパーミュート操作 (並べ替え)

次に示す命令は、Gracemont† マイクロアーキテクチャー内の単独のリザベーション・ステーションでサポートされるよりも多くのオペランドソースを必要とします。2 つの uop に分解され、最初の uop は 2 サイクルでオペランドの依存性のサブセットを解決します。従属する 2 番目の uop は、単一の 128 ビット実行ポートで、5 サイクルのレイテンシーで 2 つの連続したサイクル (合計 7 サイクルのレイテンシー) で実行を完了します。

- VPERM2I128 ymm1, ymm2, ymm3/m256, imm8
- VPERM2F128 ymm1, ymm2, ymm3/m256, imm8
- VPERMPD ymm1, ymm2/m256, imm8
- VPERMPS ymm1, ymm2, ymm3/m256

- VPERMD ymm1, ymm2, ymm3/m256
- VPERMQ ymm1, ymm2/m256, imm8

4.1.8.2 128 ビット・メモリー・オペランドを持つ 256 ビット・ブロードキャスト

以下に示すメモリーバージョンのブロードキャスト命令は、128 ビット以下のメモリー・ソース・オペランドを受け入れます。ロードオペランドと 1 つの SIMD ALU uop で構成されます。同じ命令のレジスターバージョンは、2 つの SIMD ALU uop で構成されます。

操作のレイテンシーは、ロード操作のレイテンシーに加えて 1 サイクルです。

- VBROADCASTSD ymm1, m64
- VBROADCASTSS ymm1, m32

4.1.8.3 128 ビット・メモリー・オペランドを持つアップ・コンバージョン命令

以下に示すメモリーバージョンの命令は、128 ビット以下のメモリー・ソース・オペランドを受け入れます。これは 2 つの uop に分解されます。ただし、2 番目のマイクロオペレーションは、メモリーバージョンの 1 番目のマイクロオペレーションに依存しています。同じ命令のレジスターバージョンの 2 番目のマイクロオペレーションは、1 番目のマイクロオペレーションに依存することはありません。同じ命令のレジスターバージョンでは、上位 128 ビットと下位 128 ビットのセグメントを並列に実行できます。

次に示す 256 ビットの挿入、ゼロ拡張パックド移動、符号拡張命令のレイテンシーは、ロード操作のレイテンシーに加えて 2 サイクルです。

- VPMOVZX ymm1, m128/64/32
- VPMOVSX ymm1, m128/64/32
- VINSERTI128 ymm1, ymm2, m128, imm8
- VINSERTF128 ymm1, ymm2, m128, imm8

次に示すアップコンバート命令の操作のレイテンシーは、ロード操作のレイテンシーに加えて 6 サイクルです。

- VCVTPS2PD ymm1, m128
- VCVTDQ2PD ymm1, m128
- VCVTPH2PS ymm1, m128

4.1.8.4 256 ビット可変ブレンド命令

次に示す **VBLENDVPD** と **VBLENDVPS** 命令は、マイクロコード・フローとして実装されます。スループットは 4 サイクルに 1 つで、レイテンシーは 3 サイクルです。

- VBLENDVPD ymm1, ymm2, ymm3/m256, ymm4
- VBLENDVPS ymm1, ymm2, ymm3/m256, ymm4

4.1.8.5 256 ビットのベクトル TEST 命令

次に示す 256 ビット・ベクトル **TEST** 命令は 2 つの uop に分解され、それらには依存関係があります。操作の結果は、GPR 演算フラグに書き込まれます。スループットは 1 サイクルに 1 つで、レイテンシーは 7 サイクルです。

- VTESTPS ymm1, ymm2/m256
- VTESTPD ymm1, ymm2/m256
- VPTEST ymm1, ymm2/m256

4.1.8.6 GATHER 命令

VGATHER 命令がマイクロコード・フローとして実装されました。レイテンシーは、~50 サイクルです。

4.1.8.7 マスク付きロードおよびストア命令

256 ビットの VMASKMOV ロードおよびストアのスループットは 2 サイクルです。128 ビットの VMASKMOV ロードおよびストアのスループットは 1 サイクルです。マスクされた要素のメモリーアクセスが例外を引き起こすと、マスクされた要素を含むマスク付きロードおよびストアは、パフォーマンスの低下を引き起こす可能性があります。

4.1.8.8 ADX 命令

ADX 命令がサポートされます。ADCX と ADOX は、部分的な算術フラグの更新命令です。インテル® Core™ マイクロアーキテクチャーは、Intel Atom® とは異なる方法で算術フラグをリネームして追跡します。キャリーフラグ (CF)、オーバーフロー・フラグ (OF)、およびその他のフラグ (ZF、AF、PF、SF) は、Intel Atom® プロセッサーでは単一のレジスターとして保持されますが、インテル® Core™ プロセッサーでは独立したレジスターのように名前がリネームされます。ADCX / ADOX 命令間にフラグを消費しないフルフラグ更新命令がない限り、Gracemont† マイクロアーキテクチャーでは、演算フラグレジスターが両方のソースオペランドになるため、ADCX / ADOX 命令間にオペランドの依存関係が発生します。この ADCX / ADOX 命令の依存関係は、インテル® Core™ マイクロアーキテクチャーでは発生しないため、この並列性を利用するように手動でチューニングされたバイナリーが存在します。Gracemont† マイクロアーキテクチャーではその ISA をサポートしていますが、並列性は低くなります。

4.1.8.9 BMI1、BMI2、および LZCNT 命令

ビット操作を可能にする BMI1 と BMI2、および LZCNT 命令がサポートされました。

4.2 Tremont† マイクロアーキテクチャー

Tremont† マイクロアーキテクチャーは、Goldmont Plus† マイクロアーキテクチャーの成功を基に、次の拡張を提供します。

- 拡張分岐予測ユニット。
  - パスベースの条件付きおよび間接予測の改善による容量の増加。
  - 新しく搭載されたリターン・スタック・バッファー。

- 新しくクラスター化された 6 ワイドのアウトオブオーダーのフロントエンドおよびデコード・パイプライン。
  - 2 つの 16B リードが可能なバンク化された命令キャシュ。
  - サイクルごとに最大 6 つの命令のデコードを可能にする 2 つの 3 ワイド・デコード・クラスター。

- さらに深いバックエンドのアウトオブオーダー・ウィンドウ。
- 32KB データキャッシュ。
- ラージロードとストアバッファー。
- サイクルごとに 2 つのロードと 2 つのストア、または 1 つのロードと 1 つのストアが可能な 2 つの汎用ロードおよびストア実行パイプライン。
- 専用の整数およびベクトル整数/浮動小数点ストア・データ・ポート。
- 新しく改善された暗号化機能。
  - 新しいガロア体命令 (GFNI)。
  - 2 つの AES ユニット。
  - 拡張された SHA-NI 実装。
  - 高速 PCLMULQDQ 命令。

- ユーザーレベルの低電力および低レイテンシーのスピンループ命令 (UMWAIT/UMONITOR および TPAUSE) のサポート。

4.2.1 Tremont† マイクロアーキテクチャー概要

図 4-4 に Tremont† マイクロアーキテクチャーの基本パイプライン機能を示します。

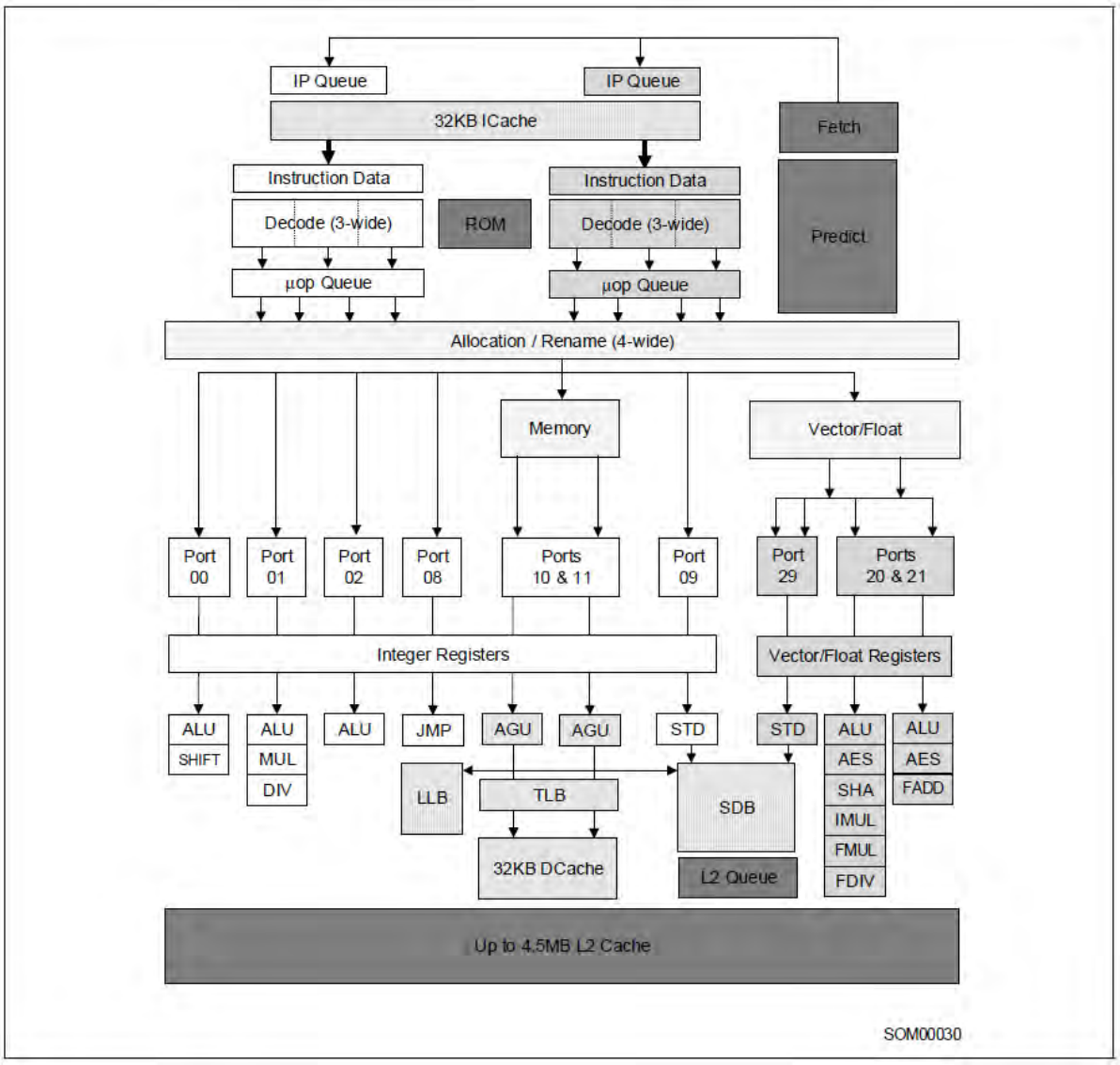


図 4-4 Tremont† マイクロアーキテクチャーのプロセッサー・コア・パイプラインの機能

Tremont† マイクロアーキテクチャーは、L3 複数スライスへのリング・インターコネクト、プロセッサー・グラフィックス、統合メモリー・コントローラー、インターコネクト・ファブリックなどを含む、多くのコンポーネントで構成される共有アンコア・サブシステムと、複数のプロセッサー・コアによる柔軟性のある統合をサポートします。

4.2.2 フロントエンド

Tremont† マイクロアーキテクチャーは、並列アウトオブオーダーの命令デコードを導入しています。命令ポインターは ITLB にアクセスし、命令キャッシュのタグ配列を参照して分岐予測器にアクセスします。分岐予測器が分岐先のターゲットを生成すると、新しいコードブロックがデコードクラスターに割り当てられます。Tremont† マイクロアーキテクチャーには 32B の予測パイプラインがあり、2 つの 3 ワイド・デコード・クラスターに供給して、サイクルごとに 6 命令をデコードできます。それぞれのクラスターは、16B/サイクルでバンク化された 32KB 命令キャッシュ

に最大 32B/サイクルでアクセスできます。ブロックあたりの命令数やデコード・レイテンシーの違いにより、コードの新しいブロックは古いブロックよりも先にデコード可能です。各デコードクラスターの最後には、デコードされた命令のキュー (uop キュー) があります。アロケーションとリネーム・パイプラインは、両方の uop キューを並列に読み取り、レジスターのリネームとリソース割り当てのため命令ストリームをインオーダーに戻します。従来の x86 方式でデコード幅を増やすと膨大なリソースが必要となり効率の低下が生じますが、クラスター化することで x86 デコードを線形的なリソースで効率の低下を最小限に抑えることができます。

クラスター化のアルゴリズムは、分岐予測器内の予測機能に依存しているため、分岐が少ない長いアセンブリー・シーケンス (浮動小数点ユニットを使用する長いアンロール済みのコードなど) は、2 つのデコードクラスターを同時に利用できないためボトルネックになる可能性があります。無条件 JMP 命令を 16 から 32 命令間隔で次のシーケンシャル命令ポインターに挿入することで、ボトルネック発生の影響を軽減できます。Tremont† マイクロアーキテクチャーは、デコードクラスターの負荷分散を行う動的なメカニズムは備えていませんが、将来の Intel Atom® プロセッサーには、このようなケースを認識して軽減するハードウェアが搭載され、アセンブリー・コードに分岐を挿入する必要はなくなります。

新しいクラスターデコード方式に加え、Tremont† マイクロアーキテクチャーでは分岐予測器が拡張され、L2 事前デコードキャッシュのサイズが Goldmont Plus† マイクロアーキテクチャーの 64KB から 128KB に倍増しています。

デコードクラスター内のマイクロアーキテクチャー・レベルの特性は、Goldmont Plus† マイクロアーキテクチャーと同じです。例えば、命令は 4 バイトを超えるプリフィクスとエスケープを回避する必要があります。詳細については、20.2.8.9 付録 F「旧世代の Intel Atom® マイクロアーキテクチャーとソフトウェアの最適化」を参照してください。

4.2.3 アウトオブオーダーと実行エンジン

Tremont† マイクロアーキテクチャーのアウトオブオーダーと実行エンジンには、次の変更があります。

- リオーダーバッファー、ロードバッファー、ストアバッファー、およびリザベーション・ステーションのサイズが大幅に増加したことで、より深い OOO 実行と高いキャッシュ帯域幅が実現されました。
- よりワイドなマシン: 8→10 実行ポート。
- 実行ポートごとの機能拡大。

表 4-2 は、異なる操作タイプをポートへディスパッチする OOO エンジンの役割をまとめています。

表 4-2 Tremont† マイクロアーキテクチャーのディスパッチ・ポートと実行スタック

Port 00 INT	Port 01 INT	Port 02 INT	Port 08 INT	Port 09 INT	Port 10	Port 11	Port 20 FP/VEC	Port 21 FP/VEC	Port 29 FP/VEC
ALU LEA[1] Shift	ALU LEA[2] Bit Ops IMUL IDIV POPCNT CRC32	ALU LEA[3]	JUMP	Store Data	Load Store Address	Load Store Address	ALU AES SHA-RND FMUL FDIV Shuffle Shift SIMUL GFNI Converts	ALU AES SHA-MSG FADD Shuffle	Store Data

注意:

1. スケーリング・インデックスのない LEA と 2 つのソース (ベース、インデックス、およびディスプレースメント入力間) のみが、ALU ポート (00、01、または 002) で 1 つの操作として実行されます。
2. 3 つのソースを持つ LEA は 2 つの操作に分割されるため、追加のレイテンシー・サイクルが生じます。インデックスを処理する部分はスケール値に関わりなくポート 02 にバインドされ、2 番目の操作ポートは 00 または 01 のどちらかにバインドされます。
3. スケーリング・インデックスを持ち、ディスプレースメントがない LEA は、ポート 02 で 1 つの操作として実行されます。

4.2.4 キャッシュとメモリー・サブシステム

Tremont† マイクロアーキテクチャーのキャッシュ階層では、次のような変更が行われています。

- L1 データ・キャッシュ・サイズを 24KB から 32KB へ 33% 増量。
- 2 x L1 ロード帯域幅: 1 つの専用ロードポートと 2 つの汎用 AGU (ロードとストアで共有)。
- 2 x L1 ストア帯域幅: 1 つの専用ストアポートと 2 つの汎用 AGU (ロードとストアで共有)。
- バッファー数が増えたことにより、多くのロードとストアを同時に処理することが可能。
- 3 サイクルの load-to-use レイテンシーを維持。
- 大きな L2 レベル TLB:
  - 512 4K エントリー → 1K 4K エントリー
  - 32 2M/4M エントリー → 64 2M/4M エントリー

- SoC の設計に応じた 1MB から 4.5MB の L2 キャッシュサイズ:
  - Snow Ridge† 製品の L2 サイズは 4.5MB、Lakefield† 製品の L2 サイズは 1.5MB。

TLB の階層は、命令キャッシュとデータキャッシュ専用の第 1 レベル TLB と、すべてのページ変換で共有される第 2 レベル TLB で構成されます。

表 4-3 Tremont† マイクロアーキテクチャーのキャッシュ・パラメーター

レベル	ページサイズ	エントリー	連想性
命令	4KB/2M/4M[1]	48	フル・アソシアチブ
第 1 レベルデータ (ロードとストア)	4KB/2M/4M[2]	32	フル・アソシアチブ
第 2 レベル	4KB	1024	4
第 2 レベル	2M/4M	64	4

注意:

1. 第 1 レベルの命令 TLB (ITLB) は、スモールページとラージページの変換をキャッシュしますが、ラージページは ITLB エントリーごとに 256KB の領域としてキャッシュされます。
2. 第 1 レベルのデータ TLB (uTLB) は、スモールページとラージページの変換をキャッシュしますが、ラージページは uTLB エントリーごとに 4KB の領域に分割されます。

4.2.5 新命令

Tremont† マイクロアーキテクチャーの新しい命令とアーキテクチャーの変更を以下に示します。実際のサポートは製品によって異なります。

- 各種暗号化アルゴリズム、エラー訂正アルゴリズム、ビット行列乗算を高速化するガロア体新命令 (Galois Field New Instructions - GFNI)。
- UMWAIT/UMONITOR/TPAUSE 命令は、ユーザーレベルのスピンループにおける省電力を有効にします。

- キャッシュラインのライトバック命令 (CLWB) により、キャッシュラインにクリーンなコピーを保ちながら、メモリーへのキャッシュラインの高速更新が可能になります。
- Tremont† マイクロアーキテクチャーのスキッドレスな PEBS 実装により、PMC0 と固定命令カウンターの両方でパフォーマンスをデバッグすることの利点がもたらされます。これは、命令や精度の高い汎用イベントのサンプリングにより、正確な分配を可能にします。PEBS はオーバーフローが通知された**後に**イベントでトリガーされるため、カウンターは大きな数 (PRIME-1) としてプログラムする必要があります。

4.2.6 Tremont† マイクロアーキテクチャーの電力管理

Tremont† マイクロアーキテクチャーは、Ice Lake† Client マイクロアーキテクチャーに備わる機能の多くをサポートします。Tremont† マイクロアーキテクチャー・ベースのプロセッサーは、インテル® スピード・シフト・テクノロジーをサポートする最初の Intel Atom® プロセッサーです。電力管理機能は SoC のニーズによって異なる場合があります。

第 5 章
SIMD アーキテクチャー向けのコーディング

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3 をサポートします。拡張版インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1 をサポートします。Nehalem† マイクロアーキテクチャー・ベースのプロセッサーは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートします。Westmere† マイクロアーキテクチャー・ベースのプロセッサーは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2、インテル® AES-NI をサポートします。Sandy Bridge† マイクロアーキテクチャー・ベースのプロセッサーは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2、インテル® AES-NI、PCLMULQDQ、インテル® AVX をサポートします。

インテル® Pentium® 4 プロセッサー、インテル® Xeon® プロセッサー、インテル® Pentium® M プロセッサーは、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジーをサポートしています。90nm テクノロジーを採用した、インテル® ハイパースレッディング・テクノロジー対応インテル® Pentium® 4 プロセッサーでは、インテル® SSE3 が導入されました。インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーは、インテル® SSE3/インテル® SSE2/インテル® SSE、インテル® MMX® テクノロジーをサポートしています。

SIMD (Single Instruction, Multiple Data) 技術により、マルチメディア、信号処理、モデリングなどの高度なアプリケーションを開発できます。

SIMD 手法は、テキスト/文字列処理、字句解析、構文解析のアプリケーションにも適用できます。これについては、第 14 章「テキスト処理/字句解析/構文解析向けのインテル® SSE4.2 と SIMD プログラミング」で説明しています。インテル® AES-NI を最適化する手法については、6.10 節で説明します。

これらの機能によって可能となる高度なパフォーマンスを十分に活用するには、以下のことを行う必要があります。

- プロセッサーが、インテル® MMX® テクノジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートしていることを確認します。
- オペレーティング・システムが、インテル® MMX® テクノロジーとインテル® SSE をサポートしていることを確認します (OS がインテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートする条件は、インテル® SSE をサポートする条件と同じです)。
- 本書で説明する最適化手法とスケジューリング手法を導入します。
- スタック・アライメントおよびデータ・アライメント手法を使用して、データの適切なアライメントを維持し、メモリーの使用効率を高めます。
- 必要に応じて、インテル® SSE とインテル® SSE2 のキャッシュ制御命令を利用します。

5.1 プロセッサーによる SIMD 技術のサポートをチェック

この節では、プロセッサーがインテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートしているかを確認する方法について説明します。

次の 3 つの方法で、アプリケーションに SIMD 技術を実装できます。

1. アプリケーションのインストール時に、SIMD 技術のサポートをチェックします。希望の SIMD 技術が使用できる場合は、適切な DLL をインストールします。
2. プログラムの実行時に SIMD 技術のサポートをチェックし、実行時に適切な DLL をインストールします。この方法は、異なるマシン上で実行されるプログラムに効果的です。

3. 複数のバージョン (SIMD 技術を使用するバージョンと使用しないバージョン) のルーチンを含む「ファット」バイナリーコードを作成します。プログラムの実行時に SIMD 技術のサポートをチェックし、適合するバージョンのルーチンを実行します。この方法は、異なるマシン上で実行されるプログラムに特に効果的です。

5.1.1 インテル® MMX® テクノロジーのサポートをチェック

インテル® MMX® テクノロジーが利用可能な場合、CPUID.01H:EDX[BIT 23] が 1 です。例 5-1 のコードを使用して、プロセッサーがインテル® MMX® テクノロジーを利用できるかどうかをテストします。

例 5-1 CPUID によるインテル® MMX® テクノロジーの識別

```
; CPUID 命令の実装を確認
…                       ;
…                       ; シグネチャーが "GENUINE INTEL" であることを確認
…                       ;
mov eax, 1              ; 機能フラグを要求
cpuid                   ; 0FH, 0A2H CPUID 命令
test edx, 00800000h     ; 機能フラグでインテル® MMX テクノロジー・ビット (ビット 23)が
                        ; 1 であることを確認
jnz Found
```

CPUID 命令の詳細については、『インテル® プロセッサーの識別と CPUID 命令』(資料番号 241618) を参照してください。

5.1.2 インテル® ストリーミング SIMD 拡張命令のサポートをチェック

プロセッサーがインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) をサポートしているかどうかをチェックする手順は、インテル® MMX® テクノロジーのチェックとよく似ています。しかし、インテル® SSE 命令を使用するアプリケーションの動作の一貫性を保証するには、オペレーティング・システム (OS) がコンテキスト・スイッチ時にインテル® SSE 状態の保存と復元をサポートしている必要があります。

システムがインテル® SSE をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサーが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサーがインテル® SSE をサポートしていることを確認します。

例 5-2 に、CPUID 機能フラグのインテル® SSE 機能ビット (ビット 25) を確認する方法を示します。

例 5-2 CPUID によるインテル® SSE の識別

```
; CPUID 命令の実装を確認
                        ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1              ; 機能フラグを要求
cpuid                   ; 0FH, 0A2H cpuid 命令
test EDX, 002000000h    ; 機能フラグのビット 25が 1 であることを確認
jnz Found
```

5.1.3 インテル® ストリーミング SIMD 拡張命令 2 のサポートをチェック

プロセッサーがインテル® SSE2 をサポートしているかをチェックする手順は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE2 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE2 をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサーが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサーがインテル® SSE2 をサポートしていることを確認します。

例 5-3 に、CPUID 機能フラグのインテル® SSE2 機能ビット (ビット 26) を確認する方法を示します。

例 5-3 CPUID によるインテル® SSE2 の識別

```
; CPUID 命令の実装を確認
                          ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1                ; 機能フラグを要求
cpuid                     ; 0FH, 0A2H CPUID 命令
test EDX, 004000000h      ; 機能フラグのビット 26 が 1 であることを確認
jnz Found
```

5.1.4 インテル® ストリーミング SIMD 拡張命令 3 のサポートをチェック

インテル® SSE3 には 13 の命令が含まれ、そのうち 11 個は SIMD 形式または x87 形式のプログラミングに適しています。プロセッサーがインテル® SSE3 をサポートしているかをチェックする手順は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE3 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE3 の x87 命令と SIMD 命令をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサーが CPUID 命令をサポートしていることを確認します。
2. CPUID の ECX 機能ビット 0 をチェックして、プロセッサーがインテル® SSE3 テクノロジーをサポートしていることを確認します。

例 5-4 に、CPUID 機能フラグのインテル® SSE3 機能ビット (ビット 0) を確認する方法を示します。

例 5-4 CPUID によるインテル® SSE3 の識別

```
; CPUID 命令の実装を確認
                          ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1                ; 機能フラグを要求
cpuid                     ; 0FH, 0A2H CPUID 命令
test ECX, 000000001h      ; 機能フラグのビット 0 が 1 であることを確認
jnz Found
```

ソフトウェアは、MONITOR と MWAIT 命令を実行する前に、これらの命令がサポートされているかどうかをチェックする必要があります。MONITOR と MWAIT 命令が利用可能であるかどうかは、例 4-4 と同様のコードシーケンスによって確認できます。これらの命令が利用可能であるかどうかは、ecx に返される値のビット 3 によって示されます。

5.1.5 インテル® ストリーミング SIMD 拡張命令 3 補足命令のサポートをチェック

プロセッサーがインテル® SSSE3 をサポートしているかをチェックする方法は、インテル® SSE のチェックとよく似ています。OS がインテル® SSSE3 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSSE3 をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサーが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサーがインテル® SSSE3 テクノロジーをサポートしていることを確認します。

例 5-5 に、CPUID 機能フラグのインテル® SSSE3 機能ビット (ビット 9) を確認する方法を示します。

例 5-5 CPUID によるインテル® SSSE3 の識別

```
; CPUID 命令の実装を確認
                        ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1              ; 機能フラグを要求
cpuid                   ; 0FH, 0A2H CPUID 命令
test ECX, 000000200h ; 機能フラグのビット 9が 1 であることを確認
jnz Found
```

5.1.6 インテル® ストリーミング SIMD 拡張命令 4.1 のサポートをチェック

プロセッサーがインテル® SSE4.1 をサポートしているかをチェックする方法は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE4.1 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE4.1 をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサーが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサーがインテル® SSE4.1 をサポートしていることを確認します。

例 5-6 に、CPUID 機能フラグのインテル® SSE4.1 機能ビット (ビット 19) を確認する方法を示します。

例 5-6 CPUID によるインテル® SSE4.1 の識別

```
; CPUID 命令の実装を確認
                        ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1              ; 機能フラグを要求
cpuid                   ; 0FH, 0A2H CPUID 命令
test ECX, 000080000h ; 機能フラグのビット 19が 1 であることを確認
jnz Found
```

5.1.7 インテル® ストリーミング SIMD 拡張命令 4.2 のサポートをチェック

プロセッサーがインテル® SSE4.2 をサポートしているかをチェックする方法は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE4.2 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE4.2 をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサーが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサーがインテル® SSE4.2 をサポートしていることを確認します。

例 5-7 に、CPUID 機能フラグのインテル® SSE4.2 機能ビット (ビット 20) を確認する方法を示します。

例 5-7 CPUID によるインテル® SSE4.2 の識別

```
; CPUID 命令の実装を確認
                        ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1              ; 機能フラグを要求
cpuid                   ; 0FH, 0A2H CPUID 命令
test ECX, 000100000h ; 機能フラグのビット 20が 1 であることを確認
jnz Found
```

5.1.8 PCLMULQDQ およびインテル® AES-NI 命令のサポートを検出

アプリケーションは、インテル® AES-NI 命令 (AESDEC/AESDECLAST/AESENC/AESENCLAST/AESIMC/AESKEYGENASSIST) を使用する前に、プロセッサーがインテル® AES-NI 拡張命令をサポートしていることをチェックする必要があります。CPUID.01H:ECX.AESNI[bit 25] = 1 である場合、インテル® AES-NI 拡張命令がサポートされています。

また、PCLMULQDQ 命令を使用する前に、アプリケーションは CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1 であるかチェックする必要があります。

インテル® SSE ステートをサポートしているオペレーティング・システムは、インテル® AES-NI 拡張命令および PCLMULQDQ 命令を使用するアプリケーションもサポートします。これは、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4 の必要条件と同じです。

例 5-8 インテル® AES-NI 命令の検出

```
; CPUID 命令の実装を確認
                        ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1              ; 機能フラグを要求
cpuid                   ; 0FH, 0A2H CPUID 命令
test ECX, 002000000h ; 機能フラグのビット 25が 1 であることを確認
jnz Found
```

例 5-9 PCLMULQDQ 命令の検出

```
; CPUID 命令の実装を確認
                        ; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1              ; 機能フラグを要求
cpuid                   ; 0FH, 0A2H CPUID 命令
test ECX, 000000002h ; 機能フラグの 1 ビットが 1 であることを確認
jnz Found
```

5.1.9 インテル® AVX 命令の検出

インテル® AVX は、256 ビットの YMM レジスターステートを処理します。アプリケーションは、図 5-1 に示す手順に従って、YMM ステートを処理する新しい拡張命令が利用できることを検出します。

アプリケーションは、インテル® AVX を使用する前に、オペレーティング・システムが XGETBV 命令と YMM レジスターステートをサポートすることに加え、プロセッサーが XSAVE/XRSTOR およびインテル® AVX 命令を使用して YMM ステートを管理できることを確認する必要があります。以下の手順により両方をチェックします (強く推奨)。

1. CPUID.1:ECX.OSXSAVE[bit 27] = 1 を検出します (アプリケーションで使用できるように XGETBV が有効になっている[8])。

[8] CPUID.01H:ECX.OSXSAVE が 1 を示す場合、プロセッサーで XSAVE、XRSTOR、XGETBV、プロセッサーによる拡張ステート・ビット・ベクトル XFEATURE_ENALBED_MASK レジスターがサポートされていることも間接的に示しています。そのため、アプリケーションは、XSAVE および OSXSAVE に対する CPUID 機能フラグのチェックを簡略化できます。XSETBV は特権命令です。

2. XGETBV を発行し、XFEATURE_ENABLED_MASK[2:1] = '11b' であることを確認します (XMM ステートおよび YMM ステートが OS で有効になっている)。
3. CPUID.1:ECX.AVX[bit 28] = 1 を検出します (AVX 命令がサポートされている)。

注意: 手順 3 は手順 1 と 2 に対して任意の順序で実行できます。

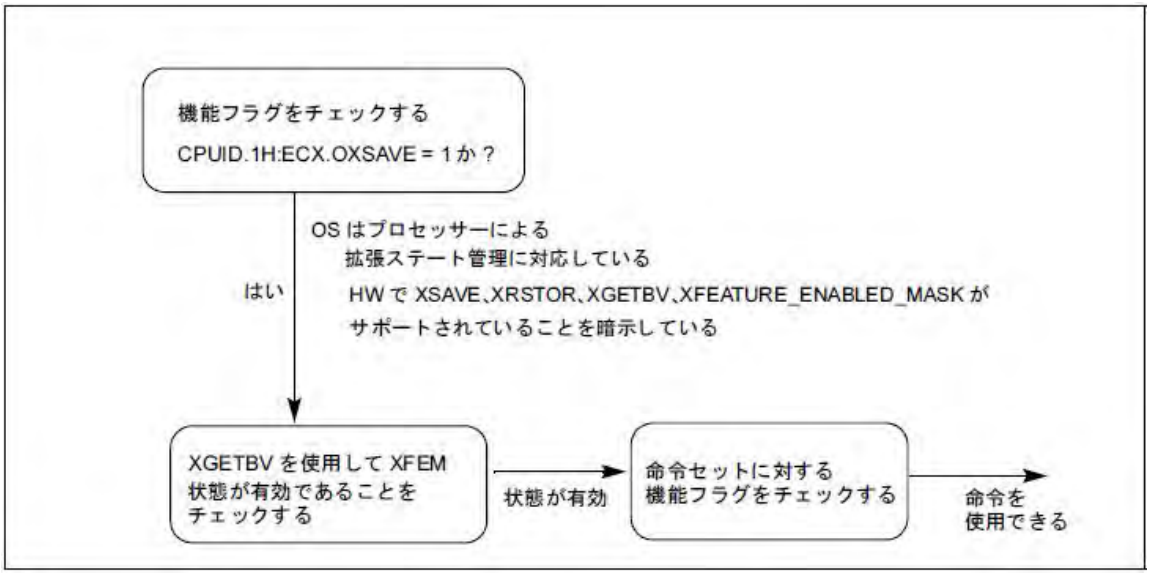


図 5-1 アプリケーションでインテル® AVX を検出する一般的な手順

以下の擬似コードは、アプリケーションでインテル® AVX を検出するときの推奨プロセスです。

例 5-10インテル® AVX 命令の検出

```
INT supports_AVX()
{ mov eax, 1
   cpuid
   and ecx, 018000000H
   cmp ecx, 018000000H ; OSXSAVE とインテル® AVX 機能フラグの両方をチェック
   jne not_supported
   ; プロセッサーはインテル® AVX 命令をサポートし OS で XGETBV が有効
   mov ecx, 0        ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
   XGETBV ; 結果は EDX:EAX
   and eax, 06H
   cmp eax, 06H      ; OS が XMM と YMM ステートをサポートしているのをチェック
   jne not_supported
   mov eax, 1
   jmp done
NOT_SUPPORTED:
   mov eax, 0
done:
```

注意: アプリケーションが CPUID.1:ECX.AVX[bit 28] に排他的に依存したり、CPUID.1:ECX.XSAVE[bit 26] に全面的に依存することは賢明とは言えません。これらは、オペレーティング・システムのサポートではなく、ハードウェアのサポートを示すためです。YMM ステート管理がオペレーティング・システムで有効になっていない場合、インテル® AVX 命令は CPUID.1:ECX.AVX[bit 28] に関わりなく #UD を生成します。"CPUID.1:ECX.XSAVE[bit 26] = 1" は、OS がステート管理で実際に XSAVE プロセスを使用することを保証するものではありません。

5.1.10 VEX エンコードされた AES および VPCLMULQDQ の検出

VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST 命令は YMM ステートを処理します。検出手順では、CPUID.1:ECX.AES[bit 25] = 1 のチェックと、インテル® AVX に対するアプリケーション・サポートの検出の手順を組み合わせる必要があります。

例 5-11 VEX エンコードされたインテル® AES-NI 命令の検出

```
INT supports_VAESNI()
{  mov eax, 1
   cpuid
   and ecx, 01A000000H
   cmp ecx, 01A000000H  ; OSXSAVE、インテル® AVX、インテル® AES-NI 機能フラグをチェック
   jne not_supported
   ; プロセッサーはインテル® AVX と VEX エンコードのインテル® AES-NI 命令をサポートし
   ; OS で XGETBV が有効
   mov ecx, 0      ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
   XGETBV          ; 結果は EDX:EAX
   and eax, 06H
   cmp eax, 06H    ; OS が XMM と YMM ステートをサポートしているのをチェック
   jne not_supported
   mov eax, 1
   jmp done
NOT_SUPPORTED:
   mov eax, 0
done:
```

同様に、VPCLMULQDQ の検出手順では、CPUID.1:ECX.PCLMULQDQ[bit 1] = 1 のチェックと、インテル® AVX に対するアプリケーション・サポートの検出手順を組み合わせる必要があります。

以下に擬似コードを示します。

例 5-12 VEX エンコードされたインテル® AES-NI 命令の検出

```
INT supports_VPCLMULQDQ)
{  mov eax, 1
   cpuid
   and ecx, 018000002H
   cmp ecx, 018000002H   ; OSXSAVE、インテル® AVX、PCLMULQDQ 機能フラグのチェック
   jne not_supported
   ; プロセッサーはインテル® AVX と VEX エンコード PCLMULQDQ 命令をサポートし
   ; OS で XGETBV が有効
   mov ecx, 0            ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
   XGETBV ; 結果は EDX:EAX
   and eax, 06H
   cmp eax, 06H       ; OS が XMM と YMM ステートをサポートしているのをチェック
   jne not_supported
   mov eax, 1
   jmp done
NOT_SUPPORTED:
   mov eax, 0
done:
```

5.1.11 F16C 命令の検出

float16 命令を使用するアプリケーションは、インテル® AVX と同様の検出手順に従って命令が利用できるか確認する必要があります。

- OS は YMM ステート管理のサポートを有効にします。
- インテル® AVX をサポートするプロセッサーは、CPUID 機能フラグの CPUID.01H:ECX.AVX[bit 28] = 1 で示されます。
- 16 ビット浮動小数点変換命令をサポートするプロセッサーは、CPUID 機能フラグ (CPUID.01H:ECX.F16C[bit 29] = 1) で確認できます。

アプリケーションは、図 5-2 の一般的な手順に従って float16 変換命令を検出します。

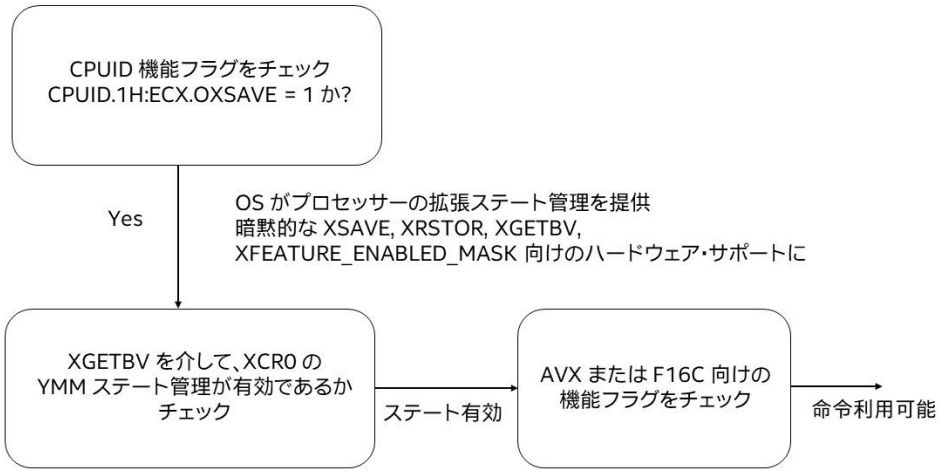


図 5-2 float 16 を検出する一般的な手順

```
INT supports_f16c()
{ ; 結果は eax に生成
    mov eax, 1
    cpuid
    and ecx, 038000000H
    cmp ecx, 038000000H    ; OSXSAVE、インテル® AVX、F16C 機能フラグのチェック
    jne not_supported
    ; プロセッサーはインテル® AVX、F16C 命令をサポートし OS で XGETBV が有効
    mov ecx, 0             ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
    XGETBV                 ; 結果は EDX:EAX
    and eax, 06H
    cmp eax, 06H           ; OS が XMM と YMM ステートをサポートしているのをチェック
    jne not_supported
    mov eax, 1
    jmp done
    NOT_SUPPORTED:
    mov eax, 0
    done:
}
```

5.1.12 FMA 命令の検出

FMA のハードウェア・サポートは、CPUID.1:ECX.FMA[bit 12]=1 によって示されます。

アプリケーション・ソフトウェアは、ハードウェアがインテル® AVX をサポートしていることを確認した後、CPUID 機能フラグ CPUID.1:ECX.FMA[bit 12] によって FMA のサポートを識別する必要があります。以下に、推奨される FMA の検出疑似シーケンスを示します。

```
INT supports_fma()
{ ; 結果は eax に生成
    mov eax, 1
    cpuid
    and ecx, 018001000H
    cmp ecx, 018001000H       ; OSXSAVE、インテル® AVX と FMA 機能フラグのチェック
    jne not_supported
    ; プロセッサーは AVX、FMA 命令をサポートし OS で XGETBV が有効
    mov ecx, 0                ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
    XGETBV                    ; 結果は EDX:EAX
    and eax, 06H
    cmp eax, 06H              ; OS が XMM と YMM ステートをサポートしているのをチェック
    jne not_supported
    mov eax, 1
    jmp done
    NOT_SUPPORTED:
    mov eax, 0
    done:
}
```

5.1.13 インテル® AVX2 の検出

ハードウェアがインテル® AVX2 をサポートするかどうかは、CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5] = 1 によって確認できます。

アプリケーション・ソフトウェアは、ハードウェアがインテル® AVX をサポートしていることを確認した後、CPUID 機能フラグ CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5] によってインテル® AVX2 のサポートを検出する必要があります。以下に、推奨されるインテル® AVX2 の検出疑似シーケンスを示します。

```
INT supports_avx2()
{ ; 結果は eax に生成
    mov eax, 1
    cpuid
    and ecx, 018000000H
    cmp ecx, 018000000H       ; OSXSAVE とインテル® AVX 機能フラグの両方をチェック
    jne not_supported
    ; プロセッサーはインテル® AVX 命令をサポートし OS で XGETBV が有効
    mov eax, 7
    mov ecx, 0
    cpuid
    and ebx, 20H
    cmp ebx, 20H               ; インテル® AVX2 機能フラグをチェック
    jne not_supported
    mov ecx, 0                 ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
    XGETBV                     ; 結果は EDX:EAX
    and eax, 06H
```

```
    cmp eax, 06H                ; OS が XMM と YMM ステートをサポートしているのをチェック
    jne not_supported
    mov eax, 1
    jmp done
    NOT_SUPPORTED:
    mov eax, 0
    done:
}
```

5.2 SIMD プログラミングにおけるコード変換に関する留意事項

インテル® VTune™ パフォーマンス拡張環境は、コードの評価とチューニングを支援するツールを提供します。これらのツールを使用する前に、次の点を確認する必要があります。

1. 現在のコードで、インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 を使用するメリットがあるか。
2. コードは整数コードか浮動小数点コードか。
3. 必要とする整数ワードサイズまたは浮動小数点精度はどれくらいか。
4. どのコーディング手法を使用すべきか。
5. いずれのガイドラインに従ったら良いか。
6. データ型をどのように配置し、アライメントするか。

図 5-3 に、既存のコードを、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 コードに変換するプロセスのフローチャートを示します。

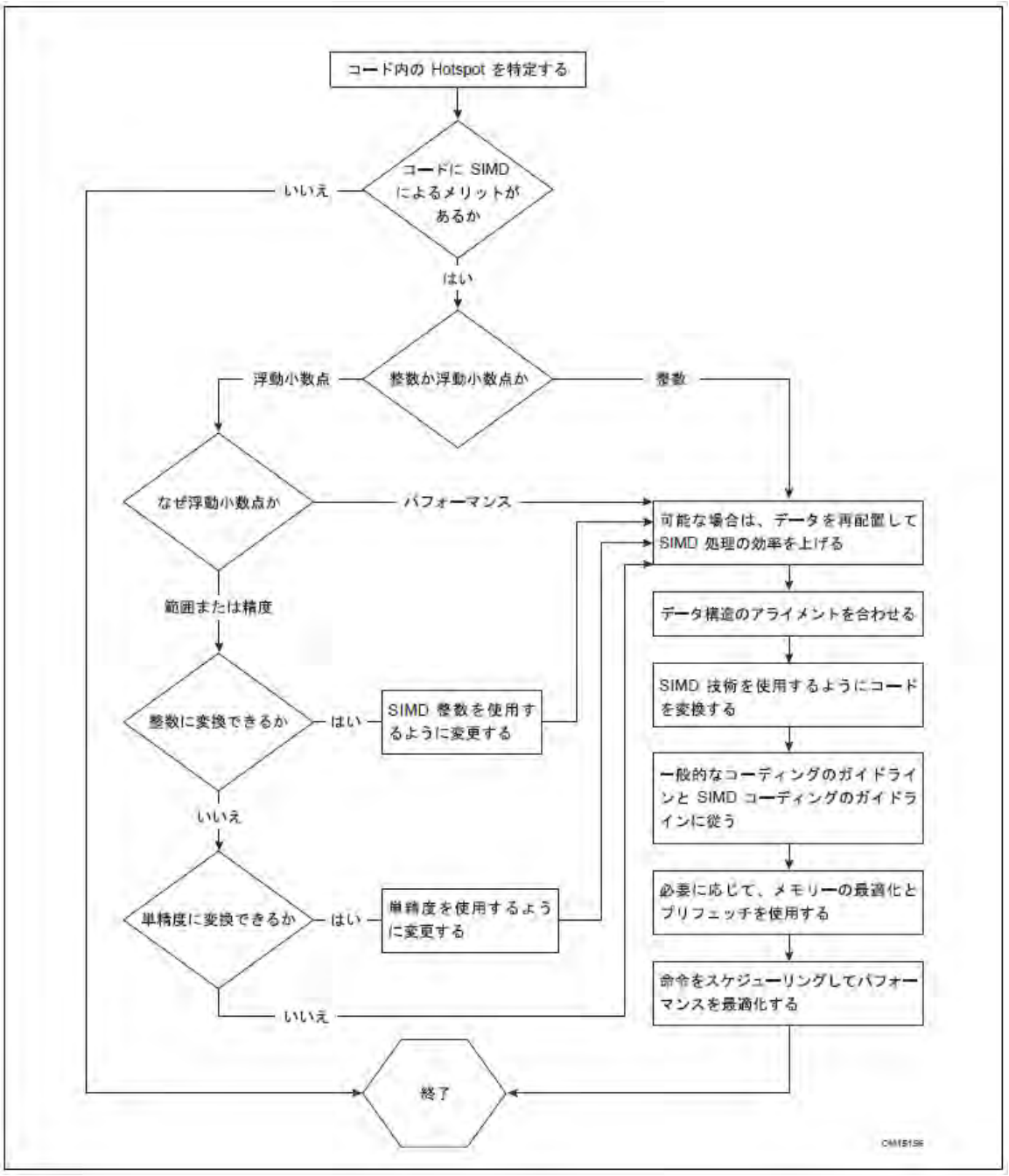


図 5-3 インテル® ストリーミング SIMD 拡張命令コードへの変換チャート

SIMD 技術の効果を最大限に発揮するには、コード内の以下の個所を特定する必要があります。

- 大量の計算を必要とする部分
- 頻繁に実行され、パフォーマンスに影響を与える部分
- ほとんどデータに依存しない制御フローを使用する部分
- 浮動小数点計算が必要な部分
- 一度に 16 バイトのデータを移動することでメリットを得られる部分
- 少ない命令を使ってコード化できる計算の部分
- キャッシュ階層を効率的に使用するために、プログラマーの指示を必要とする部分

5.2.1 ホットスポットの特定

パフォーマンスを最適化するには、インテル® VTune™ プロファイラーを使用して、計算時間の大部分を占有するコードセクションを特定します。このセクションは、ホットスポットと呼ばれます。付録 A「アプリケーション・パフォーマンス・ツール」を参照してください。

インテル® VTune™ プロファイラーは、特定のモジュールのホットスポットを検出して、大部分の CPU 時間を消費するパフォーマンス上問題があるコードセクションを識別します。ホットスポットを表示することで、大部分の CPU 時間を消費しているパフォーマンス上問題の可能性があるコード内のセクションを識別できます。

インテル® VTune™ プロファイラーは、メモリー・ロケーション、関数、クラス、またはソースファイル別にホットスポットを表示できます。ホットスポットをダブルクリックすると、そのホットスポットのソースまたはアセンブリー表示が開き、ホットスポット内の各命令のパフォーマンスに関する詳しい情報が表示されます。

インテル® VTune™ プロファイラーによって、ソースコードのすべてのレベルで、詳しい分析データとパフォーマンス・データが得られ、アセンブリー言語レベルでの助言も得られます。コードコーチは、C/C++、Fortran、Java* プログラムのパフォーマンスを向上できる個所を分析して特定し、具体的な最適化手法を提案します。必要に応じて、コードコーチは、(インテル® パフォーマンス・ライブラリーに含まれている) 高度に最適化された組込み関数と関数の使用を提案する疑似コードを表示します。インテル® VTune™ プロファイラーは、インテル® Pentium® 4 プロセッサーを含むインテル® アーキテクチャー (IA) ベースのプロセッサー向けに設計されており、IA プロセッサー上のプログラムを最適化するための詳しい手法を提案できます。詳細と最新情報については、A.1.1 節「インテル® 64 プロセッサーと IA-32 プロセッサーの推奨される最適化設定インテル® 64 プロセッサーと IA-32 プロセッサーの推奨される最適化設定」を参照してください。

5.2.2 SIMD 実行コードへの変換にメリットがあるかどうか判定

SIMD 技術を使用することでメリットが得られるコードを特定するのは、時間と困難を伴う作業です。計算集約型のアプリケーションは、SIMD コードに変換することでパフォーマンスが向上する可能性があり、これには次のものが考えられます。

- 音声圧縮アルゴリズム/フィルター
- 音声認識アルゴリズム
- ビデオ表示/キャプチャ・ルーチン
- レンダリング・ルーチン
- 3D グラフィックス (ジオメトリー)
- 画像/動画処理アルゴリズム
- 空間的 (3D) オーディオ
- 物理的モデリング (グラフィックス、CAD)
- ワークステーション・アプリケーション
- 暗号化アルゴリズム
- 複素数演算

一般に、SIMD コードへの変換に適したコードとは、8 ビット、16 ビットまたは 32 ビット整数、単精度 32 ビット浮動小数点データ、または倍精度 64 ビット浮動小数点データ (インテル® SSE2) の連続した配列を操作する、小さいサイズの反復ループを含むコードです (整数および浮動小数点データアイテムは、メモリーに連続して配置されている必要があります)。これらのコードでは、ループが繰り返されるため、アプリケーションの処理時間が長くなります。しかし、これらのルーチンは、いずれかの SIMD 技術を採用すると、パフォーマンスが大幅に向上する可能性があります。

SIMD 技術を使用できる場所が特定できたら、現在のアルゴリズムと変更後のアルゴリズムのどちらが優れたパフォーマンスを得られるか評価する必要があります。

5.3 コーディング手法

インテル® SSE4.2、インテル® SSE4.1、インテル® SSE3、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジーの SIMD 機能を利用するには、コーディング・アルゴリズムにおいて新しい手法が必要となります。その 1 つはベクトル化です。ベクトル化とは、シーケンシャルに実行される (スカラー) コードを、SIMD アーキテクチャーの並列処理を利用する、並列実行が可能なコードに変換するプロセスを指します。

この節では、アプリケーションに SIMD アーキテクチャー導入するコーディング手法について説明します。

コードをベクトル化して、SIMD アーキテクチャーを利用するには、以下の手順に従います。

- メモリーアクセスにおいて並列実行の妨げとなる依存関係があるかどうかを判断します。
- 内部ループで「ストリップマイニング」を行い、ループの反復回数を、1/(SIMD 操作の長さ) に減らします (例えば、単精度浮動小数点 SIMD 演算の場合は 1/4、16 ビット整数 SIMD 演算の場合は 1/8 にします)。
- SIMD 命令を使用してループを再構成します。

本章の各節では、上記の手順について詳しく述べていきます。また、インテル® C++ コンパイラーの自動ベクトル化機能についても説明します。

5.3.1 各種コーディング手法

ソフトウェア開発者は、アセンブリー・コードから得られるパフォーマンスの向上と、改善されたコストを比較する必要があります。対象とするプラットフォーム向けにアセンブリー言語で直接プログラミングすれば、必要とするパフォーマンスが得られる可能性は高まりますが、アセンブリー・コードにはプロセッサー・アーキテクチャー間の移植性がなく、開発と保守に大きなコストがかかります。

各種の SIMD 技術を利用することで、アセンブリー言語の代わりに高水準言語を使用して、パフォーマンス目標を達成できる可能性があります。インテル® SSE4.2、インテル® SSE4.1、インテル® SSSE3、インテル® SSE3、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジー向けに設計された、新しい C/C++ 言語拡張がこのようなコーディングを可能にします。

図 5-4 に、手作業でコーディングしたアセンブリー言語と各種のコーディング手法を比較した、パフォーマンスとプログラミングの容易さ/移植性のトレードオフを示します。

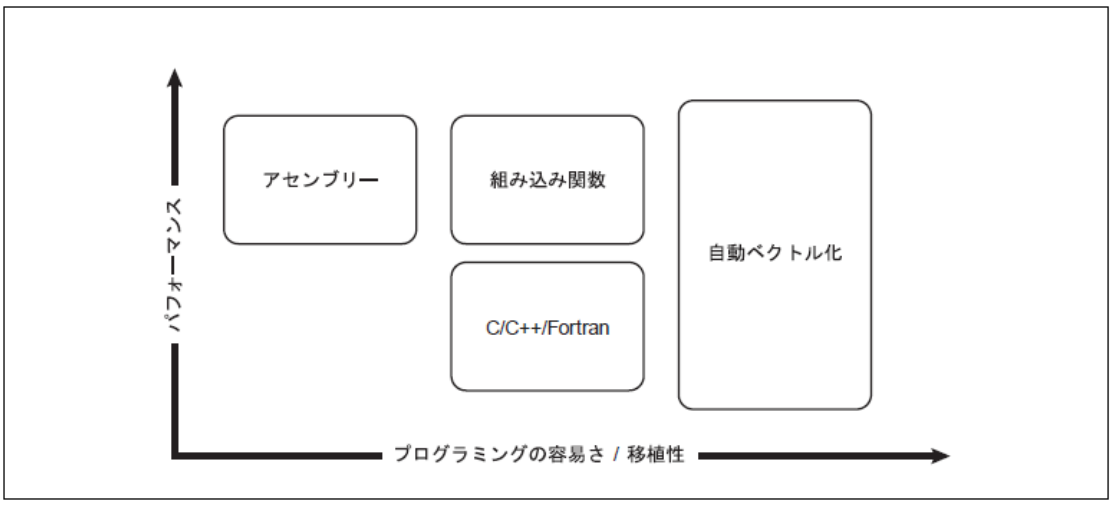


図 5-4 手作業でコーディングされたアセンブリーと高水準コンパイラーのパフォーマンスのトレードオフ

以下の例では、各種のコーディング手法を使用して、インテル® SSE のメリットが得られるアルゴリズムを表現する方法を示します。インテル® SSE4.2、インテル® SSE4.1、インテル® SSSE3、インテル® SSE3、インテル® SSE2、インテル®

SSE、インテル® MMX® テクノロジー向けに、単精度浮動小数点データ、倍精度浮動小数点データ、整数データに対して、同じ手法を適用できます。

この節で説明する使用モデルの基礎として、例 5-13 に示す簡単なループを考えます。

例 5-13 4 回反復される簡単なループ

```
void add(float *a, float *b, float *c)
{
int i;
   for (i = 0; i < 4; i++) {
       c[i] = a[i] + b[i];
   }
}
```

このループは 4 回しか反復しません。そのため、このコードはインテル® ストリーミング SIMD 拡張命令で容易に置き換えられます。

16 バイト境界へのデータ・アライメントを必要とするインテル® ストリーミング SIMD 拡張命令を効果的に使用できるように、この節のすべての例では、ルーチンに渡される配列 a, b, c は、呼び出し元ルーチンで 16 バイト境界にアライメントされているものと仮定します。アライメントを保証する手法については、インテル® Pentium® 4 プロセッサーのアプリケーション・ノートを参照してください。

以下の各項では、インライン・アセンブリー、組込み関数、C++ ベクトルクラス、自動ベクトル化の各コーディング手法について詳しく説明します。

5.3.1.1 アセンブリー

主要なループは、アセンブラーを使用するか、C/C++ コード内でインライン・アセンブリー (C-asm) を使用して、アセンブリー言語で直接コーディングできます。インテル® コンパイラーまたはアセンブラーは、新しい命令とレジスターを認識し、それに対応するコードを直接生成できます。このモデルを適切に導入すると、各種のコーディング手法の中で最も高い性能が得られる可能性があります。しかし、異なるプロセッサー・アーキテクチャー間での移植性はありません。

例 5-14 に、インライン・アセンブリーによるインテル® ストリーミング SIMD 拡張命令のコーディングを示します。

例 5-14 インライン・アセンブリーによるインテル® ストリーミング SIMD 拡張命令のコーディング

```
void add(float *a, float *b, float *c)
{
   __asm {
       mov eax, a
       mov edx, b
       mov ecx, c
       movaps xmm0, XMMWORD PTR [eax]
       addps xmm0, XMMWORD PTR [edx]
       movaps XMMWORD PTR [ecx], xmm0
   }
}
```

5.3.1.2 組込み関数

組込み関数を使用すると、アセンブリー言語の代わりに C/C++ 形式のコーディング・スタイルで命令セットを利用できます。インテル® C++ コンパイラーでは、複数の組込み関数セットを定義しており、これらの組込み関数は、それぞれインテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令、インテル® ストリーミング SIMD 拡

張命令 2 を含む最新の SIMD 拡張命令をサポートしています。組込み関数のオペランドとして、64 ビットと 128 ビットのオブジェクトを表す 4 種類の新しい C データ型 (__m64、__m128、__m128i、__m128d) が定義されています。__m64 はインテル® MMX 命令の整数 SIMD 演算に、__m128 は単精度浮動小数点 SIMD 演算に、__m128i はインテル® ストリーミング SIMD 拡張命令 2 の整数 SIMD 演算に、__m128d は倍精度浮動小数点 SIMD 演算に、それぞれ使用されます。これらのデータ型によって、プログラマーはアルゴリズムのコーディングで命令セットを直接選択でき、コンパイラーはレジスターの割り当てと命令のスケジューリングを可能な限り最適化できます。これらの組込み関数は、コンパイラーがサポートしているすべてのインテル® アーキテクチャー・ベースのプロセッサー間で移植性があります。

組込み関数を使用したプログラムのパフォーマンスは、アセンブリーを使用したプログラムに近いレベルに達します。また、組込み関数を使用すると、アセンブリーを直接記述する場合と比べて、プログラムの作成と保守のコストを大幅に軽減できます。組込み関数とその使用方法に関する詳細は、インテル® C/C++ コンパイラーのドキュメントを参照してください。

例 5-15 に、組込み関数を使用して例 5-13 のループをコーディングした例を示します。

例 5-15 組込み関数でコーディングされた 4 回反復するループ

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
   __m128 t0, t1;
   t0 = _mm_load_ps(a);
   t1 = _mm_load_ps(b);
   t0 = _mm_add_ps(t0, t1);
   _mm_store_ps(c, t0);
}
```

組込み関数は、実際のインテル® ストリーミング SIMD 拡張命令のアセンブリー・コードに 1 対 1 で対応しています。組込み関数のプロトタイプを定義している xmmintrin.h ヘッダーファイルは、インテル® C++ コンパイラーに含まれています。

インテル® MMX® テクノロジー命令セット用の組込み関数も定義されており、これらの組込み関数は、MM レジスターを表す __m64 データ型を使用します。値は、バイト、short 型整数、32 ビット値、または 64 ビット・オブジェクトを指定できます。

ただし、この組込み関数のデータ型は、ANSI C データ型ではないため、以下の制限に従って使用する必要があります。

- 組込み関数データ型は、戻り値またはパラメーターとして、代入文の左辺でのみ使用します。ほかの算術式 (“+”、”>>“ など) にこのデータ型を使用することはできません。
- 組込み関数データ型は、バイト要素/構造にアクセスするための共用体などの集合体のオブジェクトとして使用してください。__m64 オブジェクトのアドレスを指定できます。
- 組込み関数データ型のデータは、本書で説明するインテル® MMX® テクノロジーの組込み関数でしか使用できません。

ハードウェア命令の詳細については、『Intel® Architecture MMX® Technology Programmer's Reference Manual』を参照してください。さらに詳しい情報については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』を参照してください。

5.3.1.3 クラス

インテル® C++ コンパイラーでは、一連の C++ クラスライブラリーも定義されています。これによって、高水準の抽象化が可能になり、インテル® MMX® テクノロジー、ストリーミング SIMD 拡張命令、ストリーミング SIMD 拡張命令 2 および最新の SIMD 命令セットを使用して、さらに柔軟なプログラミングが可能になります。これらのクラス

は、組込み関数向けの使いやすく柔軟なインターフェイスを提供します。これにより、開発者は、特定の操作に対してどの組込み関数またはアセンブリー言語命令を使用するかを気にせずに、自然な C++ コードを記述できます。これらの C++ クラスは、組込み関数をベースにしているため、C++ クラスを使用したアプリケーションのパフォーマンスは、組込み関数を使用したアプリケーションに近いレベルに達します。C++ クラスの使用方法に関する詳細は、『SIMD 操作向けインテル® C++ クラス・ライブラリー・ユーザー・ガイド (ドキュメント番号 693500)』を参照してください。

例 5-16 に、ベクトル・クラス・ライブラリーを使用した C++ コードを示します。この例では、ルーチンに渡される配列は、16 バイトにアライメントされているものと仮定します。

例 5-16 ベクトルクラスを使用した C++ コード

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
   F32vec4 *av=(F32vec4 *) a;
   F32vec4 *bv=(F32vec4 *) b;
   F32vec4 *cv=(F32vec4 *) c;
   *cv=*av + *bv;
}
```

この例で、fvec.h はクラス定義ファイルであり、F32vec4 は 4 つの浮動小数点値の配列を表すクラスです。"+" および "=" 演算子は多重定義されており、前の例における実際のインテル® ストリーミング SIMD 拡張命令のコードは抽象化されています。このコードは元のコードによく似ていますが、より簡単かつ短時間でプログラミング可能です。

この例では、ルーチンに渡される配列は、16 バイトにアライメントされているものと仮定します。

5.3.1.4 自動ベクトル化

インテル® C++コンパイラーの最適化では、例 5-13 のようなループを自動的にベクトル化できます (つまり、インテル® ストリーミング SIMD 拡張命令コードに変換する)。コンパイラーは、プログラマーが行うのと類似した方法で、どのループが SIMD コードへの変換に適しているかを識別します。その際に、コンパイラーは、以下の条件がベクトル化の妨げになっていないかを判断します。

- ループのレイアウトとデータ構造体
- それぞれの反復の間でのデータアクセスの依存関係

コンパイラーは、このような判断を基にループをベクトル化します。これで、アプリケーションは SIMD 命令を使用できます。

ただし、自動ベクトル化を適用できるのは、特定のタイプのループだけであり、ほとんどの場合、自動ベクトル化の機能を利用するには、ユーザーがコンパイラーに情報を与える必要があります。

例 5-17 に、例 5-13 の 4 回反復されるループを自動的にベクトル化するコードを示します。

例 5-17 ループの自動的ベクトル化

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
   int i;
   for (i = 0; i < 4; i++) {
       c[i] = a[i] + b[i];
   }
}
```

インテル®C++コンパイラー (バージョン 9.0 以降) の /Qax (Windows*)、-ax (Linux* および macOS*) と /Qrestrict (Windows*)、-restrict (Linux* および macOS*) オプションを使用して、このコードをコンパイルします。

引数リストの restrict 指示子は、ポインターが指すメモリーへのエイリアスがほかにないことをコンパイラーに知らせます。つまり、使用されるポインターは、そのポインターが有効なスコープ内でそのメモリーへアクセスする唯一の方法であることを示します。コンパイラーは、restrict 指示子がなくても、ランタイムデータの依存関係テストを利用してループをベクトル化します。この場合、生成されるコードは、パラメーターのオーバーラップに基づいて、ループのシーケンシャル実行またはベクトル実行を動的に選択します (インテル® C++ コンパイラーのドキュメントを参照)。restrict 指示子を利用すると、関連するオーバーヘッドをすべて防止できます。

詳細は、インテル® C++ コンパイラーのドキュメントを参照してください。

5.4 スタックとデータ・アライメント

SIMD 技術向けに記述されたコードのパフォーマンスを最大限に発揮するには、この節で説明するガイドラインに従って、メモリー上にデータを配置する必要があります。アセンブリー・コードでアライメントされていないデータにアクセスすると、パフォーマンスが大きく低下します。

5.4.1 アライメントとデータ・アクセス・パターンの隣接性

インテル® MMX® テクノロジーで定義された 64 ビット・パックド・データ型と、インテル® ストリーミング SIMD 拡張命令およびインテル® ストリーミング SIMD 拡張命令 2 を含む最新の SIMD 命令セットで利用される 128 ビット・パックド・データ型では、データアクセスのアライメントがずれる可能性が高くなります。インテル® MMX® テクノロジー命令とインテル® ストリーミング SIMD 拡張命令を使用する場合、多くのアルゴリズムのデータ・アクセス・パターンは基本的にアライメントされていません。以下では、パディングや、配列へのデータ要素の編成など、データアクセスを改善する手法について説明します。インテル® SSE3 では、キャッシュライン分割を防止する専用命令 LDDQU が提供されています (0 節「

メモリーフィルおよびビデオフィルの帯域幅の拡大」を参照)。

5.4.1.1 パディングによるデータのアライメント

SIMD 命令を使用して SIMD データにアクセスする場合、宣言を変更するだけでデータへのアクセスを改善できます。例えば、メモリー空間内の位置と属性を表す構造体を宣言する場合を考えてみます。

```
typedef struct {short x,y,z; char a} Point;
Point pt[N];
```

SIMD ワードの 4 つの要素のうち 3 つを使用して、X, Y, Z の計算を繰り返し実行するものと仮定します。例については、5.5.1 節「データ構造体のレイアウト」を参照してください。配列 pt 内の最初の要素がアライメントされている場合でも、2 番目の要素は 7 バイト後に始まるため、アライメントが合わなくなります (各 2 バイトの short 型値 3 つ + 1 バイト = 7 バイト)。

パディング変数 pad を追加することで、この構造体のサイズは 8 バイトになります。これにより、最初の要素のアライメントが 8 バイト (64 ビット) に合っていれば、それに続くすべての要素も 8 バイトにアライメントします。次に例を示します。

```
typedef struct {short x,y,z; char a; char pad;}
Point; Point pt[N];
```

5.4.1.2 配列のデータ隣接性を保証

次のコードについて考えてみます。

```
for (i=0; i<N; i++) pt[i].y *= scale;
```

このコードでは、2 次元の y に scale 値を乗算します。for ループが配列 pt 内の各次元の y にアクセスするため、隣接するデータにアクセスできません。そのため、キャッシュミスが増え、フェッチされる各キャッシュラインの利用効率が低下し、複数のキャッシュラインにまたがるアクセスが増えるなどの理由で、アプリケーションのパフォーマンスが低下する可能性があります。

次の宣言によって、scale 変数による乗算をベクトル化し、データ・アクセス・パターンのアライメントを改善できます。

```
short ptx[N], pty[N], ptz[N];
for (i=0; i<N; i++) pty[i] *= scale;
```

SIMD 技術を使用する場合、データ編成の選択が重要です。データに対して実行される操作の種類に基づいて、データ編成を慎重に選択する必要があります。アプリケーションによっては、従来のデータ配置では十分なパフォーマンスが得られないことがあります。

この問題の典型的な例として、FIR フィルターが上げられます。FIR フィルターは、実質的には、係数タップ数の全体にわたるベクトルドット積です。

次のコードについて考えてみます。

```
(data [ j ] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of taps-1]*coeff [num of taps-1]),
```

このコードで、データ要素 i のフィルター操作がデータ要素 j から始まるベクトルドット積であるとすれば、データ要素 i+1 のフィルター操作はデータ要素 j+i から始まります。

64 ビットにアライメントされているデータベクトルと 64 ビットにアライメントされた係数ベクトルがあると仮定すると、最初のデータ要素に対するフィルター操作は完全にアライメントが合いますが、2 番目のデータ要素では、データベクトルに対するアクセスのアライメントが合わなくなります。FIR フィルターのアライメントのずれを防ぐ方法の例は、インテル® ストリーミング SIMD 拡張命令およびフィルターに関するインテルのアプリケーション・ノートを参照してください。

データ構造体の複製とパディングによって、本来アライメントが合わないアルゴリズム内のデータアクセスの問題を回避できます。5.5.1 節「データ構造体のレイアウト」では、データ構造体の編成とトレードオフについて、詳しく説明しています。

注意

複製およびパディングは、データサイズが大きくなる代わりに、アライメントのずれの問題を解決して、アライメントの合わないデータアクセスの大きなペナルティーを排除する方法です。コードを開発する際には、このトレードオフを考慮して、目的とするパフォーマンスが得られる方法を選択する必要があります。

5.4.2 128 ビット SIMD 技術向けのスタック・アライメント

インテル® ストリーミング SIMD 拡張命令、インテル® ストリーミング SIMD 拡張命令 2、および最新の SIMD 拡張命令 (インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2) で最大限のパフォーマンスを得るには、メモリーオペランドのアライメントが 16 バイト境界に合っていなければなりません。アライメントされていないデータを操作すると、大きなペナルティーが生じます。しかし、ほとんどのコンパイラーがサポートしている既存の IA-32 向けのソフトウェア規則 (STDCALL、CDECL、FASTCALL) は、特定のローカルデータと特定のパラメー

ターの 16 バイト・アライメントを保証する機構を備えていません。そのため、インテルでは、新しい __m128* データ型 (__m128、__m128d、__m128i) をサポートするアライメントに関する新しい IA-32 ソフトウェア規則を定義しました。この規則には、以下の条件があります。

- インテル® ストリーミング SIMD 拡張命令、インテル® ストリーミング SIMD 拡張命令 2、および最新の SIMD 拡張命令 (インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2) のデータを使用する関数は、16 バイトにアライメントが合ったスタックフレームを用意する必要があります。
- __m128* パラメーターは 16 バイト境界でアライメントされていなければいけません。これによって、引数ブロック内に (パディングによる)「すき間」が生じる可能性があります。

この節で説明する、インテル® C++ コンパイラーがサポートする新しい規則は、アセンブリー言語コードのためのガイドラインとしても使用できます。この節の多くでは、インテル® C++ コンパイラーで定義された、4 つの 32 ビット浮動小数点値の配列を表す __m128* データ型の使用を前提としています。

5.4.3 インテル® MMX® テクノロジー向けのデータ・アライメント

多くのコンパイラーは、変数のアライメントを調整する機能を備えています。この機能は、変数のビット長を適切な境界に揃えます。一部の変数が、指定したとおりに正しくアライメントされない場合、例 5-18 の C アルゴリズムを使用して、それらの変数のアライメント調整できます。

例 5-18 64 ビット・データ・アライメント向けの C アルゴリズム

```
/* Make newp を 64 ビット要素の 64 ビットにアラインした配列のポインターにする*/
double *p, *newp;
p = (double*)malloc (sizeof(double)*(NUM_ELEMENTS+1));
newp = (p+7) & (~0x7);
```

例 5-18 の C アルゴリズムは、64 ビット要素の配列のアライメントを 64 ビット境界に合わせます。定数の 7 は、64 ビット要素内のバイト数より 1 小さい値 (すなわち、8-1) から求められます。この方法でデータのアライメントを合わせることで、キャッシュライン境界にまたがるアクセスによって発生するパフォーマンス・ペナルティーを避けることができます。

データ・アライメントを調整するもう 1 つの方法は、64 ビット境界にアライメントされているメモリー・ロケーションに、データをコピーする方法です。データが頻繁にアクセスされる場合、これによってパフォーマンスが大きく向上する可能性があります。

5.4.4 128 ビット・データ向けのデータ・アライメント

インテル® SSE/インテル® SSE2/インテル® SSE3/インテル® SSSE3/インテル® SSE4.1/インテル® SSE4.2 が使用する 128 ビット XMM レジスターとの間でデータをロードまたはストアする場合、データのアライメントは 16 バイトである必要があります。データがアライメントされていないと、パフォーマンスが大幅に低下したり、実行中にフォルトが発生することがあります。

アライメントされていないデータを XMM レジスターとの間でコピーできる MOVE 命令 (および組込み関数) も用意されていますが、この操作はアライメントが合ったアクセスに比べはるかに時間がかかります。データが 16 バイトにアライメントされていない場合、プログラマーやコンパイラーがこれに気付かずに、アライメントが合ったデータ用の命令を使用すると、フォルトが発生します。したがって、データの 16 バイト・アライメントを常に維持する必要があります。インテル® MMX® テクノロジーの必要条件は 8 バイト・アライメントですが、データの 16 バイト・アライメントは、インテル® MMX® テクノロジー・コードにも効果があります。

以下では、インテル® C++ コンパイラーがサポートするインテル® Pentium® 4 プロセッサー向けのデータ・アライメント手法について説明します。

5.4.4.1 コンパイラーがサポートするアライメント

インテル® C++ コンパイラーは、データのアライメントを保証するため、以下の手法を用意しています。

F32vec4 または __m128 データ型によるアライメント

コンパイラーは、F32vec4 または __m128 のデータ宣言またはパラメーターを検出すると、グローバルデータ、ローカルデータ、パラメーターに対し、オブジェクトのアライメントを強制的に 16 バイト境界に合わせます。この宣言が関数の中にある場合、コンパイラーは、関数のスタックフレームのアライメントも合わせて、ローカルデータとパラメーターの 16 バイト・アライメントを保証します。デバッグコンパイルと最適化されたコンパイル (「リリース」モード) の両方でコンパイラーが生成するスタックフレームのレイアウトの詳細は、インテル® コンパイラーのドキュメントを参照してください。

__declspec(align(16)) 指定

この定義をデータ宣言の前に置くと、強制的にデータ・アライメントを 16 バイトに合わせます。この方法は、128 ビット・データ型に代入されるローカルデータまたはグローバルデータを宣言するときに便利です。構文は次のようになります。

```
__declspec(align(integer-constant))
```

integer-constant は、2 の整数乗 (32 を超えない値) です。例えば、以下のコードは、アライメントを 16 バイトにします。

```
__declspec(align(16)) float buffer[400];
```

この宣言の後、変数 buffer は、__m128 型または F32vec4 型のオブジェクトを 100 個含むものとして参照できます。次のコードでは、F32vec4 オブジェクト x は、データのアライメントが合った状態で構成されます。

```
void foo() {
    F32vec4 x = *(__m128 *) buffer;
    ...
}
```

__declspec(align(16)) の宣言がないとフォルトが発生します。

union 構造体を使用したアライメント

union と 128 ビット・データ型を組み合わせて使用すると、デフォルトでデータ構造体のアライメントを合わせるように、コンパイラーに指示ができます。できるだけ、この方法を使用することが望ましく、この union によるアライメントはプログラムの真の意図 (__declspec(align(16)) データが使用されること) をコンパイラーに通知できるため、__declspec(align(16)) による強制的なアライメントより適しています。次に例を示します。

```
union {
    float f[400];
    __m128 m[100];
} buffer;
```

この例では、union の中に __m128 型があるため、デフォルトで 16 バイト・アライメントが適用されます。__declspec(align(16)) を使用してアライメントを強制する必要はありません。

C++ では、以下のコードのように、強制的に class/struct/union 型のアライメント指示することも可能です (ただし、C ではこの手法は使用できません)。

```
struct __declspec(align(16)) my_m128
{
   float f[4];
};
```

このような class 内のデータがインテル® ストリーミング SIMD 拡張命令またはインテル® ストリーミング SIMD 拡張命令 2 で使用される場合、union を使用して、アライメントを明示的に指定する方が推奨されます。C++ では、名前なし union を使用できるため、この方法はさらに便利です。

```
class my_m128 {
    union {
        __m128 m;
        float f[4];
    };
};
```

union が無名であるため、名前 m と f を、my_m128 の直接のメンバー名として使用できます。これに対して、__declspec(align(16)) は、C と C++ のどちらでも、class、struct、または union のメンバーに指定されると無効になります。

__m64 または double データ型を使用したアライメント

状況に応じて、コンパイラーは、__m64 または double 型データを使用するルーチンのアライメントを、デフォルトで 16 バイトに合わせます。コマンドライン・オプション /Qsfalign16 (Windows* のみ) を使用して、128 ビット・データを含むルーチンだけをアライメントするようコンパイラーに指示できます。デフォルトでは、/Qsfalign8 が適用されます。このオプションは、8 バイトまたは 16 バイト・データ型を使用するルーチンだけを 16 バイトにアライメントするようコンパイラーに指示します。

詳細については、インテル® C++ コンパイラーのドキュメントを参照してください。

5.5 メモリー使用効率の改善

インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2、およびインテル® MMX® テクノロジーの組込み関数向けにデータとアルゴリズムを再配置すると、メモリーのパフォーマンスを向上できます。メモリーのパフォーマンスを向上させる方法には、以下のものがあります。

- データ構造体のレイアウト
- ベクトル化とメモリーを効率良く使用するためのストリップマイニング
- ループ・ブロッキング

また、キャッシュ制御命令、プリフェッチ、ストリーミング・ストアを使用して、メモリーの使用効率を大幅に改善できます。

関連情報: 第 9 章「キャッシュ利用の最適化」も参照してください。

5.5.1 データ構造体のレイアウト

3D 変換や照明計算などのアルゴリズムには、頂点のデータを編成する基本的な方法が 2 つあります。従来の方法は、それぞれの頂点を 1 つの構造体で表す、構造体配列 (AoS) 編成です (例 5-19)。しかし、この方法では、SIMD 技術の機能を十分に活用することができません。

例 5-19 AoS データ構造

```
typedef struct{
   float x,y,z;
   int a,b,c;
   ...
} Vertex;
Vertex Vertices[NumOfVertices];
```

SIMD 技術を使用するコードに適した処理方法は、各座標を表す配列でデータを編成することです (例 5-20 を参照)。このデータ編成は、配列構造体 (SoA) と呼ばれます。

例 5-20 SoA データ構造

```
typedef struct{
   float x[NumOfVertices];
   float y[NumOfVertices];
   float z[NumOfVertices];
   int a[NumOfVertices];
   int b[NumOfVertices];
   int c[NumOfVertices];
   ...
} VerticesList;
VerticesList Vertices;
```

AoS 形式のデータを計算するには、2 つの方法があります。1 つは元の AoS 形式のままでデータを操作する方法で、もう 1 つはデータを入れ替えて、動的に SoA 形式に再編成する方法です。例 5-21 に、ドット積計算における 2 つの方法のコード例を示します。

例 5-21 AoS コードと SoA コードの例

```
; ベクトルの配列 (Array) と固定ベクトル (Fixed) の内積は、3D 光源処理の一般的な計算です。
; Array = (x0,y0,z0),(x1,y1,z1),... で、Fixed = (xF,yF,zF) とするとき、
; 内積はスカラー量 d0 = x0*xF + y0*yF + z0*zF として定義されます。
;
; 構造体の配列 (AoS) コード
; DC と記されている値は、"don't-care." の意味です。

; AoS モデルでは、頂点データは xyz フォーマットになります。
movaps xmm0, Array ; xmm0 = DC, x0, y0, z0
movaps xmm1, Fixed ; xmm1 = DC, xF, yF, zF
mulps xmm0, xmm1 ; xmm0 = DC, x0*xF, y0*yF, z0*zF
movhlps xmm, xmm0 ; xmm = DC, DC, DC, x0*xF

addps xmm1, xmm0 ; xmm0 = DC, DC, DC,
                 ; x0*xF+z0*zFmovaps xmm2, xmm1
shufps xmm2, xmm2,55h ; xmm2 = DC, DC, DC, y0*yF
addps xmm2, xmm1 ; xmm1 = DC, DC, DC,
                 ; x0*xF+y0*yF+z0*zF

; 配列の構造体 (SoA) コード
; X = x0,x1,x2,x3
; Y = y0,y1,y2,y3
; Z = z0,z1,z2,z3
; A = xF,xF,xF,xF
; B = yF,yF,yF,yF
; C = zF,zF,zF,zF
movaps xmm0, X ; xmm0 = x0,x1,x2,x3
movaps xmm1, Y ; xmm0 = y0,y1,y2,y3
movaps xmm2, Z ; xmm0 = z0,z1,z2,z3
```

```
mulps xmm0, A ; xmm0 = x0*xF, x1*xF, x2*xF, x3*xF
mulps xmm1, B ; xmm1 = y0*yF, y1*yF, y2*yF, y3*xF
mulps xmm2, C ; xmm2 = z0*zF, z1*zF, z2*zF, z3*zF
addps xmm0, xmm1
addps xmm0, xmm2 ; xmm0 = (x0*xF+y0*yF+z0*zF), ...
```

元の AoS 形式に対して SIMD 演算を実行すると、計算の回数が多くなり、SIMD 要素の一部の利点を活用できない計算が行われます。一般的に、この方法は効率的ではありません。

AoS 形式のデータ計算に推奨される方法は、要素の各セットを SoA 形式に再編成 (スウィズリング) してから、SIMD 技術を使用してそのデータを処理することです。データの入れ替えは、プログラムの実行時に動的に行うことも、データ構造体の生成時に静的に行うこともできます。例については、第 6 章と第 7 章を参照してください。通常は、元の AoS を使用するより、データを動的に入れ替える方が適切です。ただし、データを動的に編成すると、計算時の命令の数が増えるため、実行効率が多少低下します。これに対して、データ構造体がレイアウトされるときにデータを静的に入れ替えれば、実行時のオーバーヘッドは生じません。したがって、これが最も良い方法といえます。

すでに説明したように、SoA 配置の方が、SIMD 技術の並列処理を効率的に使用できます。これは、計算でアクセスするデータが垂直形式で準備されるためです。つまり、4 つの SIMD 実行スロットを使用して、要素 x0, x1, x2, x3 に xF, xF, xF, xF を掛けて、4 つの結果を求めることができます。これに対して、AoS データを直接計算すると、水平的な操作が行われ、SIMD 実行スロットを消費しますが、1 つのスカラー結果しか得られません (このことは、例 5-21 の多くが “don't-care” (DC) スロットであることによって示されています)。

データ構造体に SoA 形式を使用すると、キャッシュと帯域幅の使用効率も向上します。構造体の各要素に対するアクセス頻度が異なる場合 (例えば、要素 x, y, z がほかのエントリーの 10 倍の頻度でアクセスされる場合) は、SoA を使用すると、メモリーを節約でき、不要なデータアイテム a, b, c のフェッチを避けることができます。

例 5-22 ハイブリッド SoA データ構造体

```
NumOfGroups = NumOfVertices/SIMDwidth
typedef struct{
   float x[SIMDwidth];
   float y[SIMDwidth];
   float z[SIMDwidth];
} VerticesCoordList;
typedef struct{
   int a[SIMDwidth];
   int b[SIMDwidth];
   int c[SIMDwidth];
...
} VerticesColorList;
VerticesCoordList VerticesCoord[NumOfGroups];
VerticesColorList VerticesColor[NumOfGroups];
```

ただし、SoA には、独立したメモリーストリームの参照の回数が増える欠点があります。配列 x,y,z を使用する計算は、3 つの別々のデータストリームを必要とします (例 4-20 を参照)。

そのため、プリフェッチとアドレス生成計算の数が増え、DRAM ページアクセスの効率も大きな影響を受けます。

これに対して、ハイブリッド SoA 手法は、AoS と SoA の 2 つの手法を組み合わせたものです (例 5-22 を参照)。この例の場合、別々のアドレスストリームが 2 つ生成され、参照されます。1 つのストリームは xxxx, yyyy, zzzz, zzzz, ... で構成され、もう 1 つのストリームは aaaa, bbbb, cccc, aaaa, dddd, ... で構成されます。変数 x、y、z が常に一緒に使用されるとすれば (変数 a, b, c も一緒に使用されますが、変数 x、y、z と同時に使用されることはありません)、この方法によって、不要なデータのプリフェッチを排除できます。

ハイブリッド SoA 手法には、次のような利点があります。

- 垂直的な SIMD 計算を効率的に実行できるようにデータが構成される。
- AoS よりアドレス生成が簡単になり、アドレス生成の回数も少なくなる。
- ストリーム数が少ないため、DRAM ページミスが減少する。
- ストリーム数が少ないため、プリフェッチの回数が少なくなる。
- 同時に使用されるデータ要素が、キャッシュラインに効率良く格納される。

SIMD 技術の出現により、データ編成の選択が重要となりました。データに対して実行される操作の種類に基づいて、データ編成を慎重に選択する必要があります。インテル® Pentium® 4 プロセッサーおよび将来のプロセッサーでは、この重要性はますます高まります。アプリケーションによっては、従来のデータ配置では十分なパフォーマンスが得られないことがあります。アプリケーション開発者は、計算を効率良く行えるように、さまざまなデータ配置およびデータセグメントの形式を検討する必要があります。これには、特定のアプリケーションで AoS、SoA、ハイブリッド SoA を組み合わせて利用することも含まれます。

5.5.2 ストリップマイニング

ループのセクション化として知られるストリップマイニングは、メモリーのパフォーマンスを改善する手段で、ループの SIMD エンコーディングを可能にするループ変換テクニックです。この手法は、最初はベクトライザーのために導入されたもので、特定のベクトルマシン上の最大ベクトル長またはそれ以下のサイズのデータに対して各ベクトル操作が行うコードを生成できます。大きなループをより小さなセグメントやストリップに分割すると、ループ構造を変換して次のことを実現できます。

- データがアルゴリズムの複数のパスで再利用される場合、データキャッシュの時間と空間の局所性を向上させます。
- ループの反復回数を、1/(各ベクトルの長さ) に減らします (ベクトル長は、1 回の SIMD 操作で実行される操作の数に相当する)。インテル® ストリーミング SIMD 拡張命令の場合、ベクトル長 (ストリップ長) は 4 であるため、1 回のインテル® SSE 単精度浮動小数点 SIMD 操作あたり 4 つの浮動小数点データアイテムが処理されます。例えば、例 5-23 の場合を考えてみます。

例 5-23 ストリップマイニング前の疑似コード

```
typedef struct _VERTEX {
   float x, y, z, nx, ny, nz, u, v;
} Vertex_rec;
main()
{
   Vertex_rec v[Num];
   ....
   for (i=0; i<Num; i++) {
       Transform(v[i]);
   }
   for (i=0; i<Num; i++) {
       Lighting(v[i]);
   }
   ....
}
```

メインループは、変換 (Transform) と照明 (Lighting) 計算の 2 つの関数で構成されています。このメインループは、各オブジェクトに対し、変換ルーチンを呼び出して一部のデータを更新し、次に照明計算ルーチンを呼び出してさらにデータを処理しています。配列 v[Num] のサイズがキャッシュより大きい場合は、Transform(v[i]) の実行時にキャッシュに格納された v[i] の座標は、Lighting(v[i]) を実行する前にキャッシュから排出されてしまいます。そのため、v[i] をメインメモリーから 2 回フェッチすることになり、パフォーマンスが低下します。

例 5-24 では、ループのストリップマイニングが行われ、strip_size サイズに分割されています。strip_size の値は、配列 v[Num] の strip_size の要素がキャッシュ階層に納まるように選択されます。これによって、

Transform(v[i]) によってキャッシュに格納された要素 v[i] は、Lighting(v[i]) を実行する時点でまだキャッシュに残っています。したがって、ストリップマイニングが行われていないコードと比べて、パフォーマンスが向上します。

例 5-24 ストリップマイニング後のコード

```
MAIN()
{
   Vertex_rec v[Num];
   ....
   for (i=0; i < Num; i+=strip_size) {
       FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
       TRANSFORM(V[J]);
       }
       FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
       LIGHTING(V[J]);
       }
   }
}
```

5.5.3 ループ・ブロッキング

ループ・ブロッキングは、メモリー・パフォーマンスを最適化するもう 1 つの有効な手法です。ループ・ブロッキングの主な目的は、ストリップマイニングと同様に、キャッシュミスをできるだけ減らすことです。この手法は、メモリー領域全体をシーケンシャルに横断するのではなく、特定のメモリー領域を小さいブロックに変換します。各ブロックは、データを最大限に再利用できるように、特定の計算に使用されるすべてのデータがキャッシュに納まる程度に小さくなります。ループ・ブロッキングを、2 次元またはそれ以上の次元におけるストリップマイニングと見なすことができます。例えば、例 5-23 のコードと図 5-5 のアクセスパターンについて考えてみます。2 次元配列 A は、まず j (列) 方向に、その後、i (行) 方向に参照されます (列優先順)。配列 B は逆の順序で参照されます (行優先順)。メモリー配置が列優先順であると仮定します。したがって、このコードの配列 A と B のアクセスのストライドは、それぞれ、1 と MAX になります。

例 5-25 ループ・ブロッキング

```
A. オリジナルループ
float A[MAX, MAX], B[MAX, MAX]
for (i=0; i< MAX; i++) {
   for (j=0; j< MAX; j++) {
       A[i,j] = A[i,j] + B[j, i];
   }
}

B. ブロッキング後の変換されたループ
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
   for (j=0; j< MAX; j+=block_size) {
       for (ii=i; ii<i+block_size; ii++) {
           for (jj=j; jj<j+block_size; jj++) {
               A[ii,jj] = A[ii,jj] + B[jj, ii];
           }
       }
   }
}
```

内側ループの最初の反復では、配列 B にアクセスするたびにキャッシュミスが発生します。配列 A の 1 行のサイズ (つまり A[2][0:MAX-1]) が十分に大きい場合、2 回目の反復が始まるまでに、配列 B にアクセスするたびに必ずキャッシュミスが発生します。例えば、float 型の変数は 4 バイト、各キャッシュラインは 32 バイトであるため、最初の反復で、B[0][0] が参照されると、B[0][0:7] を含むキャッシュラインがキャッシュに格納されます。キャッシュの容量には制限があるため、このラインは、内側ループが終わりに達する前に、競合ミスのために排出されます。外側

ループの次の反復では、B[0][1] を参照すると、次のキャッシュミスが発生します。このように、配列 B の各要素が参照されるたびに、1 回のキャッシュミスが起こります。つまり、配列 B については、キャッシュ内のデータは全く再利用されていません。

キャッシュのサイズを考慮してループをブロック化すれば、この状態を避けることができます。図 5-5 では、ループ・ブロッキング係数として block_size が選択されています。block_size が 8 の場合、各配列の変換後のブロックのサイズは 8 キャッシュラインになります (各ラインが 32 バイトに相当)。内側ループの最初の反復では、A[0][0:7] と B[0][0:7] がキャッシュに取り込まれます。B[0][0:7] は、外側ループの最初の反復ですべて参照されます。したがって、B[0][0:7] のキャッシュミスは、元のアルゴリズムでは 8 回発生していましたが、ループ・ブロッキングの最適化を適用した後のアルゴリズムでは 1 回しか発生しません。図 5-5 に示すように、配列 A と配列 B は、A と B の変換後の 2 つのブロックの合計サイズがキャッシュのサイズより小さくなるように、小さな矩形のブロックに変換されます。これによって、データを最大限に再利用できます。

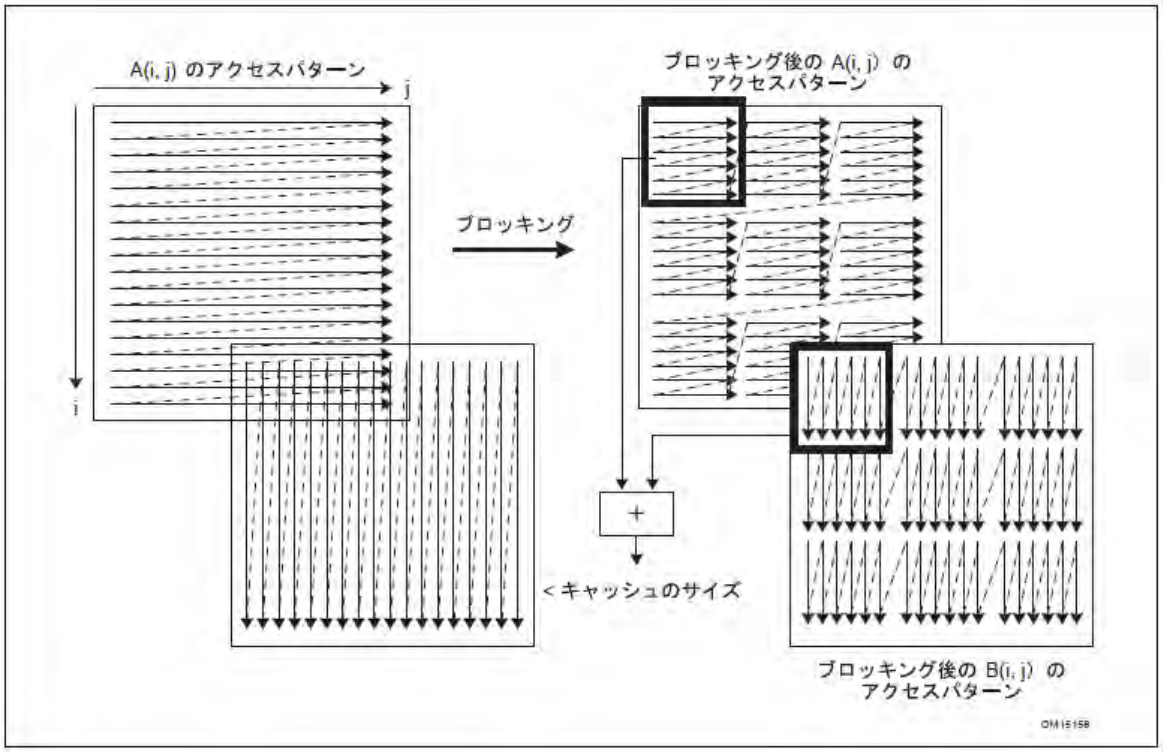


図 5-5 ループ・ブロッキングのアクセスパターン

このように、ループ・ブロッキング手法を適用すると、すべての冗長なキャッシュミスを排除できます。MAX が非常に大きい場合は、ループ・ブロッキングによって、DTLB (データ・トランスレーション・ルックアサイド・バッファー) のミスによるペナルティーも軽減できます。また、この最適化手法は、キャッシュとメモリーのパフォーマンスを向上させるだけでなく、外部バス帯域幅の節約にもなります。

5.6 命令の選択

この節では、タスクを実行する命令を選択するガイドラインについて説明します。

SIMD 計算の障害の 1 つは、データに依存する分岐の存在です。条件付き移動を使用することで、データに依存する分岐を排除できます。例 5-26 に示すように、マスクされた比較と論理演算子を使用して、SIMD 計算で条件付き移動をエミュレートします。インテル® SSE4.1 では、ループ内のデータ依存分岐をベクトル化可能にするパックド・ブレンディング命令が提供されています。

例 5-26 条件付き移動のエミュレーション

```
高レベルなコード:
__declspec(align(16)) short A[MAX_ELEMENT], B[MAX_ELEMENT], C[MAX_ELEMENT],
D[MAX_ELEMENT],
E[MAX_ELEMENT];
for (i=0; i<MAX_ELEMENT; i++) {
   if (A[i] > B[i]) {
       C[i] = D[i];
   } else {
       C[i] = E[i];
   }
}
インテル® MMX アセンブリー・コードは反復ごとに 4 つの short を処理:
   xor eax, eax
top_of_loop:
   movq mm0, [A + eax]
   pcmpgtwxmm0, [B + eax]       ; 比較マスクを生成
   movq mm1, [D + eax]
   pand mm1, mm0               ; A<B の要素をドロップ
   pandn mm0, [E + eax]        ; A>B の要素をドロップ
   por mm0, mm1                ; 単一ワードを生成
   movq [C + eax], mm0
   add eax, 8
   cmp eax, MAX_ELEMENT*2
   jle top_of_loop
インテル® SSE4.1 アセンブリー・コードは反復ごとに 8 つの short を処理:
   xor eax, eax
top_of_loop:
   movdqq xmm0, [A + eax]
   pcmpgtwxmm0, [B + eax]       ; 比較マスクを生成
   movdqa xmm1, [E + eax]
   pblendv xmm1, [D + eax], xmm0 ;
   movdqa [C + eax], xmm1       ;
   add eax, 16
   cmp eax, MAX_ELEMENT*2
   jle top_of_loop
```

レジスターの 1 つのインスタンスを複数の命令が参照する場合は、それらの命令をできるだけ近くにまとめます。ただし、データを参照する側の命令を、データを生成する側の命令の近くにスケジューリングしてはいけません。

5.7 開発の最終段階におけるアプリケーションのチューニング

アプリケーションが正常に動作することを確認した後に、アプリケーションをチューニングする最も良い方法は、システム上でアプリケーションを実行しながら、プロファイラーを使用してアプリケーションのパフォーマンスを測定することです。インテル® VTune™ プロファイラーは、パフォーマンスを向上させるためアプリケーションのどこを修正したら良いかを特定するのに役立ちます。インテル® VTune™ プロファイラーは、パフォーマンスを最適化する各種の工程で使用できます。詳細は、A.3.1 節「インテル® VTune™ プロファイラー」を参照してください。最適化手法を適用するたびに、パフォーマンスがどの程度向上したかをチェックして、どこを最適化すればパフォーマンスが大きく向上するかを確認すると良いでしょう。

第 6 章
SIMD 整数アプリケーション向けの最適化

SIMD 整数命令により、整数演算を多用する各種アプリケーションのパフォーマンスが改善でき、SIMD アーキテクチャーを有効に活用できるようになります。

第 3 章で説明したガイドラインに加え、さらに SIMD 整数命令の使用方法に関する本章のガイドラインを使用して、すべてのプロセッサー世代に対応した、高速で効率的なコードを開発できます。

インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、およびインテル® SSE4.2 の PCMPEQQ それぞれに対応した 64 ビットと 128 ビットの SIMD 整数命令は、まとめて「SIMD 整数命令」と呼ばれます。

本章のコードシーケンスは、基本的な 64 ビット SIMD 整数命令と、より効率的な 128 ビット SIMD 整数命令の使用例を示しています。

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3 をサポートします。拡張版インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーは、インテル® SSE4.1 とそれ以前のすべての世代の SIMD 整数命令をサポートしています。Nehalem† マイクロアーキテクチャー・ベースのプロセッサーは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートします。

SIMD 手法は、テキスト/文字列処理、字句解析、構文解析のアプリケーションにも適用できます。テキスト/文字列処理および字句解析のアプリケーションにおける SIMD プログラミングは通常、SIMD 整数プログラミングで一般的に使用される手法の域を超えた高度な手法を必要とします。これについては、第 14 章「テキスト処理/字句解析/構文解析向けのインテル® SSE4.2 と SIMD プログラミング」で説明しています。

インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャーの 128 ビット SIMD 整数命令の実行は、従来のマイクロアーキテクチャーよりもはるかに効率良くなりました。インテル® SSE4.1 で導入された新しい SIMD 機能は 128 ビット・オペランドを処理し、同等の 64 ビット SIMD 機能は導入されていません。64 ビット SIMD 整数コードから 128 ビット SIMD 整数コードに変換することを強く推奨します。

本章には、アプリケーションのコーディングを始める際に役立つ具体例を掲載しています。頻繁に用いられる単純な低レベルの演算をいくつか提供することが目的です。いずれの具体例も、最新世代のインテル® 64 プロセッサーと IA-32 プロセッサーで最高のパフォーマンスを得るため最小の命令数で記述されています。

具体例にはそれぞれ短い説明とサンプルコードを含めたほか、必要に応じていくつか注釈を加えています。これらの具体例は、より長いコードシーケンスに組込む場合のスケジューリングについては考慮されていません。

SIMD 整数命令の使用を検討している場合は、5.1.3 節を参照してください。

6.1 SIMD 整数コードに関する一般的な規則

以下に一般的な規則と改善案を示します。

- 64 ビット SIMD 整数命令と x87 浮動小数点命令が混在しないようにします。6.2 節「SIMD 整数と x87 浮動小数点との併用」を参照してください。すべての SIMD 整数命令は、ペナルティーなしで混在させることが可能であることに注意してください。
- 64 ビット SIMD 整数コードよりも 128 ビット SIMD 整数コードを優先します。インテル® Core™ マイクロアーキテクチャー以前のマイクロアーキテクチャーでは、大部分の 128 ビット SIMD 命令が、実行エンジン中

の 64 ビット・データ・パスが原因で、スループットが 2 サイクルになるという制限があります。インテル® Core™ マイクロアーキテクチャーでは、大部分の SIMD 命令 (シャッフル/パック/アンパック操作を除く) が 1 サイクルのスループットで実行されます。また、3 つのポートが用意されているので、複数の SIMD 命令を並列に実行できます。拡張版インテル® Core™ マイクロアーキテクチャーでは、128 ビットのシャッフル/パック/アンパック操作をそれぞれ 1 サイクルのスループットで実行できるように高速化が図られています。

- 整数データと浮動小数点データの両方を処理する SIMD コードを記述するときは、SIMD 変換命令かロード/ストア命令のサブセットを使用して、必ず正しく定義されたデータ型が XMM レジスターの入力オペランドに含まれるようにし、命令との整合がとれるようにします。
  コードシーケンスにクロスタイプの用法が含まれても、異なる実装で同じ結果が得られますが、大幅なパフォーマンス低下を招きます。型の一致しない SIMD データが XMM レジスターに格納されている場合、そのデータをインテル® SSE/インテル® SSE2/インテル® SSE3/インテル® SSSE3/インテル® SSE4.1/インテル® SSE4.2 命令を使って処理することはできる限り避けます。
- 第 3 章と第 4 章で説明した最適化の規則とガイドラインを適用します。
- 可能な場合はハードウェア・プリフェッチを利用します。PREFETCH 命令の使用は、データ・アクセス・パターンが不規則で、プリフェッチ距離を事前に特定できる場合だけに留めます。第 9 章「キャッシュ利用の最適化」も参照してください。
- 条件分岐命令を使用する代わりに、ブレンド命令、マスクによる比較命令および論理命令を使用して、条件付き移動命令をエミュレートします。

6.2 SIMD 整数と x87 浮動小数点との併用

すべての 64 ビット SIMD 整数命令は、x87 浮動小数点スタックとレジスター状態を共有する MMX レジスターを使用します。この共有により、一定の規則に従う必要があり、また考慮すべき点もいくつかあります。MMX レジスターを使用する命令は、自由に x87 浮動小数点レジスターと混在させて使用することはできません。機能の正当性を保証するため、64 ビット SIMD 整数命令と x87 浮動小数点命令を相互に切り替える場合は注意が必要です。6.2.1 節を参照してください。

6.2.1 節と 6.2.2 節は、インテル® MMX® 命令を使用するソフトウェアにのみ適用されます。高いパフォーマンスを得るには、前述のように、MMX コードの代わりに 128 ビット SIMD 整数命令を優先すべきです。これにより、EMMS を使用する必要性や、インテル® MMX® 命令と x87 命令を混在させたときに生じる切り換えによるパフォーマンスの低下も回避できます。

パフォーマンス上の考慮事項として、SIMD 浮動小数点演算、128 ビット SIMD 整数演算、および x87 浮動小数点演算を混在させてもペナルティーは生じません。

6.2.1 EMMS 命令の使用

64 ビット SIMD 整数コードの生成に際しては、8 つの x87 浮動小数点レジスター・スタックに MMX レジスターの別名が割り当てられていることに注意してください。インテル® MMX® 命令から x87 浮動小数点命令へ切り替えるとある程度の遅延が生じるため、これらの命令間の切り替えはできる限り避けることが望まれます。しかし、切り替える場合は、EMMS 命令を使用することで、後続の x87 コードの処理が正しく実行できるように効率良く x87 スタックをクリアできます。

命令の種類にかかわらず、MMX レジスターへの参照を行うと、x87 浮動小数点のタグワード内の有効なビットがすべてセットされます (x87 レジスターが有効な値を含むことを示す)。ソフトウェアの処理を正しく実行するため、MMX レジスター上での演算の後、x87 浮動小数点演算を開始する前に x87 浮動小数点スタックを空にする必要があります。

EMMS 命令を使用すると、有効なビットがすべてクリアされ、x87 浮動小数点スタックが効率良く空にされ、次の新たな x87 浮動小数点演算の準備が整います。EMMS 命令を使用すると、MMX レジスター上での演算と x87 浮動小数点スタック上での演算を交互に円滑に切り替えることができます。インテル® Pentium® 4 プロセッサーで EMMS 命令を使用するとオーバーヘッドが生じます。

EMMS 命令 (または_mm_empty( ) 組込み関数) を使用せずに MMX レジスター上での演算と x87 浮動小数点レジスター上での演算を交互に切り替えると、予期しない結果となります。

注意

インテル® MMX® 命令のいずれかを使用した後、FP 命令用のタグワードのリセットに失敗すると、実行が不完全に終わったり、パフォーマンスが低下する恐れがあります。

6.2.2 EMMS 命令を使用するガイドライン

x87 浮動小数点命令と 64 ビット SIMD 整数命令の両方を使用してコードを開発する場合、以下の手順に従います。

1. コードを x87 浮動小数点コードへ切り替えるときは必ず、64 ビット SIMD 整数コードの末尾で EMMS 命令を実行します。
2. x87 浮動小数点命令の実行時に x87 浮動小数点スタックのオーバーフロー例外が発生しないように、すべての 64 ビット SIMD 整数コードセグメントの末尾にも EMMS 命令を挿入します。

浮動小数点命令と 64 ビット SIMD 整数命令の両方を使用するアプリケーションを作成する場合、どのような状況で EMMS 命令を使用するかは、以下のガイドラインに従って判断します。

- **次の命令が x87 FP である場合** — 次の命令が x87 FP 命令である場合、64 ビット SIMD 整数命令の後に _mm_empty( ) を使用します。例えば、float、double、long double の計算を実行するよりも前に記述します。
- **すでに空になっているとき、さらに空にしないこと** — 次の命令が MMX レジスターを使用する命令である場合は、_mm_empty( ) を使用しても無駄になるだけで何の利点もありません。
- **命令のグループ化** — 64 ビット SIMD 整数命令を使用している領域から、x87 FP 命令を使用している領域を分断します。これにより、性能に大きな影響を及ぼすループの中で、EMMS 命令を使用する必要がなくなります。
- **ランタイム初期化** — __m64 および x87 FP の両方のデータ型のランタイム初期化を実行している最中に _mm_empty( ) を使用します。これにより、データ型が切り替わるときに確実に対応するレジスターがリセットされます。コーディングの例は、例 6-1 を参照してください。

例 6-1 __m64 と FP データ型のコード間でのレジスターのリセット

間違った使用方法	正しい使用方法
`__m64 x = _m_paddd(y, z);`	`__m64 x = _m_paddd(y, z);`
`float f = init();`	`float f = (_mm_empty(), init());`

インテル® C++ コンパイラーが作成したコードで インテル® MMX® 命令が生成され、その命令により MMX レジスターが使用されることを知っておく必要があります。

- インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSSE3 のいずれかの 64 ビット SIMD 整数組込み関数を使用している場合。
- インライン・アセンブリーにより、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSSE3 のいずれかの 64 ビット SIMD 整数命令を使用している場合。
- __m64 データ型の変数を参照する場合。

x87 浮動小数点プログラミング・モデルについては、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』に詳しく記載されています。EMMS 命令の詳細は、http://developer.intel.com (英語) を参照してください。

6.3 データ・アライメント

64 ビット SIMD 整数データは 8 バイト境界にアライメントし、128 ビット SIMD 整数データは 16 バイト境界にアライメントします。アライメントされていない 64 ビット SIMD 整数データを参照すると、2 つのキャッシュラインにまたがってアクセスされるため、パフォーマンスが低下します。アライメントされていない 128 ビット SIMD

整数データを参照すると、movdqu (move double-quadword unaligned) 命令を使用しない限り例外が発生します。アライメントされていないデータに対して movdqu 命令を使用すると、16 バイト境界にアライメントされている参照に比べてパフォーマンスが劣る場合があります。

詳細については 5.4 節「スタックとデータ・アライメント」を参照してください。

16 バイトの SIMD データを効率良くロードするには、データを 16 バイト境界にアライメントする必要があります。そのためインテル® SSSE3 には、PALIGNR 命令が用意されています。この命令を利用すると、ソフトウェアがアライメントの合っていないアドレスのデータ要素を処理しなければならない場合に、オーバーヘッドを削減できます。PALIGNR 命令が最も有用なのは、アドレスを数バイトシフトして、アライメントされていないデータをロードまたはストアする場合です。アライメントされていないロードの代わりに、アライメントされているロードを実行してから PALIGNR 命令と単純なレジスター間コピーを使用します。

アライメントされていないロードの代わりに PALIGNR 命令を使用すると、キャッシュライン分割やその他のペナルティーの回避により、パフォーマンスが向上します。memcpy( ) のようなルーチンでは、PALIGNR 命令によってアライメントされていないケースのパフォーマンスが向上します。例 6-2 に、PALIGNR 命令によってメリットを得られる状況を示します。

例 6-2 C 言語コードでの FIR 処理の例

```
void FIR(float *in, float *out, float *coeff, int count)
{int i,j;
   for ( i=0; i<count - TAP; i++ )
   {   float sum = 0;
       for ( j=0; j<TAP; j++ )
       {  sum += in[j]*coeff[j]; }
          *out++ = sum;
       in++;
   }
}
```

例 6-3 では、インテル® SSE2 向けに最適化された FIR ループシーケンスと、インテル® SSSE3 向けの同等のシーケンスを比較しています。どちらも、FIR 内部ループの 4 回の反復をアンロールし、SIMD コーディング手法を実装しています。インテル® SSE2 コードでは、4 回の反復ごとに 1 回、キャッシュライン分割が発生します。PALGNR 命令を使用したインテル® SSSE3 コードでは、キャッシュライン分割に関連した遅延を回避できます。

例 6-3 インテル® SSE2 およびインテル® SSSE3 で実装された FIR 処理コード

インテル® SSE2 向けに最適化	インテル® SSSE3 向けに最適化

pxor xmm0, xmm0
xor ecx, ecx
mov eax, dword ptr[input]
mov ebx, dword ptr[coeff4]
 	
inner_loop:
movaps xmm1, xmmword ptr[eax+ecx]
mulps xmm1, xmmword ptr[ebx+4*ecx]
addps xmm0, xmm1
 	
movups xmm1, xmmword ptr[eax+ecx+4]
mulps xmm1, xmmword ptr[ebx+4*ecx+16]
addps xmm0, xmm1
 	
movups xmm1, xmmword ptr[eax+ecx+8]
mulps xmm1, xmmword ptr[ebx+4*ecx+32]
addps xmm0, xmm1
 	
movups xmm1, xmmword ptr[eax+ecx+12]
mulps xmm1, xmmword ptr[ebx+4*ecx+48]
addps xmm0, xmm1
 	
add ecx, 16
cmp ecx, 4*TAP
jl inner_loop
 	
mov eax, dword ptr[output]
movaps xmmword ptr[eax], xmm0
	pxor xmm0, xmm0
	xor ecx, ecx
	mov eax, dword ptr[input]
	mov ebx, dword ptr[coeff4]
	
	inner_loop:
	movaps xmm1, xmmword ptr[eax+ecx]
	movaps xmm3, xmm1
	mulps xmm1, xmmword ptr[ebx+4*ecx]
	addps xmm0, xmm1
	movaps xmm2, xmmword ptr[eax+ecx+16]
	movaps xmm1, xmm2
	palignr xmm2, xmm3, 4
	mulps xmm2, xmmword ptr[ebx+4*ecx+16]
	addps xmm0, xmm2
	movaps xmm2, xmm1
	palignr xmm2, xmm3, 8
	mulps xmm2, xmmword ptr[ebx+4*ecx+32]
	addps xmm0, xmm2
	
	movaps xmm2, xmm1
	palignr xmm2, xmm3, 12
	mulps xmm2, xmmword ptr[ebx+4*ecx+48]
	addps xmm0, xmm2
	
	add ecx, 16
	cmp ecx, 4*TAP
	jl inner_loop
	
	mov eax, dword ptr[output]
	movaps xmmword ptr[eax], xmm0

6.4 データ移動のコーディング手法

一般に、SIMD 計算用にあらかじめデータを配置しておくと、より高いパフォーマンスが得られます (5.5 節「メモリー使用効率の改善」を参照)。この方法が常に有効とは限りません。

この節では、より効率良く SIMD 計算ができるように、データを集約して配置する手法についていくつか説明します。

6.4.1 符号なしアンパック

インテル® MMX® テクノロジーには、MMX レジスター内のデータをパックしたり、アンパックする命令がいくつか用意されています。インテル® SSE2 では、これらの命令が拡張され、128 ビットのソースとデスティネーションを処理できるようになりました。

アンパック命令を使用して、符号なし数値をゼロ拡張できます。例 6-4 は、ソースをパックドワード (16 ビット) データ型と想定しています。

例 6-4 符号なしアンパック命令を使用して 16 ビット値を 32 ビットにゼロ拡張するコード

```
; 入力:
;      XMM0      8 つの 16 ビット値のソース
;      XMM7      必要であれば、XMM7 レジスターの代わりに
;                ローカル変数を利用できる
;
; 出力:
;      XMM0      4 つの下位ワードから
;                ゼロ拡張された 4 つのダブルワード
;
;      XMM1      4 つの上位ワードから
;                ゼロ拡張された 4 つのダブルワード
;
movdqa xmm1, xmm0        ; ソースをコピー
punpcklwd xmm0, xmm7     ; 4 つの下位ワードをアンパックし
                         ; 4 つの 32 ビット・ダブルワードを生成
punpckhwd xmm1, xmm7     ; 4 つの上位ワードをアンパックし
                         ; 4 つの 32 ビット・ダブルワードを生成
```

6.4.2 符号付きアンパック

値をアンパックするときは、符号付き数値を符号拡張する必要があります。これは、PSRAD 命令 (packed shift right arithmetic) を使用して値を符号拡張する点を除いて先に示したゼロ拡張とほぼ同じです。

例 6-5 は、ソースをパックドワード (16 ビット) データ型と想定しています。

例 6-5 符号付きアンパック・コード

```
; 入力:
;      XMM0      ソース値
; 出力:
;      XMM0      4 つの下位ワードから
;                符号拡張された4 つの 32 ビット・ダブルワード
;      XMM1      4 つの上位ワードから
;                符号拡張された4 つの 32 ビット・ダブルワード
;
movdqa xmm1, xmm0        ; ソースをコピー
punpcklwd xmm0, xmm0    ; ソースの 4 つの下位ワードを、
                         ; デスティネーションの各ダブルワードの
                         ; 上位 16 ビットにアンパックする
punpckhwd xmm1, xmm1    ; ソースの 4 つの上位ワードを、
                         ; デスティネーションの各ダブルワードの
                         ; 上位 16 ビットにアンパックする
psrad xmm0, 16          ; ソースの 4 つの下位ワードを
                         ; 4 つの符号付き 32 ビット・ダブルワードへ符号拡張
psrad xmm1, 16          ; ソースの 4 つの上位ワードを
                         ; 4 つの符号付き 32 ビット・ダブルワードへ符号拡張
```

6.4.3 飽和ありインターリーブ型パック

2 つの値をあらかじめ決められた順序で目的のデスティネーション・レジスターにパックするには、パック命令を使用します。PACKSSDW 命令では、ソースオペランドから取り出した符号付きダブルワード 2 つと、デスティネーション・オペランドから取り出した符号付きダブルワード 2 つが、符号付きワード 4 つに飽和され、4 つの符号付きワードは、デスティネーション・レジスター内にパックされます。図 6-1 を参照してください。

インテル® SSE2 では PACKSSDW 命令が拡張されており、ソースオペランドから取り出した符号付きダブルワード 4 つとデスティネーション・オペランドから取り出した符号付きダブルワード 4 つが、符号付きワード 8 つに飽和され、8 つの符号付きワードは、デスティネーション内にパックされます。

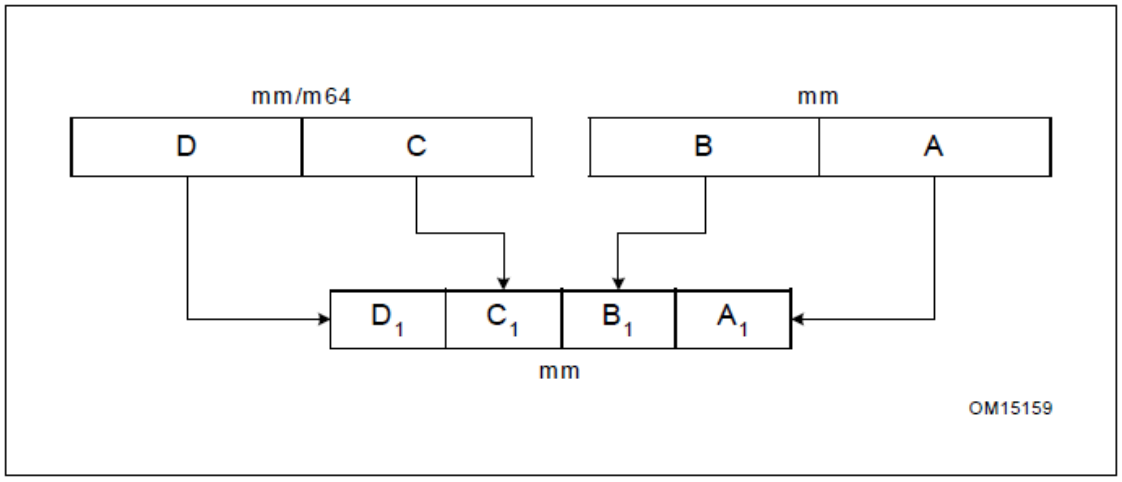


図 6-1 PACKSSDW mm, mm/m64 命令

図 6-2 に、デスティネーション・レジスター内で 2 ペアの値がインターリーブされる様子を示します。例 6-6 は、この演算を実行するインテル® MMX 命令コードです。

2 つの符号付きダブルワードがソースオペランドとして使用され、その結果、インターリーブされた符号付きワードがいくつか生成されます。インテル® SSE2 では、例 5-6 のシーケンスを拡張することにより、XMM レジスターを使用して 8 つの符号付きワードをインターリーブできます。

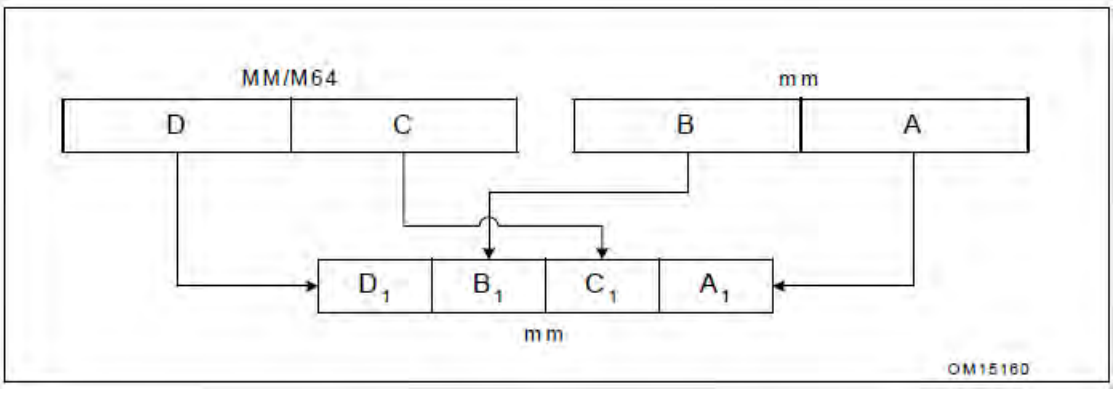


図 6-2 飽和ありインターリーブ型パック

例 6-6 飽和ありインターリーブ型パックのコード

```
; 入力:
;    MM0       符号付きソース 1 値
;    MM1       符号付きソース 2 値
; 出力:
;    MM0       MM0 からの最初と 3 番目のワードは符号拡張された飽和ダブルワードを含む
;              MM1 からの 2 番目と 4 番目のワードは符号拡張された飽和ダブルワードを含む
;
packssdw mm0, mm0    ; パックおよび符号飽和
packssdw mm1, mm1    ; パックおよび符号飽和
punpcklwd mm0, mm1   ; オペランドの下位 16 ビット値を
                     ; インターリーブする
```

パック命令は、ソースオペランドが符号付き数値であることが前提です。実行結果としてデスティネーション・レジスターに生成される内容は、目的の演算を実行したパック命令によって定義されます。例えば、PACKSSDW 命令は、2 つのソースから取り出した 2 つの符号付き 32 ビット値のそれぞれを、デスティネーション・レジスター内で、飽和した 4 つの 16 ビット符号付き値にパックします。一方、PACKUSWB 命令は、2 つのソースから取り出した 4 つの符号付き 16 ビット値を、デスティネーション・レジスター内で、飽和した 8 つの 8 ビット符号なし値にパックします。

6.4.4 飽和なしインターリーブ型パック

例 6-7 は、生成されたワードが飽和しない点を除いて例 6-6 とほぼ同じです。また、オーバーフローに対する予防策として、各ダブルワードの下位 16 ビットしか使用しません。ここでもインテル® SSE2 では、例 6-7 を拡張することにより、飽和なしで 8 つのワードをインターリーブできます。

例 6-7 飽和なしインターリーブ型パックのコード

```
; 入力:
;    MM0       符号付きソース 1 値
;    MM1       符号付きソース 2 値
; 出力:
;    MM0       MM0 からの最初と 3 番目のワードは
;                ダブルワードの下位 16 ビットを含む
;              MM1 からの 2 番目と 4 番目のワードは
;                ダブルワードの下位 16 ビットを含む
pslld mm1, 16     ; それぞれのダブルワード値の
                  ;  LSB16 を MSB16 へシフト
                  ;
pand mm0, {0,ffff,0,ffff}
                  ; 各ダブルワード値の MSB 16 をゼロでマスク
                  ;
por mm0, mm1       ; 2 つのオペランドをマージ
```

6.4.5 非インターリーブ型アンパック

デスティネーションおよびソース両者のオペランドのデータ要素をデスティネーション・レジスターにインターリーブしてマージするには、アンパック命令を使用します。

次に示す例は、インターリーブを実行せずにデスティネーション・レジスターにこの両者のオペランドを併合するものです。例えば、ソース 1 に含まれているパックドワード・データ型のうちどれか 1 つについて、隣り合っている 2 つの要素を取り出し、それを最終的に生成される下位 32 ビットに配置します。次に、ソース 2 に含まれているパックドワード・データ型のうちどれか 1 つについて、隣り合っている 2 つの要素を取り出し、それを最終的に生成される上位 32 ビットに配置します。デスティネーション・レジスターのうちの 1 つが、図 6-3 に示す組み合わせとなります。

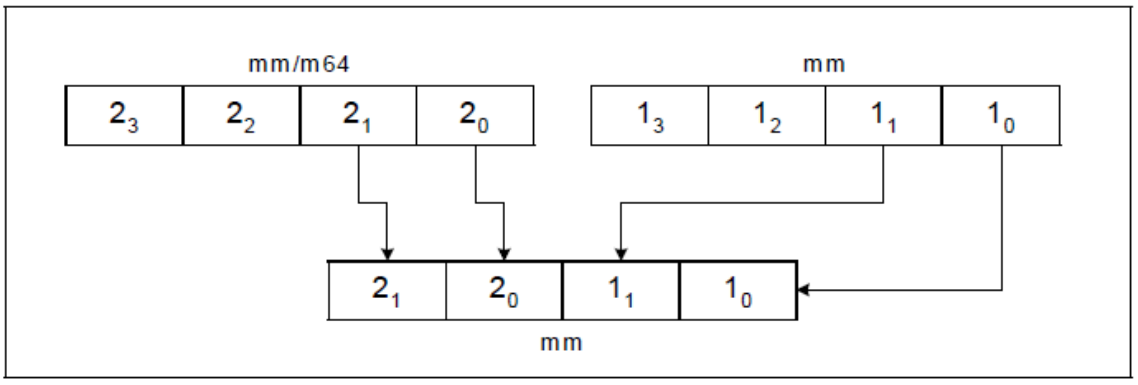


図 6-3 MM0 における非インターリーブ型アンパックの実行結果 (下位ビット)

残りのデスティネーション・レジスターは、図 6-4 に示すように組み合わせが逆になります。

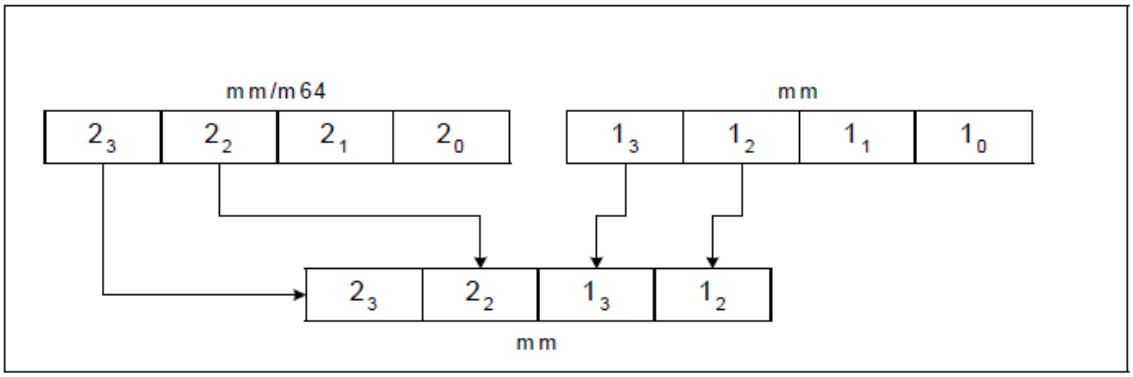


図 6-4 MM1 における非インターリーブ型アンパックの実行結果 (上位ビット)

例 6-8 のコードは、非インターリーブ方式で 2 つのパックドワード・ソースをアンパックするものです。複数のワードを複数のダブルワードにアンパックする命令を使用する代わりに、複数のダブルワードを 1 つのクワッドワードにアンパックする命令を使用することを目的としたコードです。

例 6-8 2 つのパックドワード・ソースの非インターリーブ方式コードでのアンパック

```
; 入力:
;      MM0      パックされたワードソース
;      MM1      パックされたワードソース
; 出力:
;      MM0      元のソースの下位ワードを含む
;               インターリーブはされない
;      MM2      元のソースの上位ワードを含む
;               インターリーブはされない
movq mm2, mm0        ; ソース 1 をコピー
punpckldq mm0, mm1   ; MM0 の上位ワード 2 つを
                     ; MM1 の下位ワード 2 つで置き換え
                     ; MM0 の下位ワード 2 つはそのまま残す
punpckhdq mm2, mm1   ; MM2 の上位ワード 2 つを MM2 の下位ワード 2 つへ移動
                    ; MM1 の上位ワード 2 つをMM2 の上位ワード 2 つに設定
                    ;
```

6.4.6 データ要素の抽出

即値の最下位 2 ビットによって選択された指定済み MMX レジスターに含まれるワードを抽出し、それを 32 ビット・レジスターの下位半分に移動するには、インテル® SSE の PEXTRW 命令を使用します。図 6-5 と例 6-9 を参照してください。

インテル® SSE2 の PEXTRW 命令を使用すると、XMM レジスターから整数レジスターの下位 16 ビットにワードを抽出できます。また、インテル® SSE4.1 では、XMM レジスターからメモリー・ロケーションまたは整数レジスターにバイト、ワード、ダブルワード、クワッドワードを抽出できます。

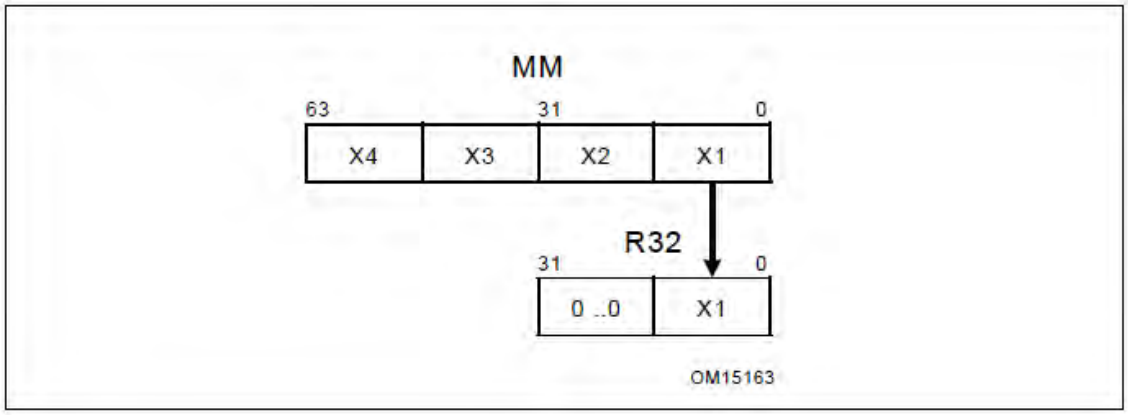


図 6-5 PEXTRW 命令

例 6-9 PEXTRW 命令のコード

```
; 入力:
;      eax       ソース値
;                即値: "0"
; 出力:
;      edx       ゼロ拡張された上位ビットと抽出されたワードを含む
;                下位ビットからなる 32 ビット·レジスター
;
movq mm0, [eax]
pextrw edx, mm0, 0
```

6.4.7 データ要素の挿入

32 ビット整数レジスターの下位半分かメモリーのいずれかから 1 ワードをロードし、それをインテル® MMX® テクノロジーのデスティネーション·レジスターに挿入するには、インテル® SSE の PINSRW 命令を使用します。挿入位置は即値定数の最下位 2 ビットによって決定されます。図 6-6 と例 6-10 を参照してください。

インテル® SSE2 の PINSRW 命令を使用すると、整数レジスターの下位 16 ビットまたはメモリーから XMM レジスターにワードを挿入できます。インテル® SSE4.1 では、メモリー·ロケーションまたは整数レジスターから XMM レジスターにバイト、ダブルワード、クワッドワードを挿入できます。

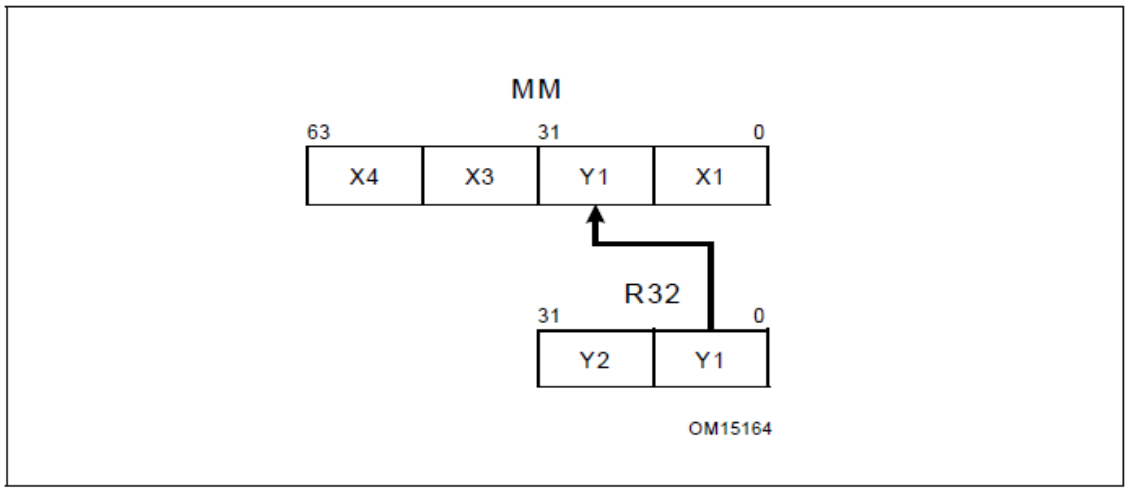


図 6-6 PINSRW 命令

例 6-10 PINSRW 命令のコード

```
; 入力:
;       edx         ソース値へのポインター
; 出力:
;       mm0         新たに 16 ビットを挿入した値を持つレジスター
;
mov eax, [edx]
pinsrw mm0, eax, 1
```

レジスターに含まれているすべてのオペランドを一連の PINSRW 命令で置き換える場合、PXOR 命令を使用するかレジスターにロードして内容をすべてクリアし、依存関係チェーンを分断すると良いでしょう。例 6-11 と 3.5.1.7 節「レジスターのクリアと依存関係解消イディオム」を参照してください。

例 6-11 PINSRW 命令を用いたコードの繰り返し実行

```
/* 入力:
;      edx      次のオフセットのソース値を含む構造体へのポインター:
;               +0, +10, +13, および +24
;               即値: "1"
; 出力:
;      MMX      新たに 16 ビットを挿入した値を持つレジスター
;
pxor mm0, mm0        ; 前の mm0 の値の依存性をなくす
mov eax, [edx]
pinsrw mm0, eax, 0
mov eax, [edx+10]
pinsrw mm0, eax, 1
mov eax, [edx+13]
pinsrw mm0, eax, 2
mov eax, [edx+24]
pinsrw mm0, eax, 3
```

6.4.8 非ユニット間隔データの移動

インテル® SSE4.1 では、メモリーから XMM レジスターにデータ要素を挿入し、XMM レジスターからメモリーにデータ要素を直接抽出する命令が提供されています。浮動小数点データと、整数のバイト、ワード、またはダブルワードを処理する個別の命令が用意されています。それぞれの命令は、メモリーから非ユニット間隔データをロード/ストアするコードのベクトル化に適しています。例 6-12 を参照してください。

例 6-12 インテル® SSE4.1 命令を使用した非ユニット間隔のロード/ストア

```
/* 目的: 非ユニットストライドな DWORD のロード */
movd xmm0, [addr]
pinsrd xmm0, [addr + stride], 1
pinsrd xmm0, [addr + 2*stride], 2
pinsrd xmm0, [addr + 3*stride], 3
```

```
/* 目的: 非ユニットストライドな DWORD のストア */
movd [addr], xmm0
pextrd [addr + stride], xmm0, 1
pextrd [addr + 2*stride], xmm0, 2
pextrd [addr + 3*stride], xmm0, 3
```

例 6-13 の 2 つの例では、それぞれ、INSERTPS と PEXTRD 命令を使用して浮動小数点データの収集操作と、EXTRACTPS と PEXTRD 命令を使用して浮動小数点データの分散操作 (scatter) を実行しています。

例 6-13 インテル® SSE4.1 命令を使用した分散/集約操作

```
/* 目的: 集約操作 */
movd eax, xmm0
movss xmm1, [addr + 4*eax]
pextrd eax, xmm0, 1
insertps xmm1, [addr + 4*eax], 1
pextrd eax, xmm0, 2
insertps xmm1, [addr + 4*eax], 2
pextrd eax, xmm0, 3
insertps xmm1, [addr + 4*eax], 3
```

```
/* 目的: 分散操作 */
movd eax, xmm0
movss [addr + 4*eax], xmm1
pextrd eax, xmm0, 1
extractps [addr + 4*eax], xmm1, 1
pextrd eax, xmm0, 2
extractps [addr + 4*eax], xmm1, 2
pextrd eax, xmm0, 3
extractps [addr + 4*eax], xmm1, 3
```

6.4.9 整数へのバイト移動マスク

ソースオペランドの各バイトの最上位ビットすべてから生成されたビットマスクを 1 つ返すには、PMOVMSKB 命令を使用します。64 ビット MMX レジスターと組み合わせた場合は、8 ビット・マスクが 1 つ生成され、デスティネーション・レジスターの上位 24 ビットはゼロとなります。128 ビット XMM レジスターと組み合わせた場合は、16 ビット・マスクが 1 つ生成され、デスティネーション・レジスターの上位 16 ビットはゼロとなります。

この命令の 64 ビット版を、図 6-7 と例 6-14 に示します。

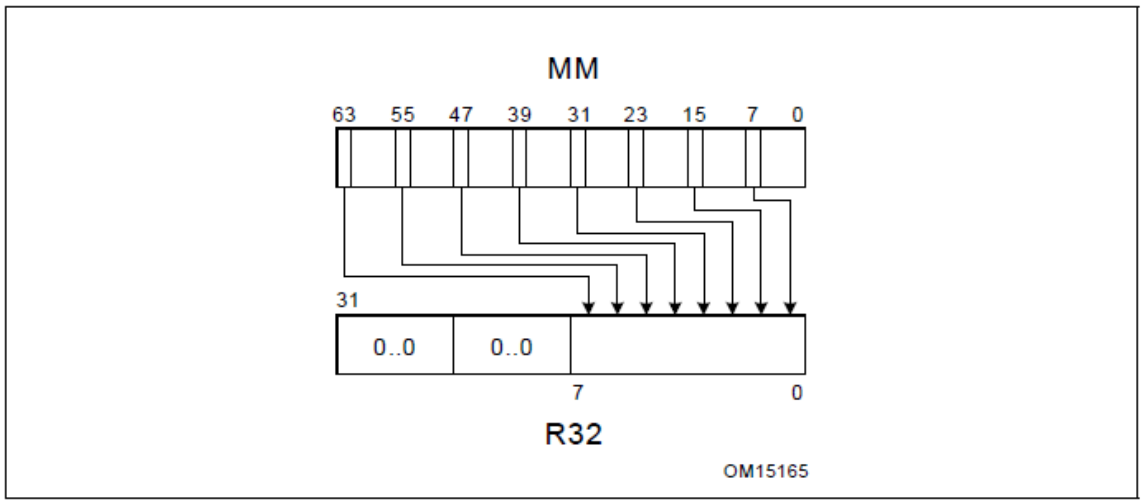


図 6-7 PMOVMSKB 命令

例 6-14 PMOVMSKB 命令のコード

```
; 入力:
;       ソース値
; 出力:
;       下位 8 ビットにバイト・マスクを含む 32 ビット・レジスター
;
;
movq mm0, [edi]
pmovmskb eax, mm0
```

6.4.10 64 ビット・レジスター用のパックド・シャッフル・ワード

PSHUF 命令は、即値 (imm8) オペランドを使用して、2 つの MMX レジスターの中に含まれている 4 つのワードを切り替えるか、あるいは 1 つの MMX レジスターと 1 つの 64 ビット・メモリー・ロケーションの中に含まれている 4 つのワードを切り替えます。インテル® SSE2 には、下位 4 ワードを XMM レジスター内にシャッフルする PSHUFLW 命令が用意されています。また、同等な PSHUFW 命令に加えて、上位 4 ワードをシャッフルする PSHUFHW 命令も用意されています。さらに、上位 4 ワードを XMM レジスター内にシャッフルする PSHUFHW 命令が用意されています。表 6-1 に示すように、この 4 つの PSHUF 命令はすべて、即値バイトを使用して、ソースからデスティネーションに供給される 8 バイト内で個々のワードのデータパスをエンコードします。

表 6-1 PSHUF のエンコード

ビット	ワード
1 – 0	0
3 – 2	1
5 – 4	2
7 - 6	3

6.4.11 128 ビット・レジスター用のパックド・シャッフル・ワード

pshuflw/pshufhw 命令は、8 ビットの即値オペランドを使用して、下位/上位 64 ビットに含まれる任意のソース・ワード・フィールドを完全にシャッフルして、デスティネーションの下位/上位 64 ビット内の任意のワード・フィールドを生成します。ほかの上位/下位 64 ビットはソースオペランドから渡される値になります。

8 ビットの即値オペランドを使用して、128 ビットのソースに含まれる任意のダブルワード・フィールドを完全にシャッフルして、128 ビットのデスティネーションの任意のダブルワード・フィールドを生成するには、pshufd 命令を使用します。

多くの一般的なデータでは、わずか 3 つの命令 (pshuflw/pshufhw/pshufd) だけでデータのシャッフル操作を実装できます。ブロードキャスト、スワップ、リバースの例を、例 6-15 と例 6-16 に示します。

例 6-15 インテル® SSE2 命令を 2 つ使用した XMM でのワードのブロードキャスト

```
/* 目的: ワード 5 からすべてのワードにブロードキャスト */
/* 命令の結果 */
| 7| 6| 5| 4| 3| 2| 1| 0|
PSHUFHW (3,2,1,1)| 7| 6| 5| 5| 3| 2| 1| 0|
PSHUFD (2,2,2,2) | 5| 5| 5| 5| 5| 5| 5| 5|
```

例 6-16 インテル® SSE2 命令を 3 つ使用した XMM でのワードのスワップ/リバース

```
/* 目的: ワード 6 と 1 の値を SWAP */
/* 命令の結果 */
                    |7|6|5|4|3|2|1|0|
PSHUFD (3,0,1,2) |7|6|1|0|3|2|5|4|
PSHUFHW (3,1,2,0)|7|1|6|0|3|2|5|4|
PSHUFD (3,0,1,2) |7|1|5|4|3|2|6|0|
```

```
/* 目的: ワードの並びを反転 */
/* 命令の結果 */
                    |7|6|5|4|3|2|1|0|
PSHUFLW (0,1,2,3) |7|6|5|4|0|1|2|3|
PSHUFHW (0,1,2,3) |4|5|6|7|0|1|2|3|
PSHUFD (1,0,3,2)  |0|1|2|3|4|5|6|7|
```

6.4.12 バイト・シャッフル

インテル® SSSE3 には、16 バイトの範囲内でバイト操作を実行する PSHUFB 命令が用意されています。PSHUFB 命令は、SHIFT、OR、AND、MOV など 12 個の命令に代わって使用できます。

代替コードで 5 つ以上の命令が使用される場合、PSHUFB 命令を使用します。

6.4.13 条件付きのデータ移動

インテル® SSE4.1 には、128 ビット・オペランド内のバイト/ワード・データ要素向けに 2 つのパックドブレンド命令が用意されています。パックドブレンド命令は、即値制御バイトまたは暗黙の XMM レジスター (XMM0) で指定されたマスクを使用して、ソース内の選択された部分から対応するデータ要素に条件付きでコピーします。マスクは、パックド比較命令などで生成します。パックドブレンド命令は、ループ内の条件付きフローをベクトル化するのに最も有効であり、状況によっては、単一の要素を一度に 1 つずつ挿入するよりも効率的です。

6.4.14 128 ビット・レジスターの 64 ビット・データのアンパック/インターリーブ

punpcklqdq/punpchqdq 命令は、ソースオペランドの下位/上位 64 ビットとデスティネーション・オペランドの下位/上位 64 ビットをインターリーブして、その結果をデスティネーション・レジスターに書き込みます。

ソースオペランドの上位/下位 64 ビットは無視されます。

6.4.15 データ移動

64 ビット SIMD 整数レジスターから 128 ビット SIMD レジスターへデータを移動できる命令がほかに 2 つあります。

1 つは movq2dq 命令で、これは MMX レジスター (ソース) から 128 ビットのデスティネーション・レジスターへ 64 ビットの整数データを移動します。デスティネーション・レジスターの上位 64 ビットはゼロになります。

もう 1 つは movdq2q 命令で、これは 128 ビットのソースレジスターから MMX レジスター (デスティネーション) へ整数データの下位 64 ビットを移動します。

6.4.16 変換命令

インテル® SSE には、単精度データとダブルワード整数データとの間で相互に 4 ワイド変換ができる命令が用意されています。インテル® SSE2 では、倍精度データからダブルワード整数データへの変換が追加されました。

インテル® SSE4.1 には、MXCSR の丸め制御とは無関係に、柔軟な方法で指定された丸め制御によって、浮動小数点値を整数値に変換する 4 つの丸め命令が用意されています。ROUNDxx 命令によって生成された整数値は、浮動小数点データとして維持されます。

インテル® SSE4.1 には、以下のように整数データに変換する命令も用意されています。

- 符号拡張またはゼロ拡張を使用して、パックドバイトからパックドワード/ダブルワード/クワッドワード形式に変換
- 符号拡張またはゼロ拡張を使用して、パックドワードからパックド・ダブルワード/クワッドワード形式に変換
- 符号拡張またはゼロ拡張を使用して、パックド・ダブルワードからパックド・クワッドワード形式に変換

6.5 定数の生成

SIMD 整数命令セットには、即値定数を SIMD レジスターにロードする命令がありません。

以下に示すコードは、使用頻度の高い定数を SIMD レジスターに生成します。インテル® SSE2 では、MMX レジスターの代わりに XMM レジスターを使用することで以下の例を拡張できます。図 6-17 を参照してください。

例 6-17 定数の生成

```
pxor mm0, mm0       ; MM0 にゼロを格納
pcmpeq mm1, mm1     ; MM1 にすべて '1' を生成。
                    ; パックされた各データフィールドは
                    ; -1 となる
pxor mm0, mm0
pcmpeq mm1, mm1
psubb mm0, mm1 [psubw mm0, mm1] (psubd mm0, mm1)
                    ; 上記の 3 つの命令は、各パックされたバイト・データ
                    ; (もしくはワードかダブルワード) フィールドに
                    ; 定数 1 を生成する
                    ;
pcmpeq mm1, mm1
psrlw mm1, 16-n(psrld mm1, 32-n)
                    ; 上記の 2 つの命令は、各パックされたバイトデータ
                    ; (もしくはダブルワード) フィールドに
                    ; 符号付き定数 2n-1 を生成する
pcmpeq mm1, mm1
psllw mm1, n (pslld mm1, n)
                    ; 上記の 2 つの命令は、各パックされたバイトデータ
                    ; (もしくはダブルワード) フィールドに
                    ; 符号付き定数 -2n を生成する
```

注意

SIMD 整数命令セットはバイトのシフト命令には対応していないため、2n-1 および -2n はパックドワードおよびパックド・ダブルワード以外には適しません。

6.6 ビルディング・ブロック

この節では、共通のコード・ビルディング・ブロックを実装する命令とアルゴリズムについて説明します。

6.6.1 符号なし数値間の絶対差

例 6-18 は、2 つの符号なし数値の絶対差を計算します。これは、符号なしパックド・バイト・データ型を対象としています。

ここでは、「符号なし飽和あり」減算命令を用いています。この命令は、符号なしオペランドを受け取って「符号なし飽和あり」減算を行う命令です。この命令は、パックドバイトとパックドワードのみに対応し、パックド・ダブルワードには対応していません。

例 6-18 2 つの符号なし数値間の絶対差

```
; 入力:
;      MM0      ソースオペランド
;      MM1      ソースオペランド
; 出力:
;      MM0      符号なしオペランドの絶対差
movq mm2, mm0      ; mm0 をコピー
psubusbmm0, mm1    ; 一方向の差を計算
psubusbmm1, mm2    ; 異なる方向の差を計算
por mm0, mm1        ; 上記を OR して統合
```

オペランドが符号付きの場合、上記のコード例は機能しません。状況によっては psadbw 命令が使用できる可能性があります。詳細については、6.6.9 節を参照してください。

6.6.2 符号付き数値間の絶対差

例 6-19 では、インテル® SSSE3 の PABSW 命令を使用して 2 つの符号付き数値の絶対差を計算しています。このシーケンスは、旧世代の SIMD 拡張命令を使用するよりも効率的です。

例 6-19 符号付き数値間の絶対差

```
; 入力:
;    XMM0      符号付きソースオペランド
;    XMM1      符号付きソースオペランド
; 出力:
;      XMM1       符号なしオペランドの絶対差
psubw xmm0, xmm1 ; ワードを減算
pabsw xmm1, xmm0 ; 結果は xmm1 に格納
```

6.6.3 絶対値

例 6-20 に、|x| の計算に使用するインテル® MMX® 命令のコードシーケンスを示します。ここで、x は符号付きです。この例では、符号付きワードがオペランドであると想定しています。

インテル® SSSE3 では、3 つの命令で構成されるこのシーケンスの代わりに PABSW 命令を使用できます。さらにインテル® SSSE3 では、XMM レジスターによる 128 ビット版も用意されており、バイト、ワード、ダブルワードの各単位がサポートされます。

例 6-20 絶対値の計算

```
; 入力:
;    MM0       符号付きソースオペランド
; 出力:
;      MM1      ABS(MM0)
pxor mm1, mm1      ; mm1 にゼロを設定
psubw mm1, mm0     ; mm1 の各ワードが mm0 の各ワードの
                   ; 負の値を含むように変換
pmaxswmm1, mm0     ; mm1 は正の (大きな) 値 - 絶対値
                   ; を含む
```

注意

絶対値が最大である負数の絶対値 (すなわち、16 ビットの場合 8000H) は、正数では表現できません。このアルゴリズムでは、絶対値 (8000H) の元の値が返されます。

6.6.4 ピクセル形式の変換

インテル® SSSE3 には、16 バイトの範囲内でバイト操作を実行する PSHUFB 命令が用意されています。PSHUFB 命令は、SHIFT、OR、AND、MOV など 12 個の命令に代わって使用できます。

代替コードで 5 つ以上の命令が使用される場合、PSHUFB 命令を使用します。例 6-21 に、カラーピクセル形式変換の基本的な手法を示します

例 6-21 RGBA から BGRA へ変換する基本的な C コード

```
// 標準的な C のコード
struct RGBA{BYTE r,g,b,a;};
struct BGRA{BYTE b,g,r,a;};
void BGRA_RGBA_Convert(BGRA *source, RGBA *dest, int num_pixels)
{
   for(int i = 0; i < num_pixels; i++){
       dest[i].r = source[i].r;
       dest[i].g = source[i].g;
       dest[i].b = source[i].b;
       dest[i].a = source[i].a;
   }
}
```

例 6-22 と例 6-23 に、ピクセル形式変換を行うインテル® SSE2 コードとインテル® SSSE3 コードを示します。インテル® SSSE3 の例では、6 つのインテル® SSE2 命令に代わって PSHUFB 命令が使用されています。

例 6-22 インテル® SSE2 を使用したカラーピクセル形式変換

```
; インテル® SSE2 向けに最適化
mov esi, src
   mov edi, dest
   mov ecx, iterations
   movdqa xmm0, ag_mask //{0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff}
   movdqa xmm5, rb_mask //{ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0}
   mov eax, remainder

convert16Pixs: // 反復ごとに 16 ピクセル、64 バイト
   movdqa xmm1, [esi]
                 // xmm1 = [r3g3b3a3,r2g2b2a2,r1g1b1a1,r0g0b0a0]
   movdqa xmm2, xmm1
   movdqa xmm7, xmm1 //xmm7 abgr
   psrld xmm2, 16 //xmm2 00ab
   pslld xmm1, 16 //xmm1 gr00

   por xmm1, xmm2 //xmm1 grab
   pand xmm7, xmm0 //xmm7 a0g0
   pand xmm1, xmm5 //xmm1 0r0b
   por xmm1, xmm7 //xmm1 argb
   movdqa [edi], xmm1

// ほかの 3* 16 バイトを繰り返す
   …
   add esi, 64
   add edi, 64
   sub ecx, 1
   jnz convert16Pixs
```

例 6-23 インテル® SSSE3 を使用したカラーピクセル形式変換

```
; インテル® SSSE3 向けに最適化
mov esi, src
   mov edi, dest
   mov ecx, iterations
   movdqa xmm0, _shufb
// xmm0 = [15,12,13,14,11,8,9,10,7,4,5,6,3,0,1,2]
   mov eax, remainder

convert16Pixs: // 反復ごとに 16 ピクセル、64 バイト
   movdqa xmm1, [esi]
// xmm1 = [r3g3b3a3,r2g2b2a2,r1g1b1a1,r0g0b0a0]
   movdqa xmm2, [esi+16]
   pshufb xmm1, xmm0
// xmm1 = [b3g3r3a3,b2g2r2a2,b1g1r1a1,b0g0r0a0]
   movdqa [edi], xmm1

// ほかの 3* 16 バイトを繰り返す
   …
   add esi, 64
   add edi, 64
   sub ecx, 1
   jnz convert16Pixs
```

6.6.5 エンディアンの変換

PSHUFB 命令を使用すると、ダブルワード内のバイト順序を逆転することもできます。これは、BSWAP 命令など従来の手法よりも効率的です。

例 6-24 (a) に、4 つの BSWAP 命令を使用してダブルワード内のバイトを逆転する従来の手法を示します。この手法では、それぞれの BSWAP 命令が 2 つのマイクロオペレーション (uop) を実行する必要があります。さらにこのコードでは、4 つのダブルワード・データを処理するのに 4 つのロードと 4 つのストアが必要です。

例 6-24 (b) に、PSHUFB 命令を使用してエンディアンを変換するインテル® SSSE3 コードを示します。この場合、4 つのダブルワード・データの逆転には、1 つのロード、1 つのストア、PSHUFB 命令が必要です。

インテル® Core™ マイクロアーキテクチャーでは、PSHUFB 命令を使用して 4 つのダブルワードを逆転すると、BSWAP 命令を使用するよりも約 2 倍高速化できます。

例 6-24 ビッグ･エンディアンからリトル･エンディアンへの変換

```
;;(a) BSWAP を利用
lea eax, src
lea ecx, dst
mov edx, elCount
start:
mov edi, [eax]
mov esi, [eax+4]
bswap edi
mov ebx, [eax+8]

bswap esi
mov ebp, [eax+12]
mov [ecx], edi
mov [ecx+4], esi
bswap ebx
mov [ecx+8], ebx
bswap ebp
mov [ecx+12], ebp

add eax, 16
add ecx, 16
sub edx, 4
jnz start
```

```
;; (b) PSHUFB を利用
__declspec(align(16)) BYTE
bswapMASK[16] =
{3,2,1,0, 7,6,5,4, 11,10,9,8,
15,14,13,12};
lea eax, src
lea ecx, dst
mov edx, elCount
movaps xmm7, bswapMASK
start:
movdqa xmm0, [eax]

pshufb xmm0, xmm7
movdqa [ecx], xmm0
add eax, 16
add ecx, 16
sub edx, 4
jnz start
```

6.6.6 任意の範囲 [High, Low] へのクリップ操作

この節では、値を任意の範囲 [High, Low] にクリップする方法について説明します。具体的には、その値が Low よりも小さいときは Low にクリップされ、High よりも大きいときは High にクリップされます。この手法では、符号付きまたは符号なしで、飽和ありのパックド加算命令とパックド減算命令を使用します。つまり、パックド･バイト･データ型とパックド･ワード･データ型以外には使用できません。

この節に記載した具体例では、packed_max および packed_min の定数を使用して、ワード値の演算を示します。簡潔にするため、以下の各定数を使用します。バイト値の演算を行う場合、対応する定数が使用されます。

packed_max は 0x7fff7fff7fff7fff に相当します。
packed_min は 0x8000800080008000 に相当します。
packed_low は、パックド･ワード･データ型の 4 つのワードすべてに Low の値を含んでいます。
packed_high は、パックド･ワード･データ型の 4 つのワードすべてに High の値を含んでいます。
packed_usmax の値はすべて 1 です。
high_us は、packed_min のすべてのデータ要素 (4 ワード) に High の値を加えたものです。
low_us は、packed_min のすべてのデータ要素 (4 ワード)に Low の値を加えたものです。

6.6.6.1 効率良いクリップ方法

任意の範囲に符号付きワードをクリップするときは、pmaxsw および pminsw 命令を使用しても良いでしょう。また任意の範囲に符号なしバイトをクリップするときは、pmaxub および pminub 命令を使用できます。

例 6-25 に、任意の範囲に符号付きワードをクリップする方法を示します。符号なしバイトをクリップするコードも同様です。

例 6-25 符号付きワード範囲 [High, Low] へのクリップ操作

```
; 入力:
;    MM0      符号付きソースオペランド
; 出力:
;     MM0      符号付きワードは、符号付き範囲 [HIGH, LOW]
;              にクリップされる
pminsw mm0, packed_high
pmaxswmm0, packed_low
```

インテル® SSE4.1 では、例 6-25 を容易に拡張して符号付きバイト、符号なしワード、符号付きダブルワード、符号なしダブルワードをクリップできます。

例 6-26 任意の符号付き範囲 [High, Low] へのクリップ操作

```
; 入力:
;    MM0      符号付きソースオペランド
; 出力:
;     MM1      符号付きオペランドは、符号なし範囲 [HIGH, LOW]
;              にクリップされる
paddw mm0, packed_min      ; 飽和なしで加算
                           ; 0x8000 で符号なしに変換
padduswmm0, (packed_usmax - high_us)
                           ; 高位へクリップ
psubuswmm0, (packed_usmax - high_us + low_us)
                           ; 低位へクリップ
paddw mm0, packed_low      ; 前の 2 つのオフセットを元に戻す
```

上記のコードは、最初に値を符号なし数値に変換してから、それを符号なし範囲にクリップしています。最後の命令でデータを元の符号付きデータに変換し、そのデータを符号付き範囲内に配置します。

(High - Low) < 0x8000 の場合は、符号なしデータに変換しないと正しい結果が得られません。(High - Low) >= 0x8000 の場合は、アルゴリズムを例 6-27 のように簡略化できます。

例 6-27 任意の符号付き範囲へのクリップ操作 (簡略版)

```
; 入力:      MM0      符号付きソースオペランド
; 出力:      MM1      符号付きオペランドは、符号なし範囲 [HIGH, LOW]
;              にクリップされる
paddssw mm0, (packed_max - packed_high)
                      ; 高位へクリップ
psubssw mm0, (packed_usmax - packed_high + packed_low)
                      ; 低位へクリップ
paddw mm0, low        ; 前の 2 つのオフセットを元に戻す
```

(High - Low) >= 0x8000 であることが分かっているときは、このアルゴリズムを使用すればサイクルを節約できます。(High - Low) < 0x8000 のときは、この 3 つの命令から成るアルゴリズムは機能しません。0xffff から 0x8000 未満の任意の数を引くと、0x8000 よりも大きな負数になるためです。

例 6-27 に示した 3 段階から成るアルゴリズムの 2 番目の命令である psubssw mm0,(0xffff- High + Low) 命令を実行すると、負数が減算されます。その場合、この減算の実行により、mm0 の値が減らずに増加するため、誤った結果となります。

6.6.6.2 任意の符号なし範囲 [High, Low] へのクリップ操作

例 6-28 のコードを実行すると、符号なし値が符号なし範囲 [High, Low] にクリップされます。その値が Low よりも小さいときは Low にクリップされ、High よりも大きいときは High にクリップされます。この手法では、符号なし飽和ありのパックド加算命令とパックド減算命令を使用します。つまり、パックド・バイト・データ型とパックド・ワード・データ型以外には使用できません。

例 6-28 では、ワード値を対象に演算を行います。

例 6-28 任意の符号なし範囲 [High, Low] へのクリップ操作

```
; 入力:
;      MM0       符号なしソースオペランド
; 出力:
;      MM1       符号なしオペランドは、符号なし範囲 [HIGH, LOW]
;                にクリップされる
paddusw mm0, 0xffff - high
                    ; 高位へクリップ
psubusw mm0, (0xffff - high + low)
                    ; 低位へクリップ
paddw mm0, low
                    ; 前の 2 つのオフセットを元に戻す
```

6.6.7 バイト、ワード、ダブルワードのパックド最大値/最小値

pmaxsw 命令は、4 つの符号付きワードの最大値を含む 2 つの SIMD レジスター、あるいは 4 つの符号付きワードの最大値を含む 1 つの SIMD レジスターとメモリー・ロケーションを返します。

pminsw 命令は、4 つの符号付きワードの最小値を含む 2 つの SIMD レジスター、あるいは 4 つの符号付きワードの最小値を含む 1 つの SIMD レジスターとメモリー・ロケーションを返します。

pmaxub 命令は、8 つの符号付きワードの最大値を含む 2 つの SIMD レジスター、あるいは 8 つの符号付きワードの最大値を含む 1 つの SIMD レジスターとメモリー・ロケーションを返します。

pminub 命令は、8 つの符号付きワードの最小値を含む 2 つの SIMD レジスター、あるいは 8 つの符号付きワードの最小値を含む 1 つの SIMD レジスターとメモリー・ロケーションを返します。

インテル® SSE2 では、PMAXSW/PMAXUB/PMINSW/PMINUB 命令が 128 ビット操作に拡張されました。インテル® SSE4.1 では、符号付きバイト、符号なしワード、符号付きダブルワード、符号なしダブルワードの 128 ビット操作が追加されています。

6.6.8 整数のパックド乗算

pmulhuw/pmulhw 命令は、ソースオペランドに含まれる符号なし/符号付きワードと、デスティネーション・オペランドに含まれる符号なし/符号付きワードの乗算を行います。32 ビットの中間結果の上位 16 ビットがデスティネーション・オペランドに書き込まれます。pmullw 命令は、デスティネーション・オペランドに含まれる符号付きワードとソースオペランドに含まれる符号付きワードの乗算を行います。32 ビットの中間結果の下位 16 ビットがデスティネーション・オペランドに書き込まれます。

インテル® SSE2 では、PMULHUW/PMULHW/PMULLW 命令が 128 ビット操作に拡張され、PMULUDQ 命令が追加されました。

2 つのソースから取り出した 64 ビット・チャンクに含まれるダブルワード・オペランドの下位ペアの符号なし乗算を実行するには、PMULUDQ 命令を使用します。各乗算から得られる 64 ビットの完全な結果は、デスティネーション・レジスターに返されます。

この命令は 64 ビット版と 128 ビット版が用意されています。128 ビット版では、128 ビット・レジスターの上位と下位を半分ずつに分けて、それぞれ独立した 2 つの演算が実行されます。

インテル® SSE4.1 では、PMULDQ と PMULLD の 128 ビット操作が追加されました。PMULLD 命令は、デスティネーション・オペランドに含まれる符号付きダブルワードとソースオペランドに含まれる符号付きダブルワードの乗算を行います。64 ビットの中間結果の下位 32 ビットがデスティネーション・オペランドに書き込まれます。PMULDQ 命令は、デスティネーション・オペランドに含まれる下位の符号付きダブルワード 2 つとソースオペランドに含まれる下位の符号付きダブルワード 2 つの乗算を行い、64 ビットの結果 2 つをデスティネーション・オペランドにストアします。

6.6.9 絶対差のパックド和

psadbw 命令を実行すると、2 つの SIMD レジスターに含まれる符号なしバイトの絶対差、あるいは SIMD レジスターとメモリー・ロケーションに含まれる符号なしバイトの絶対差が計算されます。次に、8 つのペアの符号なしバイトの差が合計され、その結果、下位 16 ビット・フィールドにワードが 1 つ生成され、上位 3 ワードは 0 になります。インテル® SSE2 では、PSADBW 命令が拡張され、2 つのワード結果を計算できます。

上記の例の減算は絶対差の計算であり、t = abs(x-y) を計算しています。バイト値は一時領域に格納され、値はすべて合計された後、その計算結果がデスティネーション・レジスターの下位ワードに書き込まれます。

動き評価には、参照フレームから最も一致するものを検索する処理が含まれます。2 つのピクセルブロックの絶対差の和 (SAD) は、一致するピクセルブロックを検索するビデオ処理アルゴリズムで一般的な構成要素です。PSADBW 命令は、4×4、8×4、8×8 のピクセルブロックに対する SAD の結果を計算することにより、最も一致するピクセルブロックを検索するビルディング・ブロックとして使用できます。

6.6.10 MPSADBW と PHMINPOSUW 命令

インテル® SSE4.1 の MPSADBW 命令は、8 つの SAD 演算を実行します。それぞれの SAD 演算の結果として、4 つのペアの符号なしバイトからワードが 1 つ生成されます。PHMINPOSUM 命令は、8 つの SAD 演算結果を 1 つの XMM レジスターに格納し、8 つの 4×4 ピクセルブロック間で最も近いものを検索します。

動き評価アルゴリズムとしては、MPSADBW 命令は以下のようにいくつかの点で PSADBW 命令よりも優れています。

- データ移動を簡素化して、パックドデータ形式をピクセルブロックの SAD 演算向けに構成します。
- 反復ごとの SAD 演算結果についてスループットが増加 (フレームごとに必要な反復数が減少) します。
- MPSADBW 命令の結果は、PHMINPOSUW 命令を使用した効率的な検索に適しています。

4×4 と 8×8 のブロックの検索に関する MPSADBW と PSADBW 命令の比較例については、第 1 章の参考文献の節に記載されているホワイトペーパーを参照してください。

6.6.11 パックド平均 (バイト/ワード)

pavgb 命令と pavgw 命令は、ソースオペランドの符号なしデータ要素を、キャリーインも含めてデスティネーション・レジスターの符号なしデータ要素に加算します。この加算結果は、独立して 1 ビットずつ右方向にシフトされます。各要素の上位ビットには、対応する総和値のキャリービットが格納されます。

このデスティネーション・オペランドは SIMD レジスターです。ソースオペランドには SIMD レジスターまたはメモリーオペランドを指定できます。

パックド符号なしバイトの演算を行うときは pavgb 命令を使用し、パックド符号なしワードの演算を行うときは pavgw 命令を使用します。

6.6.12 定数との複素乗算

複素乗算には 4 回の乗算と 2 回の加算が必要であり、pmaddwd 命令ではまさにそうした演算を実行します。この命令を使用するためには、対象となるデータのフォーマットを複数の 16 ビット値に変更する必要があります。実数成分も虚数成分もそれぞれ 16 ビットに変更します。例 6-29 について検討してみます。この例では、64 ビット MMX レジスターが使用されると想定しています。

- 入力データは Dr および Di とします。ここで、Dr はデータの実数成分、Di はデータの虚数成分です。
- メモリーに記憶されている定数の複素係数のフォーマットを 4 つの 16 ビット値 [Cr-Ci Ci Cr] に変更します。MOVQ 命令を使用して目的の値を MMX レジスターにロードすることを忘れないでください。
- 複素積の実数成分は Pr = Dr*Cr - Di*Ci です。
  複素積の虚数成分は Pi = Dr*Ci + Di*Cr です。
- 出力はパックド・ダブルワードとなります。必要な場合は、パック命令を使用して実行結果を 16 ビットに変換し、入力データのフォーマットに一致させることもできます。

例 6-29 定数との複素乗算

```
; 入力:
;      MM0      複素数値を, Dr, Di に格納
;      MM1      複素数係数の定数は
;               [Cr -Ci Ci Cr] の形式
; 出力:
;      MM0      2 つの 32ビット dword は [Pr Pi] を含む
;
punpckldq mm0, mm0          ; [dr di dr di] を作成
pmaddwd mm0, mm1            ; 完了。結果は以下のようになる。
                            ; [(Dr*Cr-Di*Ci)(Dr*Ci+Di*Cr)]
```

6.6.13 パックド 64 ビット加算/減算

2 つのソースから取り出した 64 ビット・チャンクのそれぞれに含まれるクワッドワード・オペランドを使って加算または減算を実行するには、PADDQ 命令または PSUBQ 命令を使用します。各計算の実行結果として得られた 64 ビット値は、デスティネーション・レジスターに書き込まれます。整数の ADD/SUB 命令と同様、PADDQ/PSUBQ 命令でも、2 の補数を使用することで符号なし整数オペランドと符号付き整数オペランドを処理できます。

個々の計算結果が大きすぎて 64 ビットで表現できないときは、その計算結果の下位 64 ビットがデスティネーション・オペランドに書き込まれます。したがって、計算結果がラップアラウンド (循環) することになります。これらの命令は 64 ビット版と 128 ビット版が用意されています。128 ビット版では、128 ビット・レジスターの上位と下位を半分ずつに分けて、それぞれ独立した 2 つの演算が行われます。

6.6.14 128 ビット・シフト

先頭オペランドを左方向か右方向にシフトするには、pslldq 命令または psrldq 命令を使用して、シフトするバイト数は即値オペランドで指定します。空の下位/上位バイトはクリアされます (つまりゼロに設定されます)。

即値オペランドの値が 15 より大きい場合、デスティネーション・レジスターがすべてゼロに設定されます。

6.6.15 PTEST と条件分岐

インテル® SSE4.1 には、条件分岐のあるループのベクトル化に利用可能な PTEST 命令が用意されています。PTEST 命令は、汎用命令 TEST の 128 ビット版です。PTEST 命令の結果として、EFLAGS レジスターの ZF フィールドまたは CF フィールドが変更されます。

例 6-30 (a) に、ゼロ除算の特殊なケースの処理に条件分岐を必要とするループを示します。このようなループをベクトル化するには、ゼロ除算が発生する可能性のある反復を、ベクトル化可能な反復外で処理しなければなりません。

例 6-30 PTEST 命令の使用による、ベクトル化可能なループの反復とベクトル化不可能なループの反復の分離

```
(a) /* まれな例外処理を伴うループ */
float a[CNT];
unsigned int i;
   for (i=0;i<CNT;i++)
   {
       if (a[i] != 0.0)
       { a[i] = 1.0f/a[i];
       }
       else
       { call DivException();
       }
   }
```

```
(b) /* PTEST は、まれな非ベクトル化領域を扱うことを
可能にする */
   xor eax,eax
   movaps xmm7, [all_ones]
   xorps xmm6, xmm6
lp:
   movaps xmm0, a[eax]
   cmpeqps xmm6, xmm0
     ; 各非ゼロ値を変換
   ptest xmm6, xmm7
   jnc zero_present
      ; 4 つの値が非ゼロならキャリーをセット
   movaps xmm1,[_1_0f_]
   divps xmm1, xmm0
   movaps a[eax], xmm1
   add eax, 16
   cmp eax, CNT
   jnz lp
   jmp end
zero_present:
// 1 つずつ処理し、値がゼロなら
// 例外処理を呼び出す
```

例 6-30 (b) に示すアセンブリー・シーケンスでは、PTEST 命令を使用して、xmm0 内の 4 つの浮動小数点値のいずれかがゼロであるときには分岐を早期に終了させています。フォールスルー・パスでは、4 つの値はいずれもゼロでないため、浮動小数点演算の残りをベクトル化できます。

6.6.16 ループの反復間における異種演算のベクトル化

アンロールされたループをベクトル化する手法は、一般にループの各反復間で繰り返される同種の操作に依存しています。可変ブレンド命令を使用すると、ループ反復間における異種演算のベクトル化が可能となる場合があります。

例 6-31 (a) に、単純な異種ループを示します。異種演算と条件分岐は、単純なループアンロール手法をベクトル化に適用できません。

例 6-31 可変ブレンドを使用した異種ループのベクトル化

(a) /* 反復間で異種の操作を伴うループ */

```
float a[CNT];
unsigned int i;



for (i=0;i<CNT;i++)
{
   if (a[i] > b[i])
   { a[i] += b[i]; }
   else
   { a[i] -= b[i]; }
}
```

(b) /* BLENDVPS 命令によるベクトル化条件の流れ */

```
   xor eax,eax
lp:
   movaps xmm0, a[eax]
   movaps xmm1, b[eax]
   movaps xmm2, xmm0
   // a と b の値を比較
   cmpgtps xmm0, xmm1
   // xmm3 - will hold -b
   movaps xmm3, [SIGN_BIT_MASK]
   xorps xmm3, xmm1
   // 加算操作のため値を選択
// 真の条件では a+b となり、偽では a+(-b) となる
   // ブレンドマスクは xmm0
   blendvps xmm1,xmm3, xmm0
   addps xmm2, xmm1
   movaps a[eax], xmm2
   add eax, 16
   cmp eax, CNT
   jnz lp
```

例 6-31 (b) に示すアセンブリー・シーケンスでは、BLENDVPS 命令を使用して、連続した 4 回の反復で発生する異種演算の処理をベクトル化しています。

6.6.17 ネストされたループ制御フローのベクトル化

PTEST 命令と BLENDVPX 命令は、複雑な制御フローをベクトル化するビルディング・ブロックとして使用できます。この場合、各制御フローは、マスクされた条件付きコードが動作するプレディケートとして使用される、「作業用」マスクを生成します。

ネストされたループ内に複雑な制御フローが存在する例として、マンデルブロー集合における評価が上げられます。マンデルブロー集合は、2D グリッドにマッピングされた高さ値の集合です。高さ値は、|In| > 2 を得るのに必要なマンデルブロー反復 (複素数空間で $I_n = I_{n-1}^2 + I_0$ として定義) 回数です。高さの最大しきい値を設定することでマップ生成を制限するのが一般的であり、その他のすべてのポイントには、しきい値と同じ高さが割り当てられます。例 6-32 に、C コードで実装されたマンデルブロー集合評価の例を示します。

例 6-32 マンデルブロー集合評価の基準 C コード

```
#define DIMX (64)
#define DIMY (64)
#define X_STEP (0.5f/DIMX)
#define Y_STEP (0.4f/(DIMY/2))
int map[DIMX][DIMY];

void mandelbrot_C()
{ int i,j;
   float x,y;
   for (i=0,x=-1.8f;i<DIMX;i++,x+=X_STEP)
   {
       for (j=0,y=-0.2f;j<DIMY/2;j++,y+=Y_STEP)
       {float sx,sy;
           int iter = 0;
           sx = x;
           sy = y;
           while (iter < 256)
           { if (sx*sx + sy*sy >= 4.0f) break;
              float old_sx = sx;
              sx = x + sx*sx - sy*sy;
              sy = y + 2*old_sx*sy;
              iter++;
           }
           map[i][j] = iter;
       }
   }
}
```

例 6-33 に、マンデルブロー集合評価をベクトル化したものを示します。ブレーク文がピクセルごとの反復数を一定にしないため、最内のループはベクトル化されていません。ベクトル化バージョンでは、2D の並列性を考慮した上で、連続する 4 ピクセルの Y 値を 4 回分ベクトル化し、条件付きで以下の 3 つのシナリオを処理します。

- 最も内部の反復では、4 ピクセルすべてがブレーク条件に達しない場合、4 ピクセルがベクトル化されます。
- ピクセルがブレーク条件に達した場合、ブレンド組込み関数を使用して、ブレーク条件に達しない残りのピクセルの高さ要素 (複素数) を累積し、高さ要素 (複素数) の内部反復を続行します。
- 4 ピクセルすべてがブレーク条件に達した場合、内部ループを終了します。

例 6-33 インテル® SSE4.1 組込み関数を使用したベクトル化済みマンデルブロー集合評価

```
__declspec(align(16)) float _INIT_Y_4[4] = {0,Y_STEP,2*Y_STEP,3*Y_STEP};
F32vec4 _F_STEP_Y(4*Y_STEP);
I32vec4 _I_ONE_ = _mm_set1_epi32(1);
F32vec4 _F_FOUR_(4.0f);
F32vec4 _F_TWO_(2.0f);;

void mandelbrot_C()
{ int i,j;
   F32vec4 x,y;

   for (i = 0, x = F32vec4(-1.8f); i < DIMX; i ++, x += F32vec4(X_STEP))
   {
       for (j = DIMY/2, y = F32vec4(-0.2f) +
          *(F32vec4*)_INIT_Y_4; j < DIMY; j += 4, y += _F_STEP_Y)
       { F32vec4 sx,sy;
          I32vec4 iter = _mm_setzero_si128();
          int scalar_iter = 0;
          sx = x;
          sy = y;
          while (scalar_iter < 256)
          { int mask = 0;
             F32vec4 old_sx = sx;
             __m128 vmask = _mm_cmpnlt_ps(sx*sx + sy*sy,_F_FOUR_);
             // ベクトル中のすべてのポイントが終了条件に
             // 一致したらベクトル化されたループを抜ける
             if (_mm_test_all_ones(_mm_castps_si128(vmask)))
             break;
                      (続く)

// すべてのデータポイントがゼロなら、結果を結合する追加コードを実行する必要はない
             if (_mm_test_all_zeros(_mm_castps_si128(vmask),
                 _mm_castps_si128(vmask)))
             { sx = x + sx*sx - sy*sy;
                sy = y + _F_TWO_*old_sx*sy;
                iter += _I_ONE_;
             }
             else
             {
// このコードは直前の反復の結果と現在の反復の結果をブレンドする
// 終了条件に達していない値のみがストアされる
// すでに条件に達している値は、それを維持する
                sx = _mm_blendv_ps(x + sx*sx - sy*sy,sx,vmask);
                sy = _mm_blendv_ps(y + _F_TWO_*old_sx*sy,sy,vmask);
                iter = I32vec4(_mm_blendv_epi8(iter + _I_ONE_,
                iter,_mm_castps_si128(vmask)));
             }
             scalar_iter++;
          }
          _mm_storeu_si128((__m128i*)&map[i][j],iter);
       }
   }
}
```

6.7 メモリー最適化

以下の手法を用いて、メモリーアクセスを改善できます。

- パーシャル・メモリー・アクセスを避けます。
- メモリーフィルおよびビデオフィルのいずれについても帯域幅を広げます。
- インテル® ストリーミング SIMD 拡張命令を使ってデータをプリフェッチします。第 9 章「キャッシュ利用の最適化」も参照してください。

MMX レジスターと XMM レジスターでは、プロセッサーがストールすることなく大量にデータを移動できます。8、16、または 32 のいずれかのビット長を持つ単一配列の値をいくつかロードする代わりに、シングル・クワッドワードかダブル・クワッドワードに格納されている値をすべてロードしてから、それに応じて目的の構造体または配列のポインターをインクリメントする点に注意してください。SIMD 整数命令で処理するデータはすべて、以下のいずれかを使ってロードしなければなりません。

- 64 ビットまたは 128 ビットのオペランドをロードする SIMD 整数命令 (例えば、movq mm0, m64)。
- レジスターメモリー形式の SIMD 整数命令のうち、クワッドワードかダブル・クワッドワードのメモリーオペランドを演算するもの (例えば、pmaddw mm0, m64)。

SIMD データをストアする場合、64 ビットまたは 128 ビットのオペランドをストアする SIMD 整数命令を使用しなければなりません (例えば、movq m64, mm0)。

上記の方法を推奨する理由は 2 つあります。1 つは、より大きなブロックサイズを使用して SIMD データのロード操作/ストア操作の効率化を図るためであり、もう 1 つは、8 ビット、16 ビット、32 ビットのロード操作/ストア操作と、SIMD 整数テクノロジーを用いたロード操作/ストア操作が、同じ SIMD データに混在するのを避けるためです。

こうした方法により、同一メモリー領域に対して大量のストア操作を実行した後に少量のロード操作が実行されたり、あるいは少量のストア操作を実行した後に大量のロード操作が実行されたりする状況を回避できます。インテル® Pentium® II プロセッサー、インテル® Pentium® III プロセッサー、インテル® Pentium® 4 プロセッサーは、このような状況でストールする可能性があります。詳細は、第 3 章を参照してください。

6.7.1 パーシャル・メモリー・アクセス

同じメモリーアドレス (mem) で始まる領域に対して、小さなビット幅のストア操作を連続して実行した後に大きなビット幅のロード操作を実行する状況を考えてみます。例 6-34 のコードを実行した場合、この中の大きなビット幅のロード操作命令はストールします。

例 6-34 小さなビット幅のストア操作を連続して実行した後、大きなビット幅のロード操作を実行 (良くない例)

```
mov mem, eax      ; アドレス "mem" へ dword をストア
mov mem + 4, ebx ; アドレス "mem + 4" へ dword をストア
  :
  :
movq mm0, mem     ; "mem" からの qword ロードはストール
```

上記の例の movq 命令は、ストア操作によるメモリー書き込みが終了しない限り、必要とするすべてのデータにアクセスできません。その他のデータ型の場合も同様にストールすることがあります。例えば、バイトやワードをいくつかストアしてから、ワードやダブルワードをいくつか同じメモリー領域から読み出す場合などです。例 6-35 のようにコードシーケンスを変更するとプロセッサーは遅延なくデータにアクセスできます。

例 6-35 遅延の生じないデータアクセス方法

```
movd mm1, ebx      ; メモリーへストアする前に、
                   ; データを qword に再構築
movd mm2, eax
psllq mm1, 32
por mm1, mm2
movq mem, mm1      ; SIMD 変数を
                   ; qword として "mem" へストア
  :
  :
movq mm0, mem      ; "mem" から qword SIMD をロード、ストールなし
```

次に、同じメモリーアドレス (mem) で始まる領域に対して、大きなビット幅のストア操作を実行した後に、小さなビット幅のロード操作を連続して実行する状況を考えてみます。例 6-36 にその例を示します。小さなデータ幅のロード操作は、大きなデータ幅のストア操作と整合がとれないため、ほとんどの場合ストールします。詳細については、3.6.4 節「ストア・フォワーディング」を参照してください。

例 6-36 大きなデータ幅のストアの後に小さなロードを連続して実行

```
movq mem, mm0     ; アドレス "mem" へ qword をストア
  :
  :
mov bx, mem + 2   ; "mem+2" からの word ロードはストール
mov cx, mem + 4   ; "mem+4" からの word ロードはストール
```

上記の例のワードのロード操作は、クワッドワードのストア操作によるメモリー書き込みが終了しない限り、必要なデータにアクセスできません。その他のデータ型の場合も同様にストールすることがあります。例えば、ダブルワードやワードをいくつかストアしてから、ワードやバイトをいくつか同じメモリー領域から読み出す場合です。

例 6-37 のようにコードシーケンスを変更するとプロセッサーは遅延なくデータにアクセスできます。

例 6-37 大きなデータ幅のストアの後に小さなロードを連続して実行する際の遅延の回避策

```
movq mem, mm0     ; アドレス "mem" へ qword をストア
  :
  :
movq mm1, mem     ; "mem" からの qword をロード
movd eax, mm1     ; MMX レジスターから、"mem + 2" を
                  ; eax へ移動する
psrlq mm1, 32
shr eax, 16
movd ebx, mm1      ; MMX レジスターから、"mem + 4" を
                   ; ebx へ移動する
and ebx, 0ffffh
```

一般に、上記のような変更を加えると、目的とする操作を行うために必要な命令数は増加します。命令数が増えるとパフォーマンスが低下することは確かですが、インテル® Pentium® II プロセッサー、インテル® Pentium® III プロセッサー、インテル® Pentium® 4 プロセッサーにおいては、パフォーマンスの低下よりも、先に述べた問題を避けることで得られる利点の方がはるかに大きくなります。

6.7.2 メモリーフィルおよびビデオフィルの帯域幅の拡大

どのようにメモリーがアクセスおよびフィルされるか理解するのは重要です。メモリー-メモリー間でのフィル (例えば、メモリー-ビデオ間でのフィル) は、元のメモリーに直ちにストアされるメモリー (ビデオ・フレーム・バッファーなど) から 64 バイト (キャッシュライン) ロードを行うことと定義されています。

以下に示すいくつかのガイドラインは、順次メモリーフィル (ビデオフィル) を行う帯域幅を広げ、レイテンシーを短縮することを目的とします。インテル® MMX® テクノロジーを用いるインテル® アーキテクチャー・ベースのプロセッサーのすべてが対象となります。また、ロード操作/ストア操作が 1 次キャッシュにも 2 次キャッシュにもヒットしない場合についても述べます。

6.7.2.1 MOVDQ 命令によるメモリー帯域幅の拡大

どのようなサイズのデータオペランドをロードする場合も、キャッシュライン全体がキャッシュ階層にロードされます。したがって、メモリー帯域幅の点から見ると、ロード操作はそのサイズが異なっても大体同じようなものであるといえます。ただし、データ幅の小さいロード操作の回数を増やすと、データ幅の大きなストア操作の回数を減らした場合に比べて、マイクロアーキテクチャーのリソースの消費量は増加します。マイクロアーキテクチャーのリソースの消費量が増加しすぎると、プロセッサーがストールしたり、メモリー・サブシステムに要求できる帯域幅が狭くなったりすることがあります。

32 ビット・ストア (例えば、movd) を使用する代わりに movdq 命令を使用してデータを元の UC (キャッシュ不可) メモリー、場合によっては WC (キャッシュ可能)メモリーにストアし直すことで、メモリー・フィル・サイクル 1 回あたりのストア回数を 4 分の 3 減らすことができます。

そのため、メモリー・フィル・サイクルで movdq 命令を使用すると、movd 命令を使用した場合に比べて帯域幅の効率がかなり良くなります。

6.7.2.2 同じ DRAM ページに対するロード操作/ストア操作を行うことによるメモリー帯域幅の拡大

DRAM はいくつかのページに分割されていますが、そのページはオペレーティング・システム (OS) でいうページとは異なります。DRAM ページ 1 枚あたりのサイズは、DRAM 全体のサイズと DRAM の編成によって決定されます。ページのサイズは数キロバイトが一般的です。OS のページと同様、DRAM ページもいくつかの順次アドレスで構成されます。DRAM ページへ何度か順次メモリーアクセスを行う場合、同じページにアクセスする方が毎回異なるページにアクセスするよりもレイテンシーは短くなります。

多くのシステムでは、ページミスのレイテンシー (遅延)は、メモリー・ページ・ヒットのレイテンシーの 2 倍になることがあります。「ページミス」とは、前にアクセスしたページではなく別のページにアクセスすることであり、「メモリー・ページ・ヒット」とは、前にアクセスしたのと同じページにアクセスすることです。したがって、同じ DRAM ページに対してメモリー・フィル・サイクルのロード操作/ストア操作を実行することで、メモリー・フィル・サイクルの帯域幅をかなり広げることができます。

6.7.2.3 ストア操作のアライメントを合わせることによる UC および WC のストア帯域幅の拡大

アライメントされたストア操作により UC メモリーまたは WC メモリーをフィルすることで、アライメントされていないストア操作に比べて帯域幅が広くなります。キャッシュライン境界をまたぐような UC ストアを 1 回または WC ストアを複数回実行する場合、ストア操作を 1 回実行するだけでバス上にトランザクションが 2 回発生し、バス・トランザクションの効率が低下します。ストア操作のアライメントをそのデータサイズに合わせることで、キャッシュライン境界をまたぐ可能性がなくなり、ストア操作が複数のトランザクションに分割されるのを回避できます。

6.7.3 リバース・メモリー・コピー

ソース・ロケーションからデスティネーション・ロケーションにメモリーブロックを降順でコピーするには、マイクロアーキテクチャー上のハザードを回避しながらマシンの能力を最大限に引き出す必要があり、ソフトウェアにとっては大きな課題です。例 5-40 に、最適化されていない基本 C コードを示します。

例 6-38 の簡単な C コードは、ハードウェア・プリフェッチによってデータがシステムメモリーからキャッシュに取り込まれる場合であっても、一度に 1 バイトずつロードとストアが行われるため、適切であるとは言えません。

例 6-38 最適化されていないリバース・メモリー・コピーの C コード

```
unsigned char* src;
unsigned char* dst;
while (len > 0)
{
*dst-- = *src++;
--len;
}
```

ソフトウェアでは、MOVDQA または MOVDQU 命令を使用して、一度に最大 16 バイトをロードおよびストアできます。ただし、16 バイト・アライメントの条件を満たすか (MOVDQA 命令を使用した場合)、データがキャッシュライン境界にまたがった場合 MOVDQU 命令で発生するレイテンシーを最小限に抑える必要があります。

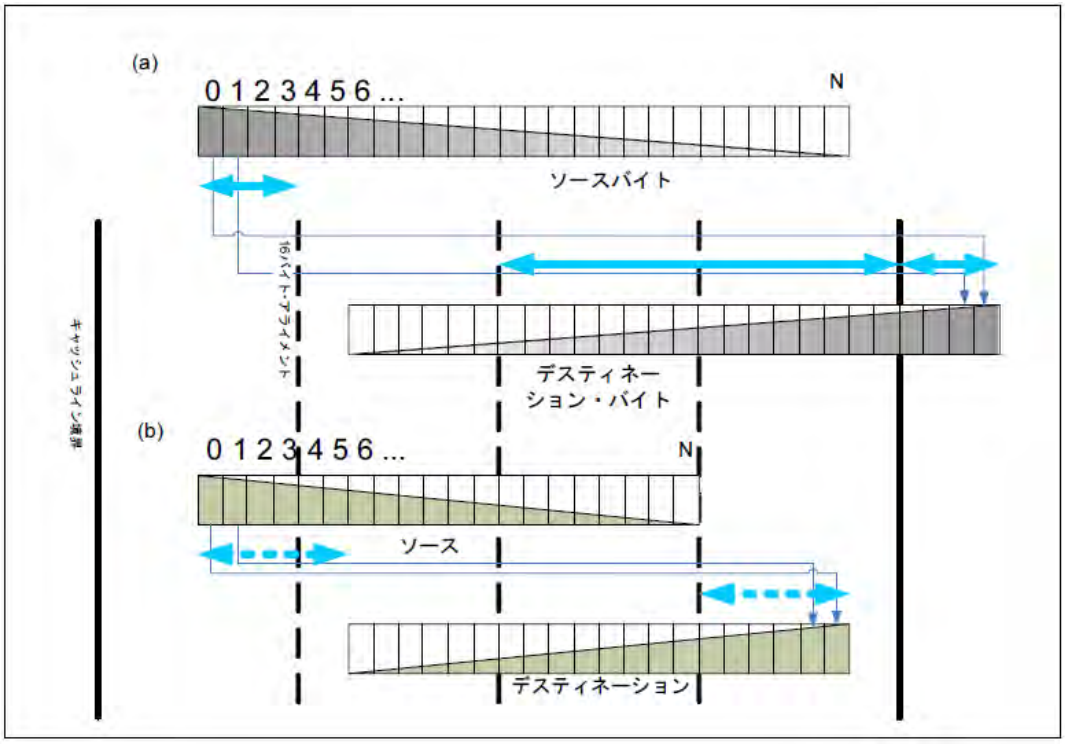


図 6-8 リバース・メモリー・コピーにおけるロードとストアのデータ・アライメント

コピーバイト数、先頭ソースバイトおよびデスティネーション・バイトのオフセット、16 バイト境界およびキャッシュライン境界に対するアドレス・アライメントなど、全般的な問題について考慮すると、このアライメントの状況は多少複雑です。図 6-8 の (a) と (b) に、N バイトのリバース・メモリー・コピーのアライメント状況を示します。

アライメントされていないロードおよびストアを扱う一般的なガイドラインを以下に示します (重要な順)。

- キャッシュライン境界をまたがったストアを回避します。
- キャッシュライン境界をまたがったロードの数を最小限に抑えます。
- アライメントされていないロードおよびストアよりも、16 バイトにアライメントされたロードおよびストアを優先します。

図 6-8 (a) では、以下のように上記のガイドラインをリバース・メモリー・コピーの問題に適用できます。

1. 16 バイト境界にアライメントされるまで先頭デスティネーション・バイトをいくつかピーリングします。その後は、残りのバイト数が 16 バイトを下回るまで、MOVAPS 命令を使用して後続デスティネーション・バイトに書き込みます。
2. 手順 1 と同様に先頭ソースバイトをピーリングした後は、図 6-8 (a) のソース・アライメントによって、残りのバイト数が 16 バイトを下回るまで、MOVAPS 命令を使用して 16 バイトをロードできます。

PSHUFB 命令を使用すると、16 バイト・マスクによって各 16 バイト・データ内のバイト順序を切り替えることができます。例 6-39 に、コードシーケンスを示します。

例 6-39 PSHUFB 命令を使用して一度に 16 バイトのバイト順序を逆転

```
__declspec(align(16)) static const unsigned char BswapMask[16] =
{15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0};
      mov esi, src
      mov edi, dst
      mov ecx, len
      movaps xmm7, BswapMask
start:
      movdqa xmm0, [esi]
      pshufb xmm0, xmm7
      movdqa [edi-16], xmm0
      sub edi, 16
      add esi, 16
      sub ecx, 16
      cmp ecx, 32
      jae start
      // 残りを処理
```

図 6-8 (b) でも、デスティネーション・バイトのピーリングから開始します。

1. 16 バイト境界にアライメントされるまで先頭デスティネーション・バイトをいくつかピーリングします。その後は、残りの d バイト数が 16 バイトを下回るまで、MOVAPS 命令を使用して後続デスティネーション・バイトに書き込みます。ただし、残りのソースバイトは 16 バイト境界にアライメントされず、MOVDQA 命令に代わって MOVDQU 命令を使用してロードを行うと、必然的にキャッシュラインが分割されます。
2. MOVDQU 命令を使用してアライメントされていないバイトのロードよりも高いデータ・スループットを達成するには、アライメントされている 2 つのロードを使用して、デスティネーション・アドレスの各 16 バイトにターゲットを定めた 16 バイト・データを構成するようにします。この手法を図 6 -9 に示します。

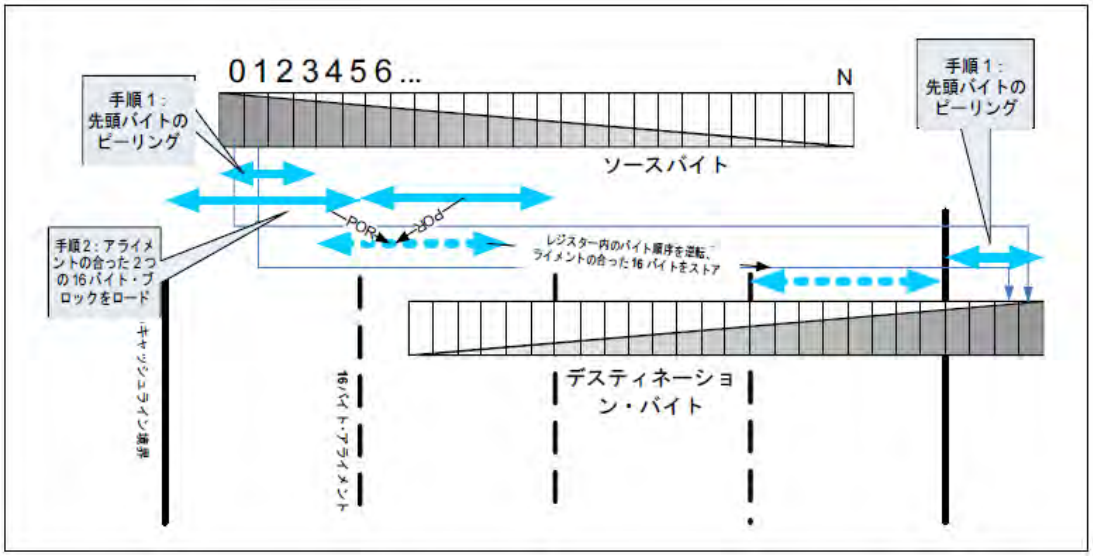


図 6-9 アライメントされている 2 つのロードを使用したリバース・メモリー・コピーにおけるキャッシュライン分割ロードを回避する手法

6.8 64 ビットから 128 ビット SIMD 整数への変換

インテル® SSE2 では、インテル® MMX® テクノロジーで利用できる操作の 128 ビット整数命令のスーパーセットが定義されています。そのため、拡張命令の動作は同じままです。スーパーセットでは、幅が 2 倍のデータを簡単に処理できます。

そのため、64 ビット整数アプリケーションの移植は容易です。ただし、以下に示すような考慮すべき点があります。

- 16 バイト境界にアライメントできないメモリーオペランドを使用する演算命令は、アライメントされていない 128 ビット・ロード (movdqu 命令) に置き換え、その後に、レジスターオペランドに代わり同じ演算処理を置く必要があります。

  16 バイト境界にアライメントされていないメモリーオペランドを用いた 128 ビット整数演算命令を使用すると、一般保護エラー (#GP) が発生します。アライメントされていない 128 ビットのロード操作/ストア操作は、アライメントされた操作ほど効率は良くなく、128 ビット SIMD 整数拡張命令を使用してもパフォーマンス改善は見込めません。

- メモリーオペランドのアライメントを合わせる一般的なガイドラインを次に示します。
  - メモリーストリームをすべて 16 バイト境界にアライメントすれば、パフォーマンス改善の度合いが最大になります。
  - メモリーストリーム全体の約半分を 16 バイト境界にアライメントすれば、残り半分をアライメントしない場合でも、ほどほどにパフォーマンスは改善されます。
  - メモリーストリームを全く 16 バイト境界にアライメントしない場合、パフォーマンスはほとんど改善されません。この場合、64 ビット SIMD 整数命令を使用した方が望ましい結果が得られることがあります。

- 各 128 ビット整数命令は、64 ビット整数命令の 2 倍の量のデータを処理するため、ループカウンターを更新する必要があります。
- 128 ビット・オペランド全体に pshufw 命令 (64 ビット整数オペランド全体にワードをシャッフル) を拡大適用する処理では、pshufhw、pshuflw、pshufd の各命令を組み合わせてエミュレートします。
- ビット命令 (psrlq、psllq) による 64 ビット・シフトを使用する場合、以下の方法で 128 ビットに拡張します。
  - 論理演算のマスク操作と組み合わせて psrlq 命令と psllq 命令を使用します。
  - psrldq 命令と pslldq 命令 (ダブル・クワッドワード・オペランドを数バイト、シフトする) を使用するようにコードシーケンスを書き直します。

6.8.1 SIMD の最適化とマイクロアーキテクチャー

インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、およびインテル® Core™ Duo プロセッサーのマイクロアーキテクチャーは、Intel NetBurst® マイクロアーキテクチャーとは異なります。以下の節では、インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーを対象とした SIMD コードの最適化について説明します。

インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーの場合、LDDQU 命令の動作は MOVDQU 命令と同じであり、アドレス・アライメントとは無関係に 16 バイトのデータをロードします。

6.8.1.1 インテル® SSE2 のパックド整数命令とインテル® MMX® 命令の比較

インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーでは通常、64 ビットのインテル® MMX® 命令よりも 128 ビットの SIMD 整数命令を優先すべきです。理由は以下のとおりです。

- インテル® Pentium® M プロセッサーと比べてデコーダーの帯域幅が広く、マイクロオペレーション (uop) のフローが効率的であるため。
- 幅が広い XMM レジスターは、デコーダーの帯域幅または実行レイテンシーによって制限されるコードに対してメリットがあるため。XMM レジスターでは、パイプラインで処理中のデータを格納するスペースが 2 倍になるため。幅の広い XMM レジスターを使用することで、ループのアンロールを容易にでき、またループの反復回数を半分にすると、ループのオーバーヘッドを削減できるため。

インテル® Core™ マイクロアーキテクチャー以前のマイクロアーキテクチャーでは、128 ビット SIMD 整数演算の実行スループットは、基本的には 64 ビットのインテル® MMX 命令と同じです。一部のシャッフル/アンパック/シフト操作では、フロントエンドの改善によるメリットが得られません。インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーで 128 ビット SIMD 整数命令を使用すると、全体的にわずかのプラスになりますが、特定の状況ではパフォーマンスへの悪影響が生じます。

インテル® Core™ マイクロアーキテクチャーでは一般に、レイテンシーとスループットの面で従来のマイクロアーキテクチャーよりも効率的に 128 ビット SIMD 命令が実行されます。インテル® Core™ Duo プロセッサーとインテル® Core™ Solo プロセッサーに固有の制限の多くは適用されません。Intel NetBurst® マイクロアーキテクチャーに対するインテル® Core™マイクロアーキテクチャーの関係についても、同じことが当てはまります。

拡張版インテル® Core™ マイクロアーキテクチャーには、インテル® Core™ マイクロアーキテクチャーよりもさらに強力な 128 ビット SIMD 実行機能と包括的な SIMD 拡張命令セットが用意されています。インテル® SSE4.1 で提供される SIMD 整数命令は、128 ビット XMM レジスターでのみ処理を行います。このような理由から、拡張版インテル® Core™ マイクロアーキテクチャーやインテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーを活用するには、ベクトル化可能な 128 ビット・コードを優先することを強く推奨します。

6.8.1.2 誤った依存関係の問題の回避策

Nehalem† マイクロアーキテクチャー・ベースのプロセッサーでは、PMOVSX 命令と PMOVZX 命令を使用して 1 つの命令でデータ型変換とデータ移動を組み合わせると、ハードウェア上の制限による誤った依存関係が発生します。誤った依存関係の問題を防止する簡単な回避策として、データ型変換にのみ PMOVSX 命令と PMOVZX 命令を使用し、移動先や移動元とのデータの移動には別の命令を使用します。

例 6-40 PMOVSX/PMOVZX における誤った依存関係の回避策

```
# 以下の命令を発行すると、xmm0 で誤った依存性が発生する
   pmovzxbd xmm0, dword ptr [eax]
// 次の命令では、xmm0 が他の命令により
// インフライトで更新されるとブロックされる
.....................................................................
# 誤った依存性を回避する方法
   movd xmm0, dword ptr [eax]
                          ; 000 ハードウェアは、遅延を隠蔽するためロードをずらすことができる
   pmovsxbd xmm0, xmm0
```

6.9 部分的にベクトル化可能なコードのチューニング

一部のループ構造コードは、ほかのコードに比べてベクトル化が困難なことがあります。例 6-41 に、テーブル・ルックアップ操作と算術演算を実行するループを示します。

例 6-41 C コードでのテーブル・ルックアップ操作

```
// pIn1 整数入力の配列
// pOut 整数出力の配列
// count 配列サイズ
// LookUpTable 整数値
TABLE_SIZE ルックアップ・テーブルのサイズ
for (unsigned i=0; i < count; i++)
{  pOut[i] =
   ( ( LookUpTable[pIn1[i] % TABLE_SIZE] + pIn1[i] + 17 ) | 17
   ) % 256;
}
```

一部の算術演算と各反復でのデータ配列への入出力は容易にベクトル化可能ですが、インデックス配列を利用したテーブル・ルックアップはベクトル化が困難です。そのため、さまざまなチューニング手法が考えられました。コンパイラーは、スカラー手法を利用して各反復を逐次的に実行できます。このようなループに対する手動でのチューニングでは、2 つの異なる手法を利用して、ベクトル化が困難なテーブル・ルックアップ操作を処理できます。

1 つ目のベクトル化手法としては、4 回の反復に対する入力データを一度にロードした後、インテル® SSE2 命令を使用して、XMM レジスターから個々のインデックスをシフトアウトし、テーブル・ルックアップを逐次実行します。例 6-42 にこのシフト手法を示します。もう 1 つの手法では、インテル® SSE4.1 の PEXTRD 命令を使用して、XMM からインデックスを直接抽出した後、テーブル・ルックアップを逐次実行します。例 6-43 に PEXTRD 命令による手法を示します。

例 6-42 ベクトル化不可能なテーブル·ルックアップのシフト手法

```
int modulo[4] = {256-1, 256-1, 256-1, 256-1};
int c[4] = {17, 17, 17, 17};
   mov esi, pIn1
   mov ebx, pOut
   mov ecx, count
   mov edx, pLookUpTablePTR
   movaps xmm6, modulo
   movaps xmm5, c
lloop:
// ベクトル化可能な連続したデータアクセス
   movaps xmm4, [esi] // pIn1 から 4 つのインデックスを読み込み
   movaps xmm7, xmm4
   pand xmm7, tableSize
// テーブル参照はベクトル化できない。テーブルを参照し 1 つずつデータ要素をシフト
   movd eax, xmm7 // 最初のインデックスを取得
   movd xmm0, word ptr[edx + eax*4]
   psrldq xmm7, 4
   movd eax, xmm7 // 2 番目のインデックスを取得
   movd xmm1, word ptr[edx + eax*4]
   psrldq xmm7, 4
   movd eax, xmm7 // 3 番目のインデックスを取得
   movd xmm2, word ptr[edx + eax*4]
   psrldq xmm7, 4
   movd eax, xmm7 // 4 番目のインデックスを取得
   movd xmm3, word ptr[edx + eax*4]
// スカラーコードの終わり
// パック操作
   movlhps xmm1,xmm3
   psllq xmm1,32
   movlhps xmm0,xmm2
   orps xmm0,xmm1
// パック操作の終わり
// ベクトル化可能な操作
   paddd xmm0, xmm4 //+pIn1
   paddd xmm0, xmm5 // +17
   por xmm0, xmm5
   andps xmm0, xmm6 //mod
   movaps [ebx], xmm0
// ベクトル化可能な操作の終わり
   add ebx, 16
   add esi, 16
   add edi, 16
   sub ecx, 1
   test ecx, ecx
   jne lloop
          (続く)
```

例 6-43 ベクトル化不可能なテーブル・ルックアップの PEXTRD 手法

```
int modulo[4] = {256-1, 256-1, 256-1, 256-1};
int c[4] = {17, 17, 17, 17};
   mov esi, pIn1
   mov ebx, pOut
   mov ecx, count
   mov edx, pLookUpTablePTR
   movaps xmm6, modulo
   movaps xmm5, c
lloop:
// ベクトル化可能な連続したデータアクセス
   movaps xmm4, [esi] // pIn1 から 4 つのインデックスを読み込み
   movaps xmm7, xmm4
   pand xmm7, tableSize
// テーブル参照はベクトル化できない。テーブルを参照し 1 つずつデータ要素を抽出
   movd eax, xmm7 // 最初のインデックスを取得
   mov eax, [edx + eax*4]
   movd xmm0, eax
   pextrd eax, xmm7, 1 // 2 番目のインデックスを取得
   mov eax, [edx + eax*4]
   pinsrd xmm0, eax, 1
   pextrd eax, xmm7, 2 // 2 番目のインデックスを取得
   mov eax, [edx + eax*4]
   XMM0, eax, 2
   pextrd eax, xmm7, 3 // 2 番目のインデックスを取得
   mov eax, [edx + eax*4]
   XMM0, eax, 2
// スカラーコードの終わり
// パックする必要はない
// ベクトル化可能な操作
   paddd xmm0, xmm4 //+pIn1
   paddd xmm0, xmm5 // +17
   por xmm0, xmm5
   andps xmm0, xmm6 //mod
   movaps [ebx], xmm0
   add ebx, 16
   add esi, 16
   add edi, 16
   sub ecx, 1
   test ecx, ecx
   jne lloop
```

部分的にベクトル化可能なコードに対するこの 2 つの手動チューニング手法の効果は、各種のパック/アンパック命令を使用してデータレイアウト形式を変換する際の相対コストによって異なります。

シフト手法では、スカラーテーブル値を XMM にパックして、ベクトル化算術演算に移行させる命令の追加が必要です。この手法による最終的なパフォーマンスの向上/低下は、マイクロアーキテクチャーごとの特性によって異なります。もう 1 つの PEXTRD 命令による手法では、各インデックスの抽出に使用する命令が少なく、ベクトル化算術演算を開始する際に、パック (スカラーデータをパックド SIMD データ形式にパック) は不要です。

6.10 並列モードの AES 暗号化/復号

ソフトウェアで計算を順序付けて複数のブロックを並列に処理することによって、インテル® AES-NI を使用して最適な暗号化および復号スループットを実現できます。これにより、シリアルモードの演算である CBC-Encrypt と比較して、ECB、CTR、CBC-Decrypt などの並列モードの演算での暗号化 (および復号) の速度が向上します。詳細は、『Recommendation for Block Cipher Modes of Operation』を参照してください。関連ドキュメントの節でこのドキュメントへの参照先を示します。

Sandy Bridge† マイクロアーキテクチャーでは、AES ラウンド命令 (AESENC/AESECNLAST/AESDEC/AESDECLAST) のスループットは 1 サイクル、レイテンシーは 8 サイクルです。そのため、データを十分高速に提供できる場合、複数のブロックに対する単独の AES 命令をサイクルごとにディスパッチできます。スループットが 2 サイクル、レイテンシーが 6 サイクルの命令を使用する以前の Westmere† マイクロアーキテクチャーと比較して、並列モードの演算の AES 暗号化/復号スループットが大幅に向上しています。

複数のブロックに対して最適な並列演算を実現するには、1 つのラウンドキーを使用して複数のブロックに対して 1 つの AES ラウンドを計算し、その後、別のラウンドキーを使用して複数のブロックに対して後続のラウンドを計算し続けるように、AES ソフトウェア・シーケンスを記述します。

ソフトウェアをこのように最適化するには、並列に処理するブロック数を定義する必要があります。Sandy Bridge† マイクロアーキテクチャーでは、最適な並列化パラメーターは 8 ブロックです (以前のマイクロアーキテクチャーでは 4 ブロック)。

6.10.1 AES カウンターモードの演算

例 6-44 は、並列に 8 ブロックを処理しながら、カウンターモード (CTR モード) の演算を実装する関数の一例です。以下の擬似コードでは、それぞれが 16 バイトの n 個のデータブロックを暗号化します (PT[i])。

例 6-44 AES カウンターモード演算の擬似コードフロー

```
CTRBLK := NONCE || IV || ONE
FOR i := 1 to n-1 DO
   CT[i] := PT[i] XOR AES(CTRBLK)
   CTRBLK := CTRBLK + 1) % 256;
END
CT[n] := PT[n] XOR TRUNC(AES(CTRBLK)) CTRBLK := NONCE || IV || ONE
FOR i := 1 to n-1 DO
   CT[i] := PT[i] XOR AES(CTRBLK)// CT [i] は、i 番目の暗号テキストのブロック
   CTRBLK := CTRBLK + 1
END
CT[n]:= PT[n] XOR TRUNC(AES(CTRBLK))
```

例 6-45 は、上記のコードを Sandy Bridge† マイクロアーキテクチャー用に最適化したアセンブリー実装を示しています。

例 6-45 8 ブロックを並列に処理する AES128-CTR 実装

```
/*************************************************************************/
/* この関数は CTR モードの AES 命令を利用して入力バッファーを暗号化する */
/* 引数: */
/* const unsigned char *in - 暗号化する平文字もしくは復号化する */
/* 暗号文へのポインター */
/* unsigned char *out - 暗号化/ 復号化されたデータがストアされる */
/* バッファーへのポインター */
/* const unsigned char ivec[8] - 8 バイトの初期化ベクトル */
/* const unsigned char nonce[4] - 臨時の 4 バイト */
/* const unsigned long length - 入力長をバイトで示す */
/* AES 丸めの値 10 = AES128, 12 = AES192,14 = AES256 */
/* unsigned char *key_schedule - AES キーへのポインター*/
/*************************************************************************/
// void AES_128_CTR_encrypt_parallelize_8_blocks_unrolled (
// const unsigned char *in,
// unsigned char *out,
// const unsigned char ivec[8],
// const unsigned char nonce[4],
// const unsigned long length,
// unsigned char *key_schedule)
.align 16,0x90
.align 16
ONE: .quad 0x00000000,0x00000001
.align 16
FOUR: .quad 0x00000004,0x00000004
.align 16
EIGHT: .quad 0x00000008,0x00000008
.align 16
TWO_N_ONE: .quad 0x00000002,0x00000001
.align 16
TWO_N_TWO: .quad 0x00000002,0x00000002
.align 16

LOAD_HIGH_BROADCAST_AND_BSWAP: .byte 15,14,13,12,11,10,9,8
                               .byte 15,14,13,12,11,10,9,8
align 16
BSWAP_EPI_64: .byte 7,6,5,4,3,2,1,0
              .byte 15,14,13,12,11,10,9,8
.globl AES_CTR_encrypt

AES_CTR_encrypt:
# parameter 1: %rdi # parameter 2: %rsi
# parameter 3: %rdx # parameter 4: %rcx
# parameter 5: %r8 # parameter 6: %r9
# parameter 7: 8 + %rsp
movq %r8, %r10
   movl 8(%rsp), %r12d
   shrq $4, %r8
   shlq $60, %r10
   je NO_PARTS
   addq $1, %r8
   (続く)
```

```
NO_PARTS:
   movq %r8, %r10
   shlq $61, %r10
   shrq $61, %r10

   pinsrq $1, (%rdx), %xmm0
   pinsrd $1, (%rcx), %xmm0
   psrldq $4, %xmm0
   movdqa %xmm0, %xmm4
   pshufb (LOAD_HIGH_BROADCAST_AND_BSWAP), %xmm4
   paddq (TWO_N_ONE), %xmm4
   movdqa %xmm4, %xmm1
   paddq (TWO_N_TWO), %xmm4
   movdqa %xmm4, %xmm2
   paddq (TWO_N_TWO), %xmm4
   movdqa %xmm4, %xmm3
   paddq (TWO_N_TWO), %xmm4
   pshufb (BSWAP_EPI_64), %xmm1
   pshufb (BSWAP_EPI_64), %xmm2
   pshufb (BSWAP_EPI_64), %xmm3
   pshufb (BSWAP_EPI_64), %xmm4

shrq $3, %r8
   je REMAINDER
   subq $128, %rsi
   subq $128, %rdi
LOOP:
   addq $128, %rsi
   addq $128, %rdi

   movdqa %xmm0, %xmm7
   movdqa %xmm0, %xmm8
   movdqa %xmm0, %xmm9
   movdqa %xmm0, %xmm10
   movdqa %xmm0, %xmm11
   movdqa %xmm0, %xmm12
   movdqa %xmm0, %xmm13
   movdqa %xmm0, %xmm14

   shufpd $2, %xmm1, %xmm7
   shufpd $0, %xmm1, %xmm8
   shufpd $2, %xmm2, %xmm9
   shufpd $0, %xmm2, %xmm10
   shufpd $2, %xmm3, %xmm11
   shufpd $0, %xmm3, %xmm12
   shufpd $2, %xmm4, %xmm13
   shufpd $0, %xmm4, %xmm14

   pshufb (BSWAP_EPI_64), %xmm1
   pshufb (BSWAP_EPI_64), %xmm2
   pshufb (BSWAP_EPI_64), %xmm3
   pshufb (BSWAP_EPI_64), %xmm4
   (続く)
```

```
movdqa (%r9), %xmm5
movdqa 16(%r9), %xmm6

paddq (EIGHT), %xmm1
paddq (EIGHT), %xmm2
paddq (EIGHT), %xmm3
paddq (EIGHT), %xmm4

pxor %xmm5, %xmm7
pxor %xmm5, %xmm8
pxor %xmm5, %xmm9
pxor %xmm5, %xmm10

pxor %xmm5, %xmm11
pxor %xmm5, %xmm12
pxor %xmm5, %xmm13
pxor %xmm5, %xmm14

pshufb (BSWAP_EPI_64), %xmm1
pshufb (BSWAP_EPI_64), %xmm2
pshufb (BSWAP_EPI_64), %xmm3
pshufb (BSWAP_EPI_64), %xmm4

aesenc %xmm6, %xmm7
aesenc %xmm6, %xmm8
aesenc %xmm6, %xmm9
aesenc %xmm6, %xmm10
aesenc %xmm6, %xmm11
aesenc %xmm6, %xmm12
aesenc %xmm6, %xmm13
aesenc %xmm6, %xmm14

movdqa 32(%r9), %xmm5
movdqa 48(%r9), %xmm6

aesenc %xmm5, %xmm7
aesenc %xmm5, %xmm8
aesenc %xmm5, %xmm9
aesenc %xmm5, %xmm10
aesenc %xmm5, %xmm11
aesenc %xmm5, %xmm12
aesenc %xmm5, %xmm13
aesenc %xmm5, %xmm14

aesenc %xmm6, %xmm7
aesenc %xmm6, %xmm8
aesenc %xmm6, %xmm9
aesenc %xmm6, %xmm10
aesenc %xmm6, %xmm11
aesenc %xmm6, %xmm12
aesenc %xmm6, %xmm13
aesenc %xmm6, %xmm14
(続く)
```

```
movdqa 64(%r9), %xmm5
movdqa 80(%r9), %xmm6

aesenc %xmm5, %xmm7
aesenc %xmm5, %xmm8
aesenc %xmm5, %xmm9
aesenc %xmm5, %xmm10
aesenc %xmm5, %xmm11
aesenc %xmm5, %xmm12
aesenc %xmm5, %xmm13
aesenc %xmm5, %xmm14

aesenc %xmm6, %xmm7
aesenc %xmm6, %xmm8
aesenc %xmm6, %xmm9
aesenc %xmm6, %xmm10
aesenc %xmm6, %xmm11
aesenc %xmm6, %xmm12
aesenc %xmm6, %xmm13
aesenc %xmm6, %xmm14

movdqa 96(%r9), %xmm5
movdqa 112(%r9), %xmm6

aesenc %xmm5, %xmm7
aesenc %xmm5, %xmm8
aesenc %xmm5, %xmm9
aesenc %xmm5, %xmm10
aesenc %xmm5, %xmm11
aesenc %xmm5, %xmm12
aesenc %xmm5, %xmm13
aesenc %xmm5, %xmm14

aesenc %xmm6, %xmm7
aesenc %xmm6, %xmm8
aesenc %xmm6, %xmm9
aesenc %xmm6, %xmm10
aesenc %xmm6, %xmm11
aesenc %xmm6, %xmm12
aesenc %xmm6, %xmm13
aesenc %xmm6, %xmm14

movdqa 128(%r9), %xmm5
movdqa 144(%r9), %xmm6
movdqa 160(%r9), %xmm15
cmp $12, %r12d

aesenc %xmm5, %xmm7
aesenc %xmm5, %xmm8
aesenc %xmm5, %xmm9
aesenc %xmm5, %xmm10
aesenc %xmm5, %xmm11
(続く)
```

```
aesenc %xmm5, %xmm12
aesenc %xmm5, %xmm13
aesenc %xmm5, %xmm14

aesenc %xmm6, %xmm7
aesenc %xmm6, %xmm8
aesenc %xmm6, %xmm9
aesenc %xmm6, %xmm10
aesenc %xmm6, %xmm11
aesenc %xmm6, %xmm12
aesenc %xmm6, %xmm13
aesenc %xmm6, %xmm14

jb LAST

movdqa 160(%r9), %xmm5
movdqa 176(%r9), %xmm6
movdqa 192(%r9), %xmm15
cmp $14, %r12d

aesenc %xmm5, %xmm7
aesenc %xmm5, %xmm8
aesenc %xmm5, %xmm9
aesenc %xmm5, %xmm10
aesenc %xmm5, %xmm11
aesenc %xmm5, %xmm12
aesenc %xmm5, %xmm13
aesenc %xmm5, %xmm14

aesenc %xmm6, %xmm7
aesenc %xmm6, %xmm8
aesenc %xmm6, %xmm9
aesenc %xmm6, %xmm10

aesenc %xmm6, %xmm11
aesenc %xmm6, %xmm12
aesenc %xmm6, %xmm13
aesenc %xmm6, %xmm14
jb LAST

movdqa 192(%r9), %xmm5
movdqa 208(%r9), %xmm6
movdqa 224(%r9), %xmm15

aesenc %xmm5, %xmm7
aesenc %xmm5, %xmm8
aesenc %xmm5, %xmm9
aesenc %xmm5, %xmm10
aesenc %xmm5, %xmm11
aesenc %xmm5, %xmm12
aesenc %xmm5, %xmm13
aesenc %xmm5, %xmm14
(続く)
```

```
   aesenc %xmm6, %xmm7
   aesenc %xmm6, %xmm8
   aesenc %xmm6, %xmm9
   aesenc %xmm6, %xmm10
   aesenc %xmm6, %xmm11
   aesenc %xmm6, %xmm12
   aesenc %xmm6, %xmm13
   aesenc %xmm6, %xmm14

LAST:
    aesenclast %xmm15, %xmm7
    aesenclast %xmm15, %xmm8
    aesenclast %xmm15, %xmm9
    aesenclast %xmm15, %xmm10
    aesenclast %xmm15, %xmm11
    aesenclast %xmm15, %xmm12

    aesenclast %xmm15, %xmm13
    aesenclast %xmm15, %xmm14
    pxor (%rdi), %xmm7
    pxor 16(%rdi), %xmm8

    pxor 32(%rdi), %xmm9
    pxor 48(%rdi), %xmm10
    pxor 64(%rdi), %xmm11
    pxor 80(%rdi), %xmm12

    pxor 96(%rdi), %xmm13
    pxor 112(%rdi), %xmm14

    dec %r8
    movdqu %xmm7, (%rsi)
    movdqu %xmm8, 16(%rsi)
    movdqu %xmm9, 32(%rsi)
    movdqu %xmm10, 48(%rsi)
    movdqu %xmm11, 64(%rsi)
    movdqu %xmm12, 80(%rsi)
    movdqu %xmm13, 96(%rsi)
    movdqu %xmm14, 112(%rsi)
    jne LOOP

    addq $128,%rsi
    addq $128,%rdi

REMAINDER:
    cmp $0, %r10
    je END
    shufpd $2, %xmm1, %xmm0
IN_LOOP:
    movdqa %xmm0, %xmm11
    pshufb (BSWAP_EPI_64), %xmm0
    pxor (%r9), %xmm11
    (続く)
```

```
    paddq (ONE), %xmm0
    aesenc 16(%r9), %xmm11
    aesenc 32(%r9), %xmm11
    pshufb (BSWAP_EPI_64), %xmm0
    aesenc 48(%r9), %xmm11
    aesenc 64(%r9), %xmm11
    aesenc 80(%r9), %xmm11

    aesenc 96(%r9), %xmm11
    aesenc 112(%r9), %xmm11
    aesenc 128(%r9), %xmm11
    aesenc 144(%r9), %xmm11
    movdqa 160(%r9), %xmm2

    cmp $12, %r12d
    jb IN_LAST
    aesenc 160(%r9), %xmm11
    aesenc 176(%r9), %xmm11
    movdqa 192(%r9), %xmm2

    cmp $14, %r12d
    jb IN_LAST
    aesenc 192(%r9), %xmm11
    aesenc 208(%r9), %xmm11
    movdqa 224(%r9), %xmm2
IN_LAST:
    aesenclast %xmm2, %xmm11
    pxor (%rdi) ,%xmm11
    movdqu %xmm11, (%rsi)
    addq $16,%rdi
    addq $16,%rsi
    dec %r10
    jne IN_LOOP
END:
    ret
```

6.10.2 AES キー拡張の代替手法

Sandy Bridge† マイクロアーキテクチャーでは、AESKEYGENASSIST のスループットは 2 サイクルであり、レイテンシーは AESENC/AESDEC 命令よりも高くなります。ソフトウェアでは、AESENCLAST 命令の 2 番目のオペランド (つまり、ラウンドキー) を RCON 値にし、レジスターで 4 回複製することによって、AES キー拡張を実装する方法も検討できます。ROTWORD ステップを PSHUFB 命令で実行している一方、AESENCLAST 命令はサブバイトによる処理と RCON 値による xor を実行します。以下のコード例は、両方の方法を使用した AES128 キー拡張です。

例 6-46 AES128 キー拡張

```
// AESKENYGENASSIST を利用する
.align 16,0x90
.globl AES_128_Key_Expansion
AES_128_Key_Expansion:
# parameter 1: %rdi
# parameter 2: %rsi
  movl $10, 240(%rsi)
  movdqu (%rdi), %xmm1
  movdqa %xmm1, (%rsi)
  aeskeygenassist $1, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 16(%rsi)
  aeskeygenassist $2, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 32(%rsi)
  aeskeygenassist $4, %xmm1, %xmm2

ASSISTS:
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 48(%rsi)
  aeskeygenassist $8, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 64(%rsi)
  aeskeygenassist $16, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 80(%rsi)
  aeskeygenassist $32, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 96(%rsi)
  aeskeygenassist $64, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 112(%rsi)
  aeskeygenassist $0x80, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 128(%rsi)
  aeskeygenassist

  $0x1b, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 144(%rsi)
  aeskeygenassist $0x36, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 160(%rsi)
  ret

PREPARE_ROUNDKEY_128:
  pshufd $255, %xmm2, %xmm2
  movdqa %xmm1, %xmm3
  pslldq $4, %xmm3
  pxor %xmm3, %xmm1
  pslldq $4, %xmm3
  pxor %xmm3, %xmm1
  pslldq $4, %xmm3
```

```
// AESENCLASTを利用する
mask:
.long
0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d
con1:
.long 1,1,1,1
con2:
.long 0x1b,0x1b,0x1b,0x1b
.align 16,0x90
.globl AES_128_Key_Expansion
AES_128_Key_Expansion:
# parameter 1: %rdi
# parameter 2: %rsi
    movdqu (%rdi), %xmm1
    movdqa %xmm1, (%rsi)
    movdqa %xmm1, %xmm2
    movdqa (con1), %xmm0
    movdqa (mask), %xmm15
    mov $8, %ax

LOOP1:
    add $16, %rsi
    dec %ax
    pshufb %xmm15,%xmm2
    aesenclast %xmm0, %xmm2
    pslld $1, %xmm0
    movdqa %xmm1, %xmm3
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pxor %xmm2, %xmm1
    movdqa %xmm1, (%rsi)
    movdqa %xmm1, %xmm2

    jne LOOP1
    movdqa (con2), %xmm0
    pshufb %xmm15,%xmm2
    aesenclast %xmm0, %xmm2
    pslld $1, %xmm0
    movdqa %xmm1, %xmm3
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pxor %xmm2, %xmm1
    movdqa %xmm1, 16(%rsi)
    movdqa %xmm1, %xmm2
    pshufb %xmm15,%xmm2
```

pxor %xmm3, %xmm1 pxor %xmm2, %xmm1 ret	aesenclast %xmm0, %xmm2 movdqa %xmm1, %xmm3 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pxor %xmm2, %xmm1 movdqa %xmm1, 32(%rsi) movdqa %xmm1, %xmm2 ret

6.10.3 Haswell† マイクロアーキテクチャーにおける強化

6.10.3.1 AES と複数バッファー暗号化のスループット

Haswell† マイクロアーキテクチャーでは、AESINC/AESINCLAST, AESDEC/ AESDECLAST 命令のレイテンシーが大幅に改善され、1 uop になっています。これらの改善は、AES アルゴリズムの並列モード (CBC 暗号化など) と複数バッファー実装の AES アルゴリズムに貢献すると考えられます。インテル® AES-NI の使用法の詳細と用例に関するいくつかのホワイトペーパーが公開されています。次のリンクを参照してください。

- http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set (英語)

6.10.3.2 PCLMULQDQ 命令の改善

Haswell† マイクロアーキテクチャーの PCLMULQDQ 命令のレイテンシーは、以前の世代と比較して 14 から 7 サイクルに軽減され、スループットは 8 サイクルごとから 1 サイクルおきに改善されています。

これは、一般的な多項式の CRC 計算をスピードアップします。詳細と用例については以下のドキュメントを参照してください。

- http://www.intel.com/Assets/PDF/manual/323640.pdf (英語)

PCLMULQDQ を使用した AES-GCM 実装は、OpenSSL プロジェクトで採用されています。

- http://www.intel.com/content/dam/www/public/us/en/documents/software-support/enabling-highperformance-gcm.pdf (英語)

6.11 軽量な展開とデータベース処理

一般的に、データベース・ストレージは、有限のディスク I/O 帯域幅の限界を維持するため、高い圧縮比の手法を必要とします。行指向で最適化されたデータベース・アーキテクチャーにおける、データベースの処理能力を制限する主な要因は、しばしばハードウェアによるストレージ I/O 帯域幅の制限、ストレージ形式から展開する必要がある大きなテーブル行からのデータレコードの局所性に関連します。近年のデータベースにおける革新は、連続してデータをフェッチするクエリー操作向けにストレージ形式が最適化されたカラムナ・データベース・アーキテクチャーを中心に行われています。

カラムナ・データベース (インメモリー・データベースとしても知られる) における近年の進歩として、軽量の圧縮/展開テクニックとインテル® SSE4.2 とほかの SIMD 命令を使用したクエリー操作プリミティブのベクトル化が上げられます。データベース・エンジンがこれらの処理手法とソリッド・ステート・ドライブを使用する列指向に最適化されたス

トレージシステムと結合することで、数倍のクエリー・パフォーマンスの向上が報告されています[9]。この節では、カラムナ・データベースにおける軽量な圧縮/展開向けの SIMD 命令の利用法について説明します。

軽量の圧縮/展開の目的は、低い CPU 使用率で、クエリー処理と展開との間の有効総帯域幅をより適切に分配し、最大限のクエリー・スループットを達成するような高いスループットを提供することです。インテル® SSE4.2 では、汎用レジスター命令を使用したクエリー・プリミティブに比べ、ある種のクエリー操作で計算帯域幅をかなり高いレベルまで引き上げることができます (14.3.3 節を参照)。また、展開されたカラムナデータのストリーミング・データ供給に対する要求も高まります。

6.11.1 ダイナミック・レンジ・データセットの減少

高速でカラムナデータを圧縮/展開する最良のアプローチの 1 つは、固定サイズのストレージの連続したストリームにおいて整数値の集合は、パーティション化、共通参照値からのオフセット、およびその他の手法によりそのダイナミック・レンジを軽減できれば、よりコンパクトに表現できるというアイディアに基づきます[10,11]。

例えば、32 ビット整数として 5 桁の ZIPCODE をストアする列は、17 ビットのダイナミック・レンジのみを必要とします。2^N 個のエントリーの連続したブロックを分割してブロックヘッダーにオフセットを格納し、それぞれの 32 ビット整数のストレージサイズを N ビットに減らすことで、20 億の行テーブルの一意なプライマリー・キーを減らすことができます。

6.11.2 SIMD 命令を使用した圧縮と展開

ダイナミック・レンジが減少した圧縮/展開において、圧縮データ要素がバイト境界でアライメントされていない状況での SIMD 命令の使用方法を例証するため、ダイナミック・レンジが値ごとに 5 ビットのみの 32 ビット整数配列について考えます。

連続した 5 ビットのバケットへ 32 ビット整数値のストリームをパックするため、例 6-49 で示す SIMD 手法は次のフェーズを含みます。

- dword からバイトへのパックとバイト配列シーケンス。dword 要素のストリームは、それぞれの反復が 32 要素を処理するバイトストリームに縮小されます。2 つの 16 バイトの結果ベクトルは、PSLLD と PSRLD 命令を使用する 4 ウェイのビットスティッチを可能にするため連続しています。

例 6-47 5 ビットのバケットに 32 ビット整数を圧縮

```
;
static __declspec(align(16)) short mask_dw_5b[16] = // dword による 4 ウェイのビットパッキング向けの 5 ビット・マスク
{0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0}; // パックドシフト
static __declspec(align(16)) short sprdb_0_5_10_15[8] = // 再アレンジのためのシャッフル制御
{ 0xff00, 0xffff, 0x04ff, 0xffff, 0xffff, 0xff08, 0xffff, 0x0cff}; // 0, 5, 10, 15 のギャップ位置へのバイト 0, 4, 8 ,12

void RDRpack32x4_sse(int *src, int cnt, char * out)
(続く)
int i, j;
__m128i a0, a1, a2, a3, c0, c1, b0, b1, b2, b3, bb;
__m128i msk4 ;
```

[9] 非クラスターの TPC-H パフォーマンス結果は、www.tpc.org (英語) でご覧いただけます。

[10] “SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units”, T. Willhalm, et. al., Proceedings of the VLDB Endowment, Vol. 2, #1, August 2009。

[11] "Super-Scalar RAM-CPU Cache Compression," M. Zukowski, et, al, Data Engineering, International Conference, vol. 0, no. 0, pp. 59, 2006。

```
__m128i sprd4 = _mm_loadu_si128( (__m128i*) &sprdb_0_5_10_15[0]);
   switch( bucket_width) {
   case 5:j= 0;
   msk4 = _mm_loadu_si128( (__m128i*) &mask_dw_5b[0]);
   // 各反復で 32 要素を処理
   for (i = 0; i < cnt; i+= 32) {
       b0 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i]),
       _mm_loadu_si128( (__m128i*) &src[i+4]));
       b1 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+8]),
       _mm_loadu_si128( (__m128i*) &src[i+12]));

       b2 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+16]),
       _mm_loadu_si128( (__m128i*) &src[i+20]));
       b3 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+24]),
       _mm_loadu_si128( (__m128i*) &src[i+28]));
       c0 = _mm_packus_epi16( _mm_unpacklo_epi64(b0, b1), _mm_unpacklo_epi64(b2, b3));
       // c0 はバイトを含む: 0-3, 8-11, 16-19, 24-27 要素
       c1 = _mm_packus_epi16( _mm_unpackhi_epi64(b0, b1), _mm_unpackhi_epi64(b2, b3));
       // c1 はバイトを含む: 4-7, 12-15, 20-23, 28-31
       b0 = _mm_and_si128( c0, msk4); // keep lowest 5 bits in each way/dword
       b1 = _mm_and_si128( _mm_srli_epi32(c0, 3), _mm_slli_epi32(msk4, 5));
       b0 = _mm_or_si128( b0, b1); // add next 5 bits to each way/dword
       b1 = _mm_and_si128( _mm_srli_epi32(c0, 6), _mm_slli_epi32(msk4, 10));
       b0 = _mm_or_si128( b0, b1);
       b1 = _mm_and_si128( _mm_srli_epi32(c0, 9), _mm_slli_epi32(msk4, 15));
       b0 = _mm_or_si128( b0, b1);
       b1 = _mm_and_si128( _mm_slli_epi32(c1, 20), _mm_slli_epi32(msk4, 20));
       b0 = _mm_or_si128( b0, b1);
       b1 = _mm_and_si128( _mm_slli_epi32(c1, 17), _mm_slli_epi32(msk4, 25));
       b0 = _mm_or_si128( b0, b1);
       b1 = _mm_and_si128( _mm_slli_epi32(c1, 14), _mm_slli_epi32(msk4, 30));
       b0 = _mm_or_si128( b0, b1); // それぞれの dword チャンネルから xmm に次の 2 ビットを加算
       *(int*)&out[j] = _mm_cvtsi128_si32( b0);// 最初の dword が圧縮され準備完了
       // 残りの 3 つの dword を再分配し、残りのストアビットにギャップバイトを加算
       b0 = _mm_shuffle_epi8(b0, gap4x3);
       b1 = _mm_and_si128( _mm_srli_epi32(c1, 18), _mm_srli_epi32(msk4, 2)); // 次の 3 ビットの 4 ウェイパッキングを行う
       b2 = _mm_and_si128( _mm_srli_epi32(c1, 21), _mm_slli_epi32(msk4, 3));
       b1 = _mm_or_si128( b1, b2);
       // バイト 0, 4, 8, 12 で 5 番目の圧縮
       //  0, 5, 10, 15 バイト·オフセットへ 5 番目のバイト結果をシャッフル
       b0 = _mm_or_si128( b0, _mm_shuffle_epi8(b1, sprd4));
       _mm_storeu_si128( (__m128i *) &out[j+4] , b0);
       j += bucket_width*4;
   }
   // cnt が 32 の倍数でなければ残りを処理
   break;
   }
}
```

- 4 ウェイのビットスティッチ: デスティネーションの各ウェイ (dword) で、5 ビットは 5 つの非ゼロ·ビット·パターンを含む対応するバイト要素にパックされます。各 dword デスティネーションは、7 つの連続した要素の内容で埋められるため、7 番目の要素と 8 番目の要素の残りの 3 ビットは、同様の 4 ウェイスティッチ操作で個別に行われますが、シャッフル操作による支援が必要です。

例 6-48 は、連続してパックされた 5 ビットのバケットを 32 ビット・データ要素に展開する逆の操作を示しています。

例 6-48 5 ビット整数のストリームを 32 ビット要素へ展開

```
;
static __declspec(align(16)) short mask_dw_5b[16] = // dword による 4 ウェイのビットパッキング向けの 5 ビット・マスク
{0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0}; // パックドシフト
static __declspec(align(16)) short pack_dw_4x3[8] = // 3 つの dwords 1-4, 6-9, 11-14 をパック
{ 0xffff, 0xffff, 0x0201, 0x0403, 0x0706, 0x0908, 0xc0b, 0x0e0d}; // バイト 0-3 を空ける
static __declspec(align(16)) short packb_0_5_10_15[8] = // 再アレンジのためのシャッフル制御バイト
{ 0xffff, 0x0ff, 0xffff, 0x5ff, 0xffff, 0xaff, 0xffff, 0x0fff}; //  3, 7, 11, 15 のギャップ位置への 0, 5, 10, 15

void RDRunpack32x4_sse(char *src, int cnt, int * out)
{int i, j;
__m128i a0, a1, a2, a3, c0, c1, b0, b1, b2, b3, bb, d0, d1, d2, d3;
__m128i msk4 ;
__m128i pck4 = _mm_loadu_si128( (__m128i*) &packb_0_5_10_15[0]);
__m128i pckdw3 = _mm_loadu_si128( (__m128i*) &pack_dw_4x3[0]);

   switch( bucket_width) {
   case 5:j= 0;
   msk4 = _mm_loadu_si128( (__m128i*) &mask_dw_5b[0]);
   for (i = 0; i < cnt; i+= 32) {
      a1 = _mm_loadu_si128( (__m128i*) &src[j +4]);
      // 4, 9, 14, 19 バイトをピックアップし、オフセット 3, 7, 11, 15 へシャッフル
      c0 = _mm_shuffle_epi8(a1, pck4);
      b1 = _mm_and_si128( _mm_srli_epi32(c0, 3), _mm_slli_epi32(msk4, 24));
      // 3 つのアライメントされていない dword 1-4, 6-9, 11-14 を空きバイト 0-3 へ配置
      a1 = _mm_shuffle_epi8(a1, pckdw3);
      b0 = _mm_and_si128( _mm_srli_epi32(c0, 6), _mm_slli_epi32(msk4, 16));
      a0 = _mm_cvtsi32_si128( *(int *)&src[j ]);
      b1 = _mm_or_si128( b0, b1); // ソースバイト 4, 9, 14, 19 の展開を終了
      a0 = _mm_or_si128( a0, a1); // バイト 0-16 は圧縮されたビットを含む
      b0 = _mm_and_si128( _mm_srli_epi32(a0, 14), _mm_slli_epi32(msk4, 16));
      b1 = _mm_or_si128( b0, b1);
      b0 = _mm_and_si128( _mm_srli_epi32(a0, 17), _mm_slli_epi32(msk4, 8));
      b1 = _mm_or_si128( b0, b1);
      b0 = _mm_and_si128( _mm_srli_epi32(a0, 20), msk4);
      b1 = _mm_or_si128( b0, b1);// b1 は展開された 4-7,12-15,20-23,28-31 で満たされる
      _mm_storeu_si128( (__m128i *) &out[i+4] , _mm_cvtepu8_epi32(b1));
      b0 = _mm_and_si128( _mm_slli_epi32(a0, 9), _mm_slli_epi32(msk4, 24));
      c0 = _mm_and_si128( _mm_slli_epi32(a0, 6), _mm_slli_epi32(msk4, 16));
      b0 = _mm_or_si128( b0, c0);
      (続く)
      _mm_storeu_si128( (__m128i *) &out[i+12] ,
      _mm_cvtepu8_epi32(_mm_srli_si128(b1, 4)));
      c0 = _mm_and_si128( _mm_slli_epi32(a0, 3), _mm_slli_epi32(msk4, 8));
      _mm_storeu_si128( (__m128i *) &out[i+20] ,
      _mm_cvtepu8_epi32(_mm_srli_si128(b1, 8)));
```

```
      b0 = _mm_or_si128( b0, c0);
      _mm_storeu_si128( (__m128i *) &out[i+28] ,
      _mm_cvtepu8_epi32(_mm_srli_si128(b1, 12)));
      c0 = _mm_and_si128( a0, msk4);
      b0 = _mm_or_si128( b0, c0);// b0 は展開された 0-3,8-11,16-19,24-27 で満たされる
      _mm_storeu_si128( (__m128i *) &out[i] , _mm_cvtepu8_epi32(b0));
      _mm_storeu_si128( (__m128i *) &out[i+8] ,
      _mm_cvtepu8_epi32(_mm_srli_si128(b0, 4)));
      _mm_storeu_si128( (__m128i *) &out[i+16] ,
      _mm_cvtepu8_epi32(_mm_srli_si128(b0, 8)));
      _mm_storeu_si128( (__m128i *) &out[i+24] ,
      _mm_cvtepu8_epi32(_mm_srli_si128(b0, 12)));
      j += g_bwidth*4;
   }
   break;
   }
}
```

2 の累乗ではないダイナミック・レンジ向けの静数の圧縮/展開では、一般に部分的にスティッチされたベクトルの水平方向の再配置を加えて、同様のマスク/パックドシフト/スティッチ手法を使用します。汎用スカラー命令によるスループットの向上は、実装とバケット幅に依存します。

インテル® コンパイラーで「/O2」オプションを指定してコンパイルする場合、圧縮スループットは Sandy Bridge† マイクロアーキテクチャーでは 6 バイト/サイクルに達し、ストリーミング・データのアクセスパターンとハードウェア・プリフェッチャーの効果により、ワーキングセットのサイズによるスループットの変動はわずかです。上記の例における展開のスループットは、完全に活用された場合 5 バイト/サイクル以上であり、データベース・クエリー・エンジンが CPU 使用率を効率良く分割し、処理中の展開向けに小さな断片を割り当てることで、ベクトル化されたクエリー計算を行うことを可能にします。

SIMD 軽量圧縮手法を使用することで得られる展開スループットの向上は、データベースの設計者にクリティカルなパフォーマンス・ボトルネックをより低いスループット技術 (ディスク I/O、DRAM) からより高速なパイプラインに再配置する自由を提供します。

第 7 章
SIMD 浮動小数点アプリケーション向けの最適化

この章では、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSE4.1 で利用できる SIMD (single-instruction, multiple-data) 浮動小数点命令を最適化する際の規則について説明します。また、単精度および倍精度の SIMD 浮動小数点アプリケーション向けの最適化手法についても、具体例を示して説明します。

7.1 SIMD 浮動小数点コード向けの一般的な規則

この節で述べる規則と提案に従うことで、SIMD 浮動小数点命令を使用した浮動小数点コードの最適化が容易になります。一般に、効率の良い SIMD 浮動小数点コードを作成するには、ポート利用率を理解しバランスを取ることが重要です。次に基本的な規則を示します。

- 第 3 章と第 5 章に示したすべてのガイドラインに従います。
- 高いパフォーマンスを達成するため、例外をマスクします。例外をマスクしないと、ソフトウェアのパフォーマンスが低下します。
- デノーマル状態とアンダーフロー状態を処理するペナルティーを避け、高いパフォーマンスを実現するためゼロ・フラッシュ・モードと DAZ モードを利用します。
- 精度を上げるため、逆数命令に続けて反復処理を行います。逆数命令では精度は低下しますが、実行速度はかなり速くなります。次の点に注意してください。
  - 精度の低下が許容される場合には、反復処理を用いずに逆数命令を使用します。
  - 完全に近い精度が必要な場合は、ニュートン・ラフソン反復法を適用します。
  - 完全な精度が必要な場合は、より高い精度の得られる除算と平方根命令を使用します。ただし速度は低下します。

7.2 計画上の留意事項

既存のアプリケーションに導入する場合でも、新しくアプリケーションを作成する場合でも、SIMD 浮動小数点命令を使用して最適なパフォーマンスを得るには、考慮すべきいくつかの課題があります。一般に、最適化の候補を選ぶときは、大量の浮動小数点計算が行われているコードセグメントを特定します。また、キャッシュ・アーキテクチャーの効果的な使用も検討する必要があります。

以下の節では、実装前によく問われる次のような疑問に答えていきます。

- 並列処理やキャッシュの使用効率を向上させるデータレイアウトは可能であるか?
- コードのどの部分が SIMD 浮動小数点命令の効果を得られるか?
- 現在のアルゴリズムは SIMD 浮動小数点命令に最適であるか?
- 対象となるコードは浮動小数点を多用しているか?
- 単精度浮動小数点演算と倍精度浮動小数点演算では、どちらが十分な数値範囲と精度を得られるか?
- ゼロ・フラッシュ・モードや DAZ モードを有効にすると、計算結果に影響はあるか?
- SIMD 浮動小数点レジスターを効率良く利用できるようにデータが配置されているか?
- 対象となるアプリケーションは、SIMD 浮動小数点命令を実行できないプロセッサーをターゲットとしているか?

関連情報: 5.2 節「SIMD プログラミングにおけるコード変換に関する留意事項」も参照してください。

7.3 x87 浮動小数点と SIMD 浮動小数点との併用

SIMD 浮動小数点演算に使用される XMM レジスターは、独立したレジスターであり、既存の x87 浮動小数点スタックにはマッピングされません。そのため、SIMD 浮動小数点コードは、x87 浮動小数点コードや、64 ビット SIMD 整数コードと混在して利用できます。

インテル® Core™ マイクロアーキテクチャーでは、128 ビット SIMD 整数命令は 64 ビット SIMD 整数命令よりもはるかに高い効率を発揮します。ソフトウェアは、可能な限り SIMD 浮動小数点命令と SIMD 整数命令を XMM レジスターとともに使用すべきです。

7.4 スカラー浮動小数点コード

SIMD レジスターの最下位の要素しか演算しない SIMD 浮動小数点命令があります。これらの命令は、「スカラー命令」と呼ばれます。スカラー命令を用いると、汎用の浮動小数点演算に XMM レジスターを使用できるようになります。

パフォーマンス面から見ると、スカラー浮動小数点コードは、x87 浮動小数点コードと同等かそれ以上であり、次の利点があります。

- SIMD 浮動小数点コードはフラットなレジスターモデルを使用するのに対し、x87 浮動小数点コードではスタックモデルが使用されます。スカラー浮動小数点コードを使用することで、FXCH 命令は必要なくなります。FXCH 命令は、インテル® Pentium® 4 プロセッサーで若干の制約があります。
- インテル® MMX® テクノロジーと混在させてもペナルティーが生じません。
- ゼロ・フラッシュ・モードが利用できます。
- x87 浮動小数点よりもレイテンシーが小さくなります。

スカラー浮動小数点命令を使用する場合、データをベクトル形式で表現する必要はありません。ただし、アライメント、スケジューリング、命令の選択など、第 3 章と第 5 章で述べた最適化手法は適用する必要があります。

7.5 データ・アライメント

SIMD 浮動小数点データは 16 バイトにアライメントされます。アライメントされていない 128 ビット SIMD 浮動小数点データの参照では、MOVUPS 命令 (Move Unaligned Packed Single) や MOVUPD 命令 (Move Unaligned Packed Double) を使用しないと例外が発生します。非アライメント命令を使用すると、対象となるデータがアライメントされているかどうかに関係なく、アライメント命令と比べてパフォーマンスが劣ります。

関連情報: 5.4 節「スタックとデータ・アライメント」を参照してください。

7.5.1 データ配置

インテル® SSE にもインテル® SSE2 にも SIMD アーキテクチャーが適用されるため、SIMD レジスターを最大限に活用できるようにデータをうまく配置すれば、最適なパフォーマンスが得られます。これには、処理するデータが隙間なく連続して並んでいることが必要ですが、うまくデータを配置することでキャッシュミスが減少します。適切にデータを配置することで、インテル® SSE ではデータ・スループットが 4 倍になり、インテル® SSE2 ではデータ・スループットが 2 倍になる可能性もあります。パフォーマンス改善は、インテル® SSE (MOVAPS 命令) では 4 つの 32 ビット・データ要素を 128 ビット・ロード命令で XMM レジスターへロードできることからもたらされます。同様に、インテル® SSE2 (MOVAPD 命令) を使用することで、2 つの 64 ビット・データ要素を 128 ビット・ロード命令で XMM レジスターへロードできます。

データ配置に関する推奨事項は、5.4 節「スタックとデータ・アライメント」を参照してください。データの構造や配置によってはミスアライメントの問題が発生する場合がありますが、複製とパディングによって解決できます。このような手法はデータ領域を増加させますが、アライメントされていないデータアクセスのペナルティーを排除できます。

アプリケーション (3D ジオメトリーなど) によっては、従来のデータ配置を一部変更しないと、SIMD レジスターと並列処理を最大限に活用できません。従来、データレイアウトには、複数の構造体から 1 つの配列を構成する、いわゆる構造体配列 (AoS) が用いられてきました。こうしたアプリケーションで SIMD レジスターを最大限に活用するため、新たなデータレイアウトが提案されています。それは、複数の配列から 1 つの構造体を構成する、いわゆる配列構造体 (SoA) と呼ばれ、より高いパフォーマンスが得られます。

7.5.1.1 垂直計算と水平計算

インテル® SSE とインテル® SSE2 の浮動小数点演算命令の大半は、並列データ要素の垂直データ処理において高いパフォーマンスを発揮します。つまり、デスティネーションの各要素は、同じ垂直位置にあるソース要素によって行われた算術演算の結果であることを意味します (図 7-1)。

またインテル® SSE とインテル® SSE2 では、並列データ要素の同種算術演算を補助するため、データ移動命令 (SHUFPS、UNPCKLPS、UNPCKHPS、MOVLHPS、MOVHLPS など) を提供してデータ要素の水平移動を容易にしています。

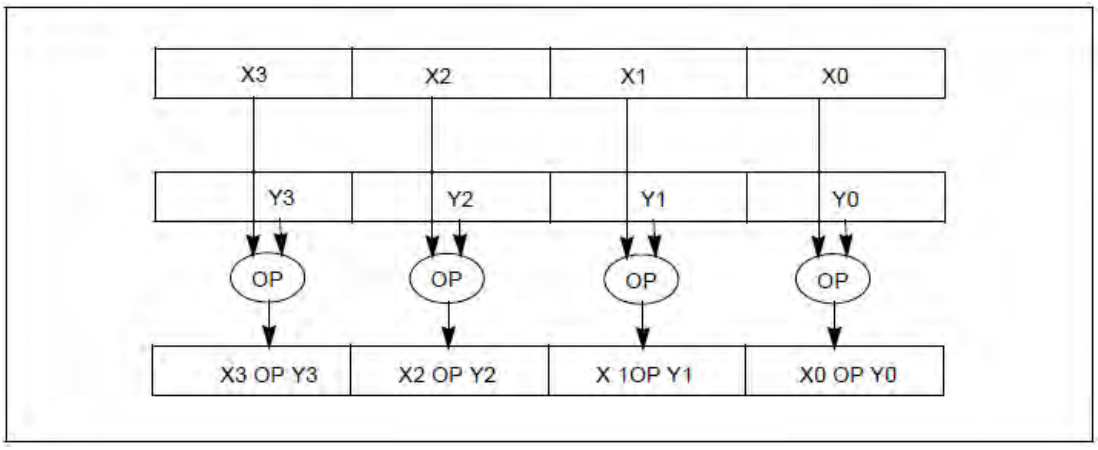


図 7-1 並列データ要素の同種演算

構造化されたデータの編成は、SIMD プログラミングの効率とパフォーマンスに大きな影響を与えます。これについては、データ構造を編成する 2 つの一般的な形式によって説明できます。

- 構造体配列 (AoS): これは、複数のデータ構造を 1 つの配列に並べたものです。データ構造内の各メンバーはスカラーであり、この様子を図 7-2 に示します。反復計算シーケンスは通常、配列の各要素、すなわちデータ構造に適用されます。構造体のスカラーメンバーに対する計算シーケンスは、各反復内で同種演算にならない可能性が高くなります。AoS は一般に、水平計算モデルに関連付けられます。

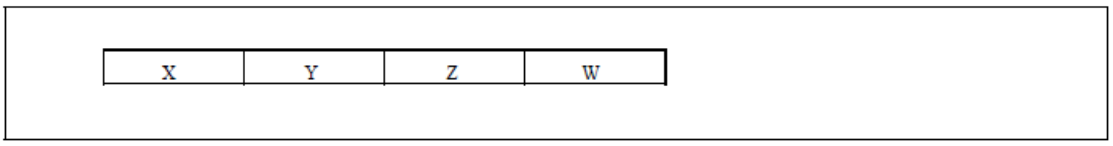


図 7-2 水平計算モデル

- 配列構造体 (SoA): データ構造体の各メンバーは配列であり、配列の各要素はスカラーです。この様子を表 7-1 に示します。反復計算シーケンスはスカラー要素に適用され、同じ構造体メンバー内の連続した反復間で同種演算を容易に実行できます。したがって、SoA は通常、垂直計算モデルに関連付けられます。

表 7-1 頂点データを SoA 形式で表現した例

Vx array	X1	X2	X3	X4	.....	Xn
Vy array	Y1	Y2	Y3	Y4	.....	Yn
Vz array	Z1	Z2	Z3	Y4	.....	Zn
Vw array	W1	W2	W3	W4	.....	Wn

SoA 配列への垂直計算に SIMD 命令を使用すると、AoS および水平計算よりも高い効率とパフォーマンスを達成できます。これは各ベクトルに対するドット積演算で見られます。図 7-3 に、SoA 配列のドット積演算を示します。

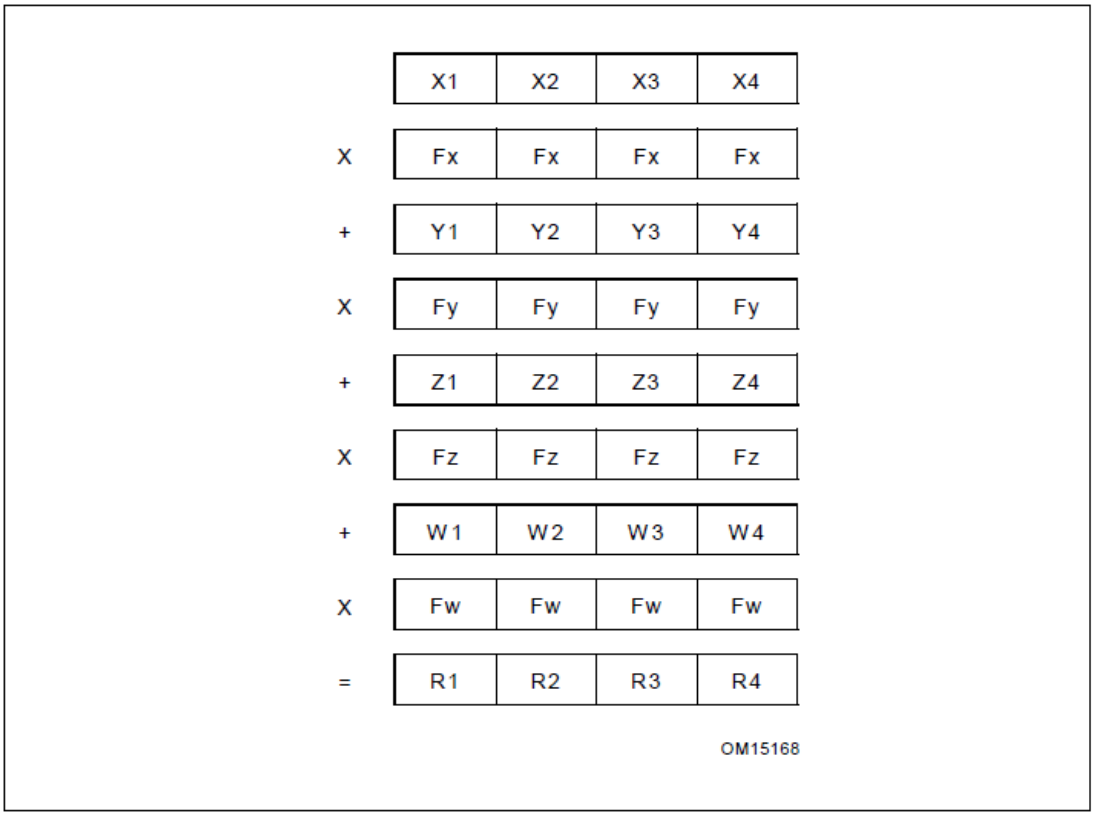


図 7-3 ドット積演算

例 7-1 を見ると、インテル® SSE のみを使用してデータを AoS として編成した場合、7 つの命令で 1 つの計算結果が得られることが分かります。つまり、計算結果を 4 つ得るには 28 個の命令が必要になります。

例 7-1 水平 (xyz, AoS) 計算の擬似コード

```
mulps ; x*x', y*y', z*z'
movaps ; 次のステップが上書きするので reg->reg 移動
shufps ; a, b, c, d から b, a, d, c を取得
addps ; a+b, a+b, c+d, c+d の取得
movaps ; reg->reg 移動
shufps ; 前の addps から c+d, c+d, a+b, a+b を取得
addps ; a+b+c+d, a+b+c+d, a+b+c+d, a+b+c+d の取得
```

次に、データが SoA として編成される場合を考えてみます。例 7-2 を見ると、5 つの命令で 4 つの計算結果が得られることが分かります。

例 7-2 垂直 (xxxx, yyyy, zzzz, SoA) 計算の擬似コード

```
mulps ; 4 頂点の 4 つの x 要素すべてに x*x' を計算
mulps ; 4 頂点の 4 つの y 要素すべてに y*y' を計算
mulps ; 4 頂点の 4 つの z 要素すべてに z*z' を計算
addps ; x*x' + y*y'
addps ; x*x'+y*y'+z*z'
```

4 つのコンポーネントを含むレジスターを最も効率良く使用するため、データを SoA 形式に再編成すると、スループットが増加します。その結果、使用する命令のパフォーマンスがより高まります。

この単純な例から分かるように、垂直計算では SIMD レジスター幅をすべて利用して 4 つの計算結果が得られます (ただし、状況によっては計算結果が異なることがあります)。データ構造体が垂直計算に向かない形式でも、実行処理と並行してその構造体を再編成すると SIMD レジスターを活用しやすくなります。このデータ再編成操作を「スウィズリング」と呼び、その逆の操作を「デスウィズリング」と呼びます。

7.5.1.2 データ・スウィズリング

SoA 形式から AoS 形式へのデータのスウィズリングは、3D ジオメトリー、ビデオ、画像処理など多くのアプリケーションに適用できます。浮動小数点データと整数データの操作には、2 つの異なるスウィズリング手法を適用できます。例 7-3 に、SHUFPS、MOVLHPS、MOVHLPS 命令を使用するスウィズリング関数を示します。

例 7-3 SHUFPS、MOVLHPS、MOVHLPS 命令を使用したデータのスウィズリング

```
typedef struct _VERTEX_AOS {
   float x, y, z, color;
} Vertex_aos; // AoS 構造体の宣言
typedef struct _VERTEX_SOA {
   float x[4], float y[4], float z[4];
   float color[4];
} Vertex_soa; // SoA 構造体の宣言
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in の内容: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// スウィズル XYZW --> XXXX
asm {
   mov ebx, in // 構造体のアドレスを取得
   mov edx, out
   movaps xmm1, [ebx ] // x4 x3 x2 x1
   movaps xmm2, [ebx + 16] // y4 y3 y2 y1
   movaps xmm3, [ebx + 32] // z4 z3 z2 z1
   movaps xmm4, [ebx + 48] // w4 w3 w2 w1
   movaps xmm7, xmm4 // xmm7= w4 z4 y4 x4
   movhlps xmm7, xmm3 // xmm7= w4 z4 w3 z3
   movaps xmm6, xmm2 // xmm6= w2 z2 y2 x2
   movlhps xmm3, xmm4 // xmm3= y4 x4 y3 x3
   movhlps xmm2, xmm1 // xmm2= w2 z2 w1 z1
   movlhps xmm1, xmm6 // xmm1= y2 x2 y1 x1
   movaps xmm6, xmm2// xmm6= w2 z2 w1 z1
   movaps xmm5, xmm1// xmm5= y2 x2 y1 x1
   shufps xmm2, xmm7, 0xDD // xmm2= w4 w3 w2 w1 => v4
   shufps xmm1, xmm3, 0x88 // xmm1= x4 x3 x2 x1 => v1
   shufps xmm5, xmm3, 0xDD // xmm5= y4 y3 y2 y1 => v2
   shufps xmm6, xmm7, 0x88 // xmm6= z4 z3 z2 z1 => v3
   movaps [edx], xmm1 // store X
   movaps [edx+16], xmm5 // store Y
   movaps [edx+32], xmm6 // store Z
   movaps [edx+48], xmm2 // store W
   }
}
```

例 7-4 に、整数ドメインに SIMD 命令を使用する同様のデータ・スウィズリング・アルゴリズムを示します。

例 7-4 UNPCKxxx 命令を使用したデータのスウィズリング

```
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in の内容: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// スウィズル XYZW --> XXXX
asm {
   mov ebx, in // 構造体のアドレスを取得
   mov edx, out
   movdqa xmm1, [ebx + 0*16] //w0 z0 y0 x0
   movdqa xmm2, [ebx + 1*16] //w1 z1 y1 x1
   movdqa xmm3, [ebx + 2*16] //w2 z2 y2 x2
   movdqa xmm4, [ebx + 3*16] //w3 z3 y3 x3
   movdqa xmm5, xmm1
   punpckldq xmm1, xmm2 // y1 y0 x1 x0
   punpckhdq xmm5, xmm2 // w1 w0 z1 z0
   movdqa xmm2, xmm3
   punpckldq xmm3, xmm4 // y3 y2 x3 x2
   punpckldq xmm2, xmm4 // w3 w2 z3 z2
   movdqa xmm4, xmm1
   punpcklqdq xmm1, xmm3 // x3 x2 x1 x0
   punpckhqdq xmm4, xmm3 // y3 y2 y1 y0
   movdqa xmm3, xmm5
   punpcklqdq xmm5, xmm2 // z3 z2 z1 z0
   punpckhqdq xmm3, xmm2 // w3 w2 w1 w0
   movdqa [edx+0*16], xmm1 //x3 x2 x1 x0
   movdqa [edx+1*16], xmm4 //y3 y2 y1 y0
   movdqa [edx+2*16], xmm5 //z3 z2 z1 z0
   movdqa [edx+3*16], xmm3 //w3 w2 w1 w0
}
```

新しいマイクロアーキテクチャーでは、MOVLPS/MOVHPS 命令を使用して各ベクトルの半分をロードする手法よりも、例 7-3 の手法 (16 バイトをロード、SHUFPS を使用して、XMM レジスターの半分をコピーする) が推奨されます。これは、MOVLPS/MOVHPS 命令を使用して 8 バイトをロードすると、コードの依存関係が生じ、実行エンジンのスループットが低下する可能性があるためです。

例 7-3 と例 7-4 に関するパフォーマンス上の考慮事項は、各マイクロアーキテクチャーの特性に依存しています。例えばインテル® Core™ マイクロアーキテクチャーでは、SHUFPS を実行すると、PUNPCKxxx 命令よりも低速になる傾向があります。拡張版インテル® Core™ マイクロアーキテクチャーでは、SHUFPS 命令と PUNPCKxxx 命令はすべて、128 ビット・シャッフル実行ユニットによって 1 サイクルのスループットで実行されます。次に重要なことは、PUNPCKxxx を実行できるポートは 1 つしかありませんが、MOVLHPS/MOVHLPS 命令は複数のポートで実行できことです。インテル® Core™ マイクロアーキテクチャーでは、SIMD 命令を実行できるポートが 3 つあるため、いずれの手法によるパフォーマンスも従来のマイクロアーキテクチャーに比べて向上しています。拡張版インテル® Core™ マイクロアーキテクチャーでは、128 ビット・シャッフル・ユニットが採用されたことにより、いずれの手法もさらにパフォーマンスが向上しています。

7.5.1.3 データ・デスウィズリング

デスウィズリングでは、XXXX、YYYY、ZZZZ が XYZ として再配置されてからメモリーに格納されるように、SoA 形式を AoS 形式に戻す必要があります。例 7-5 に、浮動小数点データ向けのデスウィズリング関数を示します。

例 7-5 単精度 SIMD データのデスウィズリング

```
void deswizzle_asm(Vertex_soa *in, Vertex_aos *out)
{
__asm {
   mov ecx, in // 構造体のアドレスをロード
   mov edx, out
   movaps xmm0, [ecx ] //x3 x2 x1 x0
   movaps xmm1, [ecx + 16] //y3 y2 y1 y0
   movaps xmm2, [ecx + 32] //z3 z2 z1 z0
   movaps xmm3, [ecx + 48] //w3 w2 w1 w0
   movaps xmm5, xmm0
   movaps xmm7, xmm2
   unpcklps xmm0, xmm1 // y1 x1 y0 x0
   unpcklps xmm2, xmm3 // w1 z1 w0 z0
   movdqa xmm4, xmm0
   movlhps xmm0, xmm2 // w0 z0 y0 x0
   movhlps xmm4, xmm2 // w1 z1 y1 x1
   unpckhps xmm5, xmm1 // y3 x3 y2 x2
   unpckhps xmm7, xmm3 // w3 z3 w2 z2
   movdqa xmm6, xmm5
   movlhps xmm5, xmm7 // w2 z2 y2 x2
   movhlps xmm6, xmm7 // w3 z3 y3 x3
   movaps [edx+0*16], xmm0 //w0 z0 y0 x0
   movaps [edx+1*16], xmm4 //w1 z1 y1 x1
   movaps [edx+2*16], xmm5 //w2 z2 y2 x2
   movaps [edx+3*16], xmm6 //w3 z3 y3 x3
   }
}
```

例 7-6 に、SIMD 整数命令を使用する同様のデスウィズリング関数を示します。どちらの手法でも、16 バイトをロードし、レジスター内で水平データ移動を行っています。この方法は、MOVLPS と MOVHPS を使用して XMM レジスターの半分の 8 バイトをストアする手法よりも効率的です。

例 7-6 SIMD 整数命令を使用したデータのデスウィズリング

```
void deswizzle_rgb(Vertex_soa *in, Vertex_aos *out)
{
//--- rgb をデスウィズリング ---
// 前提: xmm1=rrrr, xmm2=gggg, xmm3=bbbb, xmm4=aaaa
__asm {
   mov ecx, in // 構造体のアドレスをロード
   mov edx, out
   movdqa xmm0, [ecx] // load r4 r3 r2 r1 => xmm1
   movdqa xmm1, [ecx+16] // load g4 g3 g2 g1 => xmm2
   movdqa xmm2, [ecx+32] // load b4 b3 b2 b1 => xmm3
   movdqa xmm3, [ecx+48] // load a4 a3 a2 a1 => xmm4
// デスウィズリングを開始
   movdqa xmm5, xmm0
   movdqa xmm7, xmm2
   punpckldq xmm0, xmm1 // g2 r2 g1 r1
   punpckldq xmm2, xmm3 // a2 b2 a1 b1
   movdqa xmm4, xmm0
   punpcklqdq xmm0, xmm2 // a1 b1 g1 r1 => v1
   punpckhqdq xmm4, xmm2 // a2 b2 g2 r2 => v2
   punpckhdq xmm5, xmm1 // g4 r4 g3 r3
   punpckhdq xmm7, xmm3 // a4 b4 a3 b3
   movdqa xmm6, xmm5
   punpcklqdq xmm5, xmm7 // a3 b3 g3 r3 => v3
   punpckhqdq xmm6, xmm7 // a4 b4 g4 r4 => v4
```

```
    movdqa [edx], xmm0 // v1
    movdqa [edx+16], xmm4 // v2
    movdqa [edx+32], xmm5 // v3
    movdqa [edx+48], xmm6 // v4
// デスウィズリングを終了
    }
}
```

7.5.1.4 インテル® SSE による水平加算

一般に垂直計算の方が水平計算よりも SIMD のパフォーマンスをうまく引き出せますが、場合によってはコード中で水平演算を使用しなければならないことがあります。

MOVLHPS/MOVHLPS 命令とシャッフル命令を使用すると、データの水平加算が可能となります。例えば、128 ビットのレジスターが 4 つあり、それぞれを水平加算する一方、最終的な計算結果を 1 つのレジスターに格納する場合、MOVLHPS/MOVHLPS の命令を使用して各レジスターの上位部分と下位部分を合わせる必要があります。このような操作によって、垂直加算が可能になります。部分的な水平総和から全体の総和も簡単に求められます。

図 7-4 に、MOVHLPS/MOVLHPS 命令を使用した水平加算を示します。この操作を行うコードを例 7-7 と例 7-8 に示します。

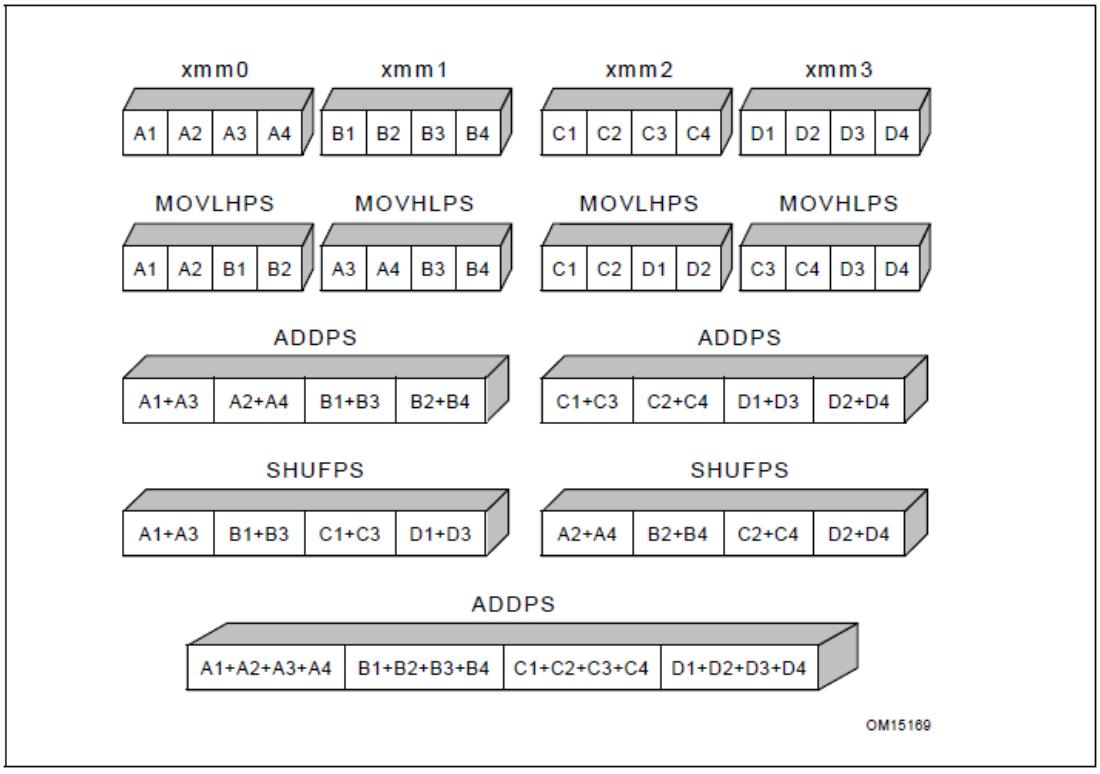


図 7-4 MOVHLPS/MOVLHPS を使用した水平加算

例 7-7 MOVHLPS/MOVLHPS を使用した水平加算

```
void horiz_add(Vertex_soa *in, float *out) {
__asm {
   mov ecx, in // 構造体のアドレスをロード
   mov edx, out
   movaps xmm0, [ecx] // load A1 A2 A3 A4 => xmm0
   movaps xmm1, [ecx+16] // load B1 B2 B3 B4 => xmm1
   movaps xmm2, [ecx+32] // load C1 C2 C3 C4 => xmm2
   movaps xmm3, [ecx+48] // load D1 D2 D3 D4 => xmm3
// 水平加算を開始
   movaps xmm5, xmm0 // xmm5= A1, A2, A3, A4
   movlhps xmm5, xmm1 // xmm5= A1, A2, B1, B2
   movhlps xmm1, xmm0 // xmm1= A3, A4, B3, B4
   addps xmm5, xmm1 // xmm5= A1+A3, A2+A4, B1+B3, B2+B4
   movaps xmm4, xmm2
   movlhps xmm2, xmm3 // xmm2= C1, C2, D1, D2
   movhlps xmm3, xmm4 // xmm3= C3, C4, D3, D4
   addps xmm3, xmm2 // xmm3= C1+C3, C2+C4, D1+D3, D2+D4
   movaps xmm6, xmm3 // xmm6= C1+C3, C2+C4, D1+D3, D2+D4
   shufps xmm3, xmm5, 0xDD
                        //xmm6=A1+A3, B1+B3, C1+C3, D1+D3
   shufps xmm5, xmm6, 0x88
                        // xmm5= A2+A4, B2+B4, C2+C4, D2+D4
   addps xmm6, xmm5 // xmm6= D, C, B, A
// 水平加算を終了
   movaps [edx], xmm6
   }
}
```

例 7-8 MOVHLPS/MOVLHPS と組込み関数を併用した水平加算

```
void horiz_add_intrin(Vertex_soa *in, float *out)
{
__m128 v, v2, v3, v4;
__m128 tmm0,tmm1,tmm2,tmm3,tmm4,tmm5,tmm6;
// 一時変数
   tmm0 = _mm_load_ps(in->x); // tmm0 = A1 A2 A3 A4
   tmm1 = _mm_load_ps(in->y); // tmm1 = B1 B2 B3 B4
   tmm2 = _mm_load_ps(in->z); // tmm2 = C1 C2 C3 C4
   tmm3 = _mm_load_ps(in->w); // tmm3 = D1 D2 D3 D4
   tmm5 = tmm0; // tmm0 = A1 A2 A3 A4
   tmm5 = _mm_movelh_ps(tmm5, tmm1); // tmm5 = A1 A2 B1 B2
   tmm1 = _mm_movehl_ps(tmm1, tmm0); // tmm1 = A3 A4 B3 B4
   tmm5 = _mm_add_ps(tmm5, tmm1); // tmm5 = A1+A3 A2+A4 B1+B3 B2+B4
   tmm4 = tmm2;
   tmm2 = _mm_movelh_ps(tmm2, tmm3); // tmm2 = C1 C2 D1 D2
   tmm3 = _mm_movehl_ps(tmm3, tmm4); // tmm3 = C3 C4 D3 D4
   tmm3 = _mm_add_ps(tmm3, tmm2); // tmm3 = C1+C3 C2+C4 D1+D3 D2+D4
   tmm6 = tmm3; // tmm6 = C1+C3 C2+C4 D1+D3 D2+D4
   tmm6 = _mm_shuffle_ps(tmm3, tmm5, 0xDD);
                                  // tmm6 = A1+A3 B1+B3 C1+C3 D1+D3
   tmm5 = _mm_shuffle_ps(tmm5, tmm6, 0x88);
                                  // tmm5 = A2+A4 B2+B4 C2+C4 D2+D4
   tmm6 = _mm_add_ps(tmm6, tmm5);
                                  // tmm6 = A1+A2+A3+A4 B1+B2+B3+B4
                                  // C1+C2+C3+C4 D1+D2+D3+D4
   _mm_store_ps(out, tmm6);
}
```

7.5.2 CVTTPS2PI/CVTTSS2SI 命令の使用

CVTTPS2PI 命令および CVTTSS2SI 命令は、暗黙的に「切り捨て/絶対値の小さくなる方向への丸め」という制御モードでエンコードされます。これは、MXCSR レジスターで指定される丸め制御モードよりも優先されます。これにより、丸め制御モードを「最も近い値へ丸め」から「切り捨て/絶対値の小さくなる方向への丸め」へ変更してから、再び「最も近い値へ丸め」へ戻さなくても計算を再開できます。

MXCSR レジスターへの書き込みにはペナルティーが生じるため、頻繁に変更するべきではありません。通常、CVTTPS2PI/CVTTSS2SI 命令を使用する場合、MXCSR レジスターの丸め制御モードは常に「近似値へ丸め」に設定します。

7.5.3 ゼロ・フラッシュ・モードと DAZ モード

ゼロ・フラッシュ (FTZ) モードとデノーマルをゼロとして扱う (DAZ) モードは、IEEE 規格 754 とは互換性がありません。これらのモードは、アンダーフローが頻発するアプリケーションや、非正規化された結果を生成する必要のないアプリケーションのパフォーマンスを改善するために用意されています。

関連情報: 3.9.2 節「浮動小数点モードと浮動小数点例外」を参照してください。

7.6 SIMD の最適化とマイクロアーキテクチャー

インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、およびインテル® Core™ Duo プロセッサーのマイクロアーキテクチャーは、Intel NetBurst® マイクロアーキテクチャーとは異なります。インテル® Core™ マイクロアーキテクチャーは、従来のマイクロアーキテクチャーよりも大幅に効率的な SIMD 浮動小数点機能を提供します。また、インテル® Core™ マイクロアーキテクチャーでは、インテル® SSE3 命令のレイテンシーとスループットが従来のマイクロアーキテクチャーよりもかなり向上しています。

7.6.1 インテル® SSE3 を使用した SIMD 浮動小数点プログラミング

インテル® SSE3 は、SIMD 浮動小数点プログラミング向けの 9 つの命令を追加しインテル® SSE とインテル® SSE2 を強化しています。インテル® SSE/インテル® SSE2 の多くの命令が並列データ要素の同種算術演算を提供し、垂直計算モデルを優先しているのに対し、インテル® SSE3 は非対称算術演算と水平データ要素の算術演算を行う命令を提供します。

ADDSUBPS と ADDSUBPD の 2 つの命令は、非対称算術処理機能を備えています (図 7-5 を参照)。HADDPS、HADDPD、HSUBPS、HSUBPD 命令は、水平算術処理機能を備えています (図 7-6 を参照)。また、MOVSLDUP、MOVSHDUP、MOVDDUP 命令は、メモリー (または XMM レジスター) からデータをロードして、一度に複数のデータ要素を複製できます。

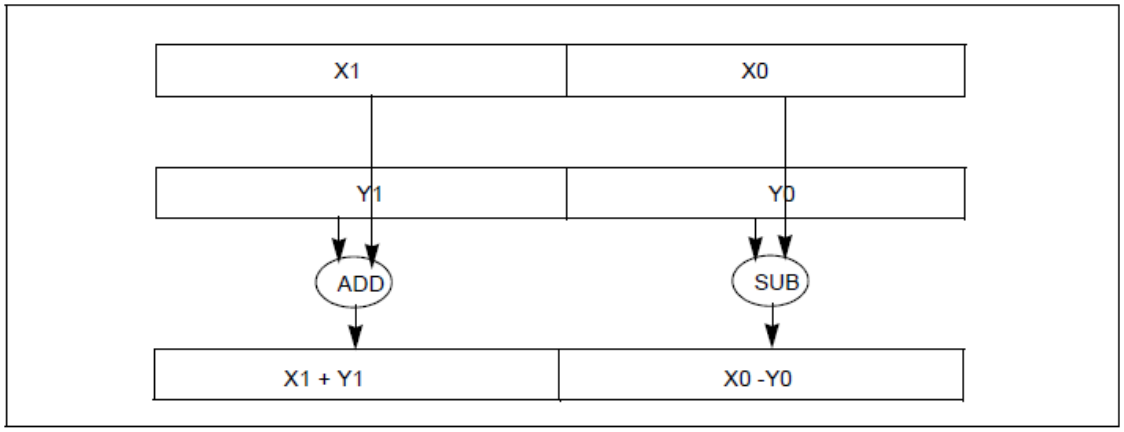


図 7-5 インテル® SSE3 命令の非対称算術演算

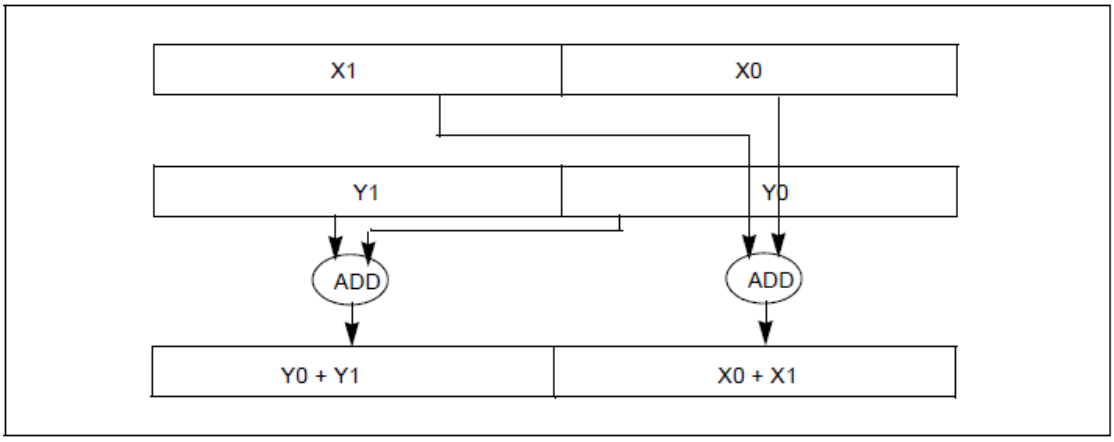


図 7-6 インテル® SSE3 命令 HADDPD の水平算術演算

7.6.1.1 インテル® SSE3 と複素数演算

AoS 形式のデータ構造体を処理する際のインテル® SSE3 の柔軟性は、複素数の乗算と除算の例を用いて示すことができます。例えば、複素数は、実数部分と虚数部分で構成された構造体に保存できます。これは自然なことですが、構造体配列を利用することになります。例 7-9 では、インテル® SSE3 命令を使用して単精度複素数の乗算を行い、例 7-10 では、インテル® SSE3 命令を使用して複素数の除算を行っています。

例 7-9 2 組の単精度複素数の乗算

```
// (ak + i bk ) * (ck + i dk ) の乗算
// a + i b はデータ構造体として格納
movsldup xmm0, Src1; 実数部分のロード a1, a1, a0, a0
movaps xmm1, src2; 2 番目のペアの複素数をロード i.e. d1, c1, d0, c0
mulps xmm0, xmm1; a1d1, a1c1, a0d0, a0c0 は一時的な結果
shufps xmm1, xmm1, b1; 実数部と虚数部の並び替え c1, d1, c0, d0
movshdup xmm2, Src1; 虚数部を b1, b1, b0, b0 にロード
mulps xmm2, xmm1; b1c1, b1d1, b0c0, b0d0 は一時的な結果
addsubps xmm0, xmm2; b1c1+a1d1, a1c1 -b1d1, b0c0+a0d0, a0c0-b0d0
```

例 7-10 2 組の単精度複素数の除算

```
// (ak + i bk ) * (ck + i dk ) の除算
movshdup xmm0, Src1; 虚数部を b1, b1, b0, b0 にロード
movaps xmm1, src2; 2 番目のペアの複素数をロード i.e. d1, c1, d0, c0
mulps xmm0, xmm1; b1d1, b1c1, b0d0, b0c0 は一時的な結果
shufps xmm1, xmm1, b1; 実数部と虚数部の並び替え c1, d1, c0, d0
movsldup xmm2, Src1; l実数部を a1, a1, a0, a0 に格納
mulps xmm2, xmm1; a1c1, a1d1, a0c0, a0d0 は一時的な結果
addsubps xmm0, xmm2; a1c1+b1d1, b1c1-a1d1, a0c0+b0d0, b0c0-a0d0
mulps xmm1, xmm1 ; c1c1, d1d1, c0c0, d0d0
movps xmm2, xmm1; c1c1, d1d1, c0c0, d0d0
shufps xmm2, xmm2, b1; d1d1, c1c1, d0d0, c0c0
addps xmm2, xmm1; c1c1+d1d1, c1c1+d1d1, c0c0+d0d0, c0c0+d0d0
divps xmm0, xmm2
shufps xmm0, xmm0, b1 ; (b1c1-a1d1)/(c1c1+d1d1), (a1c1+b1d1)/(c1c1+d1d1),
                       ; (b0c0-a0d0)/( c0c0+d0d0),
                       (a0c0+b0d0)/( c0c0+d0d0)
```

両者とも、複素数は構造体配列に保存されます。MOVSLDUP、MOVSHDUP、非対称 ADDSUBPS 命令を利用すると、データ要素間でスウィズリングを行うことなく、同時に 2 組の単精度複素数に対して複素数演算を実行できます。

マイクロアーキテクチャーの違いにより、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、インテル® SSE3 命令を使用して倍精度複素数の乗算を実装すべきです。また、インテル® Core™ Duo プロセッサーとインテル® Core™ Solo プロセッサーでは、インテル® SSE2 のスカラー命令を使用して倍精度複素数の乗算を実装すべきです。これは、SIMD 実行ユニット間のデータパスはインテル® Core™ マイクロアーキテクチャーでは 128 ビットであるのに対し、従来のマイクロアーキテクチャーでは 64 ビットしかないため 128 ビットの演算ではスループットが低下することが理由です。拡張版インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、通常、従来のマイクロアーキテクチャーよりも効率良くインテル® SSE3 命令が実行されます。また、128 ビット・シャッフル・ユニットも搭載しており、インテル® Core™ マイクロアーキテクチャーよりもさらに大きなメリットを複素数演算にもたらします。

例 7-11 に示す 2 つのコードでは、インテル® SSE2 のベクトル命令とインテル® SSE3 のベクトル命令を使用して、2 組の複素数に対し倍精度複素数乗算を行っています。例 7-12 に、インテル® SSE2 のスカラー命令を使用した同等のコードを示します。

例 7-11 2 組の複素数に対する倍精度複素数乗算

インテル® SSE2 ベクトル実装	**インテル® SSE3 ベクトル実装**

movapd xmm0, [eax] ;y x
movapd xmm1, [eax+16] ;w z
unpcklpd xmm1, xmm1 ;z z
movapd xmm2, [eax+16] ;w z
unpckhpd xmm2, xmm2 ;w w
mulpd xmm1, xmm0 ;z*y z*x
mulpd xmm2, xmm0 ;w*y w*x
xorpd xmm2, xmm7 ;-w*y +w*x
shufpd xmm2, xmm2,1 ;w*x -w*y
addpd xmm2, xmm1 ;z*y+w*x z*x-w*y
movapd [ecx], xmm2
	movapd xmm0, [eax] ;y x
	movapd xmm1, [eax+16] ;z z
	movapd xmm2, xmm1
	unpcklpd xmm1, xmm1
	unpckhpd xmm2, xmm2
	mulpd xmm1, xmm0 ;z*y z*x
	mulpd xmm2, xmm0 ;w*y w*x
	shufpd xmm2, xmm2, 1 ;w*x w*y
	addsubpd xmm1, xmm2 ;w*x+z*y z*x-w*y
	movapd [ecx], xmm1

例 7-12 インテル® SSE2 のスカラー命令を使用した倍精度複素数乗算

```
movsd xmm0, [eax] ;x
movsd xmm5, [eax+8] ;y
movsd xmm1, [eax+16] ;z
movsd xmm2, [eax+24] ;w
movsd xmm3, xmm1 ;z
movsd xmm4, xmm2 ;w
mulsd xmm1, xmm0 ;z*x
mulsd xmm2, xmm0 ;w*x
mulsd xmm3, xmm5 ;z*y
mulsd xmm4, xmm5 ;w*y
subsd xmm1, xmm4 ;z*x - w*y
addsd xmm3, xmm2 ;z*y + w*x
movsd [ecx], xmm1
movsd [ecx+8], xmm3
```

7.6.1.2 インテル® Core™ Duo プロセッサーにおけるパックド浮動小数点のパフォーマンス

インテル® Core™ Solo プロセッサーでは、ほとんどのパックド SIMD 浮動小数点コードがインテル® Pentium® M プロセッサーと比べて高速化されています。これは、パックド SIMD 命令のデコード性能の向上が要因です。

インテル® Core™ Solo プロセッサーでパックド浮動小数点のパフォーマンスがインテル® Pentium® M プロセッサーよりも向上するかどうかは、複数の要因に依存しています。一般に、デコーダー依存のコードや、整数命令とパックド浮動小数点命令が混在したコードでは、大幅な向上を期待できます。実行レイテンシーによって制限され、「1 命令あたりのサイクル数」の比率が 1 よりも大きいコードでは、デコーダーの向上によるメリットは得られません。

インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーで複素数演算を行う際に単精度のインテル® SSE3 命令を使用すると、その他の方法よりもパフォーマンスが向上することが期待できます。一方、インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーで倍精度複素数演算が必要なタスクでは、インテル® SSE2 のスカラー命令を使用した方がパフォーマンスは向上します。これは、インテル® SSE2 のスカラー命令では、2 つのポートを介してディスパッチし、2 つの独立した浮動小数点ユニットによって実行されるためです。

一部のタスクでは、インテル® SSE3 のパックド水平命令 (HADDPS と HSUBPS) を使用すると、コードシーケンスを簡素化できます。ただし、インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーのでは、これらの命令は、6 以上のマイクロオペレーション (uop) で構成されます。そのため、水平命令でのレイテンシーとデコードのペナルティーによってアルゴリズム上のメリットが相殺されないように注意する必要があります。

7.6.2 ドット積と水平 SIMD 命令

多くの代数式は、AoS 形式のデータ編成の方が適していることが多く、その典型的な例がドット積演算です。ドット積演算は、インテル® SSE/インテル® SSE2 命令セットを使用して実装できます。インテル® SSE3 では、水平計算モデルに依存したアプリケーション向けに水平加算/減算命令がいくつか追加されています。インテル® SSE4.1 では、2、3、または 4 つの成分からなるベクトルのドット積演算を直接評価できるように命令が拡張されています。

例 7-13 インテル® SSE/インテル® SSE2 を使用したベクトル長 4 のドット積演算

インテル® SSE/インテル® SSE2 を使用して 1 つのドット積を計算

```
movaps xmm0, [eax] // a4, a3, a2, a1
mulps xmm0, [eax+16] // a4*b4, a3*b3, a2*b2, a1*b1
movhlps xmm1, xmm0 // X, X, a4*b4, a3*b3, 上位半分は不要
addps xmm0, xmm1 // X, X, a2*b2+a4*b4, a1*b1+a3*b3,
pshufd xmm1, xmm0, 1 // X, X, X, a2*b2+a4*b4
addss xmm0, xmm1 // a1*b1+a3*b3+a2*b2+a4*b4
movss [ecx], xmm0
```

例 7-14 インテル® SSE3 を使用したベクトル長 4 のドット積演算

インテル® SSE3 を使用して 1 つのドット積を計算

```
movaps xmm0, [eax]
mulps xmm0, [eax+16] // a4*b4, a3*b3, a2*b2, a1*b1
haddps xmm0, xmm0 // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
movaps xmm1, xmm0 // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
psrlq xmm0, 32 // 0, a4*b4+a3*b3, 0, a4*b4+a3*b3
addss xmm0, xmm1 // -, -, -, a1*b1+a3*b3+a2*b2+a4*b4
movss [eax], xmm0
```

例 7-15 インテル® SSE4.1 を使用したベクトル長 4 のドット積演算

インテル® SSE4.1 を使用して 1 つのドット積を計算

```
movaps xmm0, [eax]
dpps xmm0, [eax+16], 0xf1 // 0, 0, 0, a1*b1+a3*b3+a2*b2+a4*b4
movss [eax], xmm0
```

例 7-13、例 7-14、例 7-15 では、1 組のベクトルに対して 1 つのドット積を計算する基本的なコードシーケンスを比較しています。

アプリケーションのメモリー・アクセス・パターンに合わせて最適なシーケンスを選択することは、各種の手法にとってメリットがあります。例えば、各ドット積が後続の計算シーケンスですぐに使用されるのであれば、各種手法の相対速度を比較するのに適しています。ベクトル配列のドット積を計算し、その結果を後続の計算のためキャッシュ内に保持できる場合、適切な選択は命令シーケンスの相対スループットに依存します。

インテル® Core™ マイクロアーキテクチャーでは、例 7-14 の方が例 7-13 よりもスループットは高くなります。ただし、HADDPS 命令のレイテンシーが比較的長いため、例 7-14 の速度は例 7-13 よりもわずかに遅くなります。

拡張版インテル® Core™ マイクロアーキテクチャーでは、例 7-15 の方が速度とスループットの両面において例 7-13 と例 7-14 よりも優れています。DPPS 命令のレイテンシーも比較的長めですが、例 7-15 では、同じ処理量に対する命令数の減少によって補間されています。

アンロールを適用すると、3 つのドット積演算コードのそれぞれでスループットをさらに向上させることができます。例 7-16 に、インテル® SSE2 とインテル® SSE3 による基本シーケンスを 2 回アンロールした例を示します。インテル® SSE4.1 でもアンロール可能であり、INSERTPS 命令を使用して 4 つのドット積をパックできます。

例 7-16 4 つのドット積に対するアンロールの実装

インテル® SSE2 実装	インテル® SSE3 実装

movaps xmm0, [eax]
mulps xmm0, [eax+16]
;w0*w1 z0*z1 y0*y1 x0*x1
movaps xmm2, [eax+32]
mulps xmm2, [eax+16+32]
;w2*w3 z2*z3 y2*y3 x2*x3
movaps xmm3, [eax+64]
mulps xmm3, [eax+16+64]
;w4*w5 z4*z5 y4*y5 x4*x5
movaps xmm4, [eax+96]
mulps xmm4, [eax+16+96]
;w6*w7 z6*z7 y6*y7 x6*x7
movaps xmm1, xmm0
unpcklps xmm0, xmm2
; y2*y3 y0*y1 x2*x3 x0*x1
unpckhps xmm1, xmm2
; w2*w3 w0*w1 z2*z3 z0*z1
movaps xmm5, xmm3
unpcklps xmm3, xmm4
; y6*y7 y4*y5 x6*x7 x4*x5
unpckhps xmm5, xmm4
; w6*w7 w4*w5 z6*z7 z4*z5
addps xmm0, xmm1
addps xmm5, xmm3
movaps xmm1, xmm5
movhlps xmm1, xmm0
movlhps xmm0, xmm5
addps xmm0, xmm1
movaps [ecx], xmm0
	movaps xmm0, [eax]
	mulps xmm0, [eax+16]
	movaps xmm1, [eax+32]
	mulps xmm1, [eax+16+32]
	movaps xmm2, [eax+64]
	mulps xmm2, [eax+16+64]
	movaps xmm3, [eax+96]
	mulps xmm3, [eax+16+96]
	haddps xmm0, xmm1
	haddps xmm2, xmm3
	haddps xmm0, xmm2
	movaps [ecx], xmm0

7.6.3 ベクトルの正規化

ベクトルの正規化は、多くの浮動小数点アプリケーションにおいて一般的な操作です。例 7-17 に、(x, y, z) ベクトルの配列を正規化する C コードの例を示します。

例 7-17 ベクトルの配列の正規化

```
for (i=0;i<CNT;i++)
{ float size = nodes[i].vec.dot();
   if (size != 0.0)
   { size = 1.0f/sqrtf(size); }
   else
   { size = 0.0; }
   nodes[i].vec.x *= size;
   nodes[i].vec.y *= size;
   nodes[i].vec.z *= size;
}
```

例 7-18 に、ベクトルの (x, y, z) 成分を正規化するアセンブリー・シーケンスを示します。

例 7-18 インテル® SSE2 を使用した、ベクトル配列 (x, y, z) の成分に対する正規化

```
Vec3 *p = &nodes[i].vec;
__asm
{ mov eax, p
   xorps xmm2, xmm2
   movups xmm1, [eax] // 入力ベクトルと次のベクトルの x を加えた (x,y,z) をロード
   movaps xmm7, xmm1 // (正規化されていないデータを戻すため) メモリーのデータのコピーを格納
   movaps xmm5, _mask // 正規化するために xmm レジスターから (x,y,z) 値を選択するためにマスクす
   る
   andps xmm1, xmm5 // 1 番目の 3 要素をマスクする
   movaps xmm6, xmm1 // 後で正規化されたベクトルを計算するために (x,y,z) のコピーを格納
   mulps xmm1,xmm1 // 0, z*z, y*y, x*x
   pshufd xmm3, xmm1, 0x1b // x*x, y*y, z*z, 0
   addps xmm1, xmm3 // x*x, z*z+y*y, z*z+y*y, x*x
   pshufd xmm3, xmm1, 0x41 // z*z+y*y, x*x, x*x, z*z+y*y
   addps xmm1, xmm3 // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
   comisd xmm1, xmm2 // サイズを 0 と比較
   jz zero
   movaps xmm3, xmm4 // 単位ベクトル (1.0, 1.0, 1.0, 1.0) のプリロード
   sqrtps xmm1, xmm1
   divps xmm3, xmm1
   jmp store
zero:
   movaps xmm3, xmm2
store:
   mulps xmm3, xmm6 // 下位 3 要素のベクトルを正規化
   andnps xmm5, xmm7 // 正規化されない値を保持するために下位 3 要素をマスクオフする
   orps xmm3, xmm5 // 正規化されたベクトルの後に正規化されない要素を並べる
   movaps [eax ], xmm3 // 正規化された x, y, z を代入、変更されない値が続く
```

例 7-19 に、インテル® SSE4.1 を使用したベクトル (x, y, z) の成分を正規化するアセンブリー・シーケンスを示します。

例 7-19 インテル® SSE4.1 を使用した、ベクトル配列 (x, y, z) の成分に対する正規化

```
Vec3 *p = &nodes[i].vec;
__asm
{ mov eax, p
   xorps xmm2, xmm2
   movups xmm1, [eax] // 入力ベクトルと次のベクトルの x を加えた (x,y,z) をロード
   movaps xmm7, xmm1 // メモリーからデータのコピーを格納
   dpps xmm1, xmm1, 0x7f // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z,
   x*x+y*y+z*z
   comisd xmm1, xmm2 // サイズを 0 と比較
   jz zero
   movaps xmm3, xmm4 // 単位ベクトル (1.0, 1.0, 1.0, 1.0) のプリロード
   sqrtps xmm1, xmm1
   divps xmm3, xmm1
   jmp store
zero:
   movaps xmm3, xmm2
store:
   mulps xmm3, xmm6 // 下位 3 要素のベクトルを正規化
   blendps xmm3, xmm7, 0x8 // 正規化されてない要素を正規ベクトルの次にコピー
   movaps [eax ], xmm3
```

例 7-18 と例 7-19 では、レイテンシーの大きな DIVPS 命令と SQRTPS 命令によって根本的に命令シーケンスのスループットが制限されます。例 7-19 では、8 つのインテル® SSE2 命令に代わって DPPS 命令を使用して、XMM レジスターの 4 つの要素に対するドット積の評価やブロードキャストを行っています。これにより、例 7-19 の相対速度は例 7-18 よりも向上します。

7.6.4 水平 SIMD 命令セットの使用とデータレイアウト

インテル® SSE とインテル® SSE2 にはパックド加算/減算、乗算/除算命令が用意されており、SoA データレイアウトなどの垂直計算モデルを活用できるのが理想的です。インテル® SSE3 とインテル® SSE4.1 では、水平加算/減算、ドット積演算などの水平 SIMD 命令が追加ました。これらの新しい SIMD 拡張命令では、垂直 SIMD 計算モデルに対応しないデータレイアウトや演算の問題を解決するツールも提供されています。

ここでは、ベクトル-行列乗算の問題について検討し、各種の水平 SIMD 命令を選択する際に関連する要素について説明します。

例 7-20 に、入出力ベクトルが構造体の配列としてストアされる AoS を使用する、ベクトル-行列データレイアウトを示します。

例 7-20 AoS ベクトル-行列乗算のためのメモリーデータ編成

```
Matrix M4x4 (pMat): M00 M01 M02 M03
                    M10 M11 M12 M13
                    M20 M21 M22 M23
                    M30 M31 M32 M33
4 input vertices V4x1 (pVert): V0x V0y V0z V0w
                               V1x V1y V1z V1w
                               V2x V2y V2z V2w
                               V3x V3y V3z V3w
Output vertices O4x1 (pOutVert): O0x O0y O0z O0w
                               O1x O1y O1z O1w
                               O2x O2y O2z O2w
                               O3x O3y O3z O3w
```

例 7-21 に示す例では、HADDPS と MULPS 命令を使用して、AoS のデータレイアウトのベクトル-行列乗算を行っています。3 つの HADDPS 命令が各出力ベクトル成分の合計を完了した後、出力成分が AoS に配置されます。

例 7-21 HADDPS 命令を使用した AoS ベクトル-行列乗算

```
   mov eax, pMat
   mov ebx, pVert
   mov ecx, pOutVert
   xor edx, edx
   movaps xmm5,[eax+16] // 列 M1 のロード
   movaps xmm6,[eax+2*16] // 列 M2 のロード
   movaps xmm7,[eax+3*16] // 列 M3 のロード
lloop:
   movaps xmm4, [ebx + edx] // 入力ベクトルのロード
   movaps xmm0, xmm4
   mulps xmm0, [eax] // m03*vw, m02*vz, m01*vy, m00*vx,
   movaps xmm1, xmm4
   mulps xmm1, xmm5 // m13*vw, m12*vz, m11*vy, m10*vx,
   movaps xmm2, xmm4
   mulps xmm2, xmm6 // m23*vw, m22*vz, m21*vy, m20*vx
   movaps xmm3, xmm4
   mulps xmm3, xmm7 // m33*vw, m32*vz, m31*vy, m30*vx,
   haddps xmm0, xmm1
   haddps xmm2, xmm3
   haddps xmm0, xmm2
   movaps [ecx + edx], xmm0 // 長さ 4 のベクトルを格納
   add edx, 16
   cmp edx, top
   jb lloop
```

例 7-22 に示す例では、DPPS 命令を使用して、AoS のベクトル-行列乗算を行っています。

例 7-22 DPPS 命令を使用した AoS ベクトル-行列乗算

```
   mov eax, pMat
   mov ebx, pVert
   mov ecx, pOutVert
   xor edx, edx
   movaps xmm5,[eax+16] // 列 M1 のロード
   movaps xmm6,[eax+2*16] // 列 M2 のロード
   movaps xmm7,[eax+3*16] // 列 M3 のロード
lloop:
   movaps xmm4, [ebx + edx] // 入力ベクトルのロード
   movaps xmm0, xmm4
   dpps xmm0, [eax], 0xf1 // 長さ 4 の内積を計算し、最下位の DWORD に格納
   movaps xmm1, xmm4
   dpps xmm1, xmm5, 0xf1
   movaps xmm2, xmm4
   dpps xmm2, xmm6, 0xf1
   movaps xmm3, xmm4
   dpps xmm3, xmm7, 0xf1
   movss [ecx + edx + 0*4], xmm0 // ベクトル長が 4 の 1 つ要素を格納
   movss [ecx + edx + 1*4], xmm1
   movss [ecx + edx + 2*4], xmm2
   movss [ecx + edx + 3*4], xmm3
   add edx, 16
   cmp edx, top
   jb lloop
```

例 7-21 と例 7-22 では、インテル® SSE3 とインテル® SSE4.1 で提供されている異なる水平処理手法を使用して、AoS データレイアウトを操作しています。それぞれの手法の効果は、内部ループでのレイテンシーが長い命令の頻度、データ移動のオーバーヘッド/効率、HADDPS と DPPS 命令のレイテンシーによって異なります。

HADDPS と DPPS の両方をサポートするプロセッサーでは、いずれの手法を選択するかは、アプリケーション固有の考慮事項に依存します。バッチ状況で出力ベクトルがメモリーに直接ライトバックされる場合、例 7-21 の方が例 7-22 よりも推奨されます。それは、DPPS 命令はレイテンシーが長く、各出力ベクトル成分を個別にストアするのは、ベクトル配列のストアにあまり適していないためです。

出力ベクトル成分がほかのベクトル化できない計算によってすぐに使用される、部分的にベクトル化可能な状況では、例 7-21 のように 3 つの HADDPS 命令によって生成されたパックド出力ベクトルを分散させるよりも、個別の成分を生成する DPPS 命令を使用する方が適しています。

7.6.4.1 SoA とベクトル-行列乗算

対象となるネイティブ・データ・レイアウトが SoA に対応している場合、MULPS 命令、レイテンシーの長い水平計算命令を使用しない ADDPS 命令、またはパックド形式へのスカラー成分のパック (例 7-22) を利用してベクトル-行列乗算を表現できます。SoA データレイアウトで高いスループットを達成するには、あらかじめ必要なデータを準備するか、または実行時のスウィズリング/デスウィズリングを理解する必要があります。例えば、ベクトル-行列乗算の SoA データレイアウトを例 7-23 に示します。

パックされた結果を生成するため、各行列要素を 4 回複製することによってデータ移動のオーバーヘッドを最小限に抑えています。

例 7-23 SoA ベクトル-行列乗算のメモリーデータ編成

```
Matrix M16x4 (pMat):
      M00 M00 M00 M00 M01 M01 M01 M01 M02 M02 M02 M02 M03 M03 M03 M03
      M10 M10 M10 M10 M11 M11 M11 M11 M12 M12 M12 M12 M13 M13 M13 M13
      M20 M20 M20 M20 M21 M21 M21 M21 M22 M22 M22 M22 M23 M23 M23 M23
      M30 M30 M30 M30 M31 M31 M31 M31 M32 M32 M32 M32 M33 M33 M33 M33
4 input vertices V4x1 (pVert): V0x V1x V2x V3x
                                V0y V1y V2y V3y
                                V0z V1z V2z V3z
                                V0w V1w V2w V3w
Ouput vertices O4x1 (pOutVert): O0x O1x O2x O3x
                                 O0y O1y O2y O3y
                                 O0z O1z O2z O3z
                                 O0w O1w O2w O3w
```

これに対応する SoA のベクトル-行列乗算の例 (ベクトル反復 4 回をアンロール) を例 7-24 に示します。

例 7-24 ネイティブ SoA データレイアウトを使用したベクトル-行列乗算

```
   mov ebx, pVert
   mov ecx, pOutVert
   xor edx, edx
   movaps xmm5,[eax+16] // 列 M1 のロード
   movaps xmm6,[eax+2*16] // 列 M2 のロード
   movaps xmm7,[eax+3*16] // 列 M3 のロード
lloop_vert:
   mov eax, pMat
   xor edi, edi
   movaps xmm0, [ebx ] // V3x, V2x, V1x, V0x のロード
   movaps xmm1, [ebx ] // V3y, V2y, V1y, V0y のロード
   movaps xmm2, [ebx ] // V3z, V2z, V1z, V0z のロード
   movaps xmm3, [ebx ] // V3w, V2w, V1w, V0w のロード
loop_mat:
   movaps xmm4, [eax] // m00, m00, m00, m00,
   mulps xmm4, xmm0 // m00*V3x, m00*V2x, m00*V1x, m00*V0x,
   movaps xmm4, [eax + 16] // m01, m01, m01, m01,
   mulps xmm5, xmm1 // m01*V3y, m01*V2y, m01*V1y, m01*V0y,
   addps xmm4, xmm5
   movaps xmm5, [eax + 32] // m02, m02, m02, m02,
   mulps xmm5, xmm2 // m02*V3z, m02*V2z, m02*V1z, m02*V0z,
   addps xmm4, xmm5
   movaps xmm5, [eax + 48] // m03, m03, m03, m03,
   mulps xmm5, xmm3 // m03*V3w, m03*V2w, m03*V1w, m03*V0w,
   addps xmm4, xmm5
   movaps [ecx + edx], xmm4
   add eax, 64
   add edx, 16
   add edi, 1
   cmp edi, 4
   jb lloop_mat
   add ebx, 64
   cmp edx, top
   jb lloop_vert
```

第 8 章
INT8 ディープラーニング推論

この章では、インテル® テクノロジー上のディープラーニング推論向けの INT8 データ型について説明します。ここでは、インテル® AVX-512 実装と新しいインテル® ディープラーニング・ブースト (インテル® DL ブースト) 命令を使用した実装の両方についてカバーします。

この章はいくつかの節に分けて説明します。最初の節では、INT8、具体的にはインテル® DL ブースト命令を主なデータタイプとして説明し、ML ワークロードで使用する命令を紹介します。2 番目の節では、効率良い推論計算向けの一般的な方法論とガイドラインでについて説明します。3 番目の節では CNN 向けの最適化について説明し、最後の節で LSTM/RNN 固有の最適化について説明します。

関連がある場合、インテル® DL ブースト命令の有無にかかわらず例を示します。多くの場合 (量子化やメモリーレイアウトなど)、オフラインで実行可能なステップと、実行時に行うステップがあります。各ステップについてその都度明確に説明していきます。

8.1 ディープラーニング推測向けの INT8 データ型について

伝統的にディープラーニングでは、32 ビット浮動小数点 (F32) データ型が使用されてきました (https://itpeernetwork.intel.com/myth-busted-general-purpose-cpus-cant-tackle-deep-neural-networktraining/#gs.rGp9lgWH を参照)。INT8 は精度をほとんど低下させることなく大幅なパフォーマンス向上をもたらし、ディープラーニングの推論で採用されはじめています。単一の F32 FMA 命令に対し、INT8 MAC 操作に必要な 4 つのナローデータ型と 3 つのインテル® AVX-512 命令は、ほぼ 1.33 倍のゲインをもたらします。Skylake† Server マイクロアーキテクチャー・ベースのインテル® Xeon® スケーラブル・プロセッサーで、ResNet-50、Inception-Resnet V2、SRGAN および NMT を評価した結果、INT8 の少ないメモリー・フットプリントにより、1.5 倍以上のスピードアップを観測できました。8.2 節では、Cascade Lake† 製品ベースのプロセッサーで導入されたインテル® ディープラーニング・ブースト命令について説明します。これにより、DL 推論のパフォーマンスがさらに向上します。

ここでは、DL 推論の 2 つの利用ケースを検討します。最初のケースはスループット・モデルです。ここでは、要素 (イメージや文章) が単一の要素にかかる時間に関係なく処理されます。通常、この使用モデルは、大量のイメージを処理して分類したり、特定のユーザーに合わせた推奨事項をオフラインで準備するサーバーに適しています。次の使用モデルは、単一要素の計算時間に制限があるスループット・レイテンシー・モデルです。この使用モデルは、オンライン計算 (言語翻訳、リアルタイム・オブジェクト検出など) に適しています。

8.2 インテル® DL ブーストについて

インテル® ディープラーニング・ブースト (インテル® DL ブースト) 命令はインテル® AVX-512 命令に含まれ、ニューラル・ネットワークのワークロードをスピードアップするように設計されています。これらの命令は、CPUID.07H.0H:ECX.AVX512_VNNI[bit 11] = 1 の場合にサポートされます。

ここでは、この新しい命令について説明し、以前のインテル® AVX-512 コードとの簡単な比較を示します。命令の完全な定義 (VPDP プリフィクスと命令) については、『インテル® アーキテクチャー命令セット拡張プログラミング・リファレンス』または『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発マニュアル』を参照してください。

8.2.1 符号なしおよび符号ありバイトの積和演算 (VPDPBUSD 命令)

VPDPBUSD は、32 ビット整数アキュムレーターで 8 ビット整数積和ベクトル演算を行います。2 つのソース・ベクトル・オペランドと 1 つのソース/デスティネーション・ベクトル・オペランドから成ります。2 つのソースオペランドは、8 ビット整数データ要素を含みます。ソース/デスティネーション・オペランドは、32 ビット・データ要素を含みます。

例えば、各ソースオペランドが 64 x 8 ビット要素を含み、ソース/デスティネーション・オペランドが 16 x 32 ビット要素を含む 512 ビットのベクトルオペランドについて考えます。

この命令は、各ソースオペランドの 64 個の要素を 16 個の要素から成る 4 つのグループに分割し、それぞれのソースオペランドから 1 グループずつ、そのグループの 4 要素を垂直方向に乗算して、4 つの 32 ビット中間結果を生成します。次に、4 つの 32 ビット中間結果と 3 番目のベクトルオペランド (ソースオペランド) の垂直方向に対応する 32 ビット整数要素の 5 方向加算を実行し、その結果を 3 番目のベクトルオペランド (デスティネーション・オペランド) に 32 ビット・データ要素として格納します。インテル® AVX-512 は 16 ビットの中間結果を飽和させるため、同じ機能をより高い精度で実現する VPDPBUSD でインテル® AVX-512 の 3 つの命令シーケンス (VPMADDUBSW + VPMADDWD + VPADDD) を置き換えます。図 8-1 を参照してください。

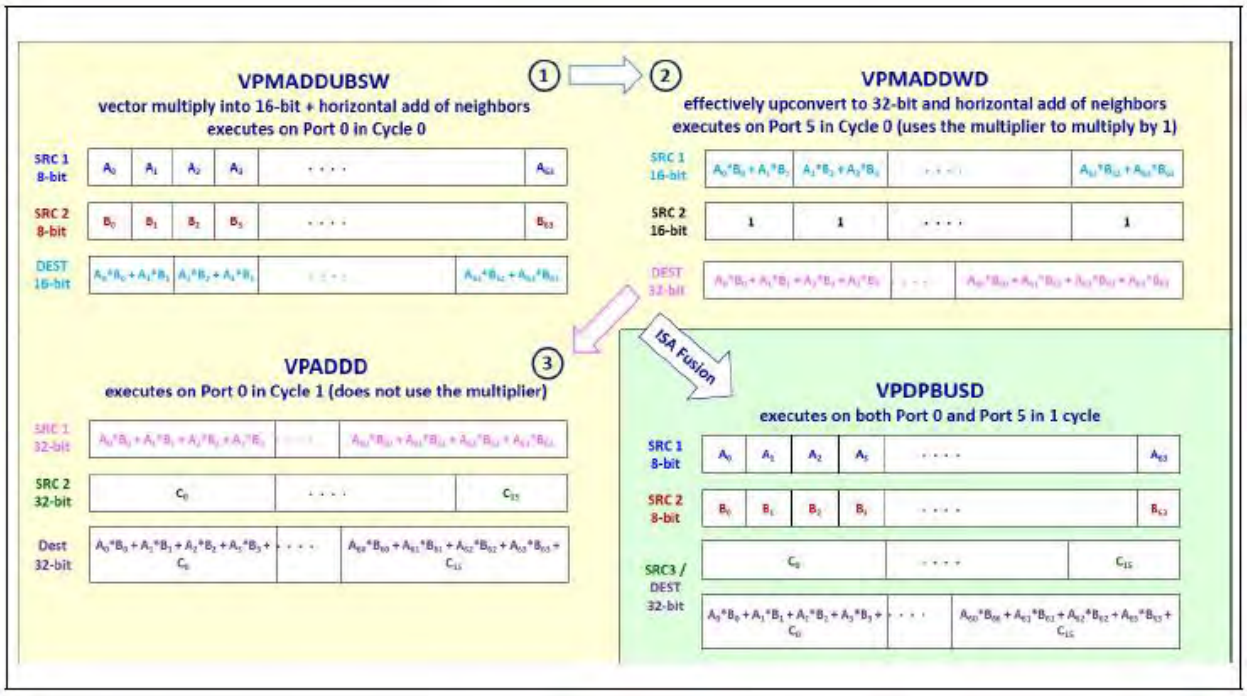


図 8-1 VPMADDUBSW + VPMADDWD + VPADDD を VPDPBUSD に融合
(サーバー・アーキテクチャーでは 3x ピーク Ops、クライアント・アーキテクチャーでは 2x ピーク Ops)

例 8-1 では、VPDPBUSD 命令を使用して SIGNAL と WEIGHT の 2 つのバイト行列の行列乗算を高速化しています。ソース行列はそれぞれ M x K および K x N の次元を持ち、行優先であると仮定すると、ソース行列は次のレイアウトを保持します。

- 次の手順で行列 SIGNAL[M][K] からなる行列信号 [K/64][M][64]:
  FOR m = 0 … M-1
  FOR k = 0 … K-1
  signal[k/64][m][k%64] = SIGNAL[m][k]

- 次の手順で行列 WEIGHT[K][N] からなる行列の重み [K/64][N][4]:
  FOR k = 0 … K-1
  FOR n = 0 … N-1
  weight[k/4][n][k%4] = WEIGHT[k][n]

例 8-1 VPDPBUSD 実装

インテル® DL ブースト以前のベクトル実装 (インテル® AVX-512)	インテル® DL ブーストの VPDPBUSD 実装
// アンロールされた行列乗算の内部ループ vpbroadcastd zmm31, dword ptr [onew] vpbroadcastd zmm24, [signal] vmovups zmm25, [weight] vmovups zmm26, [weight + 64] vmovups zmm27, [weight + 128] vmovups zmm28, [weight + 192] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm0 , zmm0 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm6 , zmm6 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm12, zmm12, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm18, zmm18, zmm30 vpbroadcastd zmm24, [signal + 64] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm1 , zmm1 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm7 , zmm7 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm13, zmm13, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm19, zmm19, zmm30 vpbroadcastd zmm24, [signal + 128] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm2 , zmm2 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm8 , zmm8 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm14, zmm14, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm20, zmm20, zmm30 vpbroadcastd zmm24, [signal + 192] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm3 , zmm3 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31	// アンロールされた行列乗算の内部ループ vpbroadcastd zmm24, [signal] vpbroadcastd zmm25, [signal + 64] vpbroadcastd zmm26, [signal + 128] vpbroadcastd zmm27, [signal + 192] vmovups zmm28, [weight] vmovups zmm29, [weight + 64] vmovups zmm30, [weight + 128] vmovups zmm31, [weight + 192] vpdpbusd zmm0 , zmm24, zmm28 vpdpbusd zmm6 , zmm24, zmm29 vpdpbusd zmm12, zmm24, zmm30 vpdpbusd zmm18, zmm24, zmm31 vpdpbusd zmm1 , zmm25, zmm28 vpdpbusd zmm7 , zmm25, zmm29 vpdpbusd zmm13, zmm25, zmm30 vpdpbusd zmm19, zmm25, zmm31 vpdpbusd zmm2 , zmm26, zmm28 vpdpbusd zmm8 , zmm26, zmm29 vpdpbusd zmm14, zmm26, zmm30 vpdpbusd zmm20, zmm26, zmm31 vpdpbusd zmm3 , zmm27, zmm28 vpdpbusd zmm9 , zmm27, zmm29 vpdpbusd zmm15, zmm27, zmm30 vpdpbusd zmm21, zmm27, zmm31

vpaddd zmm9 , zmm9 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm15, zmm15, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm21, zmm21, zmm30	
ベースライン	スピードアップ: 2.75x[1]

注意:

1. Ice Lake† Client マイクロアーキテクチャー・ベースのプロセッサーなど、インテル® ディープラーニング・ブースト (インテル® DL ブースト) をサポートするクライアント・アーキテクチャーでは、2 倍のゲインしか得られません。これは、VPADDD 命令がポート 5 のベクトル SIMD ユニットを使用するため、ベースラインがインテル® DL グースとの 1 サイクルに対して 64 MAC (ピーク) ごとに 2 サイクル要するためです。

8.2.2 符号ありワード整数の積和演算 (VPDPWSSD 命令)

VPDPWSSD は、32 ビット整数アキュムレーターで 16 ビット整数積和ベクトル演算を行います。2 つのソース・ベクトル・オペランドと 1 つのソース/デスティネーション・ベクトル・オペランドから成ります。2 つのソースオペランドは、16 ビット整数データ要素を含みます。ソース/デスティネーション・オペランドは、32 ビット・データ要素を含みます。

8 ビットの VPDPBUSD 命令で推論の精度に損失の問題が生じる場合、代わりに 16 ビットの VPDPWSSD 命令を使用できます。新しい BFLOAT16 データ型とそれに関連する命令がインテル® プロセッサーでサポートされるまで、このようなシナリオでは FP32 命令に戻すことを推奨します。

Ice Lake† Client マイクロアーキテクチャーなどクライアント・アーキテクチャーでは VPDPWSSD 命令によるパフォーマンス向上は期待できません。

8.3 一般的な最適化

8.3.1 メモリーレイアウト

NHWC メモリーレイアウトが TensorFlow* パフォーマンス・ガイドの記載に従っていると仮定します (詳細に関しては、このドキュメントのデータ形式の節を参照)。入力がネイティブ形式 (行または列のどちらか) である場合、データは計算開始時にスカラーコードによって最適化されたレイアウトに変換されます。詳細については、インテル® アカデミーを参照してください。

8.3.2 量子化

量子化は、活性化と重みにおいてデータ型のサイズを減らす (通常 float から int8/uint8 に) プロセスであり、ディープ・ニューラル・ネットワーク向けインテル® マス・カーネル・ライブラリー (インテル® MKL-DNN) ドキュメントやインテル® AI アカデミーなどさまざまなリソースで詳しく説明されています。

8.3.2.1 重みの量子化

重みは、量子化係数 127/max_range を使用して量子化されます。これは、精度を高めるため OFM ごとに実行できます。重みは事前に判明しているため、プロセスはオフラインで実行できます。

8.3.2.2 活性化の量子化

次のコード例に、量子化係数を指定してデータをスカラーまたはベクトル形式で量子化する方法を示します。

例 8-2 活性化の量子化

指定された係数で入力を量子化 (スカラー)

```
void quantize_activations(const float* data, u8* quantized_data, int count, Dtype factor, int bits, int offset = 0)
{
    int quant_min = 0;
    int quant_max = (1 << bits) - 1;
    #pragma unroll (4)
    for (int i = 0; i < count; i++) {
        int int32_val = offset + (int) round(data[i] * factor);
        int32_val = std::max(std::min(int32_val, quant_max), quant_min);
        u8 quant_val = (u8)int32_val;
        quantized_data[i] = quant_val;
    }
}
```

指定された係数で入力を量子化 (ベクトル)

```
void quantize_activations(const float* data, u8* quantized_data, int count, Dtype factor, int bits, int offset = 0)
{
    int quant_min = 0;
    int quant_max = (1 << bits) - 1;
    int count_aligned = ALIGN(count, INTR_VECTOR_LENGTH_32_bit);
    __m512i offset_broadcast = _mm512_set1_epi32(offset);
    __m512 factor_broadcast = _mm512_set1_ps(factor);
    __m512i quant_min_broadcast = _mm512_set1_epi32(quant_min);
    __m512i quant_max_broadcast = _mm512_set1_epi32(quant_max);
    #pragma unroll (4)
        for (int i = 0; i < count_aligned; i += INTR_VECTOR_LENGTH_32_bit) {
        __m512 data_m512 = _mm512_load_ps(&data[i]);
        data_m512 = _mm512_mul_ps(data_m512, factor_broadcast);
        __m512i data_i32 = _mm512_cvt_roundps_epi32
                                (data_m512,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
        data_i32 = _mm512_add_epi32(data_i32, offset_broadcast);
        data_i32 = _mm512_max_epi32(data_i32, quant_min_broadcast);
        data_i32 = _mm512_min_epi32(data_i32, quant_max_broadcast);
        __m128i q = _mm512_cvtusepi32_epi8(data_i32);
        _mm_store_si128((__m128i*)(&quantized_data[i]), q);
    }
}
```

8.3.2.3 負の活性化を量子化

VPMADDUBSW と VPDPBUSD は、最初のパラメーターで符号なしの値と 2 番目のパラメーターで符号付きの値の組み合わせのみをサポートします。2 番目のパラメーターでは符号付きの重みを容易にサポートできますが、最初のパラメーターで符号付きの活性化をサポートするには何らかの操作が必要です。負の活性化の可能性がある場合 (例えば、現在のレイヤーの前に ReLU がないなど)、最初に -128、127 の値に量子化して、次に結果に 128 を加算して負ではない活性化を行います。OFM バイアスから 128* (OFM フィルターのすべての重みの合計) を引くことで、このオフセットを補正します。詳細は、インテル® AI アカデミーを参照してください。

8.3.3 マルチコアに関する考慮事項

8.3.3.1 大規模バッチ (スループット・ワークロード)

大規模バッチ計算では、ワークを複数のコアに分割することでマルチコア処理から大きな恩恵を受けることができますが、キャッシュの局所性の観点から同じオブジェクト (イメージや文など) を同じ物理コアで処理することが最善です。また、活性化はオブジェクトごとに異なりますが、重みは通常共有できます。マルチスレッド・モデルでは、コア間で重みを容易に共有できます。マルチコアシステムで大規模バッチ入力を処理するガイドラインを次に示します。

1. スレッドモデルを使用して、複数のコア間で重みを共有します。しかし、マルチソケット・マシンでは、ソケット /NUMA ドメインごとに専用のプロセスが必要です。
2. スレッド・アフィニティーとオブジェクト・アフィニティーを定義して同一物理コア上で単一のオブジェクトを処理し、コアのキャッシュ内で活性化を維持します (キャッシュサイズよりも小さな場合)。
3. ワークがコア間で均等にロードされるように、オブジェクトをコア数の倍数でバッチ処理します。
4. すべての物理コアの兄弟スレッド (論理プロセッサー) がアイドル状態であることを確認します。
5. 例えば、次のレイヤーに移行する前にコアに割り当てられているすべてのイメージで同じレイヤーが実行される BFS モードで、コアごとのミニバッチの実行を検討します。これにより、複数の活性化によってコアキャッシュが汚染されますが、重みの再利用が改善し、(場合によっては) パフォーマンスが向上します。ミニバッチは、行列 (テンソル) が非常に細く、積和演算ユニットの使用率が低い場合に有用です。

8.3.3.2 小規模バッチ (レイテンシー・ワークロードのスループット)

小規模バッチ処理には、通常、シングルコアでは解決できないレイテンシー要件があります。この場合、単一オブジェクトの処理を複数コアに分割する必要があります。

単一オブジェクトを処理するコア数を増やすと利点が減少することが多いため、最適なコア数を特定します。ときには、バッチ数を若干増やしてもレイテンシーの要件を満足することができます (例えば、1 から 2 または 3 など)。単一のバッチ・インスタンスに最適なスレッド数を特定すると、システムを完全に活用して複数のインスタンスを実行できるようになります。

8.3.3.3 NUMA

高いパフォーマンスを提供する Cascade Lake† 2 ソケットサーバーには、2 つの Cascade Lake† Advanced Performance パッケージが含まれており、各パッケージはインテル® ウルトラ・パス・インターコネクト (インテル® UPI) リンクで接続された 2 つのプロセッサー・ダイで構成され、合計 4 つの NUMA ドメインを持っています。このような構成では、NUMA ドメイン/ダイごとに個別の DL プロセスを維持することが重要です。これは、複数の NUMA ドメインを持つ以前の世代の製品による 2 ソケットの構成でも同様です。

8.4 CNN
8.4.1 畳み込みレイヤー
8.4.1.1 直接畳み込み

インテル® DL ブーストのベクトル演算を利用するため、直接畳み込み演算を一連の行列加算と乗算に置き換えるようにします。次の説明では、入力、重み、出力に代わる行列をそれぞれ A、B、C としています。

メモリーレイアウト

入力を行列形式で表現するため、空間次元を A の行 (以降 M 次元と称します)、チャネル (IFM) の次元を A の列 (以降 K 次元と称します) となるように平坦化します。同様に、出力の空間次元は C の行となり、チャネル (OFM) 次元は C の列になります (以降 N 次元と称します)。図 8-2 では、サイズ 5x5 の 6 つのチャネルが畳み込みレイヤーによってサイズ 3x3 の 4 つのチャネルに変換されています。

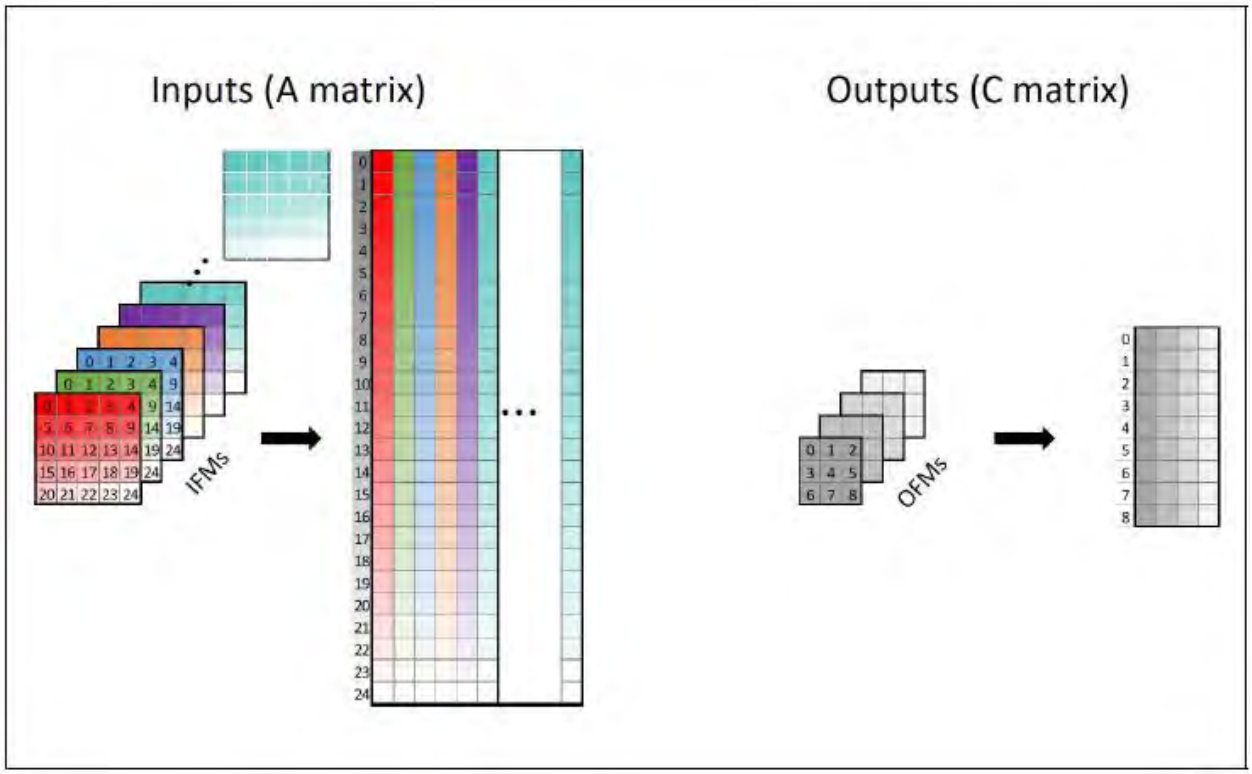


図 8-2 行列レイアウト、入力および出力

標準の 2D 畳み込みでは、各ターゲット OFM に対してサイズ KH x KW x #IFM の 3D カーネルの差 #OFM を使用しますが、ここで KH、KW、#IFM、および #OFM は畳み込みカーネルの高さと幅、入力チャネル数、および出力チャネル数です。

重みは、サイズ #IFM x #OFM の KH x KW 行列に変換されます (図 8-3 を参照)。

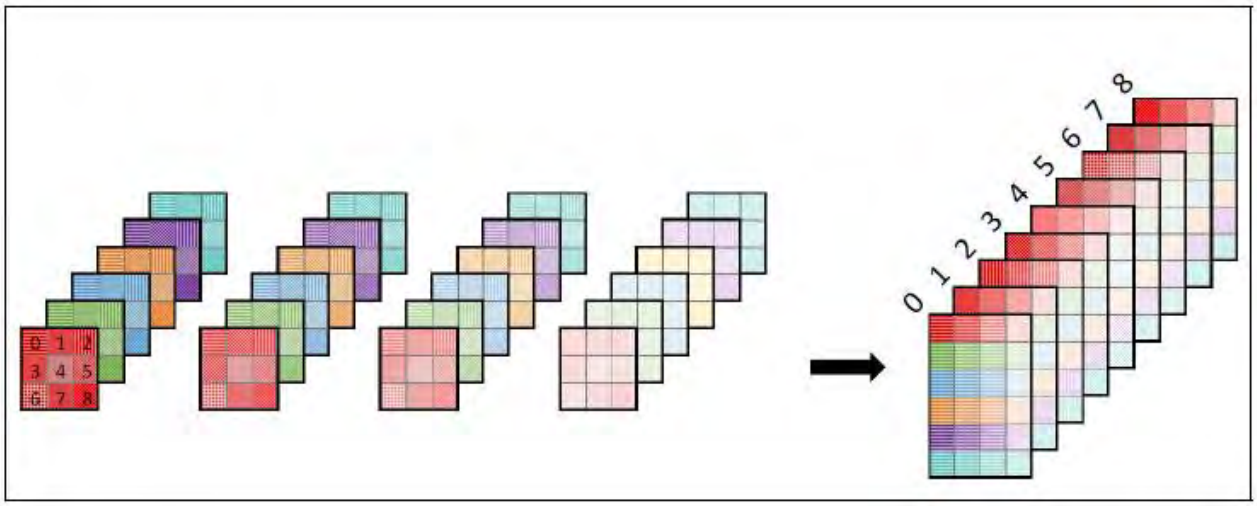


図 8-3 変換された重み

その結果、畳み込み操作 (図 8-4 を参照) は、

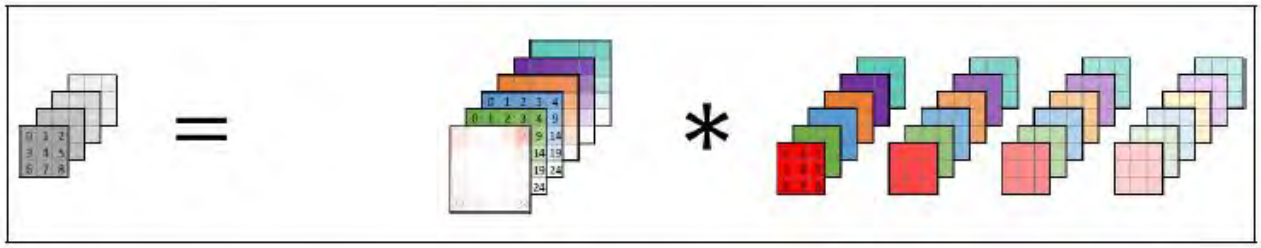


図 8-4 畳み込み操作

一連の行列の乗算と加算になります (図 8-5 を参照)。

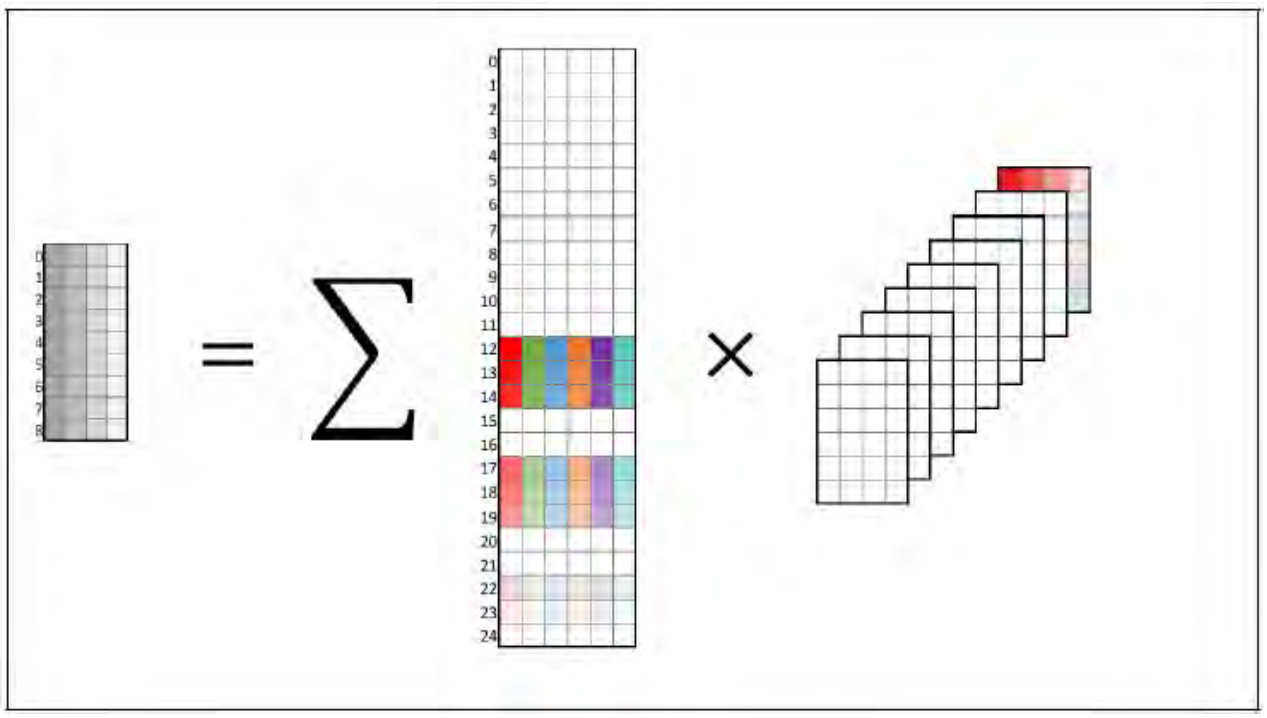


図 8-5 行列の乗算と総和

行列乗算

行列乗算は標準的な方法で行われます (第 18 章「インテル® AVX-512 命令向けのソフトウェア最適化」を参照)。

ブロック化

一般に扱う行列は大規模であるため、乗算の結果を累積しながら行列全体を反復的に横断 (トラバース) する必要があります。そのため、結果の累積にいくつかのレジスター (アキュムレーター) を割り当てて、必要に応じて再利用するため A および B 行列を一時的にキャッシュするいくつかのレジスターを用意します。

行列を横断する順番は、全体のパフォーマンスに影響する可能性があります。一般に、M または N 次元で次の要素に移動する前に、K 次元全体を移動することが推奨されます。K 次元の処理が完了すると、アキュムレーター内の結果は最終結果となります (部分的な結果については以下の説明を参照してください)。それらは、畳み込みの後処理ステージ (「畳み込みの後処理」を参照) に送られ、出力場所に保存されます。しかし、M または N 次元で移動する前に K 次元の処理が終わらないと、アキュムレーター内の結果は部分的であり、それらはいくつかの A の列と B の行の乗算結果となります。この結果は、新しいデータチャンク向けにアキュムレーターを解放するため、補助バッファーに保存する必要があります。これらの M、N 座標に戻る場合、結果を補助バッファーからロードする必要があります。そのため、K 次元の余りが生じないシナリオと比較すると、追加のストアとロードが行われることになります。さらに、M または N 次元の移動を制限することを推奨します。一般的に、行列 B のアキュムレーター K キャッシュレベル (図 8-6) が DCU にあり、キャッシュブロックの蓄積サイズ (図 8-7) が MLC に収まり可能な限り大きな場合に最良の結果が得られました。ただし、蓄積サイズがかなり大きい場合でも (最大 MLC の 3 倍) 最良の結果が得られることがあります。これらのハイパーパラメーターは、通常試行錯誤から得られます。

「K 次元の余りが生じない」ガイドラインは、すべてのケースで最適な結果をもたらすわけではありません。また、M、N の局所性の最適な範囲は状況に応じて決定されるべきです。現代の CNN におけるさまざまなシナリオに対応するため、畳み込みプロセスの制御フローを構築する簡単で効果的な方法を概説します。

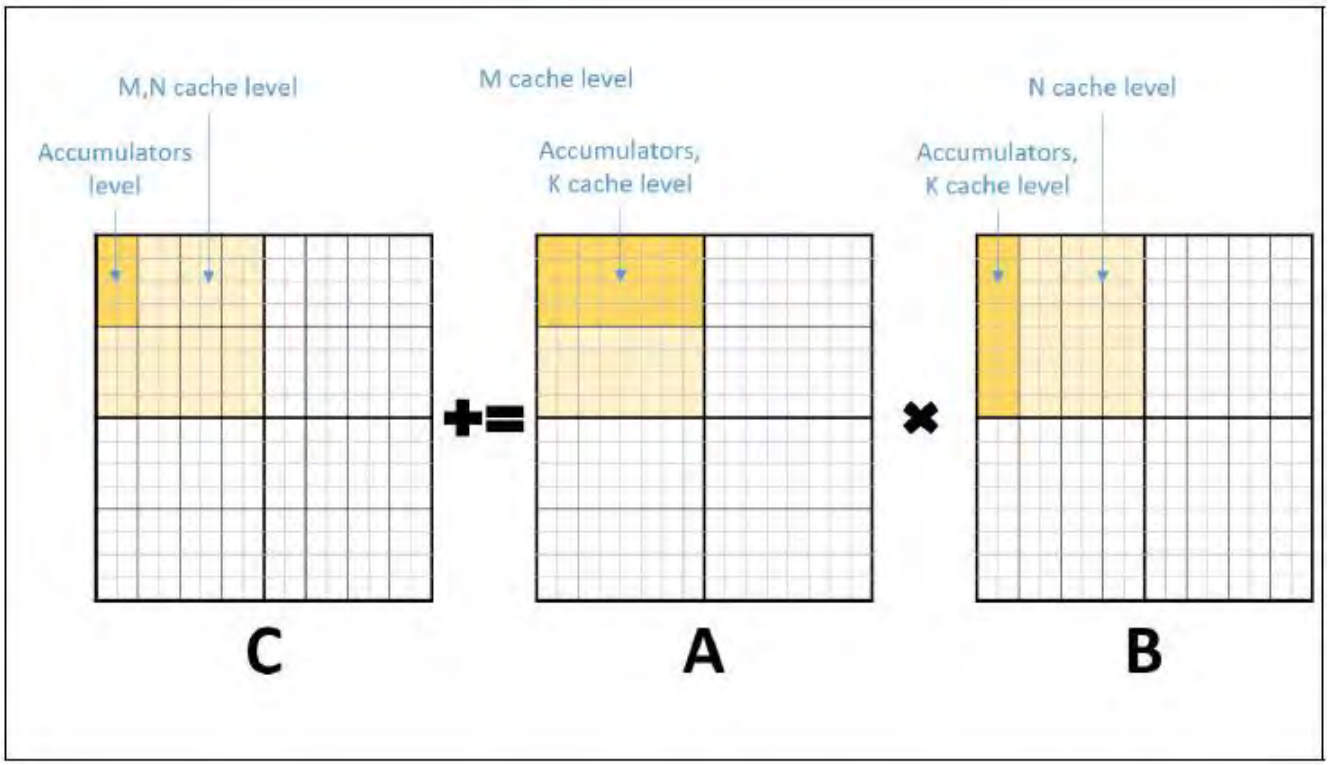


図 8-6 3 層のフレキシブルな 2D ブロック化

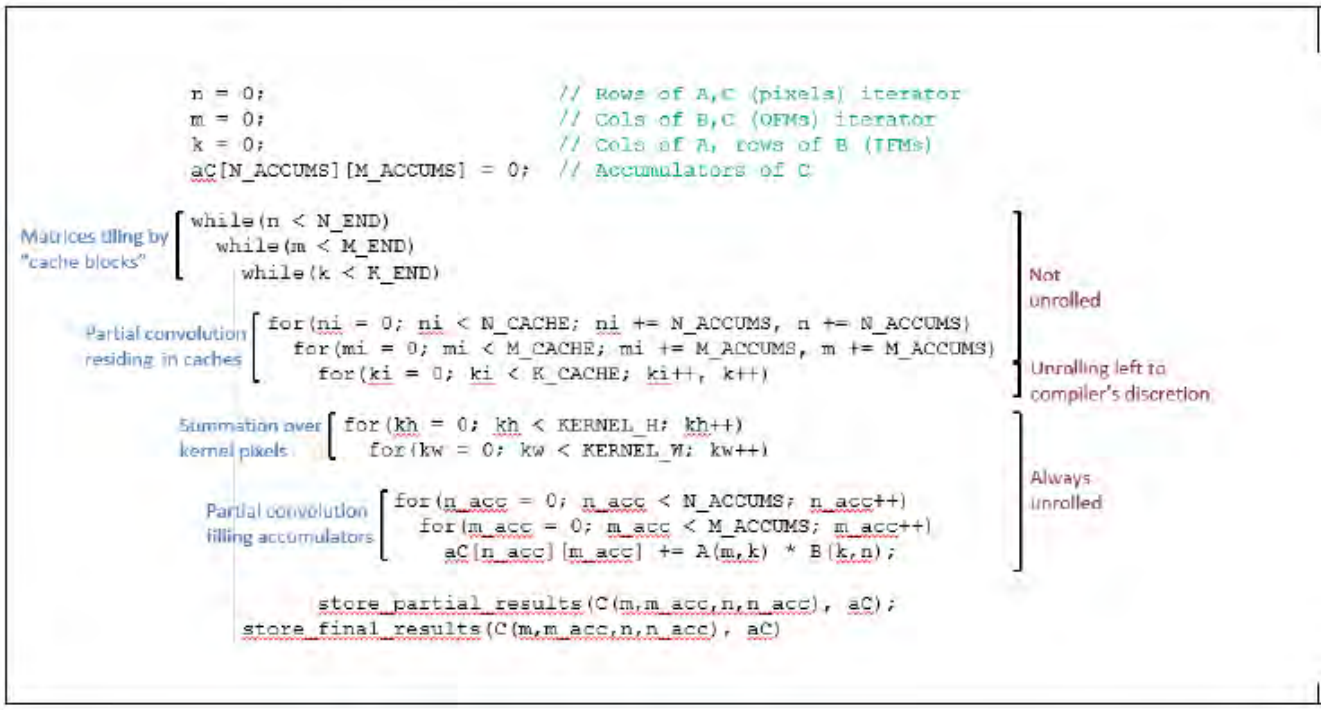


図 8-7 3 層のフレキシブルな 2D ブロック化ループ

直接畳み込みの例

次のパラメーターを持つ直接畳み込みについて考えます。

IFM サイズ = 34x34、KH = KW = 3、畳み込みストライド = 1、#IFMs = 32、#OFMs = 32、および IFM のパディングなし。つまり、OFM のサイズは 32 x 32 です。

選択されたブロック化は、M_ACCUMS = 4、N_ACCUMS = 2、M_CACHE = 1024、N_CACHE = 2、K_CACHE = 8 です。これらのブロック化パラメーターは、K 次元が完全に横断され、一時的な結果をストアする必要がないことを保証します。次に、OFM のピクセル (0,0)、(0,1)、(0,2)、および (0,3) に対して 32 出力チャネルを計算するため、以下のコードを使用できます。

例 8-3 直接畳み込み

直接畳み込み

```
vpxord zmm0 , zmm0 , zmm0 // ゼロ·アキュムレーター·タイル [m,n] = [0,0]
vpxord zmm1 , zmm1 , zmm1 // ゼロ·アキュムレーター·タイル [m,n] = [1,0]
vpxord zmm2 , zmm2 , zmm2 // ゼロ·アキュムレーター·タイル [m,n] = [2,0]
vpxord zmm3 , zmm3 , zmm3 // ゼロ·アキュムレーター·タイル [m,n] = [3,0]
vpxord zmm4 , zmm4 , zmm4 // ゼロ·アキュムレーター·タイル [m,n] = [0,1]
vpxord zmm5 , zmm5 , zmm5 // ゼロ·アキュムレーター·タイル [m,n] = [1,1]
vpxord zmm6 , zmm6 , zmm6 // ゼロ·アキュムレーター·タイル [m,n] = [2,1]
vpxord zmm7 , zmm7 , zmm7 // ゼロ·アキュムレーター·タイル [m,n] = [3,1]

for (int k = 0; k < 32 / 4; ++k) {
    vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=0,kw=0,k,n=0)] // 重みのロード [kh,kw,n] =
    [0,0,+0]
    vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=0,kw=0,k,n=1)] // 重みのロード [kh,kw,n] =
    [0,0,+1]
    vpbroadcastd zmm8 , dword ptr [IFM_ADDR(0)] // IFM ピクセルのロード +0
    vpbroadcastd zmm9 , dword ptr [IFM_ADDR(1)] // IFM ピクセルのロード +1
    vpbroadcastd zmm10 , dword ptr [IFM_ADDR(2)] // IFM ピクセルのロード +2
```

```
vpbroadcastd zmm11 , dword ptr [IFM_ADDR(3)] // IFM ピクセルのロード +3
vpdpbusd zmm0 , zmm8 , zmm12
vpdpbusd zmm1 , zmm9 , zmm12
vpdpbusd zmm2 , zmm10, zmm12
vpdpbusd zmm3 , zmm11, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=0,kw=1,k,n=0)] // 重みのロード [kh,kw,n] =
[0,1,+0]
vpdpbusd zmm4 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(4)] // IFM ピクセルのロード +4
vpdpbusd zmm5 , zmm9 , zmm13
vpdpbusd zmm6 , zmm10, zmm13
vpdpbusd zmm7 , zmm11, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=0,kw=1,k,n=1)] // 重みのロード [kh,kw,n] =
[0,1,+1]
vpdpbusd zmm0 , zmm9 , zmm12
vpdpbusd zmm1 , zmm10, zmm12
vpdpbusd zmm2 , zmm11, zmm12
vpdpbusd zmm3 , zmm8 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=0,kw=2,k,n=0)] // 重みのロード [kh,kw,n] =
[0,2,+0]
vpdpbusd zmm4 , zmm9 , zmm13
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(5)] // IFM ピクセルのロード +5
vpdpbusd zmm5 , zmm10, zmm13
vpdpbusd zmm6 , zmm11, zmm13
vpdpbusd zmm7 , zmm8 , zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=0,kw=2,k,n=1)] // 重みのロード [kh,kw,n] =
[0,2,+1]
vpdpbusd zmm0 , zmm10, zmm12
vpdpbusd zmm1 , zmm11, zmm12
vpdpbusd zmm2 , zmm8 , zmm12
vpdpbusd zmm3 , zmm9 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=1,kw=0,k,n=0)] // 重みのロード [kh,kw,n] =
[1,0,+0]
vpdpbusd zmm4 , zmm10, zmm13
vpbroadcastd zmm10 , dword ptr [IFM_ADDR(34)] // IFM ピクセルのロード +34
vpdpbusd zmm5 , zmm11, zmm13
vpbroadcastd zmm11 , dword ptr [IFM_ADDR(35)] // IFM ピクセルのロード +35
vpdpbusd zmm6 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(36)] // IFM ピクセルのロード +36
vpdpbusd zmm7 , zmm9 , zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=1,kw=0,k,n=1)] // 重みのロード [kh,kw,n] =
[1,0,+1]
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(37)] // IFM ピクセルのロード +37
vpdpbusd zmm0 , zmm10, zmm12
vpdpbusd zmm1 , zmm11, zmm12
vpdpbusd zmm2 , zmm8 , zmm12
vpdpbusd zmm3 , zmm9 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=1,kw=1,k,n=0)] // 重みのロード [kh,kw,n] =
[1,1,+0]
vpdpbusd zmm4 , zmm10, zmm13
vpbroadcastd zmm10 , dword ptr [IFM_ADDR(38)] // IFM ピクセルのロード +38
vpdpbusd zmm5 , zmm11, zmm13
vpdpbusd zmm6 , zmm8 , zmm13
vpdpbusd zmm7 , zmm9 , zmm13
```

```
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=1,kw=1,k,n=1)] // 重みのロード [kh,kw,n] = [1,1,+1]
vpdpbusd zmm0 , zmm11, zmm12
vpdpbusd zmm1 , zmm8 , zmm12
vpdpbusd zmm2 , zmm9 , zmm12
vpdpbusd zmm3 , zmm10, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=1,kw=2,k,n=0)] // 重みのロード [kh,kw,n] = [1,2,+0]
vpdpbusd zmm4 , zmm11, zmm13
vpbroadcastd zmm11 , dword ptr [IFM_ADDR(39)] // IFM ピクセルのロード +39
vpdpbusd zmm5 , zmm8 , zmm13
vpdpbusd zmm6 , zmm9 , zmm13
vpdpbusd zmm7 , zmm10, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=1,kw=2,k,n=1)] // 重みのロード [kh,kw,n] = [1,2,+1]
vpdpbusd zmm0 , zmm8 , zmm12
vpdpbusd zmm1 , zmm9 , zmm12
vpdpbusd zmm2 , zmm10, zmm12
vpdpbusd zmm3 , zmm11, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=2,kw=0,k,n=0)] // 重みのロード [kh,kw,n] = [2,0,+0]
vpdpbusd zmm4 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(68)] // IFM ピクセルのロード +68
vpdpbusd zmm5 , zmm9 , zmm13
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(69)] // IFM ピクセルのロード +69
vpdpbusd zmm6 , zmm10, zmm13
vpbroadcastd zmm10 , dword ptr [IFM_ADDR(70)] // IFM ピクセルのロード +70
vpdpbusd zmm7 , zmm11, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=2,kw=0,k,n=1)] // 重みのロード [kh,kw,n] = [2,0,+1]
vpbroadcastd zmm11 , dword ptr [IFM_ADDR(71)] // IFM ピクセルのロード +71
vpdpbusd zmm0 , zmm8 , zmm12
vpdpbusd zmm1 , zmm9 , zmm12
vpdpbusd zmm2 , zmm10, zmm12
vpdpbusd zmm3 , zmm11, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=2,kw=1,k,n=0)] // 重みのロード [kh,kw,n] = [2,1,+0]
vpdpbusd zmm4 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(72)] // IFM ピクセルのロード +72
vpdpbusd zmm5 , zmm9 , zmm13
vpdpbusd zmm6 , zmm10, zmm13
vpdpbusd zmm7 , zmm11, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=2,kw=1,k,n=1)] // 重みのロード [kh,kw,n] = [2,1,+1]
vpdpbusd zmm0 , zmm9 , zmm12
vpdpbusd zmm1 , zmm10, zmm12
vpdpbusd zmm2 , zmm11, zmm12
vpdpbusd zmm3 , zmm8 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=2,kw=2,k,n=0)] // 重みのロード [kh,kw,n] = [2,2,+0]
vpdpbusd zmm4 , zmm9 , zmm13
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(73)] // IFM ピクセルのロード +73
vpdpbusd zmm5 , zmm10, zmm13
vpdpbusd zmm6 , zmm11, zmm13
```

```
    vpdpbusd zmm7 , zmm8 , zmm13
    vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=2,kw=2,k,n=1)] // 重みのロード [kh,kw,n] =
    [2,2,+1]
    vpdpbusd zmm0 , zmm10, zmm12
    vpdpbusd zmm1 , zmm11, zmm12
    vpdpbusd zmm2 , zmm8 , zmm12
    vpdpbusd zmm3 , zmm9 , zmm12
    vpdpbusd zmm4 , zmm10, zmm13
    vpdpbusd zmm5 , zmm11, zmm13
    vpdpbusd zmm6 , zmm8 , zmm13
    vpdpbusd zmm7 , zmm9 , zmm13
}
// zmm0-zmm7 の結果で後処理を行います。
```

このコードでは、IFM 値に M_ACCUM (4) zmm レジスター zmm8-zmm11 を、重み付けに N_ACCUM (2) zmm レジスター zmm12-zmm13 を割り当てています。

最初にアキュムレーターをゼロにクリアする必要があります。次に K 次元全体 (#IFM=32) を横断する必要があります。それぞれの反復は 4 つの連続した IFM を操作します。畳み込みは、一連の 4 バイト IFM データのブロードキャスト、64 バイト重みデータのロード、および乗算と累積演算で構成されます。大規模 IFM データは異なる kh、kw 値でオーバーラップするため、IFM データを効率良く再利用でき、データロード数は大幅に減少します。

8.4.1.2 低 OFM カウントによる畳み込みレイヤー

チャネルの次元に沿ったベクトル化は、ベクトルレジスターを満たすのに十分なチャネル (入力と出力の両方) がある場合 (通常、分類トポロジーの場合) に上手く動作します。しかし、敵対的生成ネットワーク (GAN) のようなケースでは、最終結果はイメージです。つまり、最後の畳み込みレイヤーには 3 つのチャネルしかありません。このレイヤーでは、空間の次元に沿ってベクトル化する方法が理にかなっています。これには、異なるデータレイアウトが必要になります。大きな中間バッファーを使用しなくてもいいように、1 つの 4x16 ブロックに計算をオンザフライでレイアウトし直し、部分的な畳み込みを行ってブロックを破棄します (このメカニズムは 1x1 カーネルに限定され、新しいレイアウトに対応して重みが並べ替えられていると仮定します)。

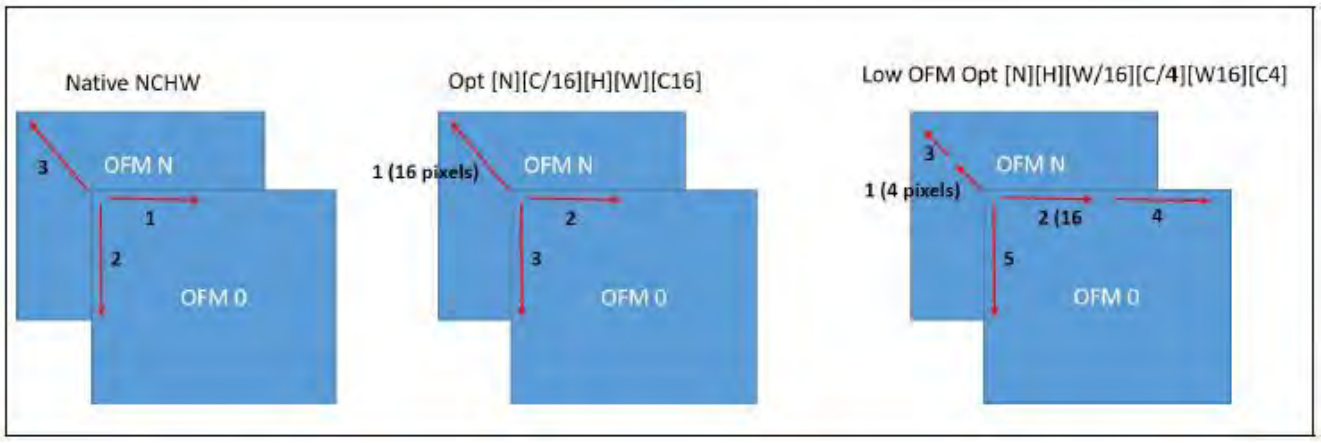


図 8-8 標準、最適化済み、および低 OFM 最適化データレイアウト[1]

注意:

1. 低 OFM 最適化の 4x16 ブロックは、オンザフライで作成され一度だけ使用されます。

例 8-4 低 OFM カウントによるレイヤーの畳み込み

低 OFM カウントによるレイヤーの畳み込み

```
# IFM_W % 16 == 0
# NUM_OFMS = 3
# NUM_IFMS = 64
# dqfs - ダウンコンバート向けの逆量子化係数の配列

int src_ifm_size = IFM_H * IFM_W * IFMBlock;
int ofm_size = IFM_W * IFM_H;

__m512i gather_indices = _mm512_setr_epi32(0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60);

      __m512 dqf_broadcast[NUM_OFMS];
      #pragma unroll(NUM_OFMS)
      for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
            dqf_broadcast[ofm] = _mm512_set1_ps(dqfs[ofm]);
      }

      for (int h = 0 ; h < IFM_H ; h++) {
          int src_line_offset = h * IFM_W * IFMBlock;
          int w = 0;
          int src_w_offset = src_line_offset;
          for ( ; w < IFM_W ; w += 16) {
          __m512i res_i32[NUM_OFMS] = { 0 };
          // オンザフライで再編成して 4x16 の OFM を畳み込み
                for (int ifm = 0 ; ifm < NUM_IFMS ; ifm += 4) {
                int src_block_offset = ifm & 0xf;
                int src_ifm_index = ifm >> 4;
                size_t src_ifm_offset = src_w_offset + src_ifm_index * src_ifm_size + src_block_offset;
                __m512i ivec = _mm512_i32gather_epi32(gather_indices, input + src_ifm_offset, 4);
                #pragma unroll(NUM_OFMS)
                    for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
                    int weight_offset = (ofm * NUM_IFMS + ifm) * 16;
                    __m512i wvec = _mm512_load_si512(weights_reorged + weight_offset);
                    res_i32[ofm] = _mm512_dpbusd_epi32(res_i32[ofm], ivec, wvec);
                    }
                }
                // ダウンコンバートして結果をネイティブレイアウトで格納
                #pragma unroll(NUM_OFMS)
                for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
                    __m512 res_f32 = _mm512_cvtepi32_ps(res_i32[ofm]);
                    res_f32 = _mm512_mul_ps(res_f32, dqf_broadcast[ofm]);
                    size_t output_offset = ofm * ofm_size + h * IFM_W + w;
                    _mm512_store_ps(output + output_offset, res_f32);
                }
                src_w_offset += 16 * IFMBlock;
          }
      }
}
```

8.4.2 畳み込みの後処理

畳み込みが行われると、レイヤーデータに対し多くの変換が実行されます。これらには、ReLU のような古典的な畳み込み後の操作、プーリングや EltWise のような個別のレイヤーとして考慮される操作、および量子化/逆量子化操作が含まれます。メモリー階層のスラッシングを減らすため、畳み込み中にこれらのステップの実行を試みます (つまり、畳み込み操作に融合します)。量子化ステップを融合することで、4 倍の計算帯域幅が得られ、メモリー帯域幅は 4 分の 1 に減少するため試みる価値があります。

8.4.2.1 融合量子化/融合逆量子化

8.3.2.1 節はオフラインでの量子化の方法を示しています。多くの場合、現在のレイヤーの逆量子化と次のレイヤーの量子化は、畳み込みステップに融合することができます。次のコードは、 OFM ブロックの畳み込み後に開始される後処理の基本操作を示しています。前述のように、この手順は同じピクセルに属する 16 個の int8 OFM を操作します。この例では、単一の係数が現在のレイヤーの逆量子化を表し、次のレイヤーへの再量子化を表すように、逆量子化係数 (存在する場合はバイアスも) が準備されていると仮定します。一般に、次のレイヤーに渡すいくつかの乗算係数 (例えば、逆量子化、レイヤー定数乗算値、および量子化) を単一の係数として示すことによってオンラインでの計算数を減らします。さらに、元の OFM が負の値 (ReLU なし) である可能性がある場合、8.3.2.1 節の手順に従ってすべての値に 127 を加算します。

例 8-5 基本 PostConv

基本 PostConv

```
// dest は同じピクセルに属する 16 個の OFM のベクトルを指します。
uint8_t* dest = (uint8_t*) (outputFeatureMaps) + offset;

// in は操作する int32 の 16 個のアキュムレーターです。
__m512 resf=_mm512_cvtepi32_ps(in); // float へ変換

// bias がある場合は bias を追加してから、シングルステップで逆量子化と再量子化を行います
if(bias) {
    resf = _mm512_fmadd_ps(resf,
                _mm512_load_ps(dqfs+OFMChannelOffset),
                _mm512_load_ps((__m512*) (bias + OFMChannelOffset)));
} else {
    resf= _mm512_mul_ps(resf,
                _mm512_load_ps(dqfs+ OFMChannelOffset));
}

#if RELU
    resf = _mm512_max_ps(resf, broadcast_zero);
#endif

// この時点では uint8 範囲内
__m512i res = _mm512_cvt_roundps_epi32
(resf,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
__m128i res8;

#if ELTWISE
    /* 融合 Eltwise 操作 */
#else
# if !RELU
    res = _mm512_add_epi8(res,_mm512_set1_epi32(128));
# endif
    res8 = _mm512_cvtusepi32_epi8(res);
```

```
#endif // ELTWISE

#if POOLING
    /* 融合プーリング操作 */
#endif
_mm_store_si128((__m128i*) dest, res8);
```

8.4.2.2 ReLu

ReLu は、畳み込みと融合した際に無視できるオーバーヘッドが生じる、ゼロレジスターを有する最大値として実装されます。

8.4.2.3 EltWise

要素ごとの操作は通常、最終結果が保存される直前にアキュムレーターで直接操作されるため、畳み込みステップに容易に融合できます。ただし、異なる入力レイヤーの量子化係数は通常同じではないため、入力を f32 に逆量子化し操作した後に再度量子化する必要があります (ベクトル化されたコードでこのステップの最適化を示します)。

次に、入力と出力のデータ型を想定した eltwise 操作の例を示します。すべての例で、畳み込み操作からのデータは VPDPBUSD 命令が返す INT32 データであり、量子化された出力は uint8 であるため、量子化されない出力は負の値になる可能性があります。負の値と uint8 への量子化をどのように行うかは、8.3.2 節「量子化」を参照してください。

次の最適化されたコードは、「dest」が出力の同じピクセルに属する 16 個の OFM ベクトルを指すと仮定する、いくつかの eltwise の使用例に対する実装を示します。原則として、次の式のように、eltwise データと畳み込みデータを逆量子化し、加算してから逆量子化します。

$$result = \left(eltwise_{f32} \times eltwiseDQfactor + conv_{i32} \times convDQfactor\right) \times NextQfactor$$

ただし、係数 ([ ] 内の演算) はオフラインで前処理できるため、オンラインでの乗算は 2 回だけになります。

$$result = \left(eltwise_{f32} + conv_{i32}\left[\frac{convDQfactor}{eltwiseDQfactor}\right]\right) \times \left([NextQfactor \times eltwiseDQfactor]\right)$$

例 8-6 Uint8 残差入力

Uint8 残差入力

```
__m128i eltwise_u8 = _mm_load_si128((const __m128i*) (eltwise_data + ew_offset));
__m512i eltwise_i32 = _mm512_cvtepu8_epi32(eltwise_u8);
if (signed_residual) {
eltwise_i32 = _mm512_sub_epi32(eltwise_i32, broadcast_128);
}

__m512 eltwise_f32 = _mm512_cvtepi32_ps(eltwise_i32);
resf = _mm512_add_ps(eltwise_f32, resf); /* 畳み込みの結果と加算 */

/* 一度の操作で逆量子化して次のレイヤーに再量子化 */
resf = _mm512_mul_ps(resf, broadcast_fused_eltwise_out_qfactor);
if (relu)
    resf = _mm512_max_ps(resf, broadcast_zero);
    __m512i data_i32 = _mm512_cvt_roundps_epi32(resf,
                                (_MM_FROUND_TO_NEAREST_INT|
                                _MM_FROUND_NO_EXC));
res8 = _mm512_cvtusepi32_epi8(data_i32);

if (!relu) {
    res8 = _mm_add_epi8(res8, _mm_set1_epi8(-128)); // 128 加算に戻る
}
```

8.4.2.4 プーリング

プーリングレイヤーを畳み込みステップに融合するのは困難かもしれませんが、場合によっては容易なこともあります。例えば、Inception ResNet 50 の最後の畳み込み平均プーリングは、各 OFM チャネルの 8x8 ピクセルすべてを単一の値に平均化するため OFM ごとに単一の値を出力します。このような操作は、すべてのピクセルで同じように動作するため、容易に融合できます。

例 8-7 8x8 レイヤーのストライド 1 の 8x8 平均プーリング

8x8 レイヤーのストライド 1 の 8x8 平均プーリング

```
__m512 pool_factor = _mm512_set1_ps((float)1.0/64);

// resf は基本 PostConv コード例で計算された 16 個の float 値です

resf = _mm512_mul_ps(resf, pool_factor ); // 64 で除算

// pool_offset は、現在の OFM (OFMItr) にのみ依存します。
int pool_offset = (BlockOffsetOFM + OFMItr);
float *pool_dest = (float *) (outputFeatureMaps) + pool_offset;
__m512 prev_val = _mm512_load_ps((const __m512 *) (pool_dest));
__m512 res_tmp_ps = _mm512_add_ps(resf, prev_val);
_mm512_store_ps((__m512 *) pool_dest, res_tmp_ps);
```

次の融合されていないベクトル化コードにより、最大および平均プーリングを実行できます。また、プーリングステップでは入力量子化範囲を出力量子化範囲に調整できます。これは、通常 NOP として実装され連結レイヤーの前に必要です。すべての連結レイヤーの出力量子化範囲は同じでなければなりません。

例 8-8 融合されていないベクトル化プーリング

融合されていないベクトル化プーリング

```
// 次の IFM セットに移動する前に、16 個の IFM を同時にプーリング
for (int ifm = 0; ifm < no_ifm; ifm+=16) {
    // 入力と出力で、プーリングするブロック位置を検出
    int block_idx = (ifm >> 4) ;
    size_t block_offset = (spatial_size_in * block_idx) << 4;
    size_t block_offset_out = (spatial_size_out * block_idx) << 4;

    for (int y = -pad_h_; y < top_y + pad_h_; y++) {
        int y_offset_out = (top_x * 16 * (y + pad_h_));
        for (int x = -pad_w_; x < top_x + pad_w_; x++) {
        __m256i res_pixel = _mm256_set1_epi16(0); // u_int である必要がありますが、0 は 0 です
            for (int i = 0; i < kernel_w; i++) {
            int y_offset_in = (bottom_x * (i + (y * stride)))<<4;
            for (int j = 0; j < kernel_h; j++) {
                int x_offset = (j + (x * stride)) << 4;
                // pad 内にあるピクセルをスキップ
                if (pad && ((j + (x * stride)) < 0 || (i + (y * stride)) < 0))
                continue;
                // ピクセルデータをロード
                __m128i data_px = _mm_load_si128((const __m128i *) (bottom_data+ ifm_image_offset +
                block_offset + y_ofset_in + x_ofsset ));
                // 平均化して u16 に変換
                __m256i data_px_u16 = _mm256_cvtepu8_epi16(data_px);
                if (MAX) {
                    res_pixel = _mm256_max_epu16(res_pixel,data_px_u16);
                } else if (AVERAGE) {
                    res_pixel = _mm256_adds_epu16(res_pixel,data_px_u16);
                }
            } // kernel_h
        } // kernl_w

        // 入力データの処理は完了ですが、調整が必要な場合があります。
        int x_offset_out = (x + pad_w_) << 4;
        if (SHOOUL_ADJUST_QUANTIZATION_RANGE) { // 通常、NOP 連結の前
            float factor = layer_param().quantization_param().bottom_range()
            / this->layer_param().quantization_param().top_range();

            __m512 broadcast_factor = _mm512_set1_ps(factor);

            __m512i res_pixel_i32 = _mm512_cvtepu16_epi32(res_pixel);
            __m512 res_pixel_f32 = _mm512_cvtepi32_ps(res_pixel_i32);
            res_pixel_f32 = _mm512_mul_ps(res_pixel_f32,broadcast_factor);

            __m512i data_i32 = _mm512_cvt_roundps_epi32 (res_pixel_f32
                                ,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
            res_pixel= _mm512_cvtusepi32_epi16(data_i32);
        }
        if (AVERAGE) {
            uint8_t kernel_size_u8 = kernel_h_ * kernel_w_;
            __m256i broadcast_kernel_size = _mm256_set1_epi16(kernel_size_u8);

            res_pixel=_mm256_div_epu16(res_pixel,broadcast_kernel_size);
```

```
            }
            // 最後のオフセットを計算して保存
            uint8_t * total_offset =
                  top_data + output_image_offset + layer_offset + block_offset_out +
                        y_ofsset_out + x_ofsset_out ;//+ vect_idx;
            _mm_store_si128((__m128i*) total_offset, _mm256_cvtusepi16_epi8(res_pixel));
            }
      }
}
```

8.4.2.5 ピクセル・シャッフラー

SRGAN トポロジーには、次のような入力を変形するレイヤーが含まれます。

1. フィーチャー・マップの幅と高さは 2 倍になります。
2. 出力フィーチャー・マップ数は 4 で割られます。

出力フィーチャー・マップのそれぞれの 2x2 クワッドは、(c mod K/4) の条件を満たす入力の同じ空間次元からの 4 つのピクセルで占められます。ここで、c は入力チャネルであり、K は入力フィーチャー・マップ数です (参考文献 SRGAN)。

例 8-9 ピクセル・シャッフラーの Caffe スカラーコード

ピクセル・シャッフラーの Caffe スカラーコード

```
void pixel_shuffler(const vector<int>& bottom_shape,
                  const vector<int>& top_shape, const pstype* bottom_data, pstype* top_data)
{
const int N = bottom_shape[0]:
const int bc = bottom_shape[1];
const int bh = bottom_shape[2];
const int bw = bottom_shape[3];
const int tc = top_shape[1];
const int th = top_shape[2];
const int tw = top_shape[3];
const int r = th / bh;
int bottom_ch_size = bw * bh;
int top_ch_size = tw * th;
pstype *cur_channel = NULL;
for(int n = 0; n < bn; n++){
    const pstype *src = bottom_data + n * bc * bottom_ch_size;
    pstype* dst = top_data + n * tc * top_ch_size;
    for(int c = 0; c < tc; c++){
        cur_channel = dst + c * top_ch_size;
        for(int h = 0; h < bh; h++){
            for(int w = 0; w < bw; w++){
                int bottom_offset = h * bw + w;
                int bottom_index = c * bottom_ch_size + bottom_offset;
                int top_index = h * r * tw + w * r;
                cur_channel[top_index] = bottom_data[bottom_index]; // 左上
                bottom_index = (c + tc) * bottom_ch_size + bottom_offset;
                top_index = h * r * tw + w * r + 1;
                cur_channel[top_index] = bottom_data[bottom_index]; // 右上
                bottom_index = (c + 2 * tc) * bottom_ch_size + bottom_offset;
```

```
                    top_index = (h * r + 1) * tw + w * r;
                    cur_channel[top_index] = bottom_data[bottom_index]; //  左下
                    bottom_index = (c + 3 * tc) * bottom_ch_size + bottom_offset;
                    top_index = (h *r + 1) * tw + w * r + 1;
                    cur_channel[top_index] = bottom_data[bottom_index]; //  右下
                    }
                }
            }
        }
}
```

ベクトル化された directConv のメモリーレイアウトにより、ピクセル・シャッフラー・レイヤーを畳み込みに融合するのは容易です。唯一の変更点は、畳み込みの結果を出力の正しい位置に保存することです。

例 8-10 融合ピクセル・シャッフラーの出力オフセット計算

融合ピクセル・シャッフラーの出力オフセット計算

```
// base_ofm -  出力ターゲットの位置  (base_ofm % 16 == 0)
// SubTileX -  クワッドの  X  位置  (0  または  1)
// SubTileY -  クワッドの  Y  位置  (0  または  1)
// ConvOutputX -  畳み込み出力の  X  位置
// ConvOutputX -  畳み込み出力の  Y  位置

int PostPSNoOutMs = (NoOutFMs / 4)
int PostPSOptOfmIndex = (base_ofm % PostPSNoOutMs) / 16;
int QuarterIndex = base_ofm / PostPSNoOutMs;
int SubTileX = QuarterIndex & 0x1;
int SubTileY = (QuarterIndex & 0x2) >> 1;
int PostPSX = ConvOutputX * 2 + SubTileX;
int PostPSY = ConvOutputY * 2 + SubTileY;
size_t offset = (OFM_H * OFM_W * PostPSOptOfmIndex + PostPSY * OFM_W + PostPSX) * 16;
```

8.5 LSTM ネットワーク

LSTM (Long short-term memory) ユニットは、音声やテキスト翻訳などのタスク向けに RNN (Recurrent Neural Network) を作成するために使用されます。LSTM セルの基本計算は、CNN などの直接畳み込みではなく行列乗算 (GEMM) です。

8.5.1 LSTM 組込み融合

LSTM セルは、入力データを入力カーネルで乗算することから開始します。ここで、入力データ (埋め込みとも言う) はすべての辞書ワードに対して 512 要素であり、入力カーネルはオフラインで分かっており変化することがありません。最後に、それぞれの埋め込みベクトルに同じ行列を掛けます。各ベクトルにオフラインでカーネルを乗算して保存し、実行時に GEMM の結果をワード・インデックスで検索してセルのアキュムレーター領域にコピーすることが推奨されます[12]。この最適化により、およそ 20% のパフォーマンス・ブーストがもたらされます。

8.5.2 GEMM 後処理融合

既存の LSTM セルの多くの異形は、活性化関数としてシグモイドおよび双曲線正弦などの超越操作を含みます。

[12] Jacob Devlin、Rabih Zbib、Zhongqiang Huang、Thomas Lamar、Richard M Schwartz、および John Makhoul。2014。統計的機械翻訳向けの高速で堅牢なニューラル・ネットワーク結合モデル。ACL (1)。Citeseer、ページ 1370-1380。

$$sigmoid(x) = \frac{1}{e^{-x} + 1}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

活性化を完全精度スカラーまたは SVML ベースのベクトル化コードとして実装すると遅くなることがあります。代替方法としては、高いパフォーマンスをもたらす近似を使用することです。超越関数を近似する方法の 1 つは、区分的多項式近似を使用することです。

例 8-11 ミニマックス多項式によるシグモイド近似

ミニマックス多項式によるシグモイド近似

```
// Clang 形式 OFF
__declspec( align(64) ) const float sigmoid_poly2_coeffs[3][16] = {
    {
    0.559464f, 0.702775f, 0.869169f, 0.968227f, 0.996341f, 0.999999f, 0.499999f, 0.499973f,
    0.499924f, 0.499791f, 0.499419f, 0.498471f, 0.496119f, 0.491507f, 0.486298f, 0.495135f,
    },
    {
    0.22038f, 0.123901f, 0.042184f, 0.00779019f, 0.000651011f, 1.12481e-7f, 0.250103f, 0.250739f,
    0.251492f, 0.252905f, 0.255751f, 0.260808f, 0.269823f, 0.282225f, 0.292552f, 0.281425f,
    }  ,
    {
    -0.0298035f, -0.0135297f, -0.00347128f, -0.000483042f, -0.0000289636f, -2.57464e-9f, -0.00292674f,
    -0.00680854f, -0.00968539f, -0.0134544f, -0.0188995f, -0.0256562f, -0.0343136f, -0.0426696f, -0.0478004f,
    -0.0443023f,
    },
};
// Clang 形式 ON

inline void sigmoid_poly_2(const __m512& arg, __m512& func)
{
    // 多項式係数をレジスターにロード (一度の操作)
    const __m512 sigmoid_coeff0 = _mm512_load_ps( sigmoid_poly2_coeffs[0] );
    const __m512 sigmoid_coeff1 = _mm512_load_ps( sigmoid_poly2_coeffs[1] );
    const __m512 sigmoid_coeff2 = _mm512_load_ps( sigmoid_poly2_coeffs[2] );

    // 引数の符号を抽出
    const __m512 ps_sign_filter = _mm512_castsi512_ps(_mm512_set1_epi32( 0x7FFFFFFF ));

    __mmask16 signs = _mm512_movepi32_mask(_mm512_castps_si512(arg));
    __m512 abs_arg = _mm512_and_ps(arg, ps_sign_filter);

    // 引数の指数と MSB から近似間隔を計算し、
    // 間隔数を 16 に制限
    const __m512i lut_low = _mm512_set1_epi32( 246 );
    const __m512i lut_high = _mm512_set1_epi32( 261 );
```

```
    __m512i indices = _mm512_srli_epi32(_mm512_castps_si512(abs_arg), 22);
    indices = _mm512_max_epi32(indices, lut_low);
    indices = _mm512_min_epi32(indices, lut_high);

    /*
    * 近似
    */
    __m512 func_p0 = _mm512_permutexvar_ps(indices, sigmoid_coeff0);
    __m512 func_p1 = _mm512_permutexvar_ps(indices, sigmoid_coeff1);
    __m512 func_p2 = _mm512_permutexvar_ps(indices, sigmoid_coeff2);
    func = _mm512_fmadd_ps(abs_arg, func_p2, func_p1);
    func = _mm512_fmadd_ps(abs_arg, func, func_p0);

    // 引数の符号を考慮
    const __m512 ps_ones = _mm512_set1_ps( 1.0 );
    func = _mm512_mask_sub_ps(func, signs, ps_ones, func);
}
```

ミニマックス多項式近似は、レイヤーごとで最も高い精度をもたらしますが、トポロジー (特に NMT) によってはエンドツーエンドの精度が損なわれる可能性があります。その場合、別の方法を検討すべきです。次の近似において、

$$e^x = 2^{x \log_2 e} = 2^{n+y}$$

ここで、

$$n = \text{round}(x \log_2 e)$$

$$y = x \log_2 e - n$$

2^n は scalef 命令で計算されます。

$$2^n = \text{scalef}(x \log_2 e)$$

2^y は次数 2 のテイラー多項式で近似できます。

例 8-12 scalef を使用したシグモイド近似

scalef を使用したシグモイド近似

```
const __m512 ps_ones = _mm512_set1_ps( 1.0 );
const __m512 half = _mm512_set1_ps( 0.5f );
const __m512 minus_log2_e = _mm512_set1_ps( -1.442695f );
const __m512 ln2sq_over_2 = _mm512_set1_ps( 0.240226507f );
const __m512 ln2__ln2sq_over_2 = _mm512_set1_ps( 0.452920674f );
const __m512 one__ln2sq_over_8 = _mm512_set1_ps( 0.713483036f );

inline void sigmoid_scalef(const __m512& arg, __m512& func)
{
    __m512 x = _mm512_fmadd_ps(arg, minus_log2_e, half);
    __m512 y = _mm512_reduce_ps(x, 1);
    __m512 _2y = _mm512_fmadd_ps(_mm512_fmadd_ps(y, ln2sq_over_2, ln2__ln2sq_over_2), y,
                                                                    one__ln2sq_over_8);
    __m512 exp = _mm512_scalef_ps(_2y, x);
    func = _mm512_rcp14_ps(_mm512_add_ps(exp, ps_ones));
}
```

8.5.3 動的バッチサイズ

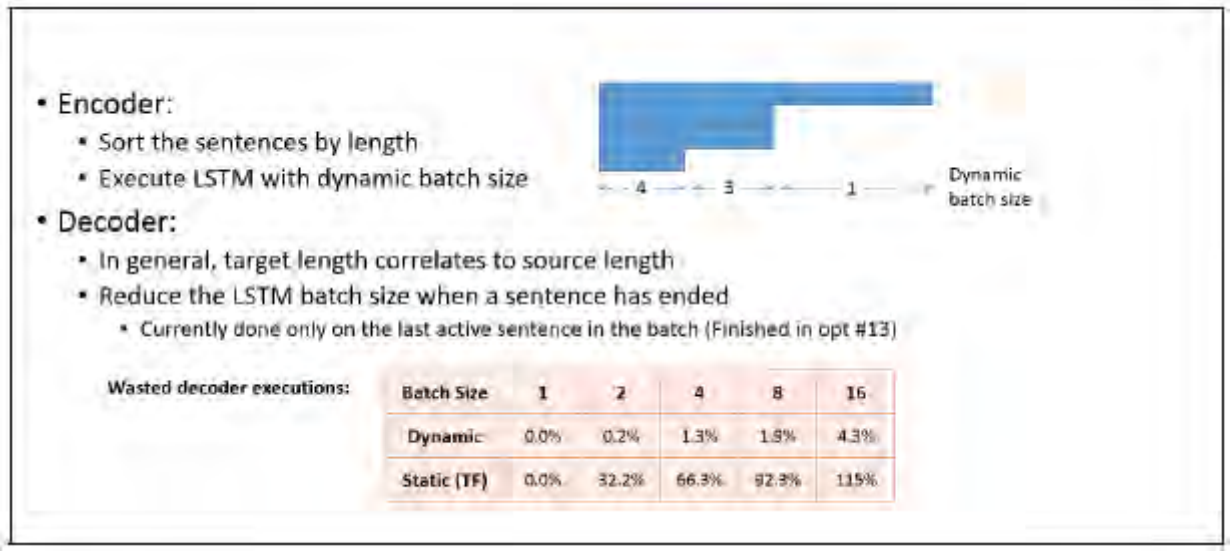


図 8-9 動的バッチサイズ1

注意:

1. NMT は、各反復の計算をアクティブな文の数に合わせることで大幅に向上します。

異なる RNN オブジェクト (例えば文) には、大きく異なる計算量が必要です (例えば、短い文と長い文向けに)。複数のオブジェクトを一括してバッチ処理する場合、このことを考慮して無駄な計算を避けることが重要です。NMT の例では (図 8-9 を参照)、文が長い順に並べられていることを確認すれば、各反復を実際にアクティブな文の数に合わせるのは容易です。

8.5.4 NMT の例: 上位 K を取得するビーム検索デコーダー

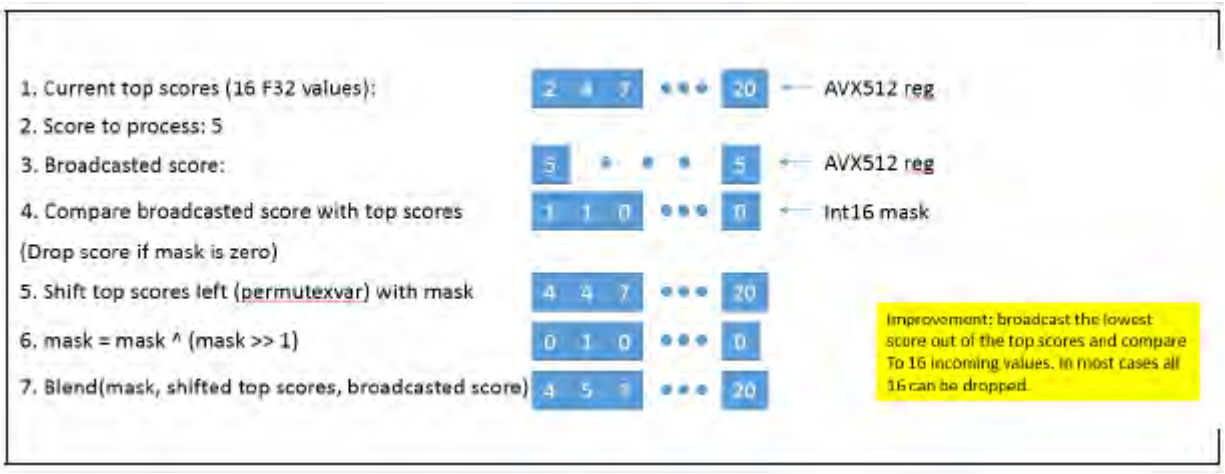


図 8-10 入力から上位 16 の値を検出

https://papers.nips.cc/paper/5346-sequence-tosequence-learning-with-neural-networks.pdf（英語）と https://github.com/tensorflow/nmt (英語) に示すように、ニューラル機械翻訳では、BEAM_WIDTH*VOCAB_SIZE 値のうち、上位 BEAM_WIDTH 算出スコアの検索にかなり時間がかかります。このステップを最適化するため次のアルゴリズムを使用することを推奨します。このプロセスの要点は、新しい値を一度の操作ですべての上位値と比較できることです (図 8-10 の 4 行目を参照)。op によって返されるマスクは、1 のシーケンスとそれに続くゼロ (1* 0 *) で構成される必要があるため、上位スコアをソートしたままにしておくことに注意してください。

例 8-13 上位 K を検出する疑似コード

上位 K を検出する疑似コード

```
// ZMM0 - ベストスコア、-MAX_FLOAT に初期化
// ZMM1 - ベストスコアのインデックス
// ZMM4 - 現在のスコアのインデックス

index = 0
pxor ZMM4
while index < MAX_DATA
    vbroadcastss ZMM2, array[index]
    VPCMPPS K1,ZMM0,ZMM2, _CMP_LT_OQ
    KTESTW K1,K1
    JZ … // K1 == 0 の場合、新しいスコアを配置します
    // K1!=0 の場合
    VPERMPS ZMM0(k1),ZMM0
    VPERMPS ZMM1(k1),ZMM0
    KSHIFT k2,k1,1
    KXOR k3,k2,k1
    VPBLENDMPS k3, ZMM0,ZMM2
    VPBLENDMD k3, ZMM1,ZMM4
    VPADD ZMM4, 1
    add index, 1
```

第 9 章
キャッシュ利用の最適化

この 10 年以上でプロセッサーの速度は向上しています。それに比べると、メモリーのアクセス速度は遅いペースで向上しています。これにより生じた差を埋めるため、次のいずれかの方法でアプリケーションをチューニングすることが重要になりました: (a) 主要なデータアクセスをプロセッサーのキャッシュで行う。(b) メモリー・レイテンシーを効果的にマスクして、ピーク時のメモリー帯域幅をできるだけ多く利用する。

ハードウェア・プリフェッチ機構は、後者を容易にするマイクロアーキテクチャーの拡張機能であり、ソフトウェア・チューニングと組み合わせるとさらに効果的です。必要なデータをプロセッサーのキャッシュからフェッチできるか、メモリー・トラフィックがハードウェア・プリフェッチを効果的に活用できれば、ほとんどのアプリケーションのパフォーマンスが大幅に向上します。

必要なデータを事前にプロセッサーに取り込むには、別にプログラムコードを追加するのが通常の方法ですが、そうしたコードは実装が難しく、またパフォーマンス低下を避けるために特殊な手順を考慮しなければならないこともあります。インテル® ストリーミング SIMD 拡張命令では、各種プリフェッチ命令によってそのような問題点を解決します。

インテル® ストリーミング SIMD 拡張命令では、何種類かの非テンポラルなストア命令がサポートされています。インテル® ストリーミング SIMD 拡張命令 2 では、そのサポート範囲が新たなデータ型にまで広がったほか、32 ビット整数レジスター用の非テンポラルなストア機能も導入されました。

本章では、主に次のテーマに焦点をあてています。

- ハードウェア・プリフェッチ機構、ソフトウェア・プリフェッチ命令、キャッシュ制御命令: アプリケーションにおけるデータキャッシュの動作を制御可能なマイクロアーキテクチャーの機能と命令について説明します。
- ハードウェア・プリフェッチ機構、ソフトウェア・プリフェッチ命令、およびキャッシュ制御命令:
  - これらの命令を使用してメモリーを最適化する手法について説明します。

- キャッシュ・パラメーターを使用してキャッシュ階層を管理します。

9.1 プリフェッチのコーディングに関する一般的なガイドライン

次のガイドラインに従うことで、メモリーシステムにおいて大量のデータ移動が発生するときでもメモリー・トラフィックを軽減し、より効率良くピーク時のメモリーシステムの帯域幅が利用できます。

- 前方もしくは後方への連続したパターンでアクセスするデータのプリフェッチは、ハードウェア・プリフェッチ機能によりさらにパフォーマンスが向上します。
- ハードウェア・プリフェッチのトリガーとなる間隔の半分未満のアクセスストライドを持つパターンでアクセスされるデータを、ハードウェア・プリフェッチャーの機能を利用してプリフェッチします。
- 以下はコンパイラーによる最適化を容易にします。
  - グローバル変数とグローバルポインターの使用を最小限に抑えます。
  - 複雑な制御フローを最小限にします。
  - const 修飾子を使用し、register 修飾子を避けます。
  - データ型の選択には注意し (下記参照)、タイプキャストは避けます。

- ストリップマイニングなど、キャッシュ・ブロッキング手法を導入します。
  - 1 次元配列に対してはストリップマイニングなどのキャッシュ・ブロッキング手法を用い、2 次元配列に対してはループ・ブロッキング手法を用いて、キャッシュのヒット率を上げます。
  - データ・アクセス・パターンに十分な規則性があり、データアクセスの別の順序付け (タイル化など) によって空間的な局所性の改善が見込める場合は、ハードウェア・プリフェッチ機構の使用を検討します。それ以外の場合は、PREFETCHNTA 命令を使用します。

- シングルパス実行とマルチパス実行のバランスをとります。
  - 「シングルパス実行」とは、「非階層化実行」とも呼ばれ、計算パイプライン全体でデータ要素を 1 つ通過させるものです。
  - 「マルチパス実行」とは、「階層化実行」とも呼ばれ、複数のデータ要素から成る 1 つのデータ群を対象にして、パイプラインのステージを 1 段実行してからそのデータ群を次のステージに渡すものです。
  - アルゴリズムがシングルパスのときは PREFETCHNTA 命令を使用し、アルゴリズムがマルチパスのときは PREFETCHT0 命令を使用します。

- メモリーバンクの競合問題を解決します。配列グループ化手法を用いて連続して使用するデータをまとめてグループ化するか、4KB のメモリーページに収まるようにデータを割り当て、できる限りメモリーバンクの競合を避けます。
- キャッシュ管理の問題を解決します。プロセッサーのキャッシュに保持されるテンポラルなデータの汚染をできる限り抑えるため、ストリーミング・ストア命令を使用します。
- ソフトウェア・プリフェッチのスケジューリング間隔を最適化します。
  - 中間の計算処理がメモリーアクセス時間をオーバーラップするように、プリフェッチのスケジューリング間隔を十分に広げます。
  - プリフェッチされたデータがデータキャッシュからのデータに置き換わらないように、プリフェッチのスケジューリング間隔を十分に狭めます。

- ソフトウェア・プリフェッチを連結します。プリフェッチをいくつかうまく並べて、不要なプリフェッチ命令が内部ループの末尾で実行されないようにし、なおかつ次の外部ループの内側でその内部ループにおける最初の数回の反復がプリフェッチされるようにします。
- ソフトウェア・プリフェッチの回数を最小限にします。プリフェッチ命令は、バスサイクル、マシンサイクル、リソースといった観点から見た場合、必ずしも完全に自由に使用できるわけではありません。プリフェッチを多用すると、アプリケーションのパフォーマンスに悪影響を与えることがあります。
- ソフトウェア・プリフェッチ命令の間に演算命令をいくつか挿入します。最高のパフォーマンスを得るには、命令シーケンスの中で複数のプリフェッチ命令を (1 カ所に固めて並べるのではなく)、各プリフェッチ命令は間隔を空けて配置し、その間に演算命令を挿入する必要があります。

9.2 プリフェッチとキャッシュ制御命令

PREFETCH 命令は、プログラマーまたはコンパイラーによって挿入されます。目的のデータが実際に必要になる前に、キャッシュラインにアクセスします。これにより、すでにキャッシュに格納されているデータを処理することで、データアクセスに要するレイテンシーが隠蔽されます。

多くのアルゴリズムでは、必要になるデータに関する情報は事前に判明しています。長いデータパターンでメモリーアクセスが行われるような場合は、ソフトウェア・プリフェッチよりも自動ハードウェア・プリフェッチを優先的に使用する必要があります。

キャッシュ制御命令を使用してデータのキャッシュ方式を制御すると、キャッシュ効率が上がり、かつキャッシュ汚染が最小限に抑えられます。

データ参照パターンは、以下のように分類できます。

- テンポラル: 時間的にそれほど間を置かずにすぐまたデータが使用されます。
- 空間的: 隣接した場所 (同じキャッシュラインなど) のデータが使用されます。
- 非テンポラル: 一度参照された後しばらくデータは再利用されません (例えば 3D グラフィックス・アプリケーションのバーテックス・バッファーのような、一部のマルチメディア・データ・タイプがこれに相当します)。

上記のデータ特性を用いて、以降の説明を進めます。

9.3 プリフェッチ

この節では、ソフトウェアによる PREFETCH 命令の仕組みについて説明します。一般に、ソフトウェア・プリフェッチ命令は、自動ハードウェア・プリフェッチ機構と合わせてアクセスパターンをチューニングする際の補助として使用すべきです。

9.3.1 ソフトウェアによるデータ・プリフェッチ

PREFETCH 命令を使用して、実際にデータが必要になる前にフェッチしておけば、アプリケーション・コードの中でも特に高いパフォーマンスの要求される部分でデータアクセスのレイテンシーが隠蔽されます。PREFETCH 命令を使用しても、ユーザーから見たプログラムの機能は変わりませんが、プログラム・パフォーマンスに影響することがあります。

PREFETCH 命令はハードウェアにヒントを与えるにすぎず、それによって例外やフォルトが発生することはまずありません。PREFETCH 命令を実行すると、非テンポラルなデータまたはテンポラルなデータが、指定するキャッシュレベルにロードされます。データ・アクセス・タイプとキャッシュレベルをヒントとして指定します。実装方式に依存しますが、プリフェッチ命令を実行すると、(指定されたアドレスバイトも含めて) アライメントされた 32 バイト以上がフェッチされ、指定されたキャッシュレベルにデータが移動します。

PREFETCH 命令の使用方法はアーキテクチャーの実装方式によって異なります。パフォーマンスを最大にするため、それぞれの実装方式に合わせてアプリケーションをチューニングします。

注意

PREFETCH 命令は、データがキャッシュに収まらない場合にのみ使用することを推奨します。ソフトウェア・プリフェッチは、管理されているまたはアプリケーチョン内で所有されるメモリーアドレスに対してのみ行われるべきです。プリフェッチするアドレスが物理ページに割り当てられていない場合、非決定論的なパフォーマンス上のペナルティーを被ります。例えば、プリフェッチするアドレスとして NULL ポインター (0L) を指定すると、長いレイテンシーの原因となります。

PREFETCH 命令はハードウェアにヒントを与え、特殊ないくつかの場合を除いて、同命令を実行しても例外やフォルトは発生しません (9.3.3 節「プリフェッチとロード命令」を参照)。ただし、PREFETCH 命令を多用するとメモリー帯域幅が浪費され、その結果、リソース上の制約を受けてパフォーマンスが低下する可能性があります。

それでも PREFETCH 命令を使用すると、キャッシュ汚染が防止され、キャッシュもメモリーも利用効率が上がるため、メモリー・トランザクションのオーバーヘッドを軽減できます。この動作は、メモリーバスに代表される重要なシステムリソースを複数のアプリケーションで共有するようなとき、特に重要となります。9.6.2.1 節「ビデオ・エンコーダー」に記載した例を参照してください。

PREFETCH の主な目的は、メモリー・レイテンシーを他の処理の背後に隠蔽することでアプリケーションのパフォーマンスを改善することにあります。アプリケーションがデータにアクセスするときにどのような方式が採られるかあらかじめ予測できる場合 (例えば、ストライド幅がすでに決まっている配列を使用する場合など)、PREFETCH 命令を使用してパフォーマンスを改善できる可能性が高まります。

PREFETCH 命令は次の場所で使用します。

- 予測可能なメモリー・アクセス・パターン
- 時間を多く消費する最も内側のループ
- データが利用できないと実行パイプラインがストールするような場所

9.3.2 プリフェッチ命令

インテル® ストリーミング SIMD 拡張命令には 4 種類の PREFETCH 命令 (非テンポラルなものが 1 つに、テンポラルなものが 3 つ) があります。それぞれテンポラルな演算と非テンポラルな演算に対応しています。

さらに、PREFETCHW 命令はプロセッサーのより近くにデータをフェッチするヒントを提供し、予測された書き込みによりキャッシュにコピーされたデータを無効化します。

ソフトウェア・プリフェッチ命令は、ソースオペランドで指定されるバイトを含む 64 バイトラインのデータをメモリーからプリフェッチします。ソフトウェア・プリフェッチ命令は、常に 64 バイトのデータをフェッチし、命令はバイト単位で操作するため、データをキャッシュライン間で分割できません。したがって、単独のソフトウェア・プリフェッチ命令で 128 バイトのデータをフェッチすることはできません。

注意

PREFETCH 命令を実行しても、その時点で、この命令によって指定されているキャッシュレベルよりもプロセッサーに近いキャッシュレベルにデータがすでに存在している場合、データは移動しません。

プリフェッチ命令におけるヒントの実装は、マイクロアーキテクチャーごとに異なります。以下の表に概要をまとめます。

表 9-1 Prefetch ヒント命令の実装の詳細

インテル® Core™ プロセッサー、インテル® Core™2 プロセッサー、Intel Atom® プロセッサー				
命令	キャッシュフィルするか?			
	L1	L2		
PrefetchT0	Yes	Yes		
PrefetchT1	No	Yes		
PrefetchT2	No	Yes		
PrefetchNTA	Yes	No		
PrefetchW[1]	Yes	Yes		
Nehalem†/Sandy Bridge†/Ivy Bridge†/Haswell†/Boradwell†/Skylake† マイクロアーキテクチャー・ベースのプロセッサー				
命令	キャッシュフィルするか?			
	L1	L2	L3	
PrefetchT0	Yes	Yes	Yes	
PrefetchT1[2]	No	Yes	Yes	
PrefetchT2[2]	No	Yes	Yes	
PrefetchNTA	Yes	No	Yes[3]	
PrefetchW[4]	Yes	Yes	Yes	
インテル® Xeon® スケーラブル・プロセッサー (L3 非インクルーシブ)				
命令	キャッシュフィルするか?			スヌープフィルターをフィルするか?
	L1	L2	L3	
PrefetchT0	Yes	Yes	No	Yes
PrefetchT1	No	Yes	No	Yes
PrefetchT2	No	Yes	No	Yes
PrefetchNTA	Yes	No	No	Yes[3]
PrefetchW[4]	Yes	Yes	No	Yes

注意:

1. PrefetchW 命令は、Intel Atom® プロセッサーのみで利用でき、インテル® Core™ プロセッサーまたはインテル® Core™2 プロセッサーでは利用できません。

2. それぞれのアーキテクチャーで、PrefetchT1/T2 の実装に違いはありません。
3. PrefetchNTA 命令による L3 キャッシュまたはスヌープフィルターへのフィルは、最も最近使用された位置には配置されず、通常のキャッシュフィルよりも早く置き換えに選択される可能性があります。
4. PrefetchW 命令は、Broadwell†/Skylake† マイクロアーキテクチャーのプロセッサーでのみ利用できます。Haswell† 以前のマイクロアーキテクチャーのプロセッサーでは使用できません。

9.3.3 プリフェッチとロード命令

最近の世代のマイクロアーキテクチャーは、非干渉型の実行とメモリー・パイプラインを備えています。これにより命令にデータやリソースの依存性がない場合、メモリーアクセスとは独立して複数の命令を実行することが可能となります。プログラムやコンパイラーは、ダミーロード命令を使用して PREFETCH 命令の機能を模倣できますが、事前ロードと PREFETCH 命令は完全に同じではありません。PREFETCH 命令の方がプリロードよりもパフォーマンス面で優れています。

それは、PREFETCH 命令が次の特長を備えているためです。

- デスティネーション・レジスターがなく、キャッシュラインをいくつか更新するだけです。
- 通常の命令のリタイアメント処理をストールさせません。
- プログラムの機能面での動作に影響しません。
- キャッシュラインの分割アクセスを行いません。
- LOCK プリフィクスを使用している場合を除いて、例外を発生しません。PREFETCH 命令とともに使用される LOCK プリフィクスは、有効なプリフィクスではありません。
- PREFETCH 命令が原因でフォルトが発生すると、命令の実行は中断されます。

PREFETCH 命令のプリロード命令に対する優位性は、プロセッサーの種類によって異なります。これも将来は変わる可能性があります。PREFETCH 命令でデータをプリフェッチできないことがあります。例えば、以下のようなケースです。

- 古いマイクロアーキテクチャーで、PREFETCH 命令を実行すると DTLB (データ・トランスレーション・ルックアサイド・バッファー) ミスが発生する場合。Nehalem†、Westmere†、Sandy Bridge† およびそれ以降の新しいマイクロアーキテクチャー、インテル® Core™2 プロセッサー、および Intel Atom® プロセッサーでは、DTLB ミスを引き起こす PREFETCH 命令でページ境界を超えたデータをフェッチできます。
- 指定されたアドレスにアクセスするとフォルトまたは例外が発生する場合。
- 1 次キャッシュと 2 次キャッシュの間にあるリクエストバッファーを、メモリー・サブシステムが使い切った場合。
- USWC や UC など、キャッシュできないメモリー領域に対して PREFETCH を行った場合。
- LOCK プリフィクスが使用されている場合。LOCK プリフィクスを使用すると、不正オペコード例外が発生します。

9.4 キャッシュ制御

この節では、キャッシュ制御命令の仕組みについて説明します。

9.4.1 非テンポラルなストア命令

この節では、ストリーミング・ストアの動作について説明し、前の節に示した情報についても一部繰り返して触れています。

インテル® ストリーミング SIMD 拡張命令の MOVNTPS、MOVNTPD、MOVNTQ、MOVNTDQ、MOVNTI、MASKMOVQ、MASKMOVDQU といった命令を実行すると、ストリーミング方式の非テンポラルなストアが行われます。メモリーの特性と順序付けに関しては、これらの命令はライト・コンバイニング (WC) メモリータイプと同様です。

- ライト・コンバイニング (Write combining): 同一キャッシュラインに対する連続書き込みが結合されます。

- ライトコラプス (Write collapsing): 同一バイトに対して連続書き込みを行ったとき、最後に書き込んだ内容のみが反映されます。
- 強制力の弱い順序付け (Weakly ordered): WC ストア同士の間にも、あるいは WC ストアとほかのロード/ストアとの間にも順序付けは設定されません。
- キャッシュが不可能で、ライトアロケート (write-allocating) が実行されない: ストアされたデータはキャッシュのあちこちに書き込まれるため、対応するキャッシュラインに対する所有権読み込みバス要求は生成されません。

9.4.1.1 フェンス操作

ストリーミング・ストアの順序付けは強制力が弱いため、ストアされたデータをプロセッサーからメモリーへ確実にフラッシュするにはフェンス操作が必要になります。フェンス操作を適切に行わないと、プロセッサー内にデータが文字どおり「閉じ込められる」結果になりかねず、そうなると、ほかのプロセッサーやシステム・エージェントから見えなくなります。

WC ストアを実行するには、ソフトウェアでフェンス操作を行いデータの整合性を保証する必要があります。詳細は 9.4.5 節「FENCE 命令」を参照してください。

9.4.1.2 ストリーミング方式の非テンポラルなストア

ストリーミング・ストアでは、以下のようにパフォーマンスが改善できます。

- キャッシュラインに収まる 64 バイトが連続的に書き込まれる場合はストア帯域幅が増加します (所有権読み出しバス要求が必要なく、64 バイトが単一のバス書き込みトランザクションとして結合されるため)。
- キャッシュされたテンポラルなデータのうち、頻繁に使用されるデータの乱れが減少します (プロセッサー・キャッシュのあちこちにキャッシュデータが書き込まれるため)。

一定のメモリー領域にストリーミング・ストアを使用すると、複数のメモリータイプによるクロスエイリアシングが可能になります。例えば、領域の 1 つを、ページ属性テーブル (PAT) やメモリー・タイプ・レンジ・レジスター (MTRRS) を使用してライトバック (WB: write-back) メモリータイプとしてマッピングすることもありますが、ストリーミング・ストアで書き込みを行います。

9.4.1.3 メモリータイプと非テンポラルなストア

メモリータイプが非テンポラルなヒントよりも優先されることがあるため、以下のような点を検討する必要があります。

- プログラマーが、強制力の高い順序付けのキャッシュ不可能メモリー (キャッシュ不可能 (UC) やライトプロテクト (WP) のメモリータイプなど) に非テンポラルなストアを指定した場合、そのストア操作は、キャッシュできないストア操作と同じように動作します。その結果、非テンポラルなヒントは無視され、その領域のメモリータイプは保持されます。
- プログラマーが、強制力の弱いライト・コンバイニング (WC) のキャッシュ不可能メモリータイプを指定した場合、非テンポラルなストア操作とその領域が同じセマンティクスとなるため、競合は発生しません。
- プログラマーが、キャッシュ可能メモリー (ライトバック(WB) かライトスルー (WT) のメモリータイプなど) に非テンポラルなストアを指定した場合、以下の 2 つの結果となる可能性があります。
  - ケース 1: データがキャッシュ階層の中に存在している場合は、その命令によって整合性が保証されます。プロセッサーによっては、別の方法でこれと同じ操作を行うものもあります。次のような方法が考えられます。(a) 当該領域に指定されたメモリー・タイプ・セマンティクスを保存する一方、キャッシュ階層の所定の位置にあるデータを更新します。(b) キャッシュからデータを排出し、WC セマンティクスで新しい非テンポラルなデータをメモリーに書き込みます。

    分割方式や結合方式は、プロセッサーごとに異なります。

1 次キャッシュに含まれているラインの 1 つにストリーミング・ストアがヒットした場合、そのストアデータは 1 次キャッシュ内のデータと結合されます。2 次キャッシュに含まれているラインの 1 つにストリーミング・ストアがヒットした場合は、そのラインとストアされたデータが 2 次キャッシュからシステムメモリーへフラッシュされます。

– ケース 2: データがキャッシュ階層の中に存在していない場合や、デスティネーション領域が WB または WT としてマッピングされている場合は、そのトランザクションの順序付けは強制力が弱くなり、すべて WC メモリー・セマンティクスに支配されることになります。非テンポラルなストアでは、ライトアロケートは行われません。実装方式が異なると、ストア操作のコラプス (折り畳み) とコンバイン (結合) が行われることもあります。

9.4.1.4 ライトコンバイン

一般的に WC セマンティクスでは、グラフィックス・カードなどほかのプロセッサーやほかのシステム・エージェントに対してソフトウェアで整合性を保証する必要があります。生産 (producer) と消費 (consumer) モデルでは、同期とフェンス操作を正しく行う必要があります (9.4.5 節「FENCE 命令」を参照してください)。

フェンス操作を実行すると、ストアされたデータはどのシステム・エージェントからも認識できるようになります。逆に、フェンス操作に失敗すると、書き込まれたキャッシュラインがプロセッサーの中に閉じ込められたままとなり、そのキャッシュラインはほかのエージェントからは見えなくなります。

キャッシュ階層にすでに存在しているデータを更新する非テンポラルなストア操作を実装しているプロセッサーは、デスティネーション領域も WC としてマッピングしなければなりません。例えば WB または WT としてマッピングした場合、プロセッサーによるスペキュレーティブな読み取り操作によってデータがキャッシュに移動する可能性があります。この場合、非テンポラルなストア操作による更新は実行されますが、その後フェンス操作を行ってもデータはプロセッサーからフラッシュされません。

メモリー・タイプ・エイリアシングが発生しているバス上で認識できるメモリータイプは、実装方式によって異なります。例えば、バスに書き込まれたメモリータイプは、プログラムの実行順で最初にキャッシュラインにストアされたメモリータイプが反映されることもあります。ただし、ほかにも可能性はあります。このような機能は予備的なものと見なした方が良いでしょう。特定の方法で実装してしまうと、将来互換性を維持できなくなる可能性があります。

9.4.2 ストリーミング・ストアの利用モデル

ストリーミング・ストアは、主にコヒーレント要求と非コヒーレント要求に使用されます。

9.4.2.1 コヒーレント要求

コヒーレント要求とは、システムメモリーに対する通常のロード操作/ストア操作のことです。マルチプロセッサー環境では、別のプロセッサーのキャッシュラインにヒットすることもあります。コヒーレント要求では、WC メモリータイプ (PAT か MTRR) でマッピングされている通常のストアと同じ方法でストリーミング・ストアを使用でます。複数プロセッサー間でデータの整合性と認識性を保証するため、生産 (producer) と消費 (consumer) モデルでは SFENCE 命令を使用する必要があります。

シングル・プロセッサー・システムでは、CPU が同じメモリー・ロケーションを読み直すと整合性が保証されます (つまり、このメモリー・ロケーションにアクセスしたときにいつも決まった同じ内容が見えます)。マルチプロセッサー (MP) システムにおいても、生産と消費の同期を行うような MP ソフトウェアが採用されていれば、シングル・プロセッサー・システムと同じことが言えます。

9.4.2.2 非コヒーレント要求

非コヒーレント要求は、AGP グラフィックス・カードなど I/O デバイスから発行されます。これらのデバイスは、非コヒーレント要求によりシステムメモリーに対してデータの読み書きを行いますが、その要求はプロセッサー・バスには反映されないため、プロセッサー・キャッシュへの照会は行われません。複数プロセッサー間でデータの整合性と認識性を保証するため、生産 (producer) と消費 (consumer) モデルでは SFENCE 命令を使用する必要があります。この場合、当該プロセッサーが I/O デバイスにデータを書き込んでいるのであれば、ケース 1 (9.4.1.3 節) のように動作するプロセッサーでストリーミング・ストアを使用できます。ただし、その領域 (PAT、MTRR) も WC メモリータイプでマッピングされている場合に限られます。

注意

対象領域を WC としてマッピングできないと、そのキャッシュラインが投機的にプロセッサー・キャッシュに読み取られる場合があります (誤った分岐予測により間違ったパスを通ることになります)。

対象領域が WC としてマッピングされていない場合は、そのストリーミングによってキャッシュのデータが更新されることがあり、その後 SFENCE 命令を実行しても、データがシステムメモリーに書き込まれません。この場合、対象領域を WC として明示的にマッピングすることで、その領域から読み取られたどのデータも、プロセッサーのキャッシュには格納されません。非コヒーレント方式の I/O デバイスからこのメモリー・ロケーションを読み取ると、不正な結果か最新でない結果が返されます。

ケース 2 (9.4.1.3 節) のみを実装しているプロセッサーは、キャッシュされたどのデータもストリーミング・ストアによってメモリーへフラッシュされるため、メモリー領域を WB としてマッピングしなくても、この非コヒーレント・ドメイン中でストリーミング・ストアを使用できます。

9.4.3 ストリーミング・ストア命令の説明

レジスターからメモリーへデータをストアするには、MOVNTQ/MOVNTDQ 命令を使用します。これは、インテル® MMX® テクノロジーまたはインテル® ストリーミング SIMD 拡張命令レジスターに格納されているパックド整数を非テンポラルにストアするものです。この命令は、暗黙的に強制力の弱い順序付けを持ち、ライトアロケートは実行しないため、キャッシュ汚染を最小化します。

MOVNTPS 命令は MOVNTQ 命令に似ており、パックド単精度浮動小数点データを非テンポラルにストアします。これは、インテル® ストリーミング SIMD 拡張命令レジスターから 16 バイト単位でメモリーへデータをストアします。MOVNTQ 命令とは異なり、メモリーアドレスを 16 バイト境界にアライメントする必要があります。そうしないと、一般保護例外エラーが発生します。この命令は、暗黙的に強制力の弱い順序付けを持ち、ライトアロケートは実行しないため、キャッシュ汚染を最小化します。

レジスターから、EDI レジスターで指定したロケーションへデータをストアするには、MASKMOVQ/MASKMOVDQU 命令を使用します。これは、インテル® MMX® テクノロジーまたはインテル® ストリーミング SIMD 拡張命令レジスターに格納されているパックド整数を非テンポラルにバイト・マスク・ストアするものです。マスクレジスターの各バイトの最上位ビットを使用して、バイト単位でソースレジスターの対応するデータを選択的に書き込みます。この命令は、暗黙的に強制力の弱い順序付けを行うため、連続してストア操作を実行しても、メモリーに書き込まれる順番が元のプログラムと一致しないことがあります。また、ライトアロケートが実行されないため、キャッシュ汚染を最小化します。

9.4.4 ストリーミング・ロード命令

インテル® SSE4.1 では、MOVNTDQA 命令が導入されました。メモリーソースが WC タイプである場合、MOVNTDQA 命令は非テンポラルなヒントを使用してメモリーから 16 バイトをロードします。WC メモリータイプでは、このデータをキャッシュせずに、アライメントの合ったキャッシュラインと同等のものを一時内部バッファーにロードすることによって、非テンポラルなヒントを実装します。バッファリング済み WC データの未読み出し部分に対して後続の MOVNTDQA 命令が読み出しを行うと、データが利用可能な場合、16 バイト・データが一時内部バッファーから XMM レジスターに転送されます。

MOVNTDQA 命令を適切に使用すると、WC メモリー領域内のデータをプロセッサーにロードする際、ほかの手段よりも大幅に高いスループットを達成できます。

MOVNTDQA 命令の利用法に関連するアプリケーション・ノートの参照先は、第 1 章に記載されています。また、MOVNTDQA 命令を適切な利用方法に関する詳細情報と条件については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 12 章「Programming with SSE3, SSSE3 and SSE4」と、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の MOVNTDQA 命令のリファレンス・ページを参照してください。

9.4.5 FENCE 命令

FENCE 命令には、SFENCE、LFENCE、MFENCE があります。

9.4.5.1 SFENCE 命令

SFENCE (STORE FENCE) 命令は、プログラム中の SFENCE よりも前に実行されたすべての STORE 命令が、SFENCE の後に実行される STORE 命令より先にメモリーに反映されることを確実にします。SFENCE は、順序付けの弱い結果を生成するルーチン間で、順序付けを確実にする方法の 1 つです。

生産と消費の関係など、特定のデータ共有関係によっては、順序付けの弱いメモリータイプの更新が重要になることがあります。順序付けの弱いメモリー更新では、効率良くデータを管理できますが、生産スレッドが更新したデータを確実に消費スレッドに渡すには、順序付けの注意が必要です。

一般的な使用モデルには、順序付けの弱いストア操作の影響を受けるものがあります。次に例を示します。

- ライブラリー関数。実行結果を書き込むときに、順序付けの弱いメモリー更新を行います。
- コンパイラーが生成したコード。これも、順序付けの弱いメモリー更新によって実行結果を書き込むことで利点がえられます。
- 手動で作成したコード。

順序付けの弱いデータ更新が行われていることをデータ消費スレッドがどの程度認識しているかは、上記のケースごとに異なります。したがって、順序付けの弱いデータを生産するルーチンと、そのデータを消費するルーチンの間で順序付けを確実にする場合は、SFENCE を使用しなければなりません。

9.4.5.2 LFENCE 命令

LFENCE (LOAD FENCE) 命令は、プログラム中の LFENCE よりも前に実行されたすべての LOAD 命令が、LFENCE の後に実行される LOAD 命令より先にメモリーを読み出すことを確実にします。

LFENCE 命令によって、特定の LOAD 命令を他の LOAD 命令から分離することが可能となります。

9.4.5.3 MFENCE 命令

MFENCE (MEMORY FENCE) 命令は、プログラム中の MFENCE よりも前に実行されたすべての LOAD/STORE 命令が、MFENCE の後に実行される LOAD/STORE 命令よりも先にメモリーアクセスを完了することを確実にします。MFENCE によって、特定のメモリー参照命令を他のメモリー参照命令から分離することが可能になります。

ロードフェンスとストアフェンスには互いを拘束する順序付けがないため、LFENCE と SFENCE 命令を組み合わせても、MFENCE と同じ結果は得られません。言い換えると、ロードフェンス命令は前のストア命令の前に実行ができ、ストアフェンス命令は前のロード命令の前に実行ができます。

プロセッサーからのスペキュレーティブ･メモリー参照を確実にするため CLFLUSH 命令 (cache line flush) でフラッシュ操作を行う場合、フラッシュを妨げないように MFENCE 命令を使用する必要があります。詳細は 9.4.6 節「CLFLUSH 命令」を参照してください。

9.4.6 CLFLUSH 命令

CLFLUSH 命令を実行すると、メモリー･ロケーションのバイト･アドレスを含むリニアアドレスに割り当てられたキャッシュラインは、プロセッサーのすべてのキャッシュ階層 (データおよび命令) で無効化されます。無効化されると、そのコヒーレンス･ドメイン全体に無効になったことがいっせい通知されます。キャッシュ階層の任意のレベルにあるキャッシュラインが、メモリーと整合がとれていない場合 (これを「ダーティー」といいます)、無効化される前にメモリーに書き戻されます。また、以下のような特長があります。

- 影響を受けるデータサイズはキャッシュのコヒーレンス･サイズ (CPUID 命令で列挙される) と同じです。大半は 64 バイトです。
- 影響を受けるキャッシュラインを含むページのメモリー属性は、この命令の動作に何の影響も与えません。
- CLFLUSH 命令は、すべての特権レベルで使用できますが、バイト･ロードに関連したアクセス権チェックやフォルトの影響を受けることがあります。

CLFLUSH 命令は、ほかの CLFLUSH 命令と互いに順序付けし、書き込み、ロック付きの読み込み-更新-書き込み命令、フェンス命令、同じキャッシュライン[13] への CLFLUSHOPT を順序付けします。CLFLUSHOPT は、異なるキャッシュラインの順序付けには影響しません。CLFLUSH とそのほかのメモリー･トラフィックのメモリー順序付けの変更に関する詳細については、『インテル® 64 および IA-32 アーキテクチャー･ソフトウェア開発者マニュアル、ボリューム 2A』の第 3 章の CLFLUSH のリファレンス･ページと、『インテル® 64 および IA-32 アーキテクチャー･ソフトウェア開発者マニュアル、ボリューム 3A』の第 8 章「Memory Ordering」を参照してください。

ビデオデータの使用モデルを例にみると、ビデオ･キャプチャー･デバイスが非コヒーレント方式のアクセスを実行してシステムメモリーにキャプチャー･ストリームを直接書き込んでいるとします。この非コヒーレント方式の書き込み操作はプロセッサー･バスにいっせい通知されないため、プロセッサーのキャッシュにある同じロケーションのコピーはフラッシュされません。したがって、プロセッサーがキャプチャー･バッファーを読み直す前に CLFLUSH 命令を使用して、キャプチャー･バッファーの古いコピーもプロセッサー･キャッシュからフラッシュしなければなりません。

例 9-1 に、CFLUSH 命令の疑似コードを示します。

例 9-1 CLFLUSH を使用した疑似コード

```
while (!buffer_ready} {}
sfence
    for(i=0;i<num_cachelines;i+=cacheline_size) {
        clflush (char *)((unsigned int)buffer + i)
    }
    prefnta buffer[0];
    VAR = buffer[0];
```

CLFLUSH を使用してキャッシュラインをフラッシュするスループットの特性にはばらつきがあり、いくつかの要因に強く依存します。一般に、多数のキャッシュラインをフラッシュするため連続して CLFLUSH を使用すると、適切なサイズのバッファー (4KB 未満) のフラッシュよりもキャッシュラインごとに大きなコストがかかります。CLFLUSH のスループットは 1 桁に落ち込むかもしれません。変更状態のキャッシュラインをフラッシュすると、非変更状態のキャッシュラインをフラッシュするよりもコストがかかります。

[13]前述のマニュアルで推奨される CLFLUSH の順序付けでは、ソフトウェアで CLFLUSH の後に MFENCE を追加することが要求されました。CLFLUSH 命令を実装するすべてのプロセッサーでは、上記に示した他の操作との相対的な順序付けを行うため、CLFLUSH の後の MFENCE は必須ではありません。

9.4.7 CLFLUSHOPT 命令

CLFLUSHOPT 命令は、第 6 世代インテル® Core™ プロセッサーで導入されました。CLFLUSH と同様に、CLFLUSHOPT 命令を実行すると、メモリー・ロケーションのバイト・アドレスを含むリニアアドレスに割り当てられたキャッシュラインは、プロセッサーのすべてのキャッシュ階層 (データおよび命令) で無効化されます。

CLFLUSHOPT 命令の実行は、ロック付きの読み込み-変更-書き込み (read-modify-write) 命令、フェンス命令、そして無効化されたキャッシュラインへの書き込みを順序付けします (CLFLUSH と CLFLUSHOPT は、同じキャッシュラインを順序付けします)。無効化されているキャッシュライン以外への書き込みは順序付けされません (CLFLUSH と CLFLUSHOPT は、異なるキャッシュラインの順序付けには影響しません)。ソフトウェアは、CFLUSHOPT と CFLUSHOPT で順序付けされるべき別のキャッシュラインへのストアとの間に SFENCE を挿入できます。

一般に、CLFLUSHOPT のスループットは CLFLUSH よりも高くなっています。これは、上記と 9.4.6 節で説明したように、CLFLUSHOPT はより小さなセットのメモリー・トラフィックに対し自身を順序付けするためです。CLFLUSHOPT のスループットにもばらつきがあります。CLFLUSHOPT を使用して変更 (M) 状態のキャッシュラインをフラッシュすると、非変更状態のキャッシュラインをフラッシュするよりも高いコストを伴います。CLFLUSHOPT は、キャッシュラインがどのようなコヒーレンス状態であっても CLFLISH を超えるパフォーマンスを提供します。(数キロバイトを超えるような) 大きなバッファーをフラッシュする場合、CLFLUSH よりも CLFLUSHOPT の方が適しています。Skylake† マイクロアーキテクチャーでは、シングルスレッドのアプリケーションが CLFLUSHOPT を使用してバッファーをフラッシュすると、CLFLUSH を使用するよりも最大 9 倍のパフォーマンスが得られます。

図 9-1 に、いくつかのバッファーサイズ (1K - 16K バイト) で CLFLUSHOPT と CLFLUSH を実行したパフォーマンス特性の比較を示します。

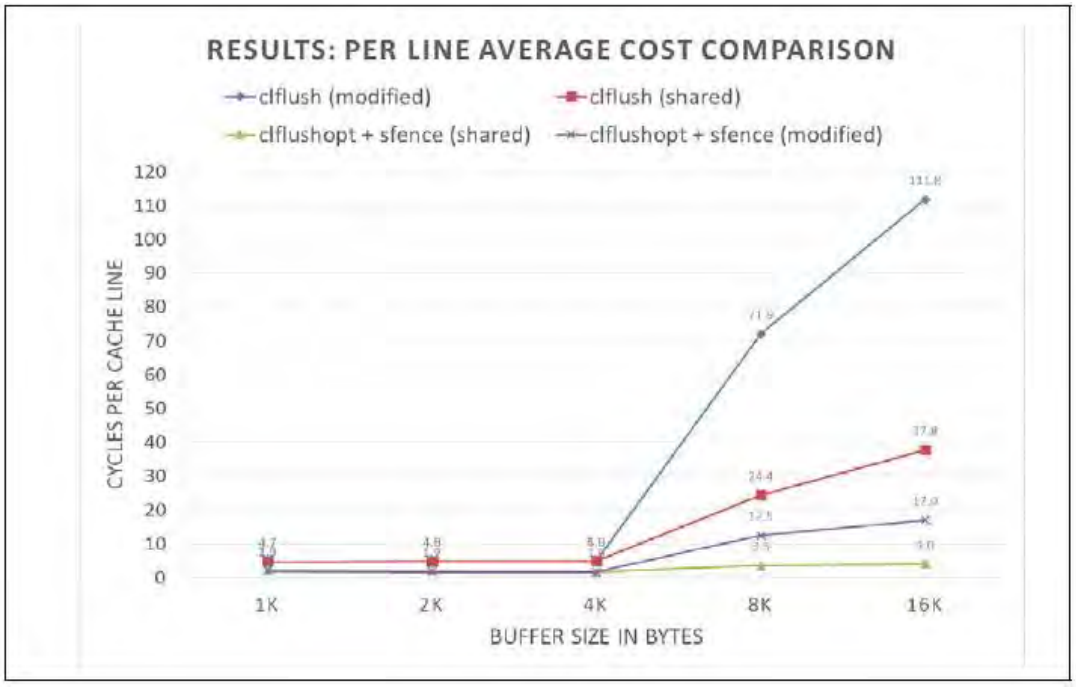


図 9-1 Skylake† マイクロアーキテクチャーにおける CLFLUSHOPT と CLFLUSH

ユーザー/ソース・コーディング規則 17: CLFLUSHOPT が利用できる場合、CLFLUSH よりも CLFLUSHOPT を使用します。CLFLUSHOPT が書き込みの順番をグローバルに反映することを確実にするため最後に SFENCE を挿入します。CLFLUSHOPT が利用できない場合、CLFLUSH を 4KB 未満の小さなチャンクで使用して、大きなバッファーをフラッシュすることを検討してください。

例 9-2 に、CLFLUSH または CLFLUSHOPT を使用してキャッシュラインをフラッシュするアセンブリーのシーケンスを示します。次に対応する C のシーケンスを示します。

```
CLFLUSH:
for (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflush( &pBufferToFlush[ i ] );

CLFLUSHOPT:
_mm_sfence();
for (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflushopt( &pBufferToFlush[ i ] );
_mm_sfence();
```

例 9-2 CLFLUSH または CLFLUSHOPT を使用してキャッシュラインをフラッシュする

CLFLUSH には MFENCE が必要なくなりました	**CLFLUSHOPT と SFENCE**

`xor rcx, rcx`
`mov r9, pBufferToFlush`
`mov rsi, iSizeOfBufferToFlush`
`;; mfence - 以前の方法`
`loop:`
`clflush [r9+rcx]`
`add rcx, 0x40`
`cmp rcx, rsi`
`jl loop`
`;; mfence - 以前の方法`
	`xor rcx, rcx`
	`mov r9, pBufferToFlush`
	`mov rsi, iSizeOfBufferToFlush`
	`sfence`
	`loop:`
	`clflushopt [r9+rcx]`
	`add rcx, 0x40`
	`cmp rcx, rsi`
	`jl loop`
	`sfence`

*メモリーの順序付け規則がアプリケーションにとって重要である場合、メモリー書き込みの順番を保証するため CLFLUSHOPT 命令の実行は SFENCE 命令で保護される必要があります。上図のように、このような解決策は CLFLUSH 命令を使用するよりも高いパフォーマンスを示しますが、これは 2048 バイト以上のバッファーの CLFLUSHOPT と同等です。	

9.5 プリフェッチを使用したメモリーの最適化

最近の世代のインテル® プロセッサーには、2 つのデータ・プリフェッチ機構があります。1 つはソフトウェア制御プリフェッチであり、もう 1 つは自動ハードウェア・プリフェッチです。

9.5.1 ソフトウェア制御プリフェッチ

ソフトウェア制御プリフェッチを行うには、ストリーミング SIMD 拡張命令とともに導入された 4 つの PREFETCH 命令を使用します。これらの命令は、データを含むキャッシュラインを要求されるレベルとモデルのキャッシュ階層に移動する際のヒントとして機能します。ソフトウェア制御プリフェッチは、コードのプリフェッチには適しません。コードが共有されているときにソフトウェア制御プリフェッチを使用すると、マルチプロセッサー・システムのパフォーマンスが著しく低下することがあります。

ソフトウェア・プリフェッチには以下の特長があります。

- ハードウェア・プリフェッチャーがトリガーされない不規則なアクセスパターンに対応します。
- ハードウェア・プリフェッチに比べて、占有するバス帯域幅が低くなります。下記を参照してください。
- ソフトウェア・プリフェッチを行うには新しいコードを追加しなければならず、既存のアプリケーションには利点がありません。

9.5.2 ハードウェア・プリフェッチ

自動ハードウェア・プリフェッチは、事前に発生したデータミスが引き金となって、ユニファイド最終レベルキャッシュにキャッシュラインを取り込みます。自動ハードウェア・プリフェッチは、プリフェッチ・ストリームの前に 2 つのキャッシュラインをプリフェッチします。ハードウェア・プリフェッチャーには次のような特長があります。

- データ・アクセス・パターンには以下のような規則性が求められます。
  - データ・アクセス・パターンに一定した間隔がある場合、アクセス間隔がハードウェア・プリフェッチのトリガーとなる間隔の半分未満のときにハードウェア・プリフェッチは有効になります。
  - アクセス間隔が一定でない場合、連続する 2 つのキャッシュミスの間隔がトリガーしきい値未満であると (間隔の狭いメモリー・トラフィック)、自動ハードウェア・プリフェッチはメモリー・レイテンシーを隠匿できます。
  - 連続する 2 つのキャッシュミスの間隔がトリガーしきい値未満で、かつ 64 バイトに近い場合に、自動ハードウェア・プリフェッチは最も効果的です。

- プリフェッチャーが起動するまで時間がかかったり、処理の終わった後もフェッチが続行する可能性があります。配列が短いと、オーバーヘッドを相殺できない可能性があります。
  - ハードウェア・プリフェッチャーは、データミスが数回発生しないと起動しません。
  - ハードウェア・プリフェッチでは、データ配列の終端を超えるプリフェッチが行われます (本来そのデータは利用されません)。この動作はバス帯域幅を浪費します。さらにこの動作は、次の配列の先頭からフェッチを開始する際のペナルティーとなります。ソフトウェア・プリフェッチでは、このような状況を認識して対処できます。

- 4KB ページ境界をまたぐプリフェッチは行われません。ハードウェア・プリフェッチャーが新たなページからのプリフェッチを開始する前に、プログラムはそのページに対するロードを要求しなければなりません。
- キャッシュミスの間隔がハードウェア・プリフェッチのトリガーしきい値の距離よりも長く (間隔の長いメモリー・トラフィック)、アプリケーションのメモリー・トラフィックの大部分を占める場合、ハードウェア・プリフェッチャーは余分なシステム帯域幅を消費することがあります。
- 既存のアプリケーションで有効であるかどうかは、メモリー・トラフィックのアクセス間隔が長いか、短いかに依存します。テンポラルな局所性に優れた、間隔の短いメモリー・トラフィックが多いアプリケーションでは、自動ハードウェア・プリフェッチから大きなメリットが得られます。
- 一部の状況では、間隔が広くキャッシュミスが多いメモリー・トラフィックを、データ・アクセス・シーケンスの再配列によって、間隔の短いキャッシュミスが集中するようなトラフィックに変換できます。ただし、間隔が短いキャッシュミスでは自動ハードウェア・プリフェッチの利点を活用できません。

9.5.3 ハードウェア・プリフェッチで実効レイテンシーを削減する例

アクセス間隔が一定な循環ポインター追尾シーケンスに対応するデータを配列にする状況について考えてみます (例 9-3 を参照)。メモリーからキャッシュラインをフェッチする際に、自動ハードウェア・プリフェッチを利用して実効レイテンシーを削減するには、循環ポインター追尾を適用する配列へデータを配置するときに、64 バイト間のアクセス間隔や、ハードウェア・プリフェッチのトリガーしきい値の距離を変化させることによって実証できます。

例 9-3 一定間隔の循環ポインター追尾向けの配列の配置

```
register char ** p;
char *next;         // 一定のストライドで続く多重間接ポインターの追加
                    // p = (char **) *p; は次の位置の値をロード
   p = (char **)&pArray;
   for ( i = 0; i < aperture; i += stride) {
      p = (char **)&pArray[i];
         if (i + stride >= g_array_aperture) {
            next = &pArray[0 ];
         }
         else {
            next = &pArray[i + stride];
         }
      *p = next; // 次のノードのアドレスを設定
   }
```

図 9-2 に、いくつかのマイクロアーキテクチャーでの実効レイテンシーの削減を示します。一定間隔のアクセスパターンでは、自動ハードウェア・プリフェッチのメリットは、トリガーしきい値の距離の半分から現れ始めます。最大のメリットは、キャッシュミスの間隔が 64 バイトのときに得られます。

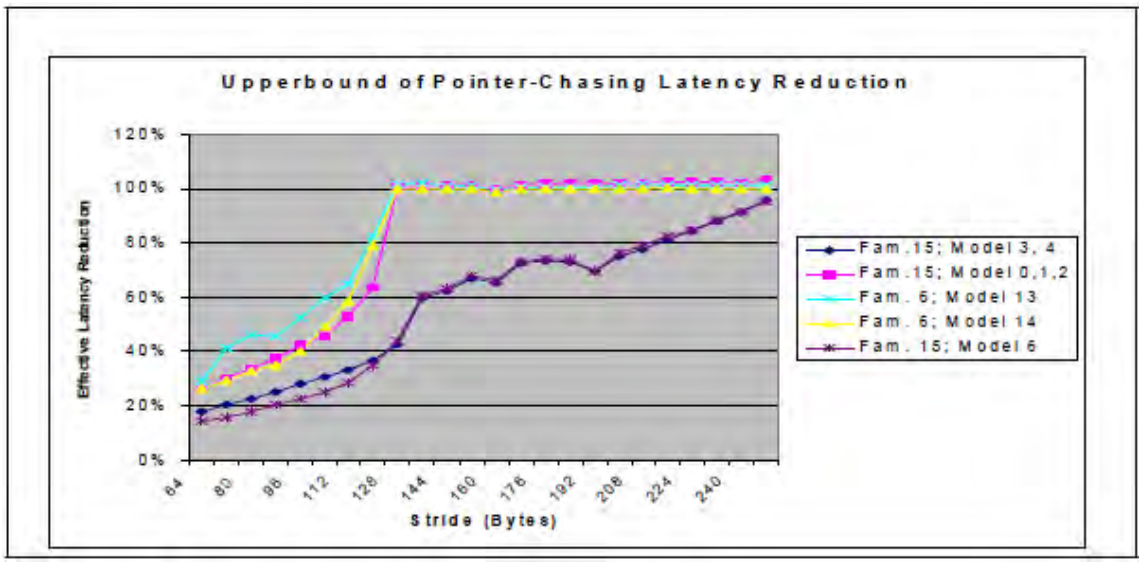


図 9-2 アクセス間隔に応じた実効レイテンシーの削減

9.5.4 ソフトウェア・プリフェッチ命令でレイテンシーを隠蔽する例

PREFETCH 命令を使用してメモリーを最大限に最適化するには、使用するマシンのアーキテクチャーを理解する必要があります。この節では、アーキテクチャーの持つ基本的な意味を、プログラマーが活用できるように簡単な形にして示します。

図 9-3 と図 9-4 に、単純化された 3D ジオメトリー・パイプラインの処理工程の 2 つの例を示します。通常、3D ジオメトリー・パイプラインは、バーテックス・レコードを一度に 1 つずつフェッチし、そのレコードに対して変換処理と照明処理を行います。どちらの図にも、実行パイプラインが 1 つにメモリー・パイプライン (フロントサイド・バス) が 1 つ、計 2 つの独立したパイプラインが描かれています。

プロセッサーの実行機能とメモリーアクセス機能は完全に分離されているため、これら 2 つのパイプラインを同時に処理できます。図 9-3 は、実行とメモリー・パイプラインで、いわゆる「バブル」が発生する様子を表しています。この図では、バーテックス・データにアクセスするロード命令が発行されると、実行ユニットはアイドル状態のままデータが取得されるのを待機しています。一方、メモリーバスは、実行ユニットがバーテックス・データを処理している間、アイドル状態になります。この方法では、分離されたアーキテクチャーを全く活用していません。

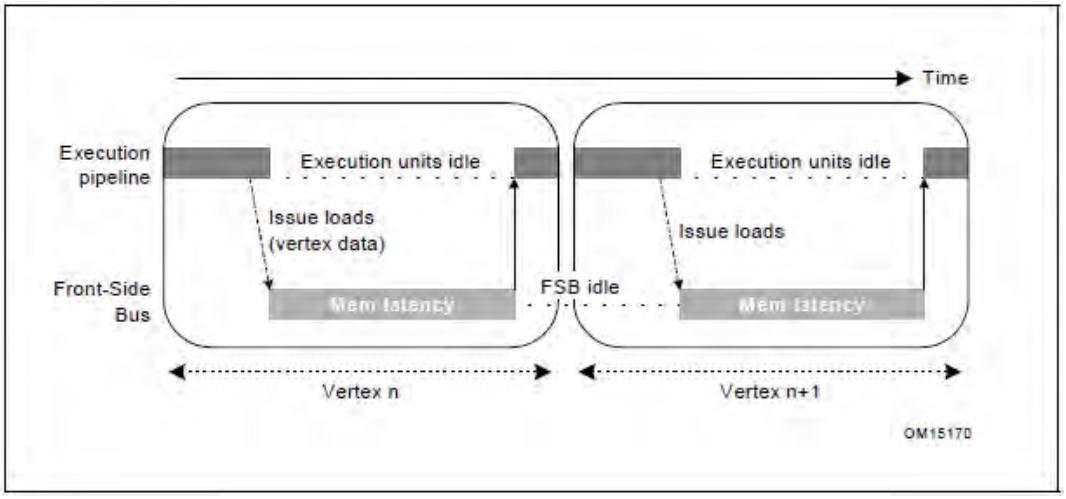


図 9-3 プリフェッチを使用しない場合のメモリー·アクセス·レイテンシーと実行

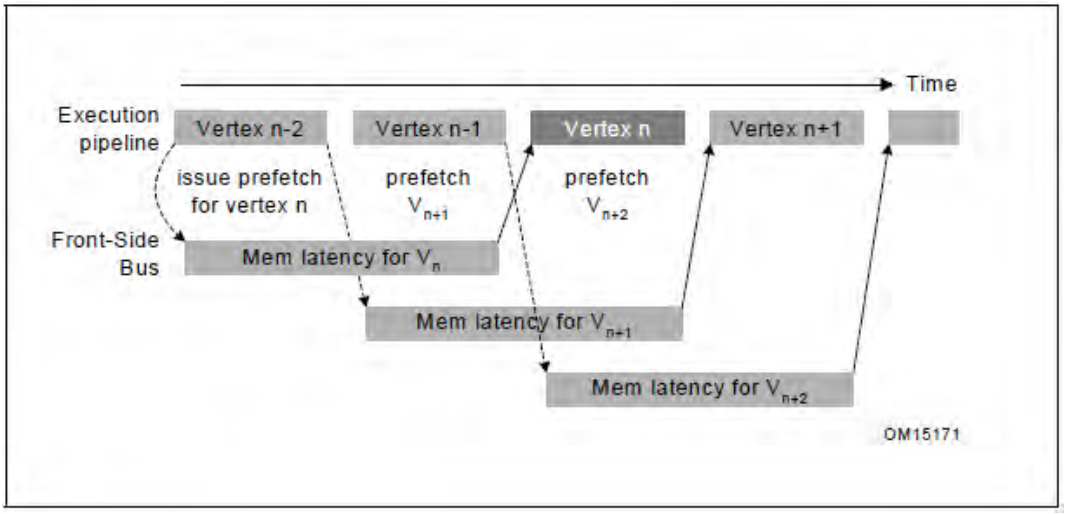


図 9-4 プリフェッチを使用した場合のメモリー·アクセス·レイテンシーと実行

リソースをうまく活用できないとパフォーマンスが低下しますが、複数の PREFETCH 命令を適切に使用すれば、そのようなパフォーマンスの低下を防ぐことができます。図 9-4 では、2 つ先のバーテックス·データに対して PREFETCH 命令を発行しています。これは、1 回の反復で処理するバーテックス·データが 1 つだけであり、反復ごとにデータのキャッシュラインが新たに 1 つずつ必要になる状況を想定しています。そのため、(反復 n、バーテックス Vn) を処理する際に、要求されたデータはキャッシュにすでに取り込まれています。その間、フロントサイド·バスは反復 n + 1、バーテックス Vn + 1 で必要となるデータを転送しています。Vn + 1 のデータと Vn での処理間に依存関係はないため、Vn + 1 のデータアクセスでレイテンシーが生じても、それを Vn の処理の背後に隠蔽できます。このような仕組みにすることで、パイプラインに「バブル」が発生しないため、可能な限り最高のパフォーマンスが得られます。

プリフェッチは、大量に計算を実行する内部ループに有用ですが、計算処理の制約とメモリー帯域幅の制約が同程度の内部ループにも有用です。プリフェッチは、メモリー帯域幅の制約が圧倒的に多いループにはそれほど有用ではありません。

すでにデータが 1 次キャッシュに格納されていると、プリフェッチを実行しても実益がなく、状況によってはパフォーマンスが低下することもあります。これは、余計なマイクロオペレーション (uop) がメモリーアクセスを待機したり、場合によっては、そのマイクロオペレーション (uop) がすべて無用になるためです。この動作は、プラットフォーム固有であり、将来は変わることもあります。

9.5.5 ソフトウェア・プリフェッチを使用する際の確認事項

ソフトウェア・プリフェッチ命令を正しく使用する際に、検討し解決しなければならない問題点を以下に示します。

- ソフトウェア・プリフェッチのスケジューリング間隔を決定します。
- ソフトウェア・プリフェッチを連結します。
- ソフトウェア・プリフェッチの回数を最小限にします。
- ソフトウェア・プリフェッチ命令と演算命令を混在させます。
- キャッシュ・ブロッキング手法 (例えばストリップマイニングなど) を使用します。
- シングルパス実行とマルチパス実行のバランスを考えます。
- メモリーバンクの競合問題を解決します。
- キャッシュ管理の問題を解決します。

上記の問題を、次の各節で説明します。

9.5.6 ソフトウェア・プリフェッチのスケジューリング間隔

コードの中にどのようにプリフェッチ命令を配置するのが理想であるかは、アーキテクチャーに関連する多くの要因によって異なります。そうした要因の中には、プリフェッチするメモリーの容量、キャッシュ・ルックアップ・レイテンシー、システムメモリー・レイテンシー、計算サイクルの予測などがあります。データをプリフェッチする理想的な間隔は、プロセッサーやプラットフォームによって異なります。間隔が狭すぎると、プリフェッチを実行しても、フェッチ操作で発生するレイテンシーを計算処理の背後に隠蔽できません。逆に、あまり先行してプリフェッチを行うと、プリフェッチされたデータが必要になったときにはすでにキャッシュからフラッシュされている可能性があります。

「プリフェッチ距離」は曖昧な用語であるため、ここでは「プリフェッチ・スケジューリング間隔 (PSD)」を使用します。「PSD」の単位は反復回数です。大きなループでは、プリフェッチ・スケジューリング間隔は 1 に設定 (つまり反復を 1 回分だけ先行してプリフェッチ命令を実行) できます。ループの本体が小さいとき (つまり計算処理の少ないループ反復のとき) は、プリフェッチ・スケジューリング間隔を反復 1 回よりも大きくする必要があります。

PSD を求める単純な数式は、数学モデルから導かれます。

例 9-4 に、ループ本体の内側でプリフェッチを使用する場合を示します。この例では、プリフェッチ・スケジューリング間隔は 3 に設定されています。ESI は任意のラインを指すポインターであり、EDX は参照されるデータのアドレスです。xmm1 ~ xmm4 は、計算に使用されるデータです。例 9-4 では、反復 1 回につき 2 つの独立したデータ・キャッシュラインを使用しています。反復 1 回あたりに使用するキャッシュラインの数が 2 つよりも多い場合は、PSD を増やし、2 つよりも少ない場合は PSD を減らす必要があります。

例 9-4 プリフェッチのスケジューリング間隔

```
top_loop:
    prefetchnta [edx + esi + 128*3]
    prefetchnta [edx*4 + esi + 128*3]
    .....
    movaps xmm1, [edx + esi]
    movaps xmm2, [edx*4 + esi]
    movaps xmm3, [edx + esi + 16]
    movaps xmm4, [edx*4 + esi + 16]
    .....
    .....
    add esi, 128
    cmp esi, ecx
    jl top_loop
```

9.5.7 ソフトウェア・プリフェッチの連結

メモリー・レイテンシーのペナルティーを被らなければ、実行パイプラインが最大スループットのときに最高のパフォーマンスが得られます。これは、ループ反復で連続して使用されるデータをプリフェッチすることで達成できます。メモリーのパイプライン処理が途切れると、パイプライン中にバブルが発生します。

パフォーマンスの面からこのような問題を説明するため、ストリップ (帯状) 形式の 3D バーテックスを処理する 3D ジオメトリー・パイプラインを例として使用します。ストリップは、バーテックスの並び順があらかじめ定義されていて、その並び順によって三角形がいくつか連続するリストを含んでいます。プリフェッチ命令がうまく配置されていないと、ストリップ境界上でメモリーのパイプライン処理が途切れますが、その様子は簡単に確認できます。この実行パイプラインは、ストリップごとに、最初の 2 回の反復処理が実行されている間ストールします。その結果、反復を 1 回終えるのに要する平均レイテンシーは 165 (FIX) クロックとなります。

メモリーのパイプライン処理が途切れると、メモリー・パイプラインと実行パイプラインの効率が悪くなります。パイプライン処理の途切れによる影響は、「プリフェッチ連結」と呼ばれる手法を適用して排除できます。この手法を使用すると、メモリーアクセス処理も実行処理も完全にパイプライン化できるため利用率は高まります。

複数のループが入れ子になっていると、内部ループの最後の反復から、対応する外部ループの次の反復までの間に、メモリーのパイプライン処理が途切れる場合があります。プリフェッチ命令を挿入するときは特別な注意を払わなければなりません。そうしないと、内部ループの最初の反復でロード命令がキャッシュミスして、データを取得するまでの間実行パイプラインがストールして、パフォーマンスが低下します。

例 9-5 では、A[ii][0] を含むキャッシュラインがプリフェッチされないため、毎回キャッシュミスが発生します。

これは、配列 A[][] のどの要素もキャッシュに存在していないのを前提としています。メモリーのパイプライン処理がストールしてペナルティーが生じても、それは内部ループを反復するうちに償却されます。しかし、内部ループが短いと、さらに有害となることもあります。さらに、PSD の最後の反復における最後のプリフェッチが無駄になり、マシンのリソースを浪費します。プリフェッチ連結の手法が導入される理由は、メモリーのパイプライン処理の途切れによるパフォーマンス低下を防ぐためです。

例 9-5 プリフェッチ連結の使用例

```
for (ii = 0; ii < 100; ii++) {
   for (jj = 0; jj < 32; jj+=8) {
       prefetch a[ii][jj+8]
       computation a[ii][jj]
   }
}
```

プリフェッチ命令を連結すると、内部ループとそれに対応した外部ループの境界で、実行パイプラインにバブルが発生しないようにできます。単純に、内部ループの最後の反復をアンロールし、それに続く反復で使用されるデータのプリフェッチ・アドレスを指定するだけで、メモリーのパイプライン処理の途切れによるパフォーマンスの低下を完全に排除できます。例 9-6 に、変更したコードを示します。

例 9-6 内部ループの最後の反復を連結しアンロールした例

```
for (ii = 0; ii < 100; ii++) {
   for (jj = 0; jj < 24; jj+=8) { /* N-1 反復 */
       prefetch a[ii][jj+8]
       computation a[ii][jj]
   }
   prefetch a[ii+1][0]
   computation a[ii][jj]/* 最後の反復 */
}
```

データ・プリフェッチの例を示すこのコードは改善されており、計算時間がメモリー・レイテンシーよりも長いと仮定すると、メモリー・アクセス・レイテンシーのペナルティーは外部ループの最初の反復時だけで生じます。

入れ子になったループ計算に入る前に必要となる最初のデータ要素を得るためのプリフェッチ命令を挿入しておけば、外部ループの最初の反復処理が始まるときのペナルティーが完全に排除できるか、軽減できます。このように複雑でないハイレベルのコードを最適化すると、メモリーのパフォーマンスを飛躍的に改善できます。

9.5.8 ソフトウェア・プリフェッチの数を最小化する

プリフェッチ命令は、たとえクロックやメモリー帯域幅をほとんど要しないとしても、バスサイクル、マシンサイクル、リソースといった観点からは、必ずしも完全に自由に使用できるわけではありません。

プリフェッチを多用すると、フロントエンドで命令発行に関連する問題が発生したり、メモリー・サブシステムでリソース競合が発生するため、パフォーマンス低下につながる可能性があります。これは、対象となるループが小さかったり、ループが命令発行の制約を受ける場合などは、問題になることがあります。

プリフェッチの多用による問題を解決する方法の 1 つは、ループをアンロールしたり、ソフトウェアでループのパイプライン処理を実装して、プリフェッチの実行回数を減らすことです。図 9-5 のコード例に示した、プリフェッチとループのアンロールの使用方法を参照してください。この例では、先行して発行されたプリフェッチ命令のアドレスと、同じアドレス指定されるプリフェッチ命令がすべて取り除かれています。特にこの例では、元のループを 1 回アンロールすることによってプリフェッチ命令が 6 つ減っています。また、ほかのすべての反復に含まれている、条件分岐を実行する命令が 9 つ減っています。

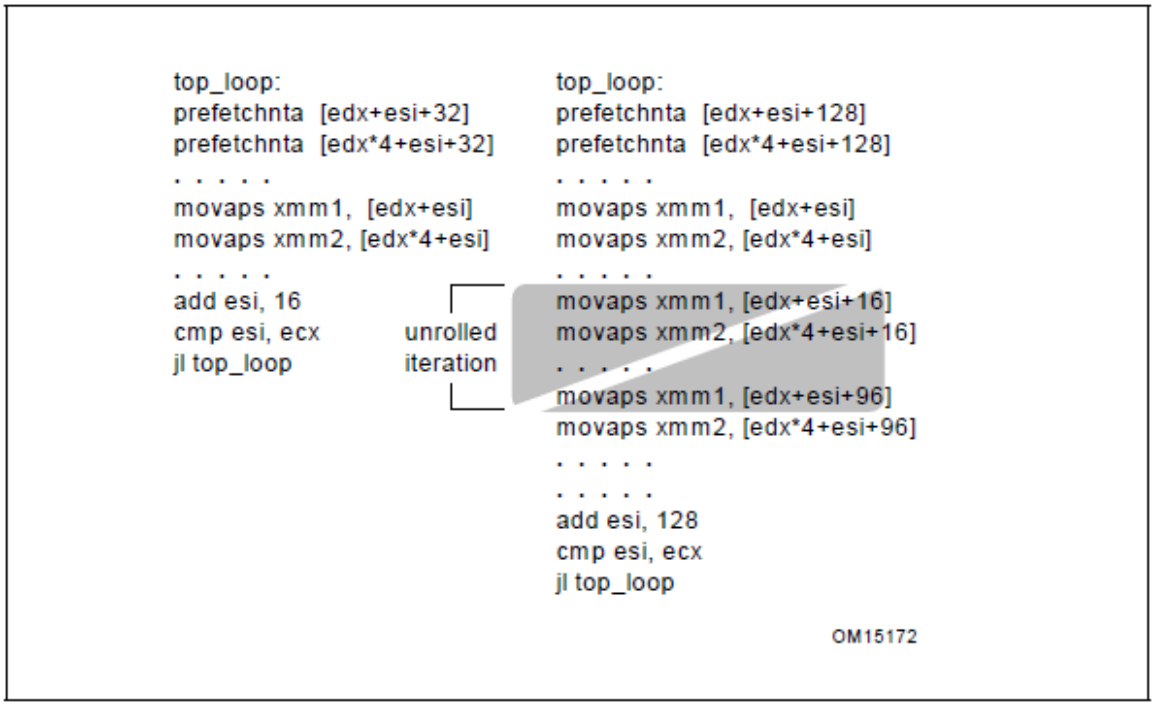


図 9-5 プリフェッチとループのアンロール

図 9-6 に、ソフトウェア・プリフェッチがレイテンシーを隠蔽する様子を図解します。

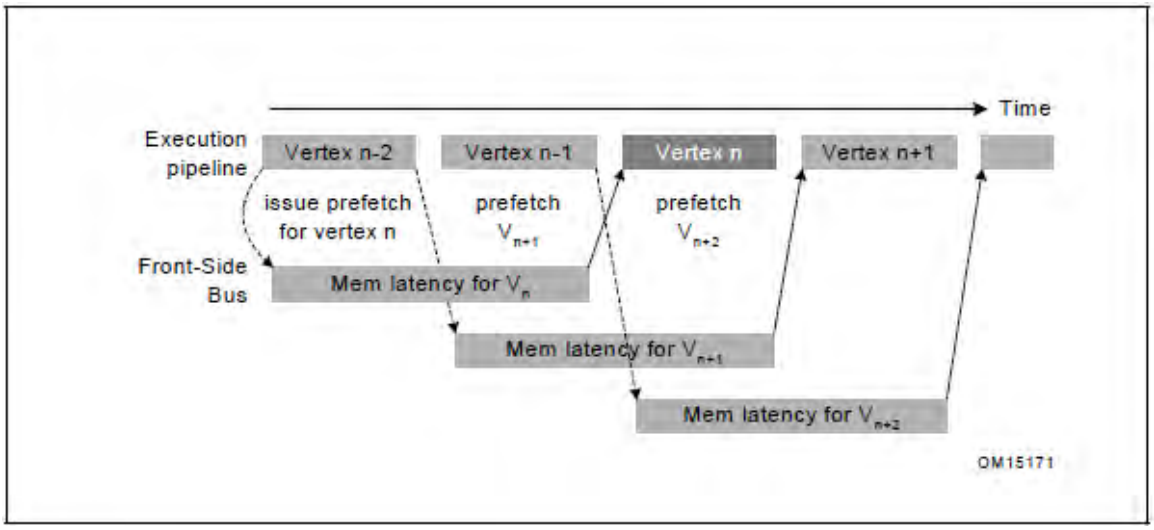


図 9-6 プリフェッチを使用した場合のメモリー・アクセス・レイテンシーと実行

図 9-6 の X 軸はループ 1 回あたりの計算クロック数であり、反復処理同士に依存関係はないものと仮定しています。左側の Y 軸には、ループ 1 回あたりのクロック数で測定した実行時間を示します。右側の Y 軸はバス帯域幅の利用率です。このテストでは、次の各要素を変数として実行しました。

- ロード/ストアストリームの数: ロードストリームもストアストリームもそれぞれ 1 回の反復で 128 バイトのキャッシュライン 1 つにアクセスします。
- ループ 1 回あたりの計算量: この変数を変えるときには、依存関係を持つ算術演算の実行回数を増やす方法を用います。
- ループ 1 回あたりのソフトウェア・プリフェッチの回数: それぞれ、16 バイト、32 バイト、64 バイト、128 バイトとして実行しています。

予想どおりですが、図 9-6 の左側を見ると分かるように、計算量が少なくてメモリーアクセスのレイテンシーが重なり合うほどでないときには、プリフェッチを実行しても効果はなく、処理は基本的にメモリーのパフォーマンスに支配されます。また、余分なプリフェッチがあると、パフォーマンスが向上しないことも分かります。

9.5.9 ソフトウェア・プリフェッチ命令と演算命令を混在させる

ループ本体の始まりか、ループの前に PREFETCH 命令をすべてまとめて配置するのが便利なように思われますが、これは著しいパフォーマンス低下につながる可能性があります。最高のパフォーマンスを引き出すには、複数の PREFETCH 命令を集中して配置のではなく、命令シーケンスの中で、ほかの計算命令の間に分散して配置する必要があります。できるだけ PREFETCH 命令はロード命令から離して配置します。そうすることで、命令レベルでの並列性が高まり、命令リソースがストールする可能性も減少します。また、複数のプリフェッチ命令を分散するとメモリー・アクセス・リソースにかかる負担が減り、その結果、PREFETCH 命令がデータをフェッチしないままリタイアする可能性が減少します。

図 9-7 に、複数の PREFETCH 命令を分散する様子を示します。キャッシュリソースに大きな負担がかかっているコードでは、PREFETCH 命令の位置を並べ替えるだけで驚くほどスピードアップすることがあります。

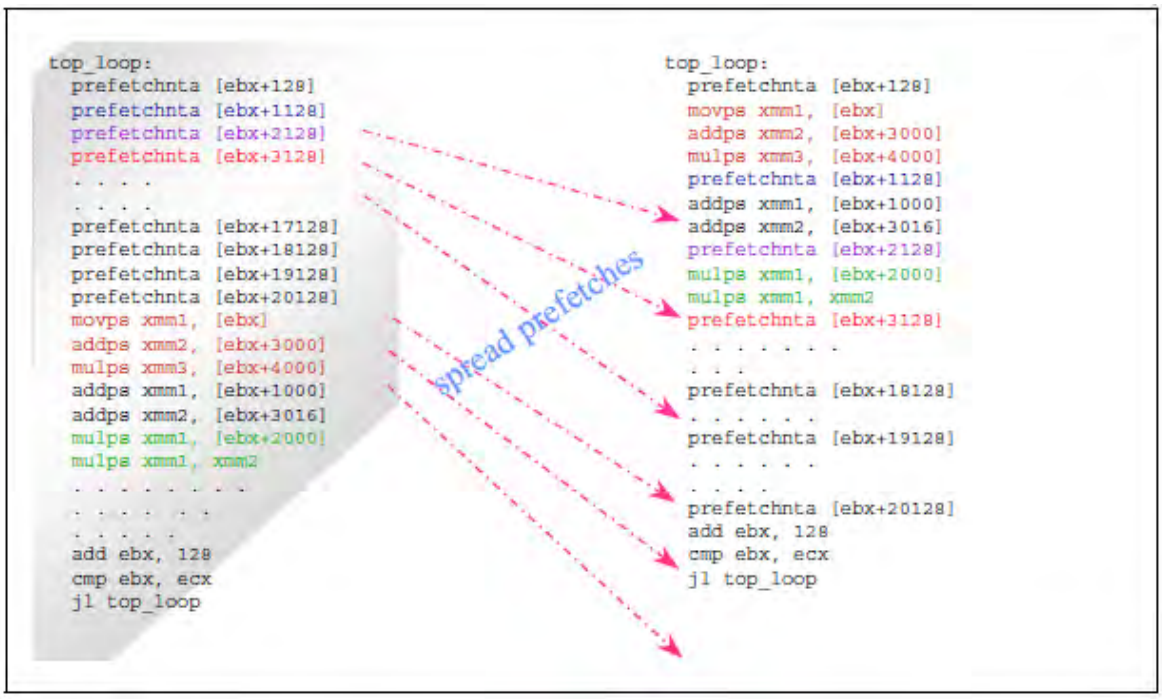


図 9-7 PREFETCH命令の分散

注意

リソースを使いすぎると命令実行がストールする場合があります。それを避けるため、計算命令の間に PREFETCH 命令を分散します。しかし、PREFETCH 命令の分散は、新しいプロセッサー向けに再調整の必要があるかもしれません。

9.5.10 ソフトウェア・プリフェッチとキャッシュ・ブロッキング

時間的局所性を改善し、それによってキャッシュヒット率を上げるには、(ストリップマイニングなどに代表される) キャッシュ・ブロッキング手法を使用します。ストリップマイニングとは、メモリーの高次元配列の時間的局所性を最適化する手法です。プログラムで 2 次元配列が使用されている場合、ループ・ブロッキング手法を利用してメモリーのパフォーマンスを改善できます。ループ・ブロッキング手法は、2 次元的である点以外はストリップマイニングとほぼ同じです。

1 つのループを繰り返すときに再利用できる大きなデータセットを使用するアプリケーションであれば、ストリップマイニングの効果があります。キャッシュよりも大きなデータセットは、キャッシュに格納できるように小さないくつかのグループに分けて処理されます。こうすることで、より長い期間にわたってテンポラルなデータをキャッシュに保持できるため、バス・トラフィックが減少します。

ストリップマイニングされたコードに PREFETCH 命令をどのように適用するかは、データセットのサイズと時間的局所性 (データ特性) によります。図 9-8 に、複数のデータが時間的に隣接している場合と、していない場合について単純化して示します。

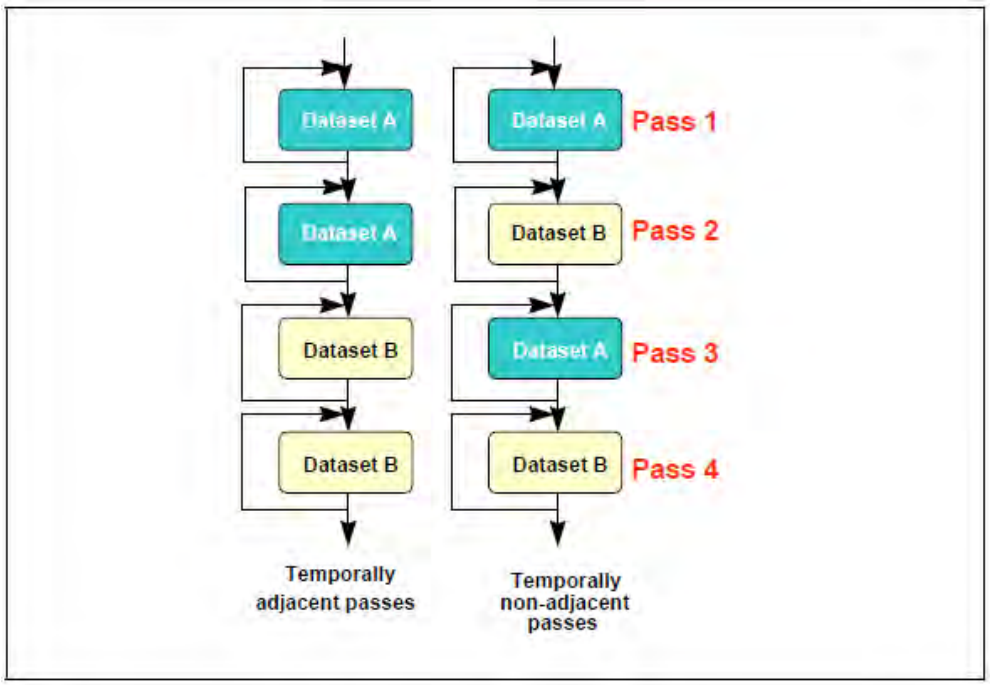


図 9-8 キャッシュ･ブロッキング – 複数のパスが時間的に隣接している場合としていない場合

「時間的に隣接している場合」は、隣接している 2 つのパスのうち、後のパスでも同じデータが使われ、しかもそのデータは 2 次キャッシュにすでに格納されています。プリフェッチに関連する問題を別にすれば、こちらが望ましいでしょう。「時間的に隣接していない場合」は、m 番目のパスで使われたデータが (m+1) 番目のパスによって強制的に追い出されるため、データをフェッチし直して 1 次キャッシュに格納し直す必要があります。また、後のパスがそのデータを再利用するような場合は、2 次キャッシュにも格納する必要があります。データセットが両方とも 2 次キャッシュに入れば、パス 3、パス 4 でのロード操作はほとんど負担になりません。

図 9-9 は、プリフェッチ命令とストリップマイニングを適用するとパフォーマンスが向上することを示したものです。

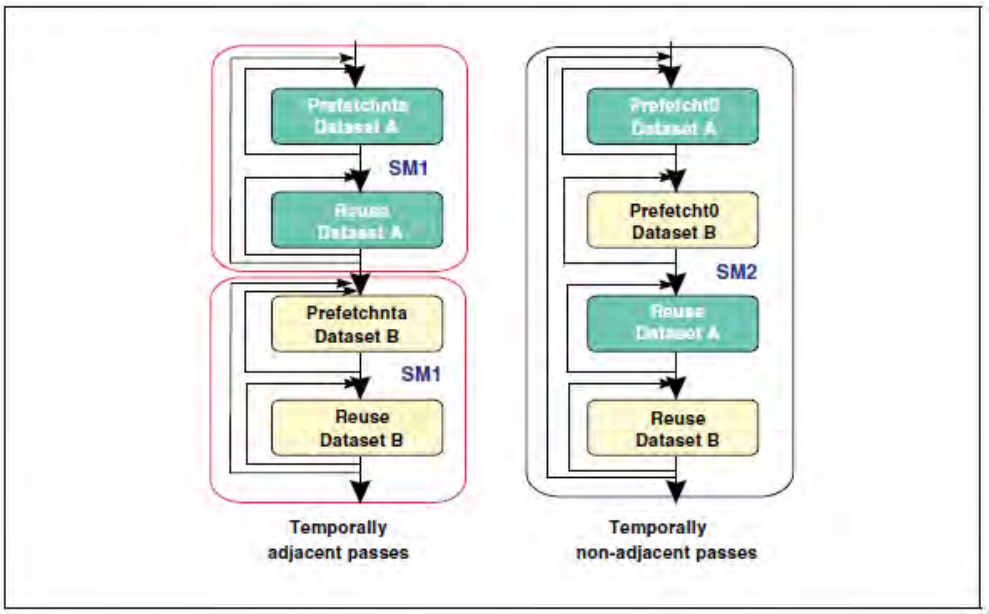


図 9-9 複数のパスループが時間的に隣接している場合と時間的に隣接していない場合でのプリフェッチとストリップマイニングの例

図 9-9 の左側は、インテル® Pentium® 4 プロセッサーでの PREFETCHNTA 命令の使用例を示したものです。この例では、プリフェッチされたデータは、2 次キャッシュのウェイのうち選択されたいくつかのウェイにのみ格納され、2 次キャッシュの汚染は最小限に抑えられます。図中の「SM1」は、2 次キャッシュのウェイのうちの 1 つがストリップマイニングで処理されることを意味します。上位キャッシュでのキャッシュ汚染を最小限に抑えるため、実行パスの処理全体を通じて一度しか利用されないデータには PREFETCHNTA 命令を使用します。これは、プリフェッチ命令が十分に先行して発行されたと仮定すれば、読み出しアクセス命令が発行されるとすぐに効果が表れます。

図 9-9 の右側では、ワークロードの大きさが L1 キャッシュには大きすぎます。したがって、データのプリフェッチには PREFETCHT0 命令を使用します。これにより、パス 1、パス 2 でのメモリー参照に要するレイテンシーが隠蔽され、また、そのデータのコピーが 2 次キャッシュに保存されるため、パス 3、パス 4 でのメモリー・トラフィックとレイテンシーが減少します。このレイテンシーをさらに減らしたいときは、パス 3、パス 4 でメモリー参照を行う前にいくつか PREFETCHNTA 命令を使用することを検討してください。

例 9-7 を用いて、3D ジオメトリー・エンジンのデータ・アクセス・パターンを考えてみます。最初に、ストリップマイニングを使用しない例を示し、その後、ストリップマイニングを適用した例を示します。

ストリップマイニングを使用しない場合は、4 つのバーテックス x、y、z 座標をすべて、2 番目のパス、すなわち照明計算ループのときに、メモリーからフェッチし直さなければなりません。そのため、照明計算ループで帯域幅が浪費されるのはもちろんですが、変換ループの最中にフェッチされたいくつかのキャッシュラインの利用率も低下します。

例 9-7 3D ジオメトリー・エンジンのデータアクセス (ストリップマイニングは未使用)

```
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data // v =[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
TRANSFORMATION コード // バーテックス x,y,z,tu,tv のみを使用
    nvtx+=4
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data // v =[x,y,z,nx,ny,nz,tu,tv]
    // x,y,z の再フェッチ
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    照明ベクトルを計算 // x,y,z のみ使用
    LOCAL 照明コード // nx,ny,nz のみ使用
    nvtx+=4
}
```

次に、ストリップマイニングがループに適用された例 9-8 のコードについて考えてみます。

例 9-8 3D ジオメトリー・エンジンのデータアクセス (ストリップマイニングを使用)

```
while (nstrip < NUM_STRIP) {
/* データを L2 キャッシュの 1 つのウェイに収めるストリップマイニング */
    while (nvtx < MAX_NUM_VTX_PER_STRIP) {
        prefetchnta vertexi data // v=[x,y,z,nx,ny,nz,tu,tv]
        prefetchnta vertexi+1 data
        prefetchnta vertexi+2 data
        prefetchnta vertexi+3 data
        TRANSFORMATION コード
           nvtx+=4
    }
    while (nvtx < MAX_NUM_VTX_PER_STRIP) {
    /* x y z 座標は L2 キャッシュ内にあり、プリフェッチを必要としない */
        照明ベクトルを計算
        POINT 照明のコード
        nvtx+=4
    }
}
```

ストリップマイニングを使用すると、ストリップマイニングが適用されたループ変換の処理中ずっとすべてのバーテックス・データをキャッシュ (例えば、2 次キャッシュのウェイの 1 つ) に保存でき、照明計算ループの中でそのバーテックス・データを再利用できるようになります。キャッシュにデータを維持しておくと、バス・トラフィックもプリフェッチの使用回数も減少します。

ストリップマイニングとプリフェッチ命令を組み合わせた基本的な使用モデルの手順を表 9-2 にまとめています。手順は以下のとおりです。

- データセットが 2 次キャッシュに収まるようにループを分割するストリップマイニングを適用します。
- データが一度しか使われないか、データセットが 32KB (2 次キャッシュのウェイの 1 つ) に収まる場合は、PREFETCHNTA 命令を使用します。データセットが 32KB を超える場合は、PREFETCHT0 命令を使用します。

上記の手順はプラットフォームによって異なり、実装例の 1 つでしかありません。特定のプラットフォーム上で特定のアプリケーションを実行する場合に最高のパフォーマンスを引き出すには、NUM_STRIP および MAX_NUM_VX_PER_STRIP 変数のヒューリスティックを決定する必要があります。

表 9-2 ストリップ・マイニング・コードへのソフトウェア・プリフェッチの組込み

読み出しが 1 回の配列参照	**読み出しが複数回の配列参照**	
	隣接パス	**非隣接パス**
Prefetchnta	Prefetch0, SM1	Prefetch0, SM1 (2 次キャッシュ汚染)
1 方向への排出。汚染を最小限に抑制	各配列の最初のパスでメモリー・アクセス・コストを消費。以降のパスで最初のパスのコストを吸収	各ストリップの最初のパスでメモリー・アクセス・コストを消費。以降のパスで最初のパスのコストを吸収

9.5.11 ハードウェア・プリフェッチとキャッシュ・ブロッキング

読み出しが複数回のメモリー参照の最初のパスと、読み出しが 1 回のメモリー参照の一部では、自動ハードウェア・プリフェッチ機構に合わせてデータ・アクセス・パターンをチューニングすると、メモリー・アクセス・コストを最小限に抑えられます。読み出しが 1 回のメモリー参照が行われる状況は、列方向優先で読み出し、行方向優先で書き込む (またはその反対)、行列またはイメージの転置よって例証できます。

例 9-9 に、典型的な行列/イメージ転置の問題である、ネストされたデータ移動ループを示します。配列の次元が大きい場合、データセットの要素が最終レベルキャッシュに収まらないだけでなく、広い間隔でキャッシュミスが発生

します。次元が 2 の累乗である場合は、ウェイ・アソシアティビティーの有限数に基づくエイリアシング条件 (3.6.7 節「キャッシュの容量制限とエイリアシング」を参照) によって、キャッシュ排出の可能性が増加します。

例 9-9 ハードウェア・プリフェッチを使用した、読み出しが 1 回のメモリー・トラフィックの向上

```
a) 最適化されていないイメージ転置
// dest と src は 2 次元配列
for(i = 0;i < NUMCOLS; i++) {
// 内部ループが 1 つの列を読み出す
    for(j = 0; j < NUMROWS; j++) {
        // 個々の読み出し参照は、ストライドの大きなキャッシュミスを引き起こす
        dest[i*NUMROWS +j] = src[j*NUMROWS + i];
    }
}
b)
// tilewidth = L2SizeInBytes/2/TileHeight/Sizeof(element)
for(i = 0; i < NUMCOLS; i += tilewidth) {
    for(j = 0; j < NUMROWS; j++) {
        // 内部ループで同じ行の複数の要素にアクセス
        // ハードウェア・プリフェッチしやすいパターンでアクセスし、ヒット率が改善
        for(k = 0; k < tilewidth; k++)
        dest[j+ (i+k)* NUMROWS] = src[i+k+ j* NUMROWS];
    }
}
```

例 9-9 (b) では、ハードウェア・プリフェッチに最適なタイルサイズとタイル幅を選択するタイル化手法を適用しています。タイル化によって、最終レベルキャッシュに収まるように、2 つのタイルのサイズを選択できます。メモリー読み出し参照で各タイルの幅を最大にすると、コードが実際にリニアアドレスを参照する前に、ハードウェア・プリフェッチはバス要求を開始してキャッシュラインを読み出せます。

9.5.12 シングルパス実行とマルチパス実行の比較

1 つのアルゴリズムで、シングルパス実行かマルチパス実行を選択できます。

- 「シングルパス実行」とは、「非階層化実行」とも呼ばれ、計算パイプライン全体でデータ要素を 1 つ通過させるものです。
- 「マルチパス実行」とは、「階層化実行」とも呼ばれ、複数のデータ要素から成る 1 つのデータ群を対象にして、パイプラインのステージを 1 段実行してからそのデータ群を次のステージに渡すものです。

シングルパス実行とマルチパス実行のどちらが良いかは、アルゴリズムの実装方式や、それら両者をどのように使用するかによって異なります。図 9-10 を参照してください。

汎用 API を実装している場合は、マルチパス実行が使いやすいことが多いでしょう。この場合、いくつかのコードパスのうちのどれが選択できるかは、対象となるアプリケーションが選ぶパターン構成要素間の組み合わせによって異なります。例えば、3D グラフィックスの場合は、使用されるバーテックス要素の種類や光源の個数および種類がパターン構成要素に含まれます。

このように、考えられる並べ替えの範囲が広いと、コードサイズや妥当性検証の点から見ると、シングルパス実行は複雑になります。これは、並べ替えの処理ごとにコードシーケンスが 1 つずつ必要になるためです。例えば、A、B、C、D のパターン構成要素を持つオブジェクトがある場合、このオブジェクトは、パターン構成要素の一部 (例えば A、B、D) を使用可能にできます。このステージではコードパスが 1 つ使われ、一方、使用可能になったパターン構成要素の別の組み合わせには別のコードパスが 1 つ与えられます。パターン構成要素を選択するためいくつかの条件節がパイプラインの各ステージ内に実装されている場合、そのステージごとに別々のパスとして実行する方が適しています。ストリップマイニングを使用すると、各ステージで処理されるバーテックスの数 (例えば、バッチサイズ) を選択して、処

理単位となるデータ群がどのパスを通過しても必ずプロセッサー・キャッシュに留めておけます。バーテックス群を現在のステージから次のステージへ、あるいは現在のパスから次のパスへ渡すときは、中間キャッシュバッファーを使用します。

任意の時点で使用できるパターン構成要素の数に制約のあるアプリケーションには、シングルパス実行が向いています。シングルパス実行を用いると、マルチパスエンジンで発生する可能性のあるデータコピーの量を減らすことができます。図 9-10 を参照してください。

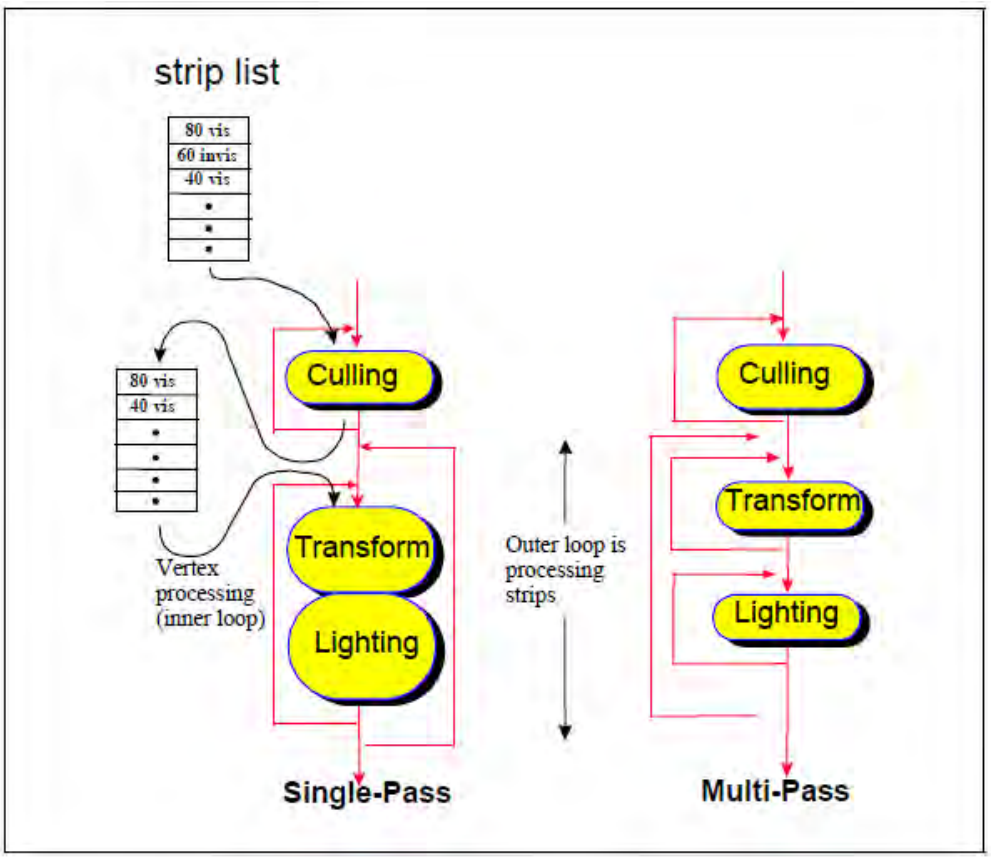


図 9-10 シングルパスとマルチパスでの 3D ジオメトリー・エンジンの比較

シングルパスとマルチパスのいずれかを選択することによって、パフォーマンスにどのような影響が出るかが異なります。例えば、マルチパスのパイプラインの場合、何段かのステージが入力または出力の帯域幅に制約されていると、それによるパフォーマンス低下の多くは全体的な実行時間として表れます。それに対し、シングルパスでは、帯域幅に制約があっても、それによるパフォーマンス低下は、計算量の多いほかの何段かのステージ全体に分散/吸収できます。また、シングルパスまたはマルチパスのどちらを使用するかは、どのプリフェッチ・ヒントを選択すべきかいう問題にも影響します。

9.6 非テンポラルなストアを使用したメモリーの最適化

キャッシュに保存するデータの管理には非テンポラルなストアも使用できます。非テンポラルなストアの用途には、次のものがあります。

- キャッシュ階層を乱すことなく、多くの書き込み処理を一体化します。
- いくつかのデータ構造のうちどれをキャッシュに残し、どれを残さないかを管理します。

上記の使用モデルの実装については、以降の節で詳しく説明します。

9.6.1 非テンポラルなストアとソフトウェアによるライトコンバイン

次のようなデータをストアする場合は、非テンポラルなストアを使用します。

- 非テンポラルな 1 回書き込み (ライトワンス) 用のデータ
- 大きすぎてキャッシュ・スラッシングの原因となるようなデータ

非テンポラルなストア命令を実行してもキャッシュラインの割り当ては行われません。つまり、非テンポラルなストア命令はライトアロケートではないため、有用なデータ帯域幅と競合してもキャッシュは汚染されず、ダーティーなライトバックも発生しません。

非テンポラルなストア命令を使用しないと、ダーティーなライトバックに起因するキャッシュ・スラッシングが発生する際に、バス帯域幅が悪影響を受けます。

インテル® ストリーミング SIMD 拡張命令では、複数の非テンポラルなストア命令が複数のライトバック・メモリー領域またはライト・コンバイニング・メモリー領域に書き込みを行っても、それらのストア命令の実行順は強制力が弱く、プロセッサーのライト・コンバイニング・バッファーの内部で結合され、1 つのライン・バースト・トランザクションとしてメモリーに書き込まれます。最高のパフォーマンスを引き出すためには、非テンポラルなストア命令を使用する一方で、そのキャッシュライン境界にデータをアライメントし、さらにそれらのデータを、同じキャッシュライン・サイズで連続して書き込むようにします。プログラミング上の制約から連続書き込みが禁止されている場合、ソフトウェア・ライト・コンバイニング (SWWC) バッファーを使用して、ライン・バースト・トランザクションを有効にします。

アプリケーションで小さな SWWC バッファーをいくつか (バッファーごとにキャッシュラインを 1 つ) 宣言して、明示的なライト・コンバイニング処理を有効にできます。このプログラムでは、データは非テンポラルなメモリーに直ちに書き込まれるのではなく、SWWC バッファーに書き込まれた後バッファーの内部で結合されます。プログラムは SWWC バッファーが一杯になると、非テンポラルなストア命令を使って 1 キャッシュラインのバッファーを書き出すだけです。このような SWWC を用いた方法では、一時的な読み出し/書き込み処理を実行する明示的な命令が必要になりますが、フロントサイド・バス上のトランザクションは、複数のパーシャル・トランザクションではなくライン・トランザクションが発生します。この方法を取り入れると、アプリケーションのパフォーマンスはかなり向上します。これらの SWWC バッファーを 2 次キャッシュに保持すると、そのプログラム全体で何度でも使用できます。

9.6.2 キャッシュ管理

PREFETCH や STORE などのストリーミング命令を使用して、データを管理したり、プロセッサーのキャッシュに格納されているテンポラルなデータの乱れを最小限にできます。

また、プロセッサー向けに、インテル® ストリーミング SIMD 拡張命令に対応した C++ 言語機能をサポートするインテル® C++ コンパイラーを利用できます。インテル® ストリーミング SIMD 拡張命令およびインテル® MMX® 命令には、キャッシュの最適化を可能にする組込み関数が用意されています。インテル® コンパイラーがサポートする組込み関数には、_mm_prefetch、_mm_stream、_mm_load、_mm_sfence などがあります。詳細は、『インテル® C/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』を参照してください。

次に、単純な 8 バイトのメモリーコピーを初め、ビデオ・エンコーダーやビデオデコーダーにおけるプリフェッチ命令の使用例を示します。プリフェッチ命令により効率良くキャッシュを管理すれば、高いパフォーマンスが引き出せることを理解できます。

9.6.2.1 ビデオ・エンコーダー

ビデオ・エンコーダーは、エンコード処理中に使用するデータの一部をプロセッサーの 2 次キャッシュに格納します。これによりシステムメモリーから読み直さなければならない参照ストリームの数を最小限にできます。他の書き込みが 2 次キャッシュのデータを乱さないように、ストリーミング・ストア命令 (MOVNTQ) を使用してすべてのプロセッサー・キャッシュに書き込みを行います。

プリフェッチによるキャッシュ管理をビデオ・エンコーダーに実装すると、メモリー・トラフィックが減少します。一度しか使用しないビデオ・フレーム・データが 2 次キャッシュに入らないようにすると、2 次キャッシュの汚染が確実に減少します。非テンポラルな PREFETCH 命令 (PREFETCHNTA) を使用すると、1 次キャッシュだけにデータが移動するため、2 次キャッシュの汚染が減少します。

2 次キャッシュに直接移動されたデータが再利用されない場合、非テンポラルなプリフェッチの方がテンポラルなプリフェッチよりもパフォーマンスを改善できます。2 次キャッシュの汚染を回避するため非テンポラルなプリフェッチ命令を使用するエンコーダーでは、2 次キャッシュでのヒット数が増え、汚染を引き起こすライトバックの回数が減少します。プリフェッチ命令だけでなく、2 次キャッシュの利用効率を高くすることで、パフォーマンスを改善できます。

9.6.2.2 ビデオデコーダー

ビデオデコーダーでは、完成したフレームデータがグラフィックス・カードの WC (ライト・コンバイニング) メモリータイプにマッピングされるローカルメモリーに書き込まれます。将来データを生成するため、プロセッサーによって WB メモリーに参照データのコピーが格納されます。これは、参照データのサイズが大きすぎてプロセッサーのキャッシュに収まらないことを想定したものです。ストリーミング・ストア命令によってデータをキャッシュ全体に書き込むと、キャッシュに格納されている他のテンポラルなデータが追い出されなくなります。後にプロセッサーは、非テンポラル (NTA) なプリフェッチ命令である PREFETCHNTA 命令により目的のデータを再読み込みします。これによって、帯域幅が最大になる一方、キャッシュされたほかのテンポラルなデータの乱れが最小限に抑えられます。

9.6.2.3 ビデオ・エンコーダー/デコーダーの実装から導かれる結論

これらの 2 つの例からいえることは、非テンポラルなプリフェッチ命令と非テンポラルなストア命令を適切に組み合わせて使用することで、2 次キャッシュの汚染が防止でき、有用なデータを 2 次キャッシュに残したたまま、コストの高いライトバックのトランザクションを減らすことができます。プリフェッチ命令でデータの準備を整えてもそれほどパフォーマンスの向上が見込まれないアプリケーションでも、2 次キャッシュとメモリーを効率良く使用すれば改善可能です。そのような設計では、メモリーバスなど重要なリソースに対するエンコーダーの要求が減り、システムのバランスが増し、パフォーマンスはさらに高くなります。

9.6.2.4 メモリー・コピー・ルーチンの最適化

大量のデータをコピーするルーチンを作成することは、ソフトウェアの最適化における一般的なタスクです。例 9-10 に、単純なメモリーコピーを実行する基本アルゴリズムを示します。

例 9-10 単純なメモリーコピーの基本アルゴリズム

```
#define N 512000
double a[N], b[N];
for (i = 0; i < N; i++) {
    b[i] = a[i];
}
```

このタスクは、さまざまなコーディング手法を用いて最適化できます。その 1 つは、ソフトウェア・プリフェッチ命令とストリーミング・ストア命令を使用する方法です。この方法について以下で述べます。例 9-11 にコード例を示します。

インテル® ストリーミング SIMD 拡張命令を使用して、以下の点を考慮することでメモリーコピーのアルゴリズムを最適化できます。

- データのアライメント
- メモリー内の各ページの正しいレイアウト
- キャッシュサイズ
- トランザクション・ルックアサイド・バッファー (TLB) とメモリーアクセスとの相互作用
- プリフェッチ命令とストリーミング・ストア命令の組み合わせ

例 9-11 ソフトウェア・プリフェッチを使用したメモリー・コピー・ルーチン

```
#define PAGESIZE 4096;
#define NUMPERPAGE 512 // ページに収まる要素数
double a[N], b[N], temp;
for (kk=0; kk<N; kk+=NUMPERPAGE) {
    temp = a[kk+NUMPERPAGE]; // 古いアーキテクチャー向けに TLB を準備
    // ブロックサイズ = ページサイズ、
    // ブロック全体をプリフェッチ、ループごとに 1 キャッシュラインを使用
    for (j=kk+16; j<kk+NUMPERPAGE; j+=16) {
        _mm_prefetch((char*)&a[j], _MM_HINT_NTA);
    }
     // ループごとに 128 バイトをコピー
     for (j=kk; j<kk+NUMPERPAGE; j+=16) {
     _mm_stream_ps((float*)&b[j],
     _mm_load_ps((float*)&a[j]));
     _mm_stream_ps((float*)&b[j+2],
     _mm_load_ps((float*)&a[j+2]));
     _mm_stream_ps((float*)&b[j+4],
     _mm_load_ps((float*)&a[j+4]));
     _mm_stream_ps((float*)&b[j+6],
     _mm_load_ps((float*)&a[j+6]));
     _mm_stream_ps((float*)&b[j+8],
     _mm_load_ps((float*)&a[j+8]));
     _mm_stream_ps((float*)&b[j+10],
     _mm_load_ps((float*)&a[j+10]));
     _mm_stream_ps((float*)&b[j+12],
     _mm_load_ps((float*)&a[j+12]));
     _mm_stream_ps((float*)&b[j+14],
     _mm_load_ps((float*)&a[j+14]));
     } // 1 ブロックのコピー終了
} // N 要素のコピー終了
_mm_sfence();
```

9.6.2.5 8 バイト・ストリーミング・ストアとソフトウェア・プリフェッチの使用

例 9-11 に 2 次キャッシュを考慮したコピー・アルゴリズムを示します。このアルゴリズムは次の順番で実行されます。

1. _mm_prefetch 組込み関数によって、ブロッキング手法を用いてメモリーから 2 次キャッシュへ 8 バイト・データを転送します。128 バイト単位で転送して 1 つのブロックを一杯にします。ブロックのサイズは、2 次キャッシュのサイズの半分未満であることが望まれますが、ループ処理による遅延を十分に吸収できる大きさでなければなりません。
2. _mm_load_ps 組込み関数を使用して、XMM レジスターにデータをロードします。
3. _mm_stream 組込み関数を使用して、別のメモリー・ロケーションにその 8 バイト・データを転送します。このときキャッシュはバイパスされます。

例 9-11 では、プリフェッチされたデータ (128 バイトのキャッシュライン) のすべてがライトバックされるように、_mm_load_ps と _mm_stream_ps の組込み関数がそれぞれ 8 つずつ使用されています。このプリフェッチ命令とストリーミング・ストア命令は、データの読み出し/書き込み処理間の移行をできるだけ減らすため、別々のループで実行されます。これによりメモリーアクセスの帯域幅が格段に広がります。

プリフェッチの前に配列 A のページ･テーブル･エントリーが古いアーキテクチャーの TLB に入るようにするには、temp = a[kk+CACHESIZE] を実行します。この文によりキャッシュラインがメモリーから取得したデータで満たされるため、本質的にはプリフェッチそのものと言えます。このループでは kk+4 からプリフェッチが開始されます。

この例では、コピー先がそのコードと時間的に隣接していないことを前提としています。コピーされたデータが、近い将来再利用される場合、これらのストリーミング･ストア命令は、通常の 128 ビット･ストア命令 (_mm_store_ps) と置き換える必要があります。

9.6.2.6 16 バイト･ストリーミング･ストアとハードウェア･プリフェッチの使用

広いメモリー領域のコピーを最適化するもう 1 つの手法は、ハードウェア･プリフェッチャーと 16 バイト･ストリーミング･ストアの利点を活用するため、バス読み出しと書き込みトランザクションを分離するセグメント化の手法を適用することです。詳細は 3.6.11 節「非テンポラルなストア･バス･トラフィック」を参照してください。

この手法は、2 つの段階で構成されます。第 1 段階では、メモリーからキャッシュ･サブシステムにデータブロックが読み込まれます。第 2 段階は、キャッシュされたデータがストリーミング･ストアによってデスティネーションに書き込まれます。

例 9-12 ハードウェア･プリフェッチとバス･セグメンテーションを利用したメモリーコピー

```
void block_prefetch(void *dst,void *src)
{ _asm {
    mov edi,dst
    mov esi,src
    mov edx,SIZE
    align 16
main_loop:
    xor ecx,ecx
    align 16

prefetch_loop:
    movaps xmm0, [esi+ecx]
    movaps xmm0, [esi+ecx+64]
    add ecx,128
    cmp ecx,BLOCK_SIZE
    jne prefetch_loop
    xor ecx,ecx
    align 16
cpy_loop:
    movdqa xmm0,[esi+ecx]
    movdqa xmm1,[esi+ecx+16]
    movdqa xmm2,[esi+ecx+32]
    movdqa xmm3,[esi+ecx+48]
    movdqa xmm4,[esi+ecx+64]
    movdqa xmm5,[esi+ecx+16+64]
    movdqa xmm6,[esi+ecx+32+64]
    movdqa xmm7,[esi+ecx+48+64]
    movntdq [edi+ecx],xmm0
    movntdq [edi+ecx+16],xmm1
    movntdq [edi+ecx+32],xmm2
    movntdq [edi+ecx+48],xmm3
    movntdq [edi+ecx+64],xmm4
    movntdq [edi+ecx+80],xmm5
    movntdq [edi+ecx+96],xmm6
```

```
    movntdq [edi+ecx+112],xmm7
    add ecx,128
    cmp ecx,BLOCK_SIZE
    jne cpy_loop
    add esi,ecx
    add edi,ecx
    sub edx,ecx
    jnz main_loop
    sfence
    }
}
```

9.6.2.7 メモリー・コピー・ルーチンのパフォーマンスの比較

広い領域のメモリー・コピー・ルーチンのスループットは、次のような複数の要因に依存します。

- メモリー・コピー・タスクを実装するコーディング手法
- システムバスの特性 (速度、最大帯域幅、読み出し/書き込みトランザクション・プロトコルのオーバーヘッド)
- プロセッサーのマイクロアーキテクチャー

パフォーマンスの比較でベースラインとなるのは、バイト・シーケンシャル手法により、400MHz システムバス対応の第 1 世代インテル® Pentium® M プロセッサー (CPUID シグネチャー 0x69n) 上で 8MB 領域のメモリーコピーを実行した場合のスループット (バイト/秒)です。システムバス速度が速く、コーディング手法が異なるいくつかの最近のプロセッサー/プラットフォームにおける、パフォーマンス・ベースラインに対する向上の度合いを比較しています。

2 番目のコーディング手法では、REP 文字列命令を使い 4 バイト単位でデータを移動しています。3 番目では、例 9-11 で示したコーディング手法のパフォーマンスを比較しています。(4) では、一度に 4KB のデータをフェッチし (ハードウェア・プリフェッチを使用してバス読み出しトランザクションを収集) 16 バイト・ストリーミング・ストアによってメモリーに書き込んだ際のスループットを比較しています。

スループット向上の主な要因は、バス速度です。例 9-12 で示した手法では、プラットフォームにおけるバス速度向上の利点をより効率良く活用できます。また、ワーキングセット全体を 2 次キャッシュ内に収めつつブロックサイズを 4KB の倍数に増加すると、スループットをわずかに高めることができます。

表 9-3 で示されている相対的なパフォーマンス値は、プロセッサー内のクリーンなマイクロアーキテクチャー条件 (例: 単純なコードシーケンスを何度もループさせる) を表しています。メモリー・コピー・ルーチンをアプリケーションに統合するメリットは、アプリケーションごとに異なります (機能が豊富なアプリケーションでは、マイクロアーキテクチャーごとの複雑な条件が大量に生じる傾向があります)。

9.6.3 キャッシュ・パラメーター

CPUID がパラメーター・リーフをサポートしている場合、ソフトウェアは、リーフを利用してキャッシュ階層の各レベルへの照会を行うことができます。各キャッシュレベルの列挙は、ECX レジスターのインデックス値 (0 から開始) を指定することで行われます (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の第 3 章「CPUID-CPU Identification」を参照してください)。

表 9-3 に、パラメーターの一覧を示します。

表 9-3 決定論的キャッシュ・パラメーター・リーフ

ビット位置	名前	意味
EAX[4:0]	キャッシュタイプ	0 = NULL - キャッシュなし 1 = データキャッシュ 2 = 命令キャッシュ 3 = ユニファイド・キャッシュ 4～31 = 予約済み
EAX[7:5]	キャッシュレベル	1 から開始
EAX[8]	キャッシュレベルの自己初期化	1: ソフトウェアの初期化が不要
EAX[9]	フル・アソシアティブ・キャッシュ	1: はい
EAX[13:10]	予約済み	
EAX[25:14]	このキャッシュを共有する論理プロセッサーの最大数	プラス 1 エンコーディング
EAX[31:26]	パッケージ中のコアの最大数	プラス 1 エンコーディング
EBX[11:0]	システム・コヒーレンシー・ライン・サイズ (L)	プラス 1 エンコーディング (バイト)
EBX[21:12]	物理ライン・パーティション (P)	プラス 1 エンコーディング
EBX[31:22]	アソシアティブ・ウェイ数 (W)	プラス 1 エンコーディング
ECX[31:0]	セット数 (S)	プラス 1 エンコーディング
EDX	予約済み	
3 より大きく 80000000 より小さい CPUID リーフは、IA32_CR_MISC_ENABLES.BOOT_NT4 (ビット 22) がクリアされている場合 (デフォルト) にのみ参照可能です。		

キャッシュ・パラメーター・リーフは、キャッシュ・パラメーターの列挙に関して、ある程度のフォワード互換をソフトウェアに実装するための手段を提供します。キャッシュ・パラメーターは、以下のような状況で役立ちます。

- キャッシュレベルのサイズを決定する。
- インテル® ハイパースレッディング・テクノロジーや、マルチコアおよびシングルコアのプロセッサー間で、異なる共有トポロジーのキャッシュレベルにキャッシュ・ブロッキング・パラメーターを適応させる。
- MP システムのマルチスレッディング・リソース・トポロジーを決定する (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A』の第 7 章「Multiple-Processor Management」を参照してください)。
- マルチコア・プロセッサーを使用したプラットフォームのキャッシュ階層トポロジーを決定する (第 1 章の最後に記載されているトポロジーの列挙に関するホワイトペーパーとサンプルコードを参照してください)。
- スレッドとプロセッサーのアフィニティーを管理する。
- プリフェッチ間隔を決定する。

特定レベルのキャッシュのサイズは、次の式で求められます。

(ウェイ数) * (パーティション) * (ラインサイズ) * (セット) = (EBX[31:22] + 1) * (EBX[21:12] + 1) * (EBX[11:0] + 1) * (ECX + 1)

9.6.3.1 キャッシュ・パラメーターを使用したキャッシュ共有

キャッシュの局所性を改善することは、ソフトウェアの最適化における重要な要素です。例えば、キャッシュ・ブロッキング・アルゴリズムは、シングル・プロセッサーおよび各種マルチプロセッサーの実行環境 (HT テクノロジー対応プロセッサーやマルチプロセッサーなど) での実行時にブロックサイズが最適化されるよう設計できます。

基本的な手法は、ターゲット・キャッシュ・レベルによって扱われる論理プロセッサーの数でそのターゲット・キャッシュ・レベルのサイズを割ったものより、ブロックサイズの上限を小さくすることです。この手法は、マルチスレッド・アプ

リケーションのプログラミングに適用されます。また、マルチタスク・ワークロードの一部であるシングルスレッド・アプリケーションにもメリットがあります。

9.6.3.2 シングルコアまたはマルチコアでのキャッシュ共有

キャッシュ・パラメーターは、より高度な状況においてマルチスレッド・アプリケーションの共有キャッシュ階層を管理する際に効果的です。特定のキャッシュレベルは、単一のプロセッサー内の複数の論理プロセッサーで共有できます。または、単一の物理プロセッサー・パッケージ内の複数の論理プロセッサーで共有するようにも実装できます。

キャッシュ・パラメーター・リーフと、プラットフォーム内の各論理プロセッサーに関連付けられた初期 APIC_ID を利用すると、ソフトウェアは、キャッシュレベルを共有する論理プロセッサーの数およびトポロジー関係に関する情報を引き出すことができます。

9.6.3.3 プリフェッチ間隔の決定

プリフェッチ間隔 (CPUID.01H.EBX の説明を参照) は、プロセッサーが PREFETCHh 命令 (PREFETCHT0、PREFETCHT1、PREFETCHT2、PREFETCHNTA) によってプリフェッチを実行する領域の長さを示します。ソフトウェアは、特定のキャッシュ階層レベルに対してプリフェッチを実行する際に、命令によって識別される間隔として、この長さを使用します。プリフェッチ・サイズは、キャッシュタイプのデータキャッシュ (1) とユニファイド・キャッシュ (3) のみに関連しており、その他のキャッシュタイプでは無視すべきです。ソフトウェアは、コヒーレンシー・ライン・サイズがプリフェッチ間隔であると見なしてはなりません。

プリフェッチ間隔フィールドがゼロの場合は、デフォルトサイズの 64 バイトがプリフェッチ間隔であると見なす必要があります。ソフトウェアは、以下のアルゴリズムを使い、キャッシュ・パラメーター機構とレガシー機構のいずれがサポートされているかに応じて、使用すべきプリフェッチ・サイズを決定します。

- プロセッサーがキャッシュ・パラメーターをサポートし、ゼロ以外のプリフェッチ・サイズを提供している場合は、そのプリフェッチ・サイズを使用します。
- プロセッサーがキャッシュ・パラメーターをサポートし、プリフェッチ・サイズを提供していない場合、各キャッシュ階層レベルのデフォルトサイズは 64 バイトです。
- プロセッサーがキャッシュ・パラメーターをサポートせず、レガシーのプリフェッチ・サイズ・ディスクリプター (0xF0 - 64 バイト、0xF1 - 128 バイト) を提供している場合、すべてのキャッシュ階層レベルのプリフェッチ・サイズはこのディスクリプターで示されます。
- プロセッサーがキャッシュ・パラメーターをサポートせず、レガシーのプリフェッチ・サイズ・ディスクリプターを提供していない場合、すべてのキャッシュ階層レベルのデフォルトサイズは 32 バイトです。

第 10 章
サブ NUMA クラスタリングの概要

サブ NUMA クラスタリング (SNC) は、ラスト・レベル・キャッシュ (LLC) からメモリーへの平均レイテンシーを改善するモードです。これは、以前の世代のインテル® Xeon® プロセッサー E5 ファミリーにおけるクラスターオンダイ (COD) の実装を置き換えるものです。

10.1 サブ NUMA クラスタリング

SNC は、LLC をアドレス範囲に基づいて複数のクラスターに分割することで LLC/メモリーの平均レイテンシーを改善します。

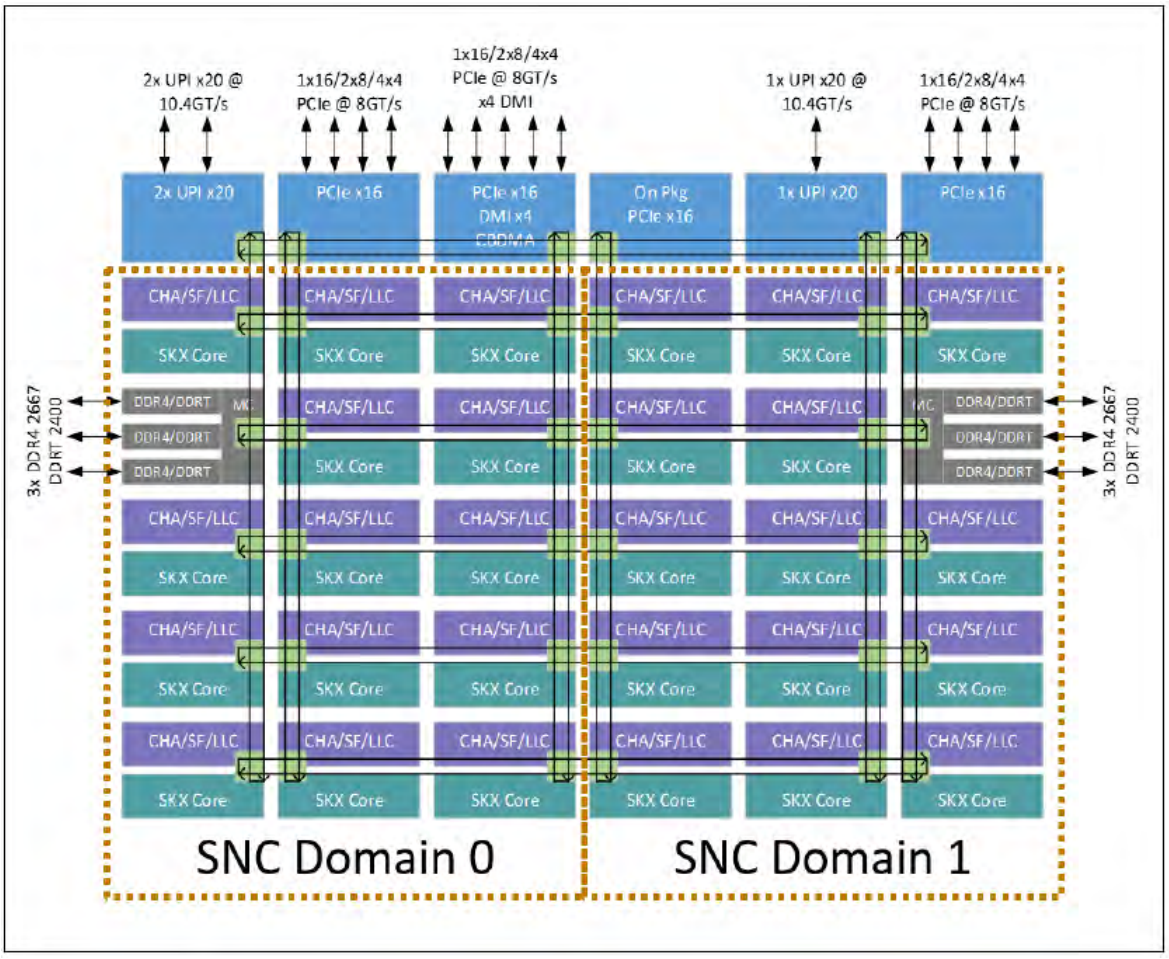


図 10-1 SNC 構成の例

10.2 クラスターオンダイとの比較

SNC は COD の不利益を被ることなく、COD と同様の局所的な利点をもたらします。COD とは異なり SNC は次の特性を持ちます。

- 単一のウルトラ・パス・インターコネクト (UPI) キャッシュ・エージェントが必要です。
- リモートクラスターへのメモリー・アクセス・レイテンシーは小さく、UPI フローは必要ありません。
- LLC でラインが複製されることはないため、LLC の容量をより効率良く利用できます。

SNC にも欠点はあります。

- リモートクラスターのアドレスがローカルクラスターの LCC にキャッシュされることがないため、クラスターオンダイ (COD) 方式に比べレイテンシーが大きくなることがあります。

10.3 SNC の利用

この節では次に示すモードと括弧で囲まれたそれらの BIOS 名について説明します (実際の BIOS パラメーター名は BIOS ベンダーとバージョンによって異なります)。

- NUMA 無効 (NUMA Optimized: Disabled)
- SNC オフ (Integrated Memory Controller (IMC) Interleaving: auto, NUMA Optimized: Enabled, Sub_NUMA Cluster: Disabled)
- SNC オン (IMC Interleaving: 1-way Interleave, NUMA Optimized: Enabled, Sub_NUMA Cluster: Enabled)

以降に示すコマンドは、2 ソケットのインテル® Xeon® プロセッサー・ベースのシステム (ソケットごとに 28 コア、インテル® ハイパースレッディング・テクノロジー有効) で実行されています。

10.3.1 NUMA 構成をチェックする方法

SNC が有効化されるとシステムには追加の NUMA ノードが存在することになります。SNC 機能の利点を活用するには、開発者は NUMA 構成を考慮する必要があります。

この節では、NUMA システムの構成を確認するいくつかの方法を紹介します。

libnuma

アプリケーションは、libnuma を使用して NUMA 構成を確認できます。

このコード例では、libnuma を使用して NUMA ノードの最大数を取得しています。

```
#include <stdio.h>
#include <stdlib.h>
#include <numa.h>
int main(int argc, char *argv[])
{
  int max_node;

/* システムが NUMA をサポートするか確認 */
  max_node = numa_max_node();
  printf("%d¥n", max_node);

  return 0;
}
```

numactl

Linux* では、numactl ユーティリティー (numactl-libs と numactl-devel パッケージが必要です) を使用して NUMA 構成を確認できます。

$ numactl --hardware

NUMA disabled:

```
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
node 0 size: 196045 MB
node 0 free: 190581 MB
node distances:
node 0
  0: 10
```

SNC off:

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83
node 0 size: 96973 MB
node 0 free: 94089 MB
node 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109 110 111
node 1 size: 98304 MB
node 1 free: 95694 MB
node distances:
node 0 1
  0: 10 21
  1: 21 10
```

SNC on:

```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 7 8 9 14 15 16 17 21 22 23 56 57 58 59 63 64 65 70
71 72 73 77 78 79
node 0 size: 47821 MB
node 0 free: 45759 MB
node 1 cpus: 4 5 6 10 11 12 13 18 19 20 24 25 26 27 60 61 62 66 67 68 69
74 75 76 80 81 82 83
node 1 size: 49152 MB
node 1 free: 47097 MB
node 2 cpus: 28 29 30 31 35 36 37 42 43 44 45 49 50 51 84 85 86 87 91 92
93 98 99 100 101 105 106 107
node 2 size: 49152 MB
node 2 free: 47617 MB
node 3 cpus: 32 33 34 38 39 40 41 46 47 48 52 53 54 55 88 89 90 94 95 96
97 102 103 104 108 109 110 111
node 3 size: 49152 MB
node 3 free: 47231 MB
node distances:
node 0 1 2 3
  0: 10 11 21 21
  1: 11 10 21 21
  2: 21 21 10 11
  3: 21 21 11 10
```

hwloc

また、Linux* では lstopo ユーティリティー (hwloc パッケージが必要) で NUMA 構成を確認できます。次に例を示します。

```
$ lstopo -p --of png --no-io --no-caches > numa_topology.png
```

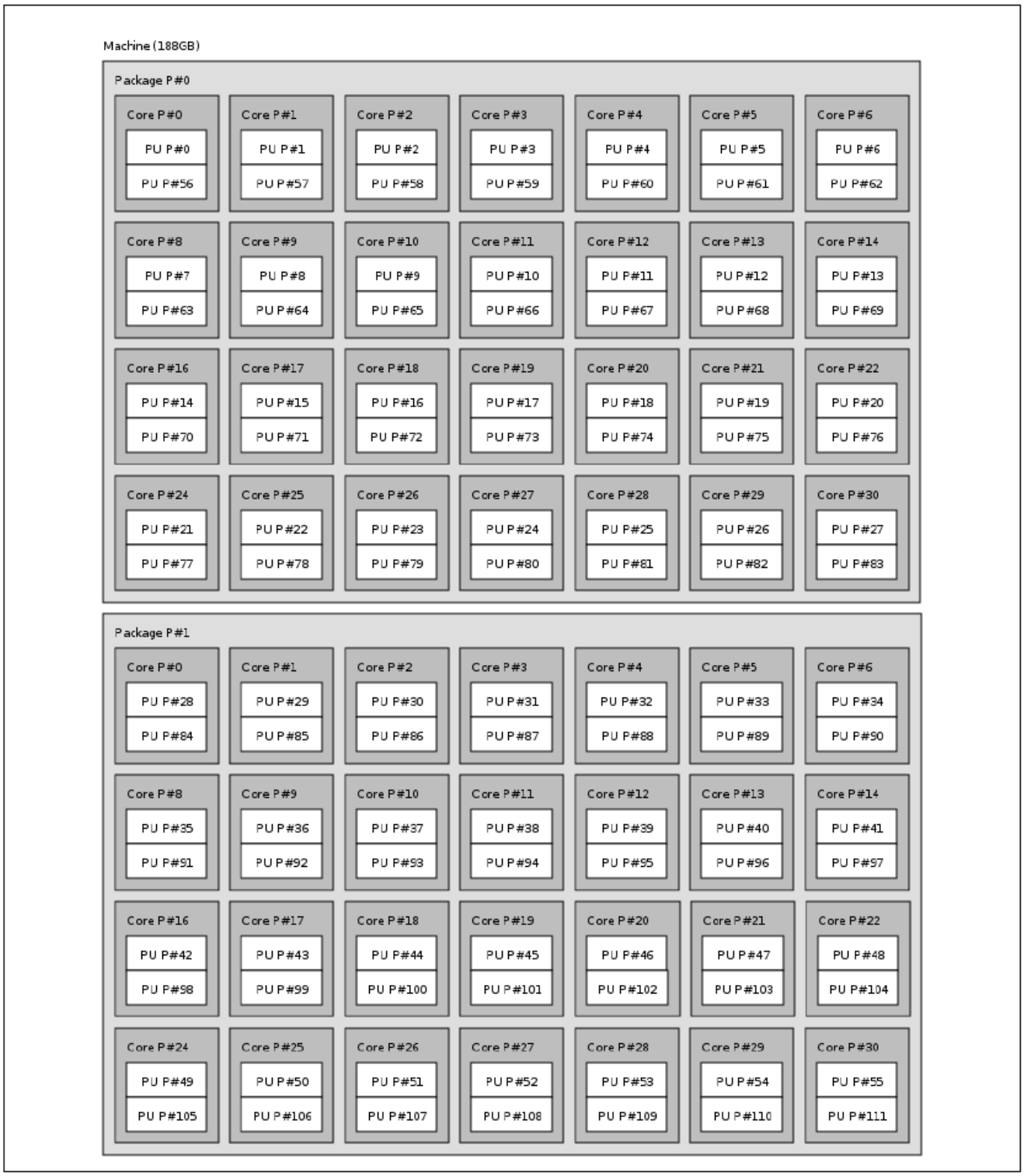


図 10-2 NUMA が無効の場合

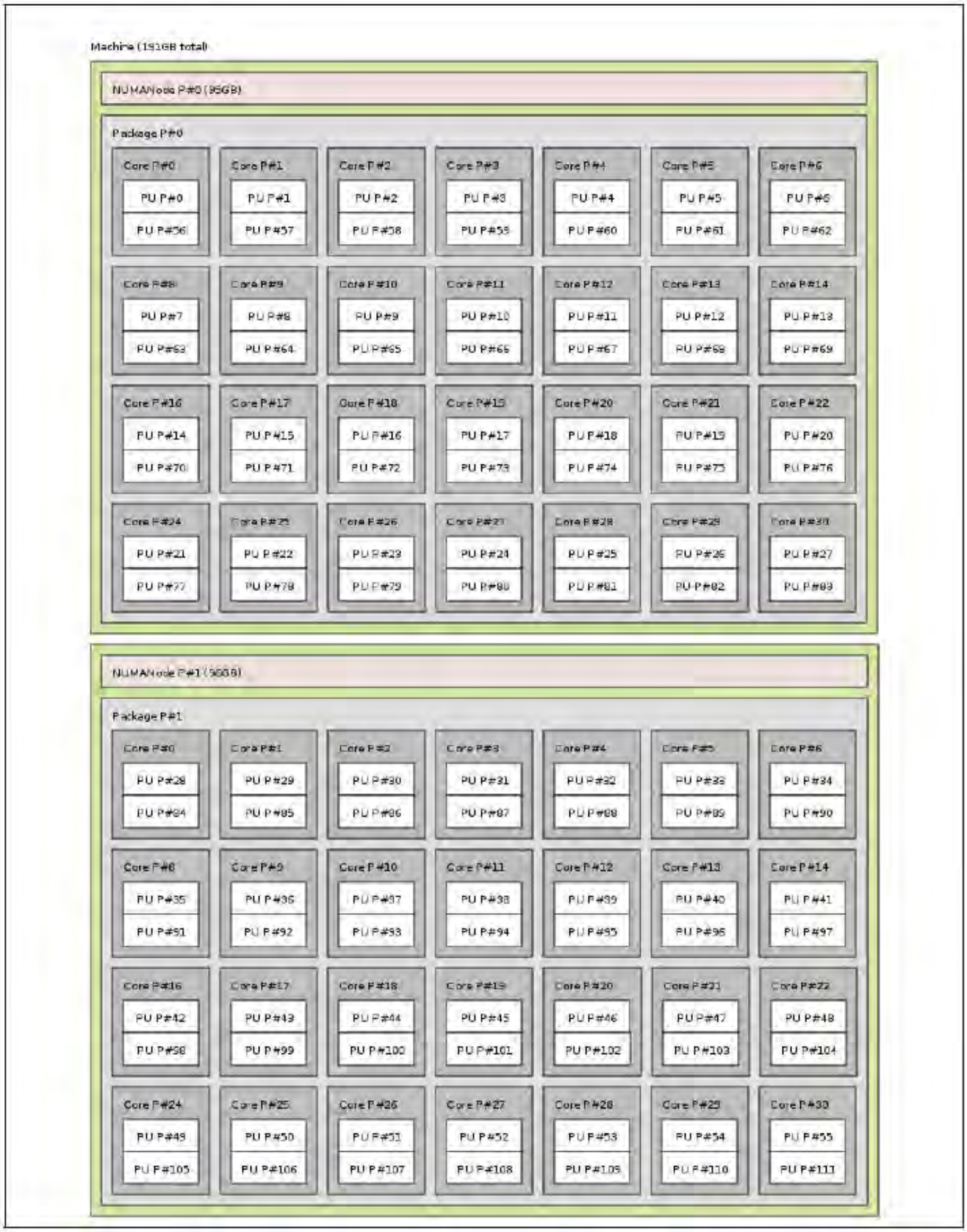


図 10-3 SNC オフの場合

図 10-4 SNC オンの場合

10.3.2 SNC 向けに MPI を最適化

ソフトウェアは SNC の恩恵を受けるため NUMA への最適化が必要です。NUMA フレンドリーな動作を保証するようにコードを変更することなく、アクセスの局所性を確実にするため NUMA 領域ごとに 1 つの MPI ランクを実行します。これは SNC を使用してパフォーマンスを向上させる最も簡単な方法です。

インテル® MPI ライブラリーには、いくつかの NUMA に関連する最適化が実装されています。インテル® MPI ライブラリーはアウトオブボックスですぐに動作し、ほとんどの状況に対応できますが、特殊な状況ではパフォーマンスを改善するため環境変数を設定して NUMA 関連の機能を制御する必要があります。

これらの環境変数は主に MPI プロセスの配置 (ピニングやバインド) に関連します。例として、I_MPI_PIN_DOMAIN 環境変数などがあります。詳細は、インテル® MPI ライブラリーのデベロッパー・リファレンスを参照してください。この環境変数は、ノード上の論理プロセッサーがオーバーラップしないサブセット数 (ドメイン) を定義し、それらのドメインへ MPI プロセスを結びつけるルールを設定することができます。次の図ではドメインごとに 1 つの MPI プロセスを割り当てています。

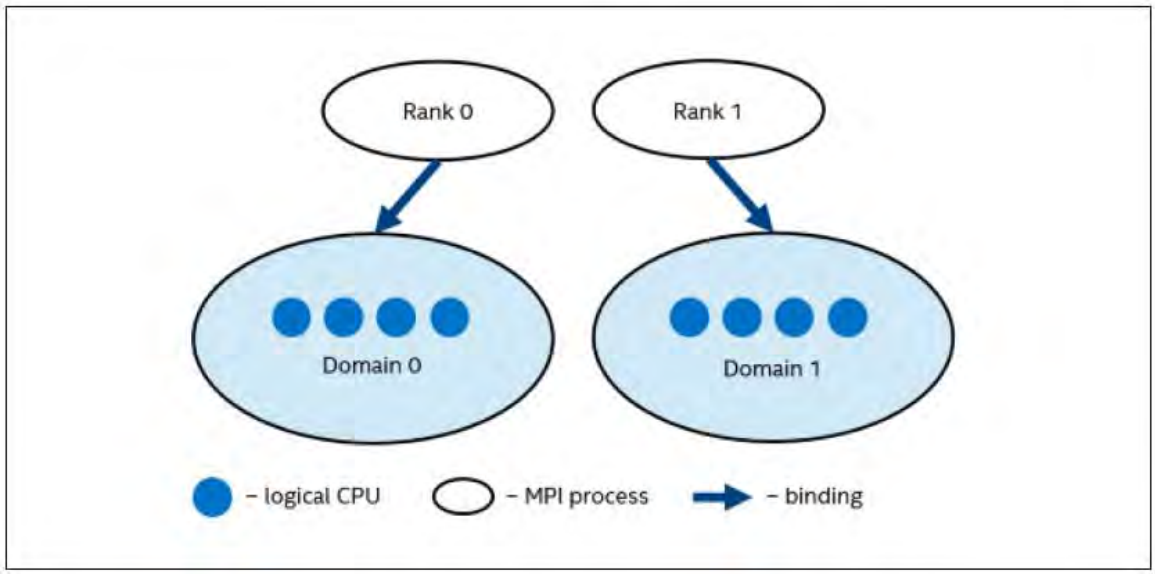


図 10-5 ドメインごとに 1 つの MPI プロセスを割り当てる例

各 MPI プロセスは、対応するドメイン内で実行する子スレッドを作成できます。プロセスのスレッドは、ドメイン内の論理プロセッサーからほかの論理プロセッサーへ自由に移行できます。

例えば、I_MPI_PIN_DOMAIN=numa は、SNC モードが有効化された MPI/OpenMP* ハイブリッド・アプリケーションで妥当なオプションであると言えます。この場合、各ドメインは特定の NUMA ノードを共有する論理プロセッサーで構成されます。マシン上のドメイン数は、マシン上の NUMA ノード数と等しくなります。

詳し情報については、インテル® MPI ライブラリーのドキュメント、https://software.intel.com/en-us/intel-mpi-library/documentation (英語) を参照してください。

10.3.3 SNC のパフォーマンス比較

この節では、異なるモードの NUMA ノードのパフォーマンス (レイテンシー) の変化を示すため、インテル® Memory Latency Cheker (インテル® MLC) を使用して収集したパフォーマンス・データを掲載しています。

アプリケーションのパフォーマンスを決定する重要な要因は、アプリケーションがプロセッサーのキャッシュ階層とメモリー・サブシステムからデータをフェッチするのにかかる時間です。NUMA 構成の複数ソケットシステムでは、ローカルメモリーとソケットを横断するメモリー・レイテンシーはかなり異なります。帯域幅もまた、パフォーマンスを決定する重要な要因です。そのため、検証するシステムでパフォーマンス解析を行う場合、ベースラインを定めてからレイテンシーと帯域幅を測定することが重要です。

インテル® MLC はメモリー・レイテンシーと帯域幅を測定するツールであり、システムの負荷が高まるにつれてそれらがどのように変化するか観察します。また、特定の組み合わせのコアからキャッシュやメモリーへの帯域幅とレイテンシーを測定できることから、さらに詳しく調査するいくつかのオプションを提供します。

インテル® MLC の詳細については、https://software.intel.com/en-us/articles/intelr-memory-latency-checker (英語) を参照してください。

パフォーマンス・データを収集するため次のコマンドを使用しました。

```
% mlc_avx512 --latency_matrix
```

このコマンドは、システムのそれぞれのソケットから別のソケットすべてへの、アイドル・メモリー・レイテンシーを測定し、表形式で結果をレポートします。デフォルトでは、システム中のすべての NUMA ノードへのレイテンシーをレ

ポートします。NUMA レベルのレポートは Linux* でのみ動作します。Windows* では、ソケットレベルのレポートがサポートされます。

注意

現代のインテル® プロセッサーは洗練された HW プリフェッチャーを備えているため、正確にメモリー・レイテンシーを計測するには課題があります。インテル® MLC はレイテンシーを計測する間これらのプリフェッチャーを自動的に無効化し、完了すると元の状態へ戻します。プリフェッチャーの制御は MSR を介して行われ (詳細は、https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors (英語) を参照)、MSR へアクセスするには root レベルの権限が必要です。そのため、インテル® MLC は 'root' で実行する必要があります。

この計測に使用したソフトウェアは、インテル® MLC v3.3-Beta2 と Red Hat* Enterprise Linux* 7.2 です。

NUMA disabled:
Using buffer size of 2000.000MB
Measuring idle latencies (in ns)...

	Memory node	
Socket	0	1
0	126.5	129.4
1	23.1	122.6

SNC off:
Using buffer size of 2000.000MB
Measuring idle latencies (in ns)...

	Numa node	
Numa node	0	1
0	81.9	153.1
1	153.7	82.0

SNC on:
Using buffer size of 2000.000MB
Measuring idle latencies (in ns)...

	Numa node			
Numa node	0	1	2	3
0	81.6	89.4	140.4	153.6
1	86.5	78.5	144.3	162.8
2	142.3	153.0	81.6	89.3
3	144.5	162.8	85.5	77.4

第 11 章
マルチコアとインテル® ハイパースレッディング・テクノロジー

この章では、マルチプロセッサー (MP) システムまたはハードウェアベースのマルチスレッディング・サポートを搭載したプロセッサーで動作するマルチスレッド・アプリケーションのソフトウェア最適化手法について説明します。マルチプロセッサー・システムとは、複数のソケットを備えたシステムであり、各ソケットは 1 つの物理プロセッサー・パッケージに対応しています。ハードウェア・マルチスレッディング・サポートを提供するインテル® 64 プロセッサーと IA-32 プロセッサーには、デュアルコア・プロセッサー、クアッドコア・プロセッサー、HT テクノロジー[14] 対応プロセッサーなどがあります。

ハードウェア・リソースを増やしてスレッドレベルまたはタスクレベルの並列化を活用すると、マルチスレッディング環境での計算スループットが向上します。ハードウェア・リソースの増加は、物理プロセッサー、パッケージごとのプロセッサー・コア、コアごとの論理プロセッサーのいずれか、またはすべてを複数搭載することによって可能となります。したがって、マルチスレッディングの最適化の一部は、MP、マルチコア、HT テクノロジーのすべてに適用されます。また、一部のマイクロアーキテクチャー・リソースは、ハードウェア・マルチスレッディングの構成によって実装方法が異なります (例えば、HT テクノロジーが有効な場合、実行リソースは、異なるコア間では共有されませんが、同じコア内の 2 つの論理プロセッサー間では共有されます)。この章では、これらの状況に適用されるガイドラインについて説明します。

次の内容が含まれます。

- 性能の特性と使用モデル
- マルチスレッド・アプリケーション向けのプログラミング・モデル
- 5 つの特定分野におけるソフトウェア最適化手法

11.1 性能および使用モデル

マルチプロセッサー、マルチコア・プロセッサー、HT テクノロジーを使用したときの性能向上は、使用モデル、ワークロードの制御フローにおける並列処理の割合に大きく左右されます。一般的な使用モデルは次の 2 つです。

- マルチスレッド・アプリケーション
- シングルスレッド・アプリケーションを使用するマルチタスキング

11.1.1 マルチスレッディング

アプリケーションでマルチスレッディング・モデルを採用し、ワークロードに対してタスクレベルの並列処理を実装する場合、マルチスレッド・ソフトウェアの制御フローは、並列タスクと順次タスクの 2 つの領域に分割できます。

アムダールの法則は、制御フロー内の並列処理の割合とアプリケーションの性能向上の関係を説明しています。この法則は、並列化するコードモジュール、関数、または命令シーケンスを選択する際に有用な指針となります。順次タスクと制御フローを並列コードに変換して、マルチスレッディング・ハードウェア・サポートを活用して得られる性能向上は、これらの要素により最大化できる可能性が高まります。

[14] インテル® 64 プロセッサーと IA-32 プロセッサーがハードウェア・マルチスレッディングをサポートしているかどうかは、CPUID の機能フラグ CPUID.01H:EDX[28] によって判別できます。ビット 28 の戻り値が 1 である場合は、少なくとも 1 種類のハードウェア・マルチスレッディングが物理プロセッサー・パッケージに搭載されていることを示します。各パッケージでサポートされている論理プロセッサーの数も CPUID から判断できます。また、アプリケーションは、適切なオペレーティング・システム呼び出しを行うことにより、実行時にアプリケーションが利用できる有効な論理プロセッサーの数を確認する必要があります。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』を参照してください。

図 11-1 は、アムダールの法則により、ワークロードに対する性能向上が実現される様子を示しています。図 11-1 のボックスは、個々のタスクユニット、またはアプリケーション全体のワークロードの集合を示しています。

一般に、N 個の物理プロセッサー (もしくはコア) を持つ MP システム上でマルチスレッドを実行すると、シングルスレッドで実行する場合と比較した速度の向上は次の式で表すことができます。

$$\text{RelativeResponse} = \frac{\text{Tsequential}}{\text{Tparallel}} = \left(1 - P + \frac{P}{N} + O\right)$$

ここで P は並列処理可能なワークロードの比率、*O* はマルチスレッディングのオーバーヘッド (オペレーティング・システムにより異なる) を示します。この場合、性能向上は相対レスポンスの逆数で示されます。

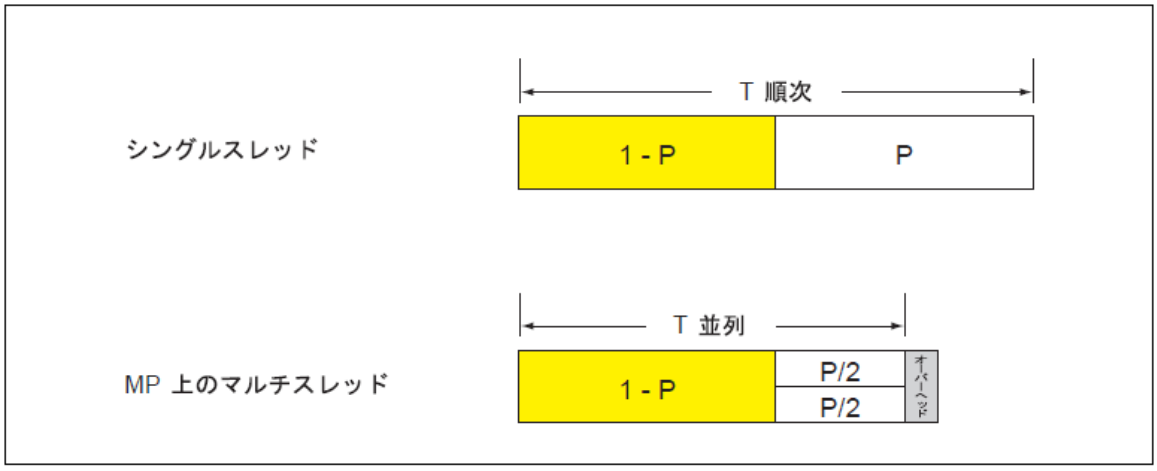


図 11-1 アムダールの法則と MP のスピードアップ

一般に、マルチスレッド環境でアプリケーションの性能を最適化する場合、物理プロセッサー数、および物理プロセッサーあたりの論理プロセッサー数に関連する性能スケーリングに最も影響するのは、全体に占める並列処理の割合です。

並列実行可能な領域が 50% しかないワークロードがマルチスレッド・アプリケーションの制御フローに含まれている場合、2 つの物理プロセッサーを使用したときの性能は、1 つのプロセッサーを使用したときと比べて、最大 33% しか向上しません。4 つのプロセッサーを使用した場合でも、60% しかスピードアップしません。したがって、並列処理を利用できる制御フローの割合を最大化することが重要です。スレッド同期の実装が不適切であると、順次制御フローの割合が大幅に増加し、アプリケーションの性能スケーリングがさらに減少します。

制御フローの並列処理を最大化することに加え、スレッド同期とタスク・スケジューリングのインバランスによるスレッド間の相互作用は、プロセッサーのスケーリング全体に大きな影響を与えることもあります。

過度なキャッシュミスは、性能スケーリングが低下する原因の 1 つです。マルチスレッド実行環境では、過度なキャッシュミスは以下が原因で発生します。

- 同じプロセス内の異なるスレッドによる別名スタックアクセス
- スレッド競合によるキャッシュラインの排出
- 異なるプロセッサー間でのキャッシュラインのフォルス・シェアリング

こうした状況 (およびその他の問題) に対処する手法は、以降の節で説明します。

11.1.2 マルチタスキング環境

インテル® 64 と IA-32 プロセッサーのハードウェア・マルチスレッディング機能がタスクレベルの並列処理を活用できるのは、ワークロードがいくつかのシングルスレッド・アプリケーションから構成され、それらのアプリケーションが MP 対応オペレーティング・システムの下で並列実行されるようにスケジュールされる場合です。こうした環境では、ハードウェア・マルチスレッディング機能はワークロードに対して高いスループットを提供できます。ただし、シングルタスクの相対的なパフォーマンス (シングルスレッド環境における同一タスクと比較した完了時間) は、共有の実行リソースおよびメモリーがどの程度利用されているかに応じて異なります。

各物理プロセッサー内でタスク・スケジューリングや共有実行リソースのバランシングを管理し、スループットを最大限に高めることができる OS カーネルコードは、一般的なオペレーティング・システム (Microsoft* Windows* XP Professional および Home、カーネル 2.4.19 以降[15] の Linux* ディストリビューションなど) に含まれています。

マルチタスキング環境では、アプリケーションは独立して実行されるため、スレッド同期の問題によってスループットのスケーリングが制限される可能性は低くなります。これは、(プロセッサー間通信が発生せず、システムバスの制約がない限り) ワークロードの制御フローは一般に 100% 並列となるためです。[16]

マルチタスキング・ワークロードでは、バス・アクティビティーやキャッシュのアクセスパターンによって、スループットのスケーリングが影響を受けやすくなります。同じアプリケーションを 2 つ、または同じアプリケーション・スイートをロックステップで実行すると、パフォーマンス測定に影響が現れることがあります。これは、1 次データキャッシュへのアクセスパターンが過度なキャッシュミスを引き起こし、パフォーマンスに影響する可能性があるためです。この問題は次の方法で解決できます。

- アプリケーションの起動時にインスタンス 1 つあたりのオフセットを含めます。
- アプリケーションの各インスタンスに対して異なるデータセットを使用し、異なるワークロードを与えます。
- 複数のコピーを実行する場合、アプリケーションの起動順序をランダムにします。

マルチタスキング・ワークロードの一部として 2 つのアプリケーションを使用すると、これらの 2 つのプロセス間では同期オーバーヘッドがほとんど発生しません。また、各アプリケーション自体の同期オーバーヘッドを最小にすることも重要です。

プロセス内部の同期を目的とした長いスピンループを使用するアプリケーションでは、マルチタスキング・ワークロードで HT テクノロジーの効果が得られる可能性は低くなります。これは、長いスピンループによって重要なリソースが消費されるためです。

11.2 プログラミング・モデルとマルチスレッディング

並列処理は、マルチスレッド・アプリケーションを設計し、マルチプロセッサーで最高のパフォーマンス・スケーリングを達成する最も重要な概念です。最適化されたマルチスレッド・アプリケーションは、並列処理の割合が高く、以下の点で依存性が低い特長があります。

- ワークロード
- スレッド間のやり取り
- ハードウェア利用効率

ワークロードの並列処理を最大限にする鍵は、アプリケーション内にある相互依存性の低い複数のタスクを識別し、それらのタスクを並列実行するため独立したスレッドを生成することです。

---

15 このコードは、Red Hat* Linux Enterprise AS 2.1 に含まれています。

16 一般に、マルチタスキング・ワークロードのスループットを評価するソフトウェア・ツールを使用すると、シリアル制御フローが増加します。スレッド同期の問題は、パフォーマンス測定手法における不可欠な要素として検討する必要があります。

独立したスレッドを並列に実行するのは、マルチスレッド・アプリケーションをマルチプロセシング・システムに導入する際の最も重要な点です。スレッド間のやり取りを管理して、スレッド同期のコストを最小限に抑えるのも、マルチプロセッサーでスケーリングを高めるために重要です。

並列スレッド間でハードウェア・リソースを効率良く使用するには、ハードウェア・リソースの競合を回避する最適化手法が必要となります。スレッド同期を最適化し、他のハードウェア・リソースを管理するコーディング手法については、以降の節で説明します。

並列プログラミング・モデルは、その後で説明します。

11.2.1 並列プログラミング・モデル

次に、独立したタスクの要件をアプリケーション・スレッドに置き換える一般的なプログラミング・モデルを示します。

- ドメイン分解
- 機能分解

11.2.1.1 ドメイン分解

通常、計算主体の大きなタスクでは、多数の小さなサブセットに分割可能なデータセットが使用されます。これらの各サブセットは、計算の独立性が高い傾向があります。これには次のような例が含まれます。

- 2 次元データに対する離散コサイン変換 (DCT) の計算: 2 次元データをいくつかのサブセットに分割し、各サブセットの変換を計算するスレッドを生成します。
- 行列乗算: 乗数行列を使用して、行列の半分の乗算を処理する複数のスレッドを生成します。

ドメイン分解は、同じまたは同等のスレッドを生成し、より小さなデータ部分を独立して処理することをベースにしたプログラミング・モデルです。このモデルでは、従来のマルチプロセッサー・システム内にある複製された実行リソースを活用できます。また、HT テクノロジーの 2 つの論理プロセッサー間で共有される実行リソースも活用できます。これは、一般にデータ・ドメイン・スレッドでは、オンチップの実行リソースの使用可能な部分のみが消費されるためです。

11.3.4 節「実行リソース最適化の主な慣例」では、データ・ドメイン・スレッドで共有実行リソースを使用し、2 つのスレッド間のハードウェア・リソースの競合を排除するガイドラインを詳しく説明しています。

11.2.2 機能分解

通常、アプリケーションは、多様な機能と多くの依存性のないデータセットを使用して、幅広いタスクを処理します。例えば、ビデオコーデックでは、DCT、動き評価、カラー変換など各種処理機能が必要となります。アプリケーションは、機能スレッドモデルを使用して、動き評価、カラー変換、その他の機能タスクを実行する独立したスレッドをプログラムできます。

機能分解は、ハードウェア・リソースの複製にあまり依存しない場合、より柔軟なスレッドレベルの並列処理を実現できます。例えば、ソート・アルゴリズムを実行するスレッドと、行列乗算ルーチンを実行するスレッドは、同じ実行ユニットを同時に使用しません。これを考慮して設計すると、従来のマルチプロセッサー・システムや、HT テクノロジー対応のプロセッサーを使用するマルチプロセッサー・システムを活用できます。

11.2.3 専用プログラミング・モデル

インテル® Core™ Duo プロセッサーとインテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、同一物理パッケージ内の 2 つのプロセッサー・コアが 2 次キャッシュを共有します。そのため、バス・トラフィックのオーバーヘッドを最小限に抑えて 2 つのアプリケーション・スレッドが同じアプリケーション・データにアクセスできます。

マルチスレッド・アプリケーションでこの種のハードウェア機能を利用するには、専用のプログラミング・モデルが必要な場合があります。このようなモデルの 1 つに、「生産 (producer) - 消費 (consumer)」モデルがあります。このモデルでは、一方のスレッドがデータをデスティネーション (2 次キャッシュが望ましい) に書き込み、同一物理パッケージ内の別のコアで実行される他方のスレッドが、最初のスレッドによって生成されたデータを読み出します。

生産-消費モデルを実装する基本的なアプローチは、一方が生産し他方が消費する 2 つのスレッドを生成することです。生産と消費スレッドは通常、バッファーを介して交互に処理を行い、バッファーの交替準備が整うと互いに通知します。生産-消費モデルでは、両者でバッファーを交替する際にスレッド同期のオーバーヘッドが生じます。コア数に応じて最適なスケーリング保つには、同期のオーバーヘッドを低く抑える必要があります。これは、各増分タスクを終了してバッファーを交替する前に、生産と消費のスレッドで同等の時定数を設定することにより実現できます。

例 11-1 に、タスクユニットのシーケンスをシングルスレッドで実行する場合のコーディング構造を示します。この例では、各タスクユニット (生産または消費) がシリアルに実行されています (図 11-2 を参照)。同等のシナリオをマルチスレッドで実行する場合、生産と消費の 1 つのペアを 1 つのスレッド関数としてラップすると、プロセッサー・リソースに 2 つのスレッドを同時にスケジューリングできます。

例 11-1 生産および消費のワークアイテムのシリアル実行

```
for (i = 0; i < number_of_iterations; i++) {
    producer (i, buff); // バッファー・インデックスとアドレスを渡す
    consumer (i, buff);
}
```

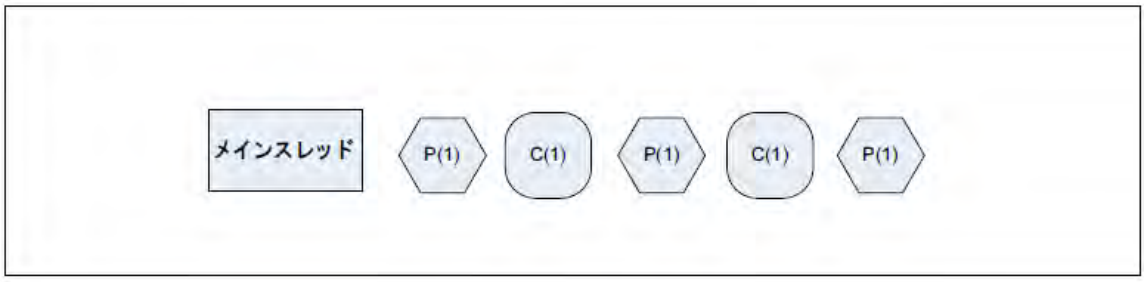


図 11-2 生産-消費スレッド化モデルをシングルスレッドで実行

11.2.3.1 生産-消費スレッド化モデル

図 11-3 に、生産スレッドと消費スレッドのペア間の相互作用の基本的な仕組みを示します。水平方向は時間を表します。各ブロックは、タスクユニットを表し、スレッドに割り当てられたバッファーを処理します。

各タスク間の隙間は、同期のオーバーヘッドを表します。括弧内の 10 進数は、バッファー・インデックスです。インテル® Core™ Duo プロセッサーの場合、生産スレッドがデータを 2 次キャッシュに格納できるため、消費スレッドは最小限のバス・トラフィックで処理を継続できます。

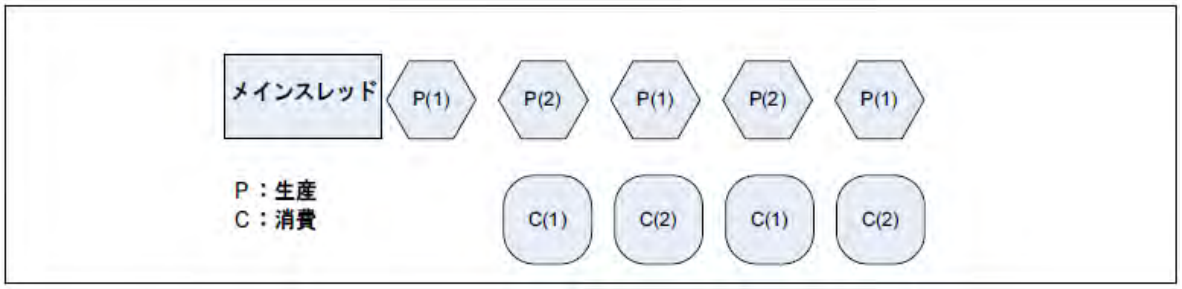


図 11-3 マルチコア・プロセッサー上で生産-消費スレッド化モデルを実行

例 11-2 に、同期によってバッファー・インデックスを通知する生産と消費のスレッド関数を実装する基本構造を示します。

例 11-2 生産および消費スレッドを実装する基本構造

```
(a) 生産スレッド関数の基本構造
void producer_thread()
{ int iter_num = workamount - 1; // ローカルコピーを作成
  int mode1 = 1; // 0 と 1 で 2 つのバッファーの利用を監視
  produce(buffs[0],count); // プレースホルダー関数
  while (iter_num--) {
      Signal(&signal1,1); // 開始する他のスレッドへ通知
      produce(buffs[mode1],count); // プレースホルダー関数
      WaitForSignal(&end1);
      mode1 = 1 - mode1; // 他のバッファーへ切り替え
  }
}
(b) 消費スレッドの基本構造
void consumer_thread()
{ int mode2 = 0; // 最初の反復はバッファー 0 で開始
  int iter_num = workamount - 1;
  while (iter_num--) {
      WaitForSignal(&signal1);
      consume(buffs[mode2],count); // プレースホルダー関数
      Signal(&end1,1);
      mode2 = 1 - mode2;
  }
  consume(buffs[mode2],count);
}
```

バス・トラフィックを最小限に抑え、共有 2 次キャッシュのないマルチコア・プロセッサーでも効果を得られるように、生産と消費モデルをインターレース方式で構造化でます。

このインターレース方式の生産−消費モデルでは、アプリケーション・スレッドの各スケジューリング単位は、生産タスクと消費タスクで構成されます。同じスレッドが 2 つ生成され、並列に実行されます。各スレッドのスケジューリングでは、まず生産タスクが開始され、生産タスクの完了後に消費タスクが開始されます。いずれのタスクも、同じバッファーを処理します。タスクの完了ごとに、一方のスレッドが他方のスレッドに信号を送り、特定のバッファーを使用するように対応するタスクに通知します。生産タスクと消費タスクはこのようにして、2 つのスレッドで並列実行されます。生産タスクによって生成されるデータが、同一コアの 1 次キャッシュまたは 2 次キャッシュ内に存在する限り、消費タスクはバス・トラフィックを発生させずにそのデータにアクセスできます。図 11-4 に、インターレース方式の生産−消費モデルのスケジューリングを示します。

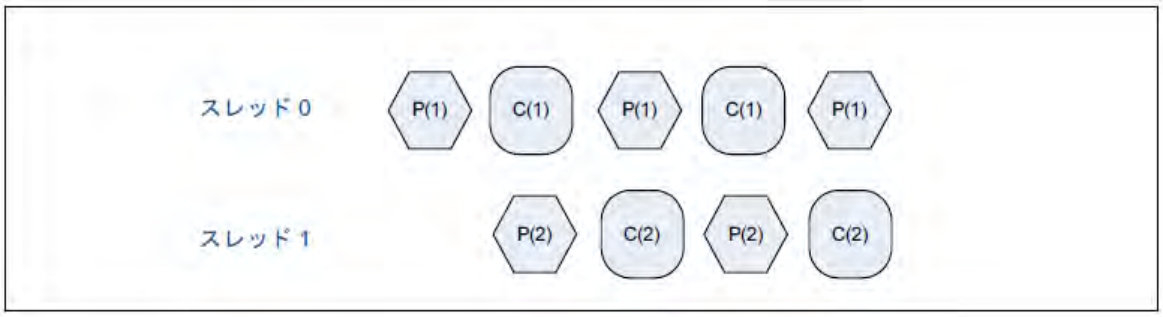


図 11-4 インターレース方式の生産-消費モデル

例 11-3 に、インターレース方式の生産-消費モデルで使用可能なスレッド関数の基本構造を示します。

例 11-3 インターレース方式の生産-消費モデルのスレッド関数

```
// マスタースレッドは最初の反復を開始
// 他のスレッドは1 つの反復を待機
void producer_consumer_thread(int master)
{
int mode = 1 - master; // スレッドとバッファー・インデックスを追跡
unsigned int iter_num = workamount >> 1;
unsigned int i=0;
iter_num += master & workamount & 1;
   if (master) // マスタースレッドは最初の反復を開始
   {
       produce(buffs[mode],count);
       Signal(sigp[1-mode1],1); // 生産タスクは後続のスレッドが開始できることを通知
       consume(buffs[mode],count);
       Signal(sigc[1-mode],1);
       i = 1;
   }
   for (; i < iter_num; i++)
   {
       WaitForSignal(sigp[mode]);
       produce(buffs[mode],count); // 生産タスクは他のスレッドに通知
       Signal(sigp[1-mode],1);
       WaitForSignal(sigc[mode]);
       consume(buffs[mode],count);
       Signal(sigc[1-mode],1);
   }
}
```

11.2.4 マルチスレッド・アプリケーション作成用のツール

マルチスレッド用のアプリケーション・プログラミング・インターフェイス (API) を直接使用してプログラミングすることだけがマルチスレッド・アプリケーションを作成する方法ではありません。マルチスレッド・アプリケーションを簡単に作成する機能を備えた (インテル® コンパイラーなどの) 新しいツールが提供されています。

最新のインテル® コンパイラーでは、次の機能が提供されています。

- OpenMP* ディレクティブによるマルチスレッド・コードの生成[17]
- ソースコードを変更することなくマルチスレッド・コードを自動生成[18]

17 OpenMP* ディレクティブをサポートするのは、インテル® コンパイラー 5.0 以降です。詳細は、http://software.intel.com (英語) を参照してください。

18 自動並列化をサポートするのは、インテル® コンパイラー 6.0 以降です。

11.2.4.1 OpenMP* ディレクティブによるプログラミング

OpenMP* は、アプリケーションの共有メモリー並列処理をサポートする Fortran と C/C++ 向けのコンパイラー・ディレクティブを提供します。これらのディレクティブは標準化されており、独自仕様ではなく、移植性があります。OpenMP*は、ディレクティブ・ベースの処理をサポートします。ディレクティブ・ベースの処理では、特別なプリプロセッサーまたはディレクティブの解釈を実装したコンパイラーを使用して、Fortran コメント、または C/C++ プラグマで表現される並列処理が解釈されます。ディレクティブ・ベースの利点を次に示します。

- 元のソースを変更せずにコンパイルできます。
- コードを段階的に変更することが可能です。これにより、元のコードのアルゴリズムを保持でき、迅速なデバッグが可能となります。
- 段階的なコードの変更は、シリアルバージョンの一貫性の維持に役立ちます。あるプロセッサー上でコードを実行すると、未変更のソースコードと同じ結果が得られます。
- スレッド・スケジューリングのインバランスを微調整するディレクティブを提供します。
- インテルの OpenMP* ランタイムは、手作業でコーディングされたマルチスレッディングのコードと比較して、スレッディングのオーバーヘッドを最小化できます。

11.2.4.2 コードの自動並列化

OpenMP* ディレクティブでは、シリアル・アプリケーションを迅速に並列アプリケーションに変換できます。ただし、アプリケーション・コード内の並列処理が可能な領域を識別し、コンパイラー・ディレクティブを追加する必要があります。インテル® コンパイラー 6.0 以降では、新しいオプションとして /Qparallel (Windows*)、-parallel (Linux* および macOS*) をサポートしています。このオプションでは、並列処理が含まれているループ構造を識別します。コンパイル中、コンパイラーは、並列処理のためにコードシーケンスを別々のスレッドに自動的に分割しようと試みます。プログラマーは、オプションを指定する以外の作業は必要ありません。

11.2.4.3 開発ツールのサポート

ソフトウェア開発向けにインテルから提供される各種ツールの詳細については、20.2.8.9 付録 A「アプリケーション・パフォーマンス・ツール」を参照してください。

11.3 最適化のガイドライン

この節では、マルチスレッド・アプリケーションをチューニングする最適化のガイドラインについて説明します。重要度の高い順で、次の 5 つの分野を取り上げます。

- スレッド間の同期
- バスの利用率
- メモリー最適化
- フロントエンドの最適化
- 実行リソースの最適化

ここでは、それぞれの分野に関連する慣例を説明しています。各分野のガイドラインについては、以降の節で詳しく説明します。

コーディングの推奨事項の大部分は、プロセッサー・コアによる性能スケーリングや HT テクノロジーによるスケーリングを改善します。どちらか一方にしか適用されない手法については、その都度注記します。

11.3.1 スレッド間の同期の主な慣例

スレッド同期のコストを最小限に抑えるための主な慣例を次に示します。

- 高速スピンループ中に PAUSE 命令を挿入し、ループの反復回数を最小限に抑え、システム全体の性能を向上させます。
- 複数のスレッドが取得可能なスピンロックを、2 つのスレッドしか 1 つのロックに書き込めないようなパイプライン化されたロックと置き換えます。2 つのスレッドが共有する変数に、1 つのスレッドしか書き込みを行う必要がない場合は、ロックを取得する必要はありません。
- スレッド・ブロッキング API を長いアイドルループ内で使用し、プロセッサーを解放します。
- 2 つのスレッド間で、スレッドごとのデータの「フォルス・シェアリング」を防止します。
- 各同期変数を 128 バイトで分離して単独に配置するか、独立したキャッシュラインに格納します。

詳細は、11.4 節「スレッド間の同期」を参照してください。

11.3.2 システムバス最適化の主な慣例

バス・トラフィックを管理すると、マルチスレッド・ソフトウェアおよび MP システム全体の性能で大きな効果が得られます。次に、高速なデータ・スループットや応答の実現に向けたシステムバス最適化の主な慣例を示します。

- データおよびコードの局所性を改善し、バスコマンド帯域幅を保持します。
- ソフトウェア・プリフェッチ命令の過度な使用は避け、自動ハードウェア・プリフェッチを機能させます。ソフトウェア・プリフェッチを過度に使用すると、バス利用が大幅かつ不必要に増加する可能性があります (不適切に使用されている場合)。
- オーバーラップする複数の連続したメモリー読み込みにより、実効キャッシュ・ミス・レイテンシーを改善することを検討します。
- フルサイズの書き込みトランザクションを使用して、より高いデータ・スループットを達成します。

詳細は、11.5 節「システムバスの最適化」を参照してください。

11.3.3 メモリー最適化の主な慣例

次に、メモリー操作を最適化する主な慣例を示します。

- キャッシュ・ブロッキングを使用して、データアクセスの局所性を改善します。HT テクノロジー対応のプロセッサーが対象の場合は、キャッシュサイズの 4 分の 1 から 2 分の 1 を目標とします。
- バスを共有する別々の物理プロセッサー上で実行されるスレッド間で、データの共有を最小限に抑えます。
- 各スレッドにおいて 64KB の倍数でオフセットされるデータ・アクセス・パターンを抑えます。
- HT テクノロジー対応のプロセッサーが対象の場合は、アプリケーション内の各スレッドのプライベート・スタックを調整し、それらのスタック間の挿入間隔が 64KB または 1MB の倍数でオフセットされないようにします (キャッシュラインの不必要な排出を避けるため)。
- HT テクノロジー対応のプロセッサーが対象の場合は、64KB または 1MB の倍数でオフセットされたメモリーアクセスを避けるため、同じアプリケーションの 2 つのインスタンスがロックステップで実行しているときに、インスタンス 1 つあたりのスタックオフセットを追加します。

詳細は、11.6 節「メモリー最適化」を参照してください。

11.3.4 実行リソース最適化の主な慣例

各物理プロセッサーは、専用の実行リソースを持っています。また、HT テクノロジーに対応した物理プロセッサー内の論理プロセッサーは、オンチップの特定の実行リソースを共有します。次に、実行リソース最適化の主な慣例を示します。

- 各スレッドを最適化し、最初に最適な周波数スケーリングを達成します。
- マルチスレッド・アプリケーションを最適化し、物理プロセッサー数に対する最適なスケーリングを達成します。
- 同一の物理プロセッサー・パッケージ内で 2 つのスレッドが実行リソースを共有している場合、オンチップの実行リソースを共有します。
- 各物理プロセッサー・パッケージ内でハードウェア・リソースの使用率を高めるため、HT テクノロジー対応のプロセッサーごとに、機能的に相関のないスレッドを追加することを検討します。

11.8 節「アフィニティーと共有プラットフォーム・リソースの管理」を参照してください。

11.3.5 一般性およびパフォーマンスの影響

次の 5 つの節では、各最適化手法について詳しく説明します。各節で説明する推奨事項は、局所的な影響および一般性の評価に関して、重要性の度合いが示されます。

優先度は、主観的で大まかなものです。これは、コーディングの形式やアプリケーション/スレッドのドメインによって変化します。高(H)、中(M)、低(L) という優先度を各推奨事項に含める目的は、推奨事項を適用したときに期待されるパフォーマンス向上の程度を相対的に評価するためです。

多岐にわたるアプリケーションのコード・インスタンスの頻度を予測できないため、影響する優先度をアプリケーション・レベルでパフォーマンス向上に直接相互に関連付けられません。一般性に対する優先度も主観的で大まかなものです。

これらのどのスケーリング要素にも影響を与えないコーディングの推奨事項は、一般に中 (M) または低 (L) として分類されます。

11.4 スレッド間の同期

複数のスレッドを持つアプリケーションは、動作を正しく行うために、同期を使用します。しかし、スレッド同期の実装が不適切であると、性能が大幅に低下する可能性があります。

スレッド同期のオーバーヘッドを削減する最適な慣例としては、アプリケーションの同期を減らすことから始めます。インテル® パフォーマンス・ツールを利用すると、各スレッドの実行タイムラインをプロファイルした上で、頻繁な同期のオーバーヘッドの発生によって性能が影響を受けている個所を検出できます。

スレッド同期で頻繁に使用されるコーディング手法およびオペレーティング・システム (OS) 呼び出しには、スピンウェイト・ループ、スピンロック、クリティカル・セクションなどがあります。状況に応じて最適な OS 呼び出しを選択し、並列処理を考慮して同期コードを実装することは、スレッド同期の処理コストを最小限に抑えるため重要です。

インテル® SSE3 では、複数のエージェント間でのマルチスレッド・ソフトウェアの同期を改善するように、2 つの命令 (MONITOR と MWAIT) を提供しています。MONITOR と MWAIT の最初の実装では、これらの命令がオペレーティング・システムで利用可能であるため、オペレーティング・システムはさまざまな領域でスレッド同期を最適化できます。例えば、オペレーティング・システムがシステム・アイドル・ループ (C0 ループ) で MONITOR と MWAIT を使用すると、消費電力を削減できます。また、C1 ループで MONITOR と MWAIT を使用すると、C1 ループの応答性を高めることができます。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 8 章を参照してください。

11.4.1 同期プリミティブの選択

スレッド間の同期では、同期プリミティブによって操作を保護しながら共有データを更新する手法が多く用いられます。利用可能なプリミティブは多数あります。同期プリミティブの選択に役立つガイドラインを紹介します。

- インクリメントや比較/交換など単純なデータ操作のアトミックな更新では、コンパイラー組込み関数や OS が提供するインターロック API を優先します。これらは、オーバーヘッドが多く複雑な同期プリミティブよりも効率的です。

  各種同期プリミティブの使用に関する詳細については、ホワイトペーパー「マルチスレッド・アプリケーションの開発のためのガイド」(https://www.isus.jp/products/psxe/intelguide-index/) を参照してください。

- 同期構造を実装するプリミティブを選択する際、インテル® パフォーマンス・ツールを使用すると、マルチスレッディング機能の正当性に関する問題や、マルチスレッド実行時におけるパフォーマンスへの影響を判断する上で効果的です。インテル® パフォーマンス・ツールの機能の詳細については、20.2.8.9 付録 A で説明しています。

表 11-1 は、マルチスレッド・アプリケーションに利用可能な 3 つのカテゴリーの同期オブジェクトの機能を比較するのに役立ちます。

表 11-1 同期オブジェクトのプロパティー

特性	**オペレーティング・システムの同期オブジェクト**	**軽量のユーザー同期**	**MONITOR/MWAIT ベースの同期オブジェクト**
取得と解放に要するサイクル (競合する場合)	数千または数万サイクル	数百サイクル	数百サイクル
消費電力	アイドル時はコアまたは論理プロセッサーを停止して節電が可能	PAUSE を使用した場合はある程度節電可能	PAUSE よりも節電可能
スケジューリングとコンテキスト・スイッチ	競合がある場合は OS スケジューラーに戻る (スピン・ループ・カウントの減少によってチューニング可能)	自発的には OS スケジューラーに戻らない	自発的には OS スケジューラーに戻らない
特権レベル	リング 0	リング 3	リング 0
その他	オブジェクトによっては、プロセス内同期やプロセス間通信を提供	複数のスレッドが同時に書き込む場合、同期変数へのアクセスをロックする必要があります。 その他の場合はロックなしで書き込み可能	軽量オブジェクトと同じ。 MONITOR/MWAIT をサポートするシステム上でのみ使用可能
推奨使用条件	• アクティブなスレッドの数がコア数より多い • 信号を数千サイクル待機 • プロセス間で同期	• アクティブなスレッドの数がコア数以下 • 競合がまれにしか発生しない • プロセス間の同期が必要	• 軽量オブジェクトと同じ • MONITOR/MWAIT が利用可能

11.4.2 短期間の同期

スレッドがほかのスレッドと同期する頻度と継続期間は、アプリケーションの特性によって異なります。同期ループで非常に高速な応答が必要な場合は、スピンウェイト・ループがアプリケーションで使用されます。

一般にスピンウェイト・ループは、あるスレッドが、別のスレッドが同期点に達するまで短時間待つ必要がある場合に使用されます。スピンウェイト・ループは、同期変数と事前定義値を比較するループから成ります。例 11-4(a) を参照してください。

スーパースケーラー・スペキュレーティブ・エグゼキューション・エンジンを搭載した最近のマルチプロセッサーでは、このようなループによって、スピン中のスレッドからの同時読み込み要求が複数発行されることがあります。これらの要求はアウトオブオーダーを通常実行し、各読み込み要求は追加のバッファーリソースを使用して割り当てられます。進行中のロードにワーカースレッドが書き込みを行ったことが検出されると、プロセッサーは、メモリーオーダー違反が発生しないよう保証する必要があります。未処理のメモリー操作の順序を正しく維持するには、プロセッサーに大きなペナルティーを課す必要があり、すべてのスレッドに影響が及ぶことが避けられません。

このペナルティーは、インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、インテル® Core™ Duo プロセッサーで発生します。ただし、これらのプロセッサーでのペナルティーは、インテル® Pentium® 4 プロセッサーやインテル® Xeon® プロセッサーと比べて小さめです。インテル® Pentium® 4 プロセッサー、インテル® Xeon® プロセッサーでは、このループを終了する性能のペナルティーは、約 25 倍以上です。

HT テクノロジー対応のプロセッサーの場合、スピンウェイト・ループは、プロセッサーの実行帯域幅を相当量消費することがあります。スピンウェイト・ループを実行している一方の論理プロセッサーは、もう一方の論理プロセッサーの性能に深刻な影響を与えることがあります。

例 11-4 スピンウェイト・ループと PAUSE 命令

(a) スピンウェイト・ループが最適化されていないと、ループの終了時に性能のペナルティーが発生します。最適化されていないスピンウェイト・ループは、計算処理に貢献せずに実行リソースを消費します。

```
do {
   // このループは最も高速にメモリーアクセスできます。
   // 他のワーカースレッドは、このスピンループが解決されるまで
   // sync_var を変更できません
} while( sync_var != constant_value);
```

(b) 高速スピンウェイト・ループに PAUSE 命令を挿入すると、スピニングスレッドおよびワーカースレッドに性能のペナルティーが生じません。

```
do {
_asm pause
   // このループのパイプラインを分断する。sync_var が参照される間、
   // 1つ以上のロード要求が行われることを防止し、
   // ワーカースレッドが sync_var を更新する時の
   // パフォーマンス・ペナルティーを避けます。スピンスレッドはループを終了します。
}
while( sync_var != constant_value);
```

(c) 同期変数の可用性を判別するために、「テスト、テストおよび設定」手法を使用しているスピンウェイト・ループ。この手法は、インテル® 64 アーキテクチャー・プロセッサーと IA-32 アーキテクチャー・プロセッサー上で動作するスピンウェイト・ループを作成する際に推奨されます。

```
Spin_Lock:
   CMP lockvar, 0 ; // ロックがフリーかチェック
   JE Get_lock
   PAUSE; // 短い遅延
   JMP Spin_Lock;
Get_Lock:
   MOV EAX, 1;
   XCHG EAX, lockvar; // ロックの取得を試みる
   CMP EAX, 0; // 取得できたかテスト
   JNE Spin_Lock;
Critical_Section:
   <クリティカル・セクションのコード>
   MOV lockvar, 0; // ロックを解放
```

ユーザー/ソース・コーディング規則 18 (影響 M、一般性 H): 高速スピンループ中に PAUSE 命令を挿入し、ループの反復回数を最小限に抑え、システム全体の性能を向上させます。

スピンウェイト・ループからの終了のペナルティーは、PAUSE 命令をループに挿入することで回避できます。PAUSE 命令は、ループ内に若干の遅延を挿入することで性能を改善し、同期変数へのストアを迅速に検出できる速度でメモリー読み込み要求を発行します。これにより、メモリーオーダー違反による長い遅延の発生を防ぐことができます。

単純なスピンウェイト・ループに PAUSE 命令を挿入する例を、例 11-4(b) に示します。PAUSE 命令は、すべてのインテル® 64 プロセッサーおよび IA-32 プロセッサーで互換性があります。Intel NetBurst® マイクロアーキテクチャーが採用される以前の IA-32 プロセッサーでは、PAUSE 命令は基本的に NOP 命令と同等です。PAUSE 命令を使用してスピンウェイト・ループを最適化するその他の例は、アプリケーション・ノート AP-949「インテル® Pentium® 4 プロセッサーおよびインテル® Xeon® プロセッサーにおけるスピンループの使用」(https://software.intel.com/sites/default/files/22/30/25602 (英語)) を参照してください。

PAUSE 命令を挿入すると、使用されるシステムリソースが少なくなるため、スピンウェイト中に消費される電力が大幅に減少する利点もあります。

11.4.3 スピンロックによる最適化

通常、スピンロックは、複数のスレッドで同期変数を変更する必要がある場合、およびその同期変数が誤って上書きされないようにロックで保護する必要がある場合に使用されます。しかしロックを解放すると、同時に複数のスレッドがそのロックを競って取得しようとすることがあります。こうしたスレッドのロック競合が発生すると、周波数、個別のプロセッサー数、HT テクノロジーに関連するパフォーマンス・ケーリングが大幅に減少する恐れがあります。

パフォーマンスのペナルティーを削減する方法の 1 つは、多数のスレッドが同じロックを競って取得する可能性を減らし、複数のスレッド間で共有する必要のあるデータをソフトウェア・パイプライン手法により処理することです。

複数のスレッドで 1 つのロックを競合させるのではなく、2 つのスレッドしか 1 つのロックに書き込みできないようにする必要があります。アプリケーションでスピンロックを使用する必要がある場合、ウェイトループに PAUSE 命令を挿入します。例 11-4(c) は、スピンウェイト・ループ内のロックの可用性を判別するための「テスト、テストおよび設定」手法の例を示しています。

ユーザー/ソース・コーディング規則 19 (影響 M、一般性 L): 複数のスレッドが取得可能なスピンロックを、2 つのスレッドしか 1 つのロックに書き込めないようなパイプライン化されたロックと置き換えます。2 つのスレッドが共有する変数に、1 つのスレッドしか書き込みを行う必要がない場合は、ロックを使用する必要はありません。

11.4.4 長期間の同期

長期間ロックを保持するスピンウェイト・ループでは、アプリケーションで次の 2 つのガイドラインに従う必要があります。

- スピンウェイト・ループの継続期間を、最小の繰り返し回数になるように維持します。
- アプリケーションで OS サービスを使用して、待機中のスレッドをブロックする必要があります。これにより、プロセッサーが解放され、その他の実行可能スレッドがプロセッサーまたは提供されている実行リソースを利用できるようになります。

HT テクノロジー対応のプロセッサーにおいて、一方の論理プロセッサーがアクティブで、もう一方の論理プロセッサーが非アクティブの場合は、オペレーティング・システムは HLT 命令を使用する必要があります。HLT 命令によって、アイドル状態の論理プロセッサーを停止状態に遷移できます。これにより、アクティブな論理プロセッサーは、物理パッケージのハードウェア・リソースをすべて利用できるようになります。この手法を持たないオペレーティング・システムは、アイドル状態の論理プロセッサー上で命令を実行する必要があり、繰り返しチェックされます。この「アイドルループ」が発生すると、他方のアクティブな論理プロセッサー上で処理を進行するために使用する実行リソースが消費されます。

アプリケーション・スレッドが長時間アイドル状態を維持する必要がある場合は、そのアプリケーションでスレッド・ブロッキング API、またはその他の手法を使用して、アイドル状態のプロセッサーを解放する必要があります。ここで

説明する手法は、従来の MP システムに適用されますが、HT テクノロジーに対応したプロセッサーでも、かなり高い効果が得られます。

一般に、オペレーティング・システムは、Sleep(dwMilliseconds)[19] などのタイミングサービスを提供しています。このような機能を使用すると、同期変数が頻繁にチェックされなくなります。

複数のワーカースレッドおよび制御ループを同期するもう 1 つの手法は、OS が提供するスレッド・ブロッキング API を使用することです。スレッド・ブロッキング API を使用すると、スピニングおよびウェイティング用のプロセッサー・サイクルが、制御スレッドであまり消費されなくなります。これにより、OS は、使用可能なプロセッサー上のワーカースレッドをスケジュールするより長い時間が得られます。さらに、スレッド・ブロッキング API を使用すると、OS が HLT 命令を使用して実装するシステム・アイドル・ループの最適化からも利点が得られます。

ユーザー/ソース・コーディング規則 20 (影響 H、一般性 M): スレッド・ブロッキング API を長いアイドルループ内で使用し、プロセッサーを解放します。

実行可能なスレッドの数が MP システム内のプロセッサーの数よりも少ない場合は、スピンウェイト・ループを従来の MP システムで使用してもあまり問題はありません。しかし、アプリケーション内のスレッド数が、シングルプロセッサー・システムまたはマルチプロセッサーのプロセッサー数よりも多くなると予測される場合、スレッド・ブロッキング API を使用してプロセッサー・リソースを解放します。1 つの制御スレッドを使用して複数のワーカースレッドを同期するマルチスレッド・アプリケーションでは、ワーカースレッド数をシステム内のプロセッサー数以下に制限し、制御スレッド内でのスレッド・ブロッキング API の使用を検討する必要があります。

11.4.4.1 スレッド同期におけるコーディングの落とし穴の回避策

複数スレッド間の同期において、個別のプロセッサー数や物理プロセッサーあたりの論理プロセッサー数に応じてパフォーマンスのスケーリングを高めるには、設計および実装を慎重に行う必要があります。1 つの手法で、どのような同期状況にも適用できる万能な解決策はありません。

次の例 11-5(a) に示す疑似コードは、制御スレッドに対するポーリングループの実装例を示しています。一般に、実行可能なワーカースレッドが 1 つしか存在しない場合は、Sleep(0) などのタイミング・サービス API を呼び出しても、スレッド同期のコストを最小限に抑える効果は得られません。制御スレッドは、依然として、高速スピニングループのように動作するため、唯一の実行可能なワーカースレッドは、実行リソースをスピンウェイト・ループと共有する必要があります (HT テクノロジー対応の同一の物理プロセッサー上で両方が実行されている場合)。実行可能なワーカースレッドが複数存在している場合、Sleep(0) などのスレッド・ブロッキング API を呼び出すと、スピンウェイト・ループを実行しているプロセッサーが解放され、スピニングループの代わりに別のワーカースレッドによってプロセッサーが使用されます。

一般に、ワーカースレッドが完了に長時間を要する場合、ワーカースレッドの完了を待機している制御スレッドは、スレッド・ブロッキング API やタイムサービスを使用して、スレッド同期を実装できます。例 11-5(b) は、スレッド同期で、制御スレッドのオーバーヘッドを削減する例を示しています。

[19] Sleep() API は、プロセッサーの解放を保証しないため、スレッド・ブロッキングではありません。例 11-5(a) は、Sleep(0) の使用例を示しています。Sleep(0) では、必ずしもプロセッサーが別のスレッドに解放されるとは限りません。

例 11-5 スピンウェイト・ループを使用するコーディングの落とし穴

(a) スピンウェイト・ループは、不適切にプロセッサーを解放しようとします。唯一のワーカースレッドと制御スレッドが同一の物理プロセッサー・パッケージ上で実行されている場合、スピンウェイト・ループでは、パフォーマンスのペナルティーが発生します。

```
// 1つのワーカースレッドのみが実行されており、
// 制御ループはワーカースレッドの終了を待つ
ResumeWorkThread(thread_handle);
While (!task_not_done ) {
   Sleep(0) // 即座にスピンループに戻る
…
}
```

(b) ポーリングループは、プロセッサーを正常に解放します。

```
// 1つのワーカースレッドを実行し、完了を待機させます。
ResumeWorkThread(thread_handle);
While (!task_not_done ) {
   Sleep(FIVE_MILISEC)

// このプロセッサーはわずかの間解放されます。
// その間他のスレッドがプロセッサーを利用できます。
…
}
```

通常、スレッドを同期するときは、OS 関数呼び出しを慎重に使用する必要があります。OS でサポートされるスレッド同期オブジェクト (クリティカル・セクション、ミューテックス、セマフォーなど) を使用するときは、クリティカル・セクションなどの最小の同期オーバーヘッドを持つ OS サービスを使用すべきです。

11.4.5 変更されたデータの共有とフォルス・シェアリングの防止

プロセッサー/コアのトポロジーと特定のマイクロアーキテクチャーにおけるキャッシュトポロジーによっては、一方のコアで実行中のスレッドが、他方のコアの 1 次キャッシュに変更状態で存在するデータに対して読み出しまたは書き込みを試みると、変更されたデータの共有によってパフォーマンスのペナルティーが発生します。この場合、変更されたキャッシュラインがメモリーに排出され、他方のコアの 1 次キャッシュに再度読み込まれます。このようなキャッシュライン転送のレイテンシーは、1 次キャッシュまたは 2 次キャッシュ内のデータを直接使用する場合よりもはるかに大きくなります。

フォルス・シェアリングが発生するのは、別のスレッドが使用する異なるデータと、スレッドが使用するデータが同じキャッシュライン上に存在する場合です。このような状況では、プラットフォーム上の論理プロセッサー/コアのトポロジーに応じて、パフォーマンス上の遅延が発生することもあります。

異なる物理プロセッサー上の論理プロセッサーでスレッドが実行されていると、フォルス・シェアリングによってパフォーマンスのペナルティーが発生する場合があります。HT テクノロジー対応のプロセッサーでは、異なるコア上、異なる物理プロセッサー上、または物理プロセッサー・パッケージ内の 2 つの論理プロセッサー上で 2 つのスレッド

が実行されていると、フォルス・シェアリングによってパフォーマンスのペナルティーが発生します。異なるコア上または異なる物理プロセッサー上で 2 つのスレッドが実行されている場合、パフォーマンスのペナルティーは、キャッシュ整合性を維持するためのキャッシュ排出によって生じます。物理プロセッサー・パッケージ内の 2 つの論理プロセッサー上で 2 つのスレッドが実行されている場合、性能のペナルティーは、メモリー・オーダー・マシンクリアー条件によって発生します。

マルチスレッド化されたソフトウェアでフォルス・シェアリングのペナルティーを防ぐ一般的な方法は、"フォルス・シェアリングのしきい値" サイズに応じてクリティカルなデータを分離して配置するか、アライメントの粒度でロックします。次のステップで、ソフトウェアは各世代のインテル® プロセッサーに適用できる "フォルス・シェアリングのしきい値" を決定できます。

1. プロセッサーが CLFLUSH 命令をサポートする場合、つまり PUID.01H:EDX.CLFLUSH[ビット 19] =1 のケース。

   CLFLUSH ラインサイズ、CPUID.01H:EBX[15:8] の整数値を "フォルス・シェアリングのしきい値" として使用します。

2. CLFLUSH ラインサイズが利用できない場合、以下に示す CPUID リーフ 4 を使用します。

   CPUID リーフ 4 のサブリーフを介して報告される有効なキャッシュタイプで最も大きなシステム・コヒーレンシーのラインサイズを評価することで、"フォルス・シェアリングのしきい値" を決定します。それぞれのサブリーフ n は、システムのコヒーレンシー・ラインサイズに関連する (CPUID.(EAX=4, ECX=n):EBX[11:0] + 1) です。

3. CLFLUSH ラインサイズと CPUID リーフ 4 のどちらも利用できない場合、ソフトウェアは次のいずれかの方法で "フォルス・シェアリングのしきい値" を選択します。

   a. CPUID リーフ 2 のディスクリプター・テーブルを取得し、利用可能なディスクリプターのエントリーから選択します。
   b. プラットフォームで利用可能なファミリー/モデル固有メカニズム、または既知のファミリー/モデル固有値。
   c. 安全なデフォルト値を 64 バイトとします。

ユーザー/ソース・コーディング規則 21 (影響 H、一般性 M): キャッシュライン、またはセクター内のフォルス・シェアリングに注意します。"フォルス・シェアリングのしきい値" ほど小さくないアライメントの粒度で分離して、重要なデータやロックを割り当てます。

パラメーターの共通ブロックが親スレッドから複数のワーカースレッドに渡される場合、各ワーカースレッドが頻繁にアクセスされるプライベート・コピー (それぞれのコピーは "フォルス・シェアリングのしきい値" の倍数に配置) をパラメーター・ブロックに作成することが理想的です。

11.4.6 共有同期変数の配置

Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサーでは通常、バス読み込みで 128 バイトがフェッチされ、キャッシュに格納されます。そのため、キャッシュデータの排出を最小限に抑えるための最適な挿入間隔は、128 バイトとなります。フォルス・シェアリングを防止するため、同期変数とシステム・オブジェクト (クリティカル・セクションなど) は、128 バイト領域内に単独で存在するように割り当て、128 バイト境界にアライメントする必要があります。

例 11-6 は、MP システムでキャッシュ整合性を維持するために必要なバス・トラフィックを最小限に抑える方法を示しています。この手法は、HT テクノロジー対応または非対応のプロセッサーを使用する MP システムにも適用されます。

例 11-6 同期変数およびレギュラー変数の配置

```
int regVar;
int padding[32];
int SynVar[32*NUM_SYNC_VARS];
int AnotherVar;
```

インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、インテル® Core™ Duo プロセッサー、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、同期変数を独立したキャッシュラインに単独で格納して、フォルス・シェアリングを回避します。ソフトウェアでは、同期変数がページ境界をまたぐことを許可してはなりません。

ユーザー/ソース・コーディング規則 22 (影響 M、一般性 ML): 各同期変数を 128 バイトで分離して単独に配置するか、独立したキャッシュラインに格納します。

ユーザー/ソース・コーディング規則 23 (影響 H、一般性 L): スピンロック変数は、キャッシュライン境界をまたいで配置してはなりません。

以下の場合は、コードレベルでフォルス・シェアリングに関する考慮が必要です。

- 同じキャッシュラインに配置され、異なるスレッドによって書き込まれるグローバルデータ変数とスタティック・データ変数。
- 異なるスレッドによって動的に割り当てられるオブジェクトが、キャッシュラインを共有する可能性がある場合。一方のスレッドによって局所的に使用される変数が、他方のスレッドとキャッシュラインを共有しない方法で割り当てられることを確認する必要があります。

同期変数のアライメントを強制し、キャッシュラインの共有を回避するための別の手法としては、データ構造の宣言時にコンパイラー・ディレクティブを使用します。例 11-7 を参照してください。

例 11-7 キャッシュラインを共有しない同期変数の宣言

```
__declspec(align(64)) unsigned __int64 sum;

struct sync_struct {…};

__declspec(align(64)) struct sync_struct sync_var;
```

フォルス・シェアリングを回避するためのその他の手法を示します。

- データ構造内の各種の変数を整理します (コンパイラーがデータ変数に与えるレイアウトは、ソースコードでの配置と異なる場合があります)。
- 各スレッドが変数セットの独自のコピーを使用する必要がある場合は、以下を利用して変数を宣言します。
  - OpenMP* 使用時は、threadprivate ディレクティブ
  - Microsoft* コンパイラー使用時は、__declspec(thread) 修飾子

- オブジェクトを自動的に割り当てるマネージド環境では、オブジェクト・アロケーターとガベージコレクターは、2つのオブジェクトによるフォルス・シェアリングが発生しないように、メモリー内でオブジェクトをレイアウトする必要があります。
- 1 つのスレッドのみが各オブジェクト・フィールドと近くのオブジェクト・フィールドに書き込めるクラスを提供して、フォルス・シェアリングを回避します。

この節で説明されている推奨事項を、配置がまばらなデータレイアウトの推奨と理解してはいけません。データレイアウトに関する推奨事項は必要時にのみ適用して、ワークセットのサイズが不用意に拡大することを避けるべきです。

11.5 システムバスの最適化

システムバスは、バス・エージェント (論理プロセッサーなど) からの要求を処理して、メモリー・サブシステムからデータやコードをフェッチします。メモリーからフェッチされるデータ・トラフィックがパフォーマンスに及ぼす影響は、ワークロードの特性、メモリーアクセスにおけるソフトウェア最適化の度合い、ソフトウェア・コードにおける局所性の改善によって異なります。ワークロードのメモリー・トラフィックの特性評価を行う手法については、付録 A で説明します。局所性の改善に関する最適化ガイドラインについては、3.6.10 節「局所性の改善」と 9.5.11 節「ハードウェア・プリフェッチとキャッシュ・ブロッキング」で説明しています。

第 3 章と第 7 章で説明した手法を利用すると、バスシステムがシングルスレッド環境に対してサービスを提供しているプラットフォーム上でアプリケーションの性能が向上します。マルチスレッド環境の場合、バスシステムは通常、多数の論理プロセッサーにサービスを提供しているため、各論理プロセッサーが独立してバス要求を発行することがあります。そのため、局所性の改善、バス帯域幅の保持、間隔の広いキャッシュミスでの遅延の削減に関する手法は、プロセッサー・スケーリングのパフォーマンスに大きな影響を与える場合があります。

11.5.1 バス帯域幅の保持

マルチスレッディング環境では、複数のバス・エージェントから要求されたメモリー・トラフィックによってバス帯域幅が共有される場合があります (バス・エージェントは、複数の論理プロセッサーや複数のプロセッサー・コアである場合があります)。バス帯域幅を保持すると、プロセッサーのスケーリング・パフォーマンスを高めることができます。また、大きなストライドのキャッシュミスが大量にある場合、一般的に実効バス帯域幅が減少します。大きなストライドのキャッシュミス (または DTLB ミス) を減らすと、それに起因する帯域幅が減少する問題が緩和されます。

バスコマンド帯域幅を保持する方法の 1 つは、コードとデータの局所性を改善することです。データの局所性を改善すると、キャッシュラインの排出回数およびデータフェッチの要求回数が減少します。この手法はまた、システムメモリーからの命令フェッチの回数を減少させます。

ユーザー/ソース・コーディング規則 24 (影響 M、一般性 H): データおよびコードの局所性を改善し、バスコマンド帯域幅を保持します。

プロファイルに基づく最適化をサポートするコンパイラーを使用して、頻繁に実行されるコードパスをキャッシュ内に留めることで、コードの局所性が改善されます。その結果、命令フェッチの回数が少なくなります。また、ループ・ブロッキングによりデータの局所性も改善できます。その他の局所性を改善する手法は、マルチスレッディング環境にも適用してバス帯域幅を保持できます (9.5 節「プリフェッチを使用したメモリーの最適化」を参照してください)。

システムバスは多くのバス・エージェント (論理プロセッサーやプロセッサー・コア) 間で共有されるため、ソフトウェアのチューニングではバスが飽和状態になる兆候を認識すべきです。効果的な手法の 1 つは、バス読み出しトラフィックのキューの深さを調べることです。バスキューが深い場合、局所性を改善してキャッシュの使用効率を高めると、その他の手法 (挿入するソフトウェア・プリフェッチの増加や、重複したバス読み出しによるメモリー・レイテンシーのマスクなど) よりもパフォーマンス上のメリットがあります。バスが飽和しない状態でソフトウェアを動作させる作業ガイドラインとして、バス読み出しキューの深さが 5 を大きく下回っているかどうかを確認する方法があります。

一部の MP プラットフォームとワークステーション・プラットフォームでは、2 つのシステムバスを備え、各バスが 1 つ以上の物理プロセッサーに対応するチップセットを搭載しています。バス帯域幅の保持に関する上記のガイドラインは、それぞれのバスドメインにも適用されます。

11.5.2 バスとキャッシュとの相互作用について

ワーキングセット全体が 2 次キャッシュに収まらないデータセットや、使用する帯域幅がバスの能力を超えたデータセットがある場合、そのコード領域の並列化には注意を払う必要があります。インテル® Core™ Duo プロセッサーでは、1 つのスレッドのみが 2 次キャッシュやバスを使用する場合、そのスレッドの処理は他方のコアによって干渉

されないため、キャッシュやバスシステムを最大限に利用できます。ただし、2 つのスレッドが 2 次キャッシュを同時に使用する場合、以下の条件のいずれかが当てはまると、パフォーマンスが低下する可能性があります。

- ワーキングセットの合計が 2 次キャッシュのサイズよりも大きい。
- バス使用量の合計がバスの能力よりも多い。
- 両者が 2 次キャッシュ内の同じセットに頻繁にアクセスし、少なくとも 1 つのスレッドがキャッシュラインへの書き込みを行う。

マルチスレッド・ソフトウェアでこのような落とし穴を回避するには、一度に 1 つのスレッドのみが 2 次キャッシュにアクセスする並列処理や、2 次キャッシュとバス使用量が制限を超えない並列処理を検討する必要があります。

11.5.3 過度なソフトウェア・プリフェッチを避ける

インテル® Pentium® 4 プロセッサーとインテル® Xeon® プロセッサーは、自動ハードウェア・プリフェッチを備えています。自動ハードウェア・プリフェッチでは、事前の参照パターンに基づいて、データと命令をユニファイド 2 次キャッシュに格納できます。ほとんどの場合、ハードウェア・プリフェッチは、ソフトウェア・プリフェッチの明確な介入がなくても、システム・メモリー・レイテンシーを削減できます。また、コードのデータ・アクセス・パターンを調整し、自動ハードウェア・プリフェッチの特性を活かして、局所性の改善やメモリー・レイテンシーのマスクを行うことが推奨されます。インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーも、先進的なハードウェア・プリフェッチ機構を提供します。旧世代のハードウェア・プリフェッチ機構を利用できるデータ・アクセス・パターンは、通常、新しい世代のハードウェア・プリフェッチ機構にも利用できます。

ソフトウェア・プリフェッチ命令を過度または無計画に使用すると、間違いなくパフォーマンスのペナルティーが生じます。これは、システムバスのコマンドとデータ帯域幅が無駄に使用されるためです。

ソフトウェア・プリフェッチを使用すると、プロセッサー・コアが必要とするデータをハードウェア・プリフェッチがフェッチする時期が遅れます。また、ソフトウェア・プリフェッチは、実行リソースを消費するため、実行がストールします。状況によっては、ソフトウェア・プリフェッチの削減または使用しないことを検討し、ハードウェア・プリフェッチ機構を有効活用する方が効果的です。ソフトウェア・プリフェッチ命令を使用するガイドラインは、第 3 章で説明しています。自動ハードウェア・プリフェッチを活用する手法については、第 9 章で説明しています。

ユーザー/ソース・コーディング規則 25 (影響 M、一般性 L): ソフトウェア・プリフェッチ命令の過度な使用は避け、自動ハードウェア・プリフェッチを機能させます。ソフトウェア・プリフェッチを過度に使用すると、バス利用が大幅かつ不必要に増加する可能性があります (不適切に使用されている場合)。

11.5.4 キャッシュミスの実効レイテンシーを改善

キャッシュミスによるシステムメモリーのアクセスレイテンシーは、バス・トラフィックの影響を受けます。これは、バス読み込み要求は、その他のバス・トランザクション要求とともに調整されるためです。未処理のバス・トランザクション数を削減すると、実効メモリー・アクセス・レイテンシーが改善されます。

メモリー読み込みトランザクションの実効レイテンシーを改善する手法として、複数のバス読み込みをオーバーラップすることにより、分散した読み込みのレイテンシーを減らす方法があります。データの局所性がほとんどない、またはメモリー読み込みと他のバス・トランザクションを調整する必要がある状況では、複数のメモリー読み込みを連続して発行し、未処理のメモリー読み込みトランザクションをオーバーラップすることにより、分散したメモリー読み込みの実効レイテンシーを改善できます。一般に、連続したバス読み込みの平均レイテンシーは、他のバス・トランザクションとともに間隔を開けて配置された (分散した) 読み込みの平均レイテンシーよりも低くなります。キャッシュミスの完全な遅延が避けられないのは、最初のメモリー読み込みだけです。

ユーザー/ソース・コーディング規則 26 (影響 M、一般性 M): オーバーラップする複数の連続したメモリー読み込みにより、実効キャッシュ・ミス・レイテンシーを改善することを検討します。

最終レベルキャッシュで連続するキャッシュミスを引き起こすアクセス間隔が、自動ハードウェア・プリフェッチを引き起こすしきい値の距離よりも狭くなるようにデータ・アクセス・パターンを調整できれば、実効メモリー・レイテンシーを削減するもう 1 つの手法が利用可能となります。9.5.3 節「ハードウェア・プリフェッチで実効レイテンシーを削減する例」を参照してください。

ユーザー/ソース・コーディング規則 27 (影響 M、一般性 M): 最終レベルキャッシュの連続するキャッシュミスの間隔が 64 バイトに近づくように、メモリー参照の順序付けを調整することを検討します。

11.5.5 フルサイズの書き込みトランザクションによる高データレートの実現

バスを介してトランザクションが書き込みを行う場合、64 バイトのキャッシュライン・サイズすべて、またはその一部分が物理メモリーへ書き込まれます。後者は、パーシャル書き込みと呼ばれます。一般に、ライトバック (WB) メモリーアドレスではフルサイズの書き込みとなり、ライトコンバイン (WC) またはキャッシュ不可能 (UC) メモリーアドレスではパーシャル書き込みとなります。キャッシュされた WB ストア操作と WC ストア操作では、6 つの WC バッファー (64 バイト幅) を使用して、これらの書き込みトランザクションのトラフィックが管理されます。WC バッファーへの書き込みがすべて完了する前に、競合するトラフィックによってバッファーがクローズされると、単一の 64 バイト書き込みトランザクションではなく、一連の 8 バイトのパーシャル・バス・トランザクションが発生します。

ユーザー/ソース・コーディング規則 28 (影響 M、一般性 M): フルサイズの書き込みトランザクションを使用して、より高いデータ・スループットを達成します。

多くの場合、WC メモリーへのパーシャル書き込みが複数発生する場合、ソフトウェアによるライト・コンバイニング手法を使用することでそれらをフルサイズの書き込みに結合し、WB ストア・トラフィックとの競合から WC ストア操作を分離できます。ソフトウェアによるライト・コンバイニングを実装するには、WC 属性が指定されたメモリーへのキャッシュ不可能書き込みが、1 次データキャッシュに収まる小さな一時バッファー (WB タイプ) に書き込まれるようにします。一時バッファーが一杯になると、アプリケーションは最終的な WC デスティネーションに一時バッファーの内容をコピーします。

バス上でパーシャル書き込みを処理すると、システムメモリーに対する有効なデータレートが、システムバス帯域幅の 1/8 に削減されます。

11.6 メモリー最適化

効率的なキャッシュ利用は、メモリー最適化の重要な要素です。キャッシュを効率的に利用するには、以下の課題を考慮する必要があります。

- キャッシュ・ブロッキング
- 共有メモリー最適化
- 64KB エイリアス・データ・アクセスの排除
- 1 次キャッシュの過度な排出防止

11.6.1 キャッシュ・ブロッキングのテクニック

ループ・ブロッキングは、キャッシュミスの削減とメモリーアクセスのパフォーマンス向上に役立ちます。ループ・ブロッキング手法を適用する場合、ブロックサイズを適切に選択することが重要です。ループ・ブロッキングは、HT テクノロジー対応または非対応のプロセッサー上で動作するマルチスレッド・アプリケーションだけではなく、シングルスレッド・アプリケーションにも適用できます。ループ・ブロッキングでは、メモリー・アクセス・パターンを目的のキャッシュサイズに効果的に収まるブロックに変換します。

HT テクノロジー対応のインテル® プロセッサーの場合、ユニファイド・キャッシュ向けのループ・ブロッキング手法では、目的のキャッシュサイズの 2 分の 1 を超えないブロックサイズを選択します (2 つの論理プロセッサーがキャッシュを共有している場合)。ループ・ブロッキングのブロックサイズの上限は、目的のキャッシュサイズを、物理プ

ロセッサー・パッケージで利用可能な論理プロセッサー数で割ることで決定します。一般に、キャッシュ・ブロッキングで使用するソースまたはターゲットバッファーに含まれないデータにアクセスするには、複数のキャッシュラインが必要となります。そのため、ブロックサイズは、ターゲットのキャッシュの 4 分の 1 から 2 分の 1 の間で選択します (第 3 章「一般的な最適化ガイドライン」を参照)。

ソフトウェアは、CPUID のキャッシュ・パラメーター・リーフを参照して、特定のキャッシュをどの論理プロセッサーのサブセットが共有しているか検出できます (第 9 章「キャッシュ利用の最適化」を参照)。したがって、特定のキャッシュをアクセスする論理プロセッサー数でそのキャッシュの合計サイズを割ったものを、ブロックサイズの上限に設定することで、上記のガイドラインの適用を拡張できます。その結果、特定のキャッシュをアクセスするすべての論理プロセッサーがそのキャッシュを同時に使用できるようになります。この手法は、マルチタスク・ワークロードの一部として使用されるシングルスレッド・アプリケーションにも適用できます。

ユーザー/ソース・コーディング規則 29 (影響 H、一般性 H): キャッシュ・ブロッキングを使用して、データアクセスの局所性を改善します。HT テクノロジー対応のインテル®プロセッサーが対象の場合、キャッシュサイズの 4 分の 1 から 2 分の 1 を目標とします。または、特定のキャッシュをアクセスするすべての論理プロセッサーがそのキャッシュを同時に共有可能なブロックサイズを目標とします。

11.6.2 共有メモリー最適化

多くの場合、個々のプロセッサー間でキャッシュの整合性を維持するには、プロセッサー周波数より大幅に低いクロックレートで動作するバスを介して、データを転送する必要があります。

11.6.2.1 物理プロセッサー間でのデータ共有の最小化

一般に、2 つのスレッドが 2 つの物理プロセッサー上でデータを共有して実行する場合、共有データに対して読み込みまたは書き込みを行うには、いくつかのバス・トランザクション (スヌーピング、所有権の変更要求、バスを介したデータのフェッチなど) が必要となります。そのため、大量の共有メモリーにアクセスするスレッドは、プロセッサーのスケーリング・パフォーマンスを低下させる可能性があります。

ユーザー/ソース・コーディング規則 30 (影響 H、一般性 M): バスを共有する別々のバス・エージェント上で実行されるスレッド間で、データの共有を最小限に抑えます。プラットフォームが複数のバスドメインで構成されている場合、バスドメイン間でのデータ共有も最小限に抑える必要があります。

データの共有を最小化する手法の 1 つは、ローカルスタック変数にデータをコピーすることです (長時間繰り返しアクセスされる場合)。必要であれば、共有メモリー・ロケーションに書き戻すとき、複数のスレッドからの結果を結合します。この方法では、共有データアクセスの同期で使用する時間を最小限に抑えられます。

11.6.2.2 バッチ方式の生産-消費モデル

スレッド化された生産-消費モデル (図 11-5 を参照) のメリットは、共有 2 次キャッシュを使って生産と消費との間でデータを共有しながら、バス・トラフィックを最小限に抑えられることです。インテル® Core™ Duo プロセッサーでワークバッファーが 1 次キャッシュ内に収まる場合、最適なパフォーマンスを得るには生産タスクと消費タスクの再順序付けが必要となります。これは、L2 から L1 にデータをフェッチする方が、一方のコアのキャッシュラインを無効化してバスからフェッチするよりも、はるかに高速であるためです。

図 11-5 に示すバッチ方式の生産-消費モデルを利用すると、標準の生産-消費モデルで小型のワークバッファーを使用する際の欠点も克服できます。バッチ方式の生産-消費モデルでは、各スケジューリング単位で 2 つまたは 3 つの生産タスクがまとめられ、それぞれ指定のバッファーで処理されます。まとめられるタスクの数は、ワーキングセット全体が 1 次キャッシュより大きく、2 次キャッシュよりも小さくなることを基準として決定されます。

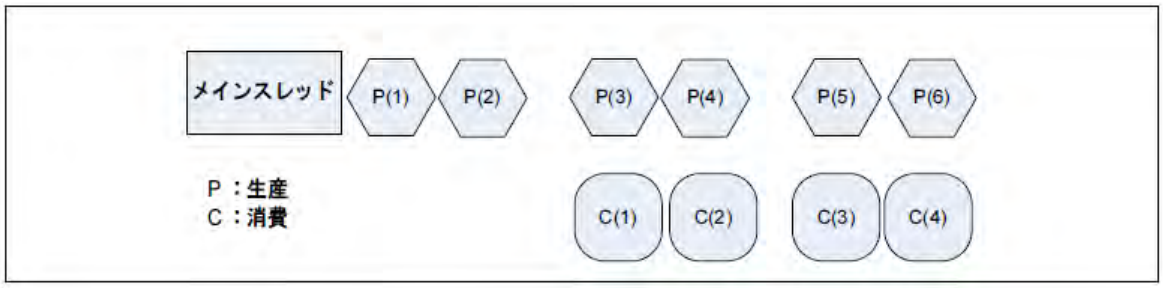


図 11-5 バッチ方式の生産-消費モデル

例 11-8 に、生産および消費のスレッド機能のバッチ方式での実装を示します。

例 11-8 生産および消費スレッドのバッチ方式での実装

```
void producer_thread()
{ int iter_num = workamount - batchsize;
   int mode1;
for (mode1=0; mode1 < batchsize; mode1++)
{ produce(buffs[mode1],count); }
   while (iter_num--)
   { Signal(&signal1,1);
      produce(buffs[mode1],count); // プレースホルダー関数
      WaitForSignal(&end1);
      mode1++;
      if (mode1 > batchsize)
         mode1 = 0;
   }
}
void consumer_thread()
{ int mode2 = 0;
  int iter_num = workamount - batchsize;
  while (iter_num--)
  { WaitForSignal(&signal1);
     consume(buffs[mode2],count); // プレースホルダー関数
     Signal(&end1,1);
     mode2++;
     if (mode2 > batchsize)
        mode2 = 0;
  }
  for (i=0;i<batchsize;i++)
  { consume(buffs[mode2],count);
     mode2++;
     if (mode2 > batchsize)
        mode2 = 0;
  }
}
```

11.6.3 64KB エイリアスのデータアクセスを排除

64KB エイリアスの条件については、第 3 章で詳しく説明しています。64KB エイリアスの条件に該当するメモリーアクセスでは、1 次データキャッシュが過度に排出される可能性があります。一般に、各スレッドに起因する 64KB エイリアス・データ・アクセスを排除すると、周波数スケーリングの改善に役立ちます。また、アプリケーションで HT テクノロジーを活用している場合、1 次データキャッシュを効率良く利用できます。

ユーザー/ソース・コーディング規則 31 (影響 H、一般性 H): 各スレッドで 64KB の倍数でオフセットされるデータ・アクセス・パターンを最小限にします。

インテル® Pentium® 4 プロセッサーのパフォーマンス監視イベントを使用すると、64KB エイリアス・データ・アクセスを検出できます。インテル® Pentium® 4 プロセッサーのパフォーマンス・メトリックは、付録 B で説明します。これらのメトリックは、インテル® VTune™ プロファイラーを使用してアクセスされるイベントに基づいています。

64KB エイリアスに関連するパフォーマンスのペナルティーは、主に HT テクノロジーまたは Intel NetBurst® マイクロアーキテクチャーを実装したプロセッサーに適用されます。次の節では、HT テクノロジー対応のプロセッサー上で動作するマルチスレッド・アプリケーションに適用されるメモリー最適化手法について説明します。

11.7 フロントエンドの最適化

ユニファイド 2 次キャッシュが 2 つのプロセッサー・コアによって共有されるデュアルコア・プロセッサー (インテル® Core™ Duo プロセッサーおよびインテル® Core™ マイクロアーキテクチャー・ベースのプロセッサー) の場合、マルチスレッド・ソフトウェアでは、ユニファイド・キャッシュからコードをフェッチする 2 つのスレッドが原因でコード・ワーキング・セットが増加するため、フロントエンドとキャッシュの最適化において考慮する必要があります。インテル® Core™ マイクロアーキテクチャー・ベースのクアッドコア・プロセッサーの場合、インテル® Core™2 Duo プロセッサーに適用される考慮事項がクアッドコア・プロセッサーにも適用されます。

11.7.1 過度なループアンロールの回避

ループをアンロールすると分岐の数が少なくなり、アプリケーション・コードの分岐予測が改善されます。ループアンロールの詳細については、第 3 章で説明しています。ループアンロールは慎重に行う必要があります。分岐予測が改善することの利点、およびループストリーム検出器 (LSD) の利用率が低下するコストについて検討する必要があります。

ユーザー/ソース・コーディング規則 32 (影響 M、一般性 L): 過度なループアンロールを回避し、LSD が効率良く動作するようにします。

11.8 アフィニティーと共有プラットフォーム・リソースの管理

現代の OS は、論理プロセッサーや NUMA (Non-Uniform Memory Access) メモリー・サブシステムなど特定の共有リソースをアプリケーションが管理できるように API やデータ構造 (アフィニティー・マスクなど) を提供しています。

マルチスレッド・ソフトウェアでは、表 11-2 の推奨事項を考慮した上で、アフィニティー API の使用を検討すべきです。

表 11-2 設計時におけるリソース管理の選択

ランタイム環境	スレッド・スケジューリング/プロセッサー・アフィニティーの考慮事項	メモリー・アフィニティーの考慮事項
シングルスレッド・アプリケーション	OS スケジューラーにスケジュールを管理させることで、システムの応答性とスループットに関する OS スケジューラーの目標をサポートします。OS は、エンドユーザーがランタイム固有の環境を最適化する機能を提供します。	該当しません。OS に任せます。
次の条件に当てはまるマルチスレッド・アプリケーション: **i) システム上のすべてのプロセッサー・リソースを使用するわけではない。** **ii) システムリソースをほかの並列アプリケーションと共有する。** **iii) ほかの並列アプリケーションのほうが優先度が高い。**	OS のデフォルト・スケジューラー・ポリシーに依存します。 ハードコードされたアフィニティー・バインディングは、システムの応答性やスループットに悪影響を与えることがあり、場合によってはアプリケーションのパフォーマンスにも影響します。	OS のデフォルト・スケジューラー・ポリシーに依存します。 明示的に NUMA を管理することなく透過的に NUMA のメリットを提供できる API を使用します。
次の条件に当てはまるマルチスレッド・アプリケーション: **i) フォアグラウンドで実行され優先度が高い。** **ii) システム上のすべてのプロセッサー・リソース使用するわけではない。** **iii) システムリソースをほかの並列アプリケーションと共有する。** **iv) ただし、ほかの並列アプリケーションのほうが優先度が低い。**	アプリケーション向けにカスタマイズされたスレッド・バインディング・ポリシーについて検討する場合、ハードコードされたスレッド・アフィニティー・バインディング・ポリシーではなく、OS スケジューラーを使用した協調的アプローチを採用すべきです。例えば、SetThreadIdealProcessor() を使用すると、局所性が最適化されたアプリケーション・バインディング・ポリシーに次のフリーコア・バインディング・ポリシーを固定するフローティング・ベースを提供して、デフォルト OS ポリシーと協調できます。	明示的に NUMA を管理することなく透過的に NUMA のメリットを提供できる API を使用します。 デフォルト OS ポリシーが原因でパフォーマンス上の問題が発生する場合、パフォーマンス・イベントを使用してローカルでないメモリーアクセスの問題を診断します。
フォアグラウンドで実行されるマルチスレッド・アプリケーションであり、システム上のすべてのプロセッサー・リソースを必要とし、システムリソースを並列アプリケーションと共有しない。MPI ベースのマルチスレッディング。	アプリケーション向けにカスタマイズされたスレッド・バインディング・ポリシーのほうがデフォルト OS ポリシーよりも効果的な場合があります。パフォーマンス・イベントを使用して、局所性とキャッシュ転送の可能性を最適化します。 独自の明示的なスレッド・アフィニティー・バインディング・ポリシーを使用するマルチスレッド・アプリケーションは、エンドユーザーまたは管理者に何らかの形式のオプトイン選択肢を提供し、許可された場合に導入すべきです。例えば、明示的なスレッド・アフィニティー・バインディング・ポリシーを導入する権限は、インストール後に権限が与えられてから有効化できるようにします。	アプリケーション向けにカスタマイズされたメモリー・アフィニティー・バインディング・ポリシーのほうがデフォルト OS ポリシーよりも効果的な場合があります。パフォーマンス・イベントを使用して、OS またはカスタムポリシーに関するローカルでないメモリーアクセスの問題を診断します。

11.8.1 トポロジー共有リソースの列挙

マルチスレッド・ソフトウェアが OS のスケジューリング・ポリシーに基づいて実行される場合でも、カスタマイズされたリソース管理向けにアフィニティー API の使用を必要とする場合でも、共有プラットフォーム・リソースのトポロジーを理解することは重要です。CPUID によって提供される情報から、プラットフォーム上の論理プロセッサー (SMT)、プロセッサー・コア、物理プロセッサーのプロセッサー・トポロジーを判断できます。これに関して、『インテル® 64 および

IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A』の第 8 章「Multiple-Processor Management」を参照してください。インテルからはホワイトペーパーとサンプルコードも提供されています。

11.8.2 NUMA (Non-Uniform Memory Access)

Nehalem† マイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサーが複数搭載されたプラットフォームでは、各物理プロセッサーが独自のローカル・メモリー・コントローラーを備えているため、Non-Uniform Memory Access (NUMA) トポロジーがサポートされています。NUMA が提供するシステムメモリー帯域幅は、物理プロセッサー数に合わせて拡張されます。システムメモリーのレイテンシーは、メモリー・トランザクションが同じソケット内でローカルに発生するか、別のソケットからリモートで発生するかに応じて、非対称な動作となります。また、OS 固有の構造や実装による動作により、API レベルの複雑性が増すため、マルチスレッド・ソフトウェアでは NUMA 環境におけるメモリー割り当て/初期化に注意を払う必要があります。

一般に、レイテンシーに拘束されるワークロードでは、リモートよりもローカルのメモリー・トラフィックを維持することが望まれます。複数のスレッドがバッファーを共有する場合、プログラマーは、NUMA システムでのメモリー割り当て/初期化に関する OS 固有の動作に注意する必要があります。

帯域幅に拘束されるワークロードでは、データ編成スレッディング・モデルを採用すると利便性が高くなります。各ソケットで実行されるアプリケーション・スレッドをまとめて、ソケットごとでローカル・トラフィックを優先することで、物理プロセッサー数に合わせて帯域幅全体を拡張できるようになります。

ローカルとリモートの NUMA トラフィックを管理するプログラミング・インターフェイスを提供する OS 構造は、メモリー・アフィニティーと呼ばれます。OS は、物理アドレス (システム RAM によって割り当てられる) とリニアアドレス (アプリケーション・ソフトウェアによってアクセスされる) のマッピングを管理しており、ページングは物理ページの動的再割り当てで異なるリニアアドレスへの動的な割り当てを可能にします。このような理由から、メモリー・アフィニティーを適切に使用するには、OS 固有の知識が求められます。

アプリケーション・プログラミングを簡素化するため、OS では特定の API と物理/リニア・アドレス・マッピングを実装し、一部の状況で NUMA の特性を透過的に利用できます。一般的な手法の 1 つとして、アプリケーション・スレッドがリニアアドレス空間で物理メモリーページの最初のメモリー参照にアクセスするまで、OS による同ページの割り当てのコミットを遅延します。この場合、メモリー割り当て API がプログラムに戻る際にどのソケットがローカル・メモリー・トラフィックを処理するかは、アプリケーション・スレッドによるリニアアドレス空間でのメモリーバッファー割り当てによって決まるとは限りません。ただし、このレベルの NUMA 透過性をサポートするメモリー割り当て API は、OS によって異なります。例えば、Linux*上では移植性に優れた C 言語 API「malloc」がある程度の透過性を提供しますが、Windows* 上では API「VirtualAlloc」が同様の透過性を提供します。OS によっては、明示的な NUMA 情報を必要とするメモリー割り当て API も提供されており、リニアアドレスとローカル/リモート・メモリー・トラフィックとのマッピングが割り当て時に固定されます。

例 11-9 では、マルチスレッド・アプリケーションが最小限の処理で OS 固有 の API を扱い、NUMA ハードウェア機能を利用しています。メモリーバッファーを初期化するこの並列アプローチは、NUMA システム上で各ワーカースレッドにローカルのメモリー・トラフィックを維持するのに役立ちます。

例 11-9 OpenMP* と NUMA を使用した並列メモリーの初期化手法

```
#ifdef _LINUX // Linux* の malloc の実装では、最初に参照/ アクセスしたときに物理ページがコミットされます
   buf1 = (char *) malloc(DIM*(sizeof (double))+1024);
   buf2 = (char *) malloc(DIM*(sizeof (double))+1024);
   buf3 = (char *) malloc(DIM*(sizeof (double))+1024);
#endif
#ifdef windows
// Windows* の malloc の実装では割り当て時に物理ページがコミットされるので、
// 代わりに VirtualAlloc を使用します
   buf1 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
   buf2 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
   buf3 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
#endif
   (続き )
   a = (double *) buf1;
   b = (double *) buf2;
   c = (double *) buf3;
#pragma omp parallel
{ // ループの各反復を OpenMP* スレッドで処理します
// OpenMP* のスレッド数はデフォルトか、もしくは環境変数で指定できます
#pragma omp for private(num)
// 各ループ反復を実行するため、プライベートなイテレーターを持つ異なる OpenMP* スレッドに割り当てられます
   for(num=0;num<len;num++)
   {// 各スレッドは最初にアクセスした時に、メモリーアドレスのサブセットを取得し、
   // 物理ページは該当するスレッドのローカル・メモリー・コントローラーへマップされます
        a[num]=10.;
        b[num]=10.;
        c[num]=10.;
   }
}
```

例 11-9 では、OpenMP* によって生成されたワーカースレッドが終了した後に、メモリーバッファーが解放されることに注意してください。ここでは、異なるアプリケーション・スレッド間で malloc と解放が繰り返し使用される問題を回避しています。ローカルメモリーがあるスレッドによって初期化され、その後別のスレッドによって解放された場合、OS では NUMA トポロジーに関連するリニアアドレス空間のメモリープールを追跡/再割り当てすることが困難になります。Linux* では、別の API「numa_local_alloc」が使用できます。

11.9 その他の共有リソースの最適化

マルチスレッド・アプリケーションのリソースの最適化は、プロセッサー・トポロジー階層内のキャッシュトポロジーと関連する実行リソースに依存します。プロセッサー・トポロジーと、ソフトウェアによるプロセッサー・トポロジーの識別アルゴリズムについては、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A』の第 8 章で説明されています。

共有バスを備えたプラットフォームでは、バスシステムは、SMT レベルとプロセッサー・トポロジーのプロセッサー・コア・レベルで複数のエージェントによって共有されます。そのため、マルチスレッド・アプリケーションの設計は、同じバスリンクを共有する複数のプロセッサー・エージェント間で利用可能なバス帯域幅を公平に管理することから始めるべきです。これは、個々のアプリケーション・スレッドのデータ局所性を改善するか、2 つのスレッドが共有 2 次キャッシュを利用することで (そのような共有キャッシュトポロジーを利用可能な場合) 実現できます。

マルチスレッド・アプリケーションのビルディング・ブロックの最適化は通常、個々のスレッドから開始できます。第 3 章から第 13 章のガイドラインは、主にマルチスレッドの最適化に適用されます。

チューニングの推奨事項 2: 実行スループットを最大化するため、最初にシングルスレッド・コードを最適化します。

チューニングの推奨事項 3: 効率良いスレッディング・モデルを採用し、入手可能なツール (インテル® スレッディング・ビルディング・ブロック、インテル® パフォーマンス・ツールなど) を活用して、物理プロセッサーやプロセッサー・コア数に応じた最適なプロセッサー・スケーリングを達成します。

11.9.1 HT テクノロジー最適化の可能性を拡大

Nehalem† マイクロアーキテクチャーのハイパースレッディング (HT) テクノロジーは、前世代の HT テクノロジーの実装とは異なっています。幅広いアプリケーションで、マルチスレッド・ソフトウェアが HT テクノロジーを利用してシステム・スループットを高められる可能性が拡大しています。ここでは、ヒューリスティック的な推奨事項をいくつか紹介し、Nehalem† マイクロアーキテクチャーで提供される HT テクノロジーの最適化の可能性ついて説明します。

第 2 章「インテル® 64 および IA-32 プロセッサー・アーキテクチャー」では、HT テクノロジーにおけるマイクロアーキテクチャー上の機能強化について説明しています。これらの機能強化の多くは、複数のスレッド・コンテキストで使用可能なハードウェア・リソースの共有に関するマルチスレッド・ソフトウェアの基本要件を中心に行われています。

ソフトウェア・アルゴリズムやワークロード特性が異なると、複数の論理プロセッサー間で共有されるマイクロアーキテクチャー・リソースの要件から、パフォーマンス特性も異なります。表 11-3 では、HT テクノロジー向けのソフトウェア・チューニングで重要な役割を果たす各種マイクロアーキテクチャー・サブシステムを簡単に比較します。

表 11-3 HT テクノロジーのマイクロアーキテクチャー・リソースの比較

マイクロアーキテクチャー・サブシステム	**Nehalem† マイクロアーキテクチャー 06_1AH**	**Intel NetBurst® マイクロアーキテクチャー 0F_02H、0F_03H、0F_04H、0F_06H**
発行ポート、実行ユニット	3 つの発行ポート (0、1、5) を ALU、SIMD、FP の各演算向けに分散	ポートのバランスが取れていない。高速 ALU、SIMD、FP で同じポート (ポート 1) を共有
バッファリング	適度なパイプラインの深さを確保しながら、ROB、RS、フィルバッファーなどのエントリーを増加	バッファーエントリーとパイプラインの深さとのバランスレベルが低い
分期予測とアライメントされないメモリーアクセス	スペキュレーティブ・エグゼキューションが強化され、予測ミスの直後に再利用が可能。キャッシュ分割を効率的に処理	マイクロアーキテクチャー・ハザードが多く発生し、両方のスレッドでパイプラインがクリアーされる
キャッシュ階層	大容量かつ効率的	回避すべきマイクロアーキテクチャー・ハザードが多い
メモリーと帯域幅	NUMA、DDR3 に対してソケットごとに 3 チャネル、ソケットごとに最大 32GB/秒	SMP、FSB またはデュアル FSB、FSB ごとに最大 12.8GB/秒

演算量の多いワークロードの場合、Intel NetBurst® マイクロアーキテクチャーの HT テクノロジーでは、比較的高い CPI (連続した命令がリタイアするまでの平均サイクル) で実行されるスレッド・コンテキストが優先される傾向にあります。ハードウェア・レベルでは、高速 ALU、低速 ALU (負荷の高い整数演算)、SIMD、FP の各演算によってポート 1 が共有されるという、マイクロアーキテクチャーにおける発行ポートのインバランスがこの原因です。ソフトウェア・レベルでは、HT テクノロジーのメリットをもたらす高い CPI の要因として、レイテンシーの長い命令 (ポート 1)、一部の L2 ヒット、時折発生する分岐予測ミスなどがあります。ただし、Intel NetBurst® マイクロアーキテクチャーのパイプライン長は、ハードウェア内部の制約を増加させ、ソフトウェアによる HT テクノロジーの利用を制限する傾向があります。

表 11-3 に示したマイクロアーキテクチャーの強化点は、演算量の多いワークロードにおいてソフトウェア最適化の可能性を高めることが期待されます。ただし、演算量の多い 2 つのスレッドが同じ実行ユニット内で競合することが、データ分解スレッドモデルよりも機能分解スレッドモデルを選択する上での課題となる可能性があります。プログラマーによる最適なスレッド分解モデルの選択をサポートするには、Nehalem† マイクロアーキテクチャーのほうが適しています。

大量のメモリーを必要とするワークロードは、完全に並列なメモリー・トラフィック (Stream の例に見られるような、システムメモリー帯域幅の飽和)、メモリー・レイテンシーに左右されるメモリー・トラフィック、演算操作といずれかの種類のメモリー・トラフィックとの各種組み合せなど、幅広いパフォーマンス特性が見られます。

Intel NetBurst® マイクロアーキテクチャーの HT テクノロジーは、後者の 2 種類のワークロード特性の一部においてメリットが得られます。Nehalem† マイクロアーキテクチャーの HT テクノロジーを利用すると、NUMA のサポート、効率的なリンクプロトコル、物理プロセッサー数に合わせて拡張可能なシステムメモリー帯域幅により、後者の 2 種類のワークロード特性における処理範囲を広げ、システム・スループットの向上を図ることが可能です。

キャッシュ階層中の一部のキャッシュレベルは、複数の論理プロセッサーによって共有されることがあります。キャッシュ階層の利用は、ソフトウェアによってメモリー・トラフィックを効率化し、システムメモリー帯域幅の飽和を回避する重要な手段です。キャッシュ・ブロッキング手法が採用されたマルチスレッド・アプリケーションでは、HT テクノロジーを利用するためにターゲット・キャッシュ・レベルの分割が必要となることがあります。また、同じ L1 キャッシュと L2 キャッシュを共有する 2 つの論理プロセッサーや、L3 キャッシュを共有する論理プロセッサーは、それぞれに関連するトポロジーに応じた共有リソース管理を必要とすることがあります。プロセッサー・トポロジーの列挙およびキャッシュトポロジーの列挙とそのサンプルコードについては、ホワイトペーパーが公開されています (第 1 章の最後にある「関連情報」を参照してください)。

第 12 章
インテル® Optane™ DC パーシステント・メモリー

Cascade Lake† をベースとするインテル® Xeon® スケーラブル・プロセッサー製品は、インテル® Optane™ DC パーシステント・メモリー・モジュールをサポートします。インテル® Optane™ DC パーシステント・メモリー・モジュールは、DRAM と比較して大きな容量を持ち、かつ永続的 (システムの電源を落としても内容が保持されます) です。しかし、DRAM DIMM よりもレイテンシーが長く、帯域幅は低くなります。

12.1 Memory モードと App Direct モード

インテル® Optane™ DC パーシステント・メモリー・モジュール DIMM は、2 つの異なるモードで使用できます。

12.1.1 Memory モード

Memory モードでは、メモリーは不揮発性メモリーとして動作します。これは、オペレーティング・システムとアプリケーションからは透過です。ソフトウェアは変更を加えることなく、大容量メモリーの恩恵を受けることができます。システム上の DRAM メモリーは、メモリー側のキャッシュとして使用されます。これが意図することは、ソフトウェアがインテル® Optane™ DC パーシステント・メモリー・モジュール層の「インメモリー」データ容量を活用しながら、DRAM 層のレイテンシーを維持することです。

Memory モードでは、リブート後にインテル® Optane™ DC パーシステント・メモリー・モジュール上のデータにアクセスできなくなります。デバイス自体は不揮発性であるため、これは電源投入前に破棄されるキーによってデータを暗号化することで実装されます。ソケット上の DRAM メモリーは、インテル® Optane™ DC パーシステント・メモリー・モジュールのダイレクトマップ・キャッシュとして使用されます。これは、プロセッサーのキャッシュとは異なり、キャッシュに LRU ポリシーが適用されないことを意味します。インテル® Optane™ DC パーシステント・メモリー・モジュールのキャッシュラインは、常に DRAM から同じキャッシュラインを排出します。オペレーティング・システムは、DRAM キャッシュから排出すべきではないデータを保持するページと競合しないページのアドレスを使用することによって、Memory モードを最適化できます。例えば、ページテーブルを常に DRAM に保持することは有益です。ワーキングセットの大きさはパフォーマンスを決定付けます。アプリケーションのワーキングセットが DRAM に収まる場合、パフォーマンスはインテル® Optane™ DC パーシステント・メモリー・モジュールのレイテンシーと帯域幅の影響をそれほど受けません。

12.1.2 App Direct モード

App Direct モードではメモリーはデバイスとして扱われ、ファイルシステムとしてフォーマットできます。1 つの選択肢は、インテル® Optane™ DC パーシステント・メモリー・モジュールを超高速ブロックデバイスである、「App Direct ストレージ」と呼ばれるファイルシステムとして使用することです。これには、ソフトウェアに変更を加えることなく、I/O 集約型のアプリケーションがインテル® Optane™ DC パーシステント・メモリー・モジュールの恩恵を受けられる利点があります。しかしこれは、App Direct 高速ストレージとしてデバイスを使用しますが、パーシステント・メモリーとしては使用していません。それとは対照的に、パーシステント・メモリーとして使用するためアプリケーションを書き換えることには、大きな利点があります。「App Direct ストレージ」との大きな違いは、データをキャッシュライン単位でアクセスできることです。データをデバイスからロードまたはストアするためプロセッサーのロードやストア命令が実行され、いったんページがプロセスに割り当てられると OS との対話は必要ありません。App Direct モードはパーシステント・メモリーを実装します。オペレーティング・システムは、RAM としてパーシステント・メモリーにアクセスするのではなく、「dax」と呼ばれる特殊なフラグを持ってマウントされたファイルシステムとしてアクセスします。「dax」フラグは、「App Direct モード」(dax 付きでマウント) と「App Direct モードのストレージ」(dax なしでマウント) を区別するために使用されます。dax 付きでマウントすると、次のような利点があります。

パーシステント・メモリーがアプリケーションの仮想アドレス空間にマッピングされると、読み書きはプロセッサーのロードおよびストア命令で行えるようになります。これには、App Direct ストレージモードに対し、次にような利点があります。

- システムコールを呼び出す必要がありません。
- ページ単位 (例えば 4KB) で転送する代わりに、キャッシュライン単位 (64B) で転送されます。
- メモリーの内容はパーシステント・メモリーと DRAM 間で転送する必要がなく、データのコピーは 1 つのみ存在します。
- アクセスは同期的です。

パーシステント・メモリーは、ファイルシステムを介してアクセスできるため、オペレーティング・システムや従来の OS メモリー監視ツールは、システムに存在する DRAM のみを報告することに注意してください。2 つのメモリープールは互いに異なり、ソフトウェアがデータを DRAM と NVDIMM のいずれに配置するか完全に制御できることで App Diect モードは差別化されます。最適なパフォーマンスを得るため、ソフトウェアはコピーを作成するか再構築することによって、レイテンシーに敏感なデータ構造を DRAM に配置できます。例としてインデックス・データ構造が上げられます。インデックス・データ構造は通常ランダムアクセスされますが、再起動後に再構築できます。

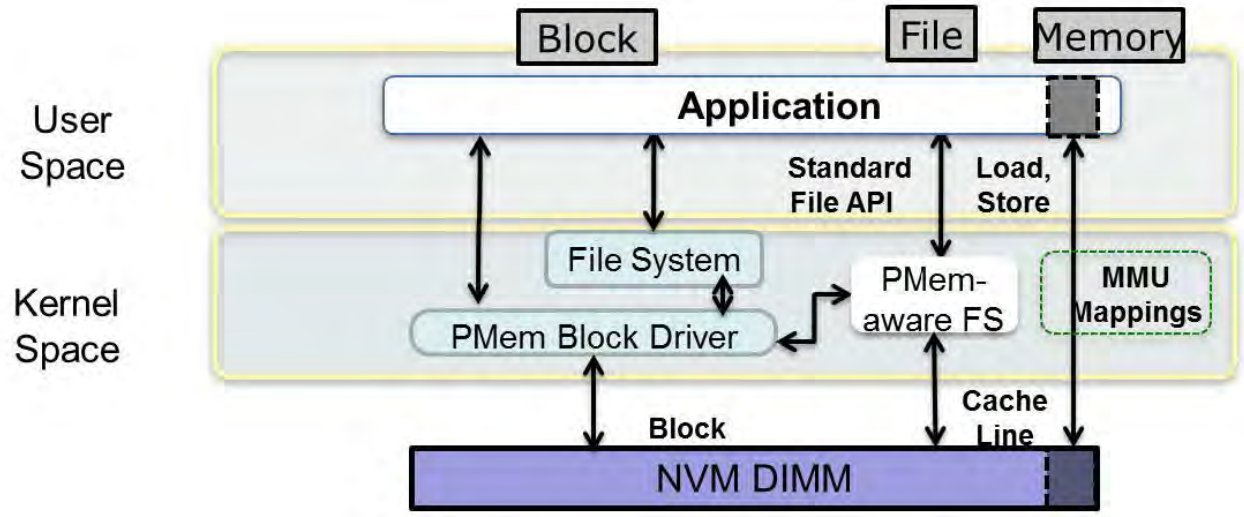


図 12-1 App Direct モードではインテル® Optane™ DC パーシステント・メモリー・モジュール上のデータはロードおよびストアで直接アクセスされる

12.1.3 モードの選択

ソフトウェア開発者は、アプリケーションと使用シナリオにどのモードが最も適しているか判断するため、さまざまな要因を考慮する必要があります。

- 大きな容量のメモリー (プラットフォームの DRAM 容量より大きい) から利点は得られますか? 例えば、アプリケーションが頻繁にディスクへページングされる場合、メモリー容量が大きいほどページングは少なくなる可能性があります。また、大きなメモリー容量が利用可能な場合に異なるアルゴリズムを選択的に導入することで、パフォーマンスが向上するアプリケーション、および大きな容量が与えられると中間結果を保持して再利用することを選択するアプリケーションなども考えられます。
- メモリー・サブシステムで永続性を使用する利点はありますか? これには、起動時間の短縮 (ディスクからメモリーへのデータロードの回避、再起動時のリンクリストやツリーなど「インメモリー」ポインターベース構造の再構築の回避など) が含まれます。これはまた、永続性への高速パスという利点を含みます。例えば、ディスクに代えてメモリーをデータの最終的な永続性のある保存先にすることができます。ディスクのレイテンシーや帯域幅に制約されるアプリケーションは、永続性のあるメモリーを使用することで利益を得られます。
- アプリケーションはメモリー・レイテンシーからどれくらいの影響を受けますか? インテル® Optane™ DC パーシステント・メモリー・モジュールのレイテンシーは DRAM より長く、通常 DRAM の 3-4 倍となります。メモリーをインテル® Optane™ DC パーシステント・メモリー・モジュールと置き換えると、それらのアクセスをどの程

度予測できるか、またはメモリーアクセスがレイテンシーにどれくらい影響を受けるかはさまざまな要因により異なります。その状況を説明するため、アプリケーションがインテル® Optane™ DC パーシステント・メモリー・モジュールから数 GB の連続した配列を読み取るシナリオを考えてみます。この場合、アクセスは空間的に予測できるため、ハードウェアおよびソフトウェア・プリフェッチャーによるプリフェッチは可能です。これにより、アプリケーションがデータを要求する前にデータは常にプロセッサー・キャッシュに存在する可能性があり、インテル® Optane™ DC パーシステント・メモリー・モジュールのレイテンシーは隠匿されます。しかし、アプリケーションがリンクリストを追跡するような場合、最初に現在のノードを読み取らなければ次のノードを識別することはできません (これを「ポインター追尾」と呼びます)。この場合、インテル® Optane™ DC パーシステント・メモリー・モジュールのレイテンシーは、アプリケーションから隠匿できません。もう 1 つの重要な考慮事項は、アプリケーションがメモリー・レイテンシーからどれくらいの影響を受けるかということです。アプリケーションのメモリー参照がインテル® Optane™ DC パーシステント・メモリー・モジュールへ展開されるのを待機する間に、プロセッサー・コアが別の有用なワークを実行できることがあります。この場合、有用なワークが実行されているため、パフォーマンスに大きな影響はありません。一方、インテル® Optane™ DC パーシステント・メモリー・モジュールからのメモリー参照を待機する間コアが停止すると、パフォーマンスに影響します。

アプリケーション全体がメモリー・レイテンシーに敏感な場合、どのメモリー構造が敏感であるか検証します。インテル® Optane™ DC パーシステント・メモリー・モジュール向きの用途は、上記の考慮事項に基づくメモリー・レイテンシーの影響をそれほど受けない大容量のデータ構造です。アクセス頻度が高く、メモリー・レイテンシーの影響を受ける小さなデータ構造ほど DRAM に適しています。

次の図は、考慮事項に基づく決定条件を示しています。

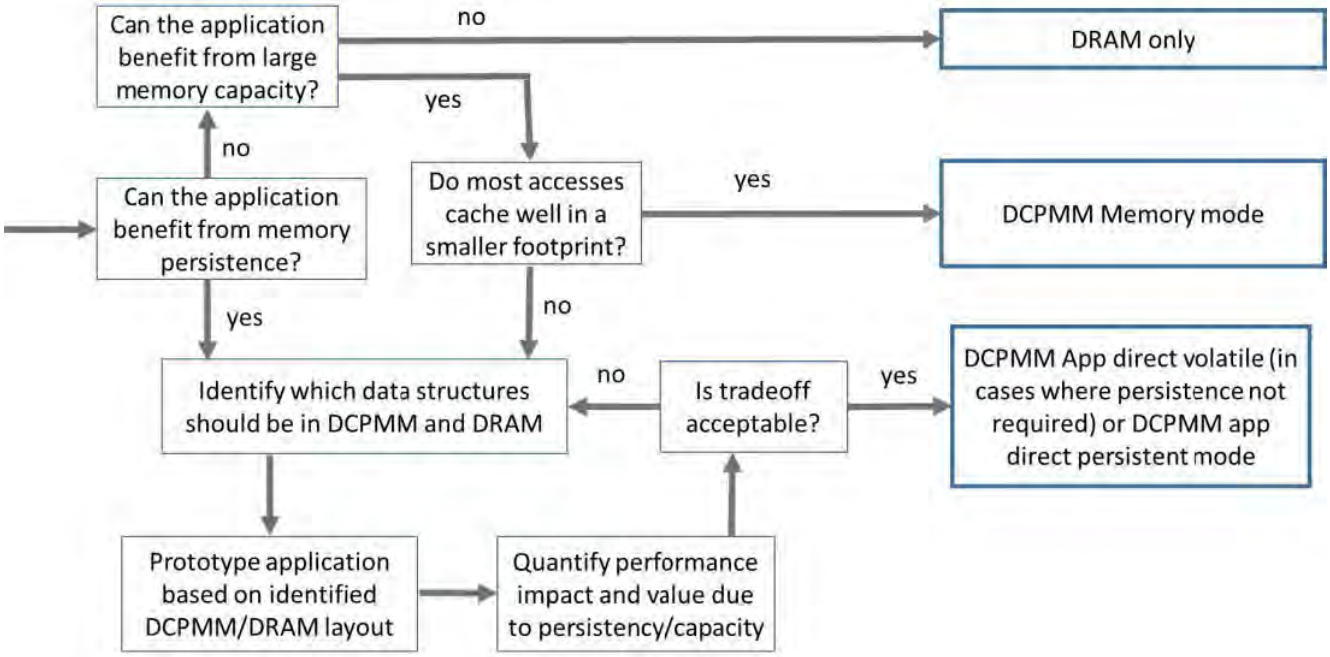


図 12-2 インテル® Optane™ DC パーシステント・メモリー・モジュールと DRAM の利用を決定する条件

12.2 インテル® Optane™ DC パーシステント・メモリー・モジュールのデバイス特性

前の節では、ソフトウェア開発者がデータ構造をインテル® Optane™ DC パーシステント・メモリー・モジュールに配置する際の考慮事項の 1 つは「メモリー・レイテンシーに対するパフォーマンスの影響」であることを述べました。この節では、この影響を判断するいくつかの考慮事項について説明します。これには、DRAM と異なるデバイス特性、メモリーの永続性など新しい機能を利用するために必要なソフトウェアの変更などが含まれます。

12.2.1 インテル® Optane™ DC パーシステント・メモリー・モジュールのレイテンシー

インテル® Optane™ DC パーシステント・メモリー・デバイスは、DRAM の素材と異なるマテリアルで構成されるため DRAM とは異なるアクセス特性を持っています。次の表はシーケンシャルとランダムアクセスのリード・レイテンシーをまとめたものです。

表 12-1 インテル® Optane™ DC パーシステント・メモリー・モジュールのアクセス・レイテンシー

レイテンシー	インテル® Optane™ DC パーシステント・メモリー・モジュール	DRAM
アイドル・シーケンシャル・リードのレイテンシー	~170ns	~75ns
アイドル・ランダム・リードのレイテンシー	~320ns	~80ns

DRAM の場合、シーケンシャルとランダムアクセスのレイテンシーの差は数ナノ秒です。これは、シーケンシャル・アクセスにより、DRAM の行バッファーのヒット数が増加するためです。しかし、インテル® Optane™ DC パーシステント・メモリー・モジュールでは、レイテンシーが全体的に DRAM と異なるだけでなく、シーケンシャルとランダムアクセスでも大きく異なります。

インテル® Optane™ DC パーシステント・メモリー・モジュールと DRAM のアクセス・レイテンシーの差は、パフォーマンスの観点からソフトウェア開発者にとって特別な考慮が求められます。プロセッサー・キャッシュ使用率を最適化する一般的なガイドラインについては、第 9 章「キャッシュ利用の最適化」を参照してください。

Memory モードでは、DRAM キャッシュがほとんどのアクセスを吸収するため、アプリケーションでは DRAM に近いレイテンシーが生じることが予測されます。Memory モードでのインテル® Optane™ DC パーシステント・メモリー・モジュールへのアクセス・レイテンシーは、最初に DRAM キャッシュをルックアップするオーバーヘッドが生じるため、App Direct モードよりも最大 30 ~ 40 ns 長くなります。Memory モードでのパフォーマンスは、ワーキングセットを DRAM キャッシュサイズに収める従来のキャッシュのタイル化、および局所性の最適化手法を利用することで改善できます。

また、インテル® Optane™ DC パーシステント・メモリー・モジュールは、256 バイト単位でのバッファー機能を備えています。これは、シーケンシャルとランダムアクセスを区別する単位の 1 つであるといえます。そのため、ソフトウェア・データ構造を設計する考慮事項として、256 バイト内にデータを配置しまとめて読み取ることで、ランダムではなくシーケンシャル・アクセスのレイテンシーが得られます。

12.2.2 リードとライトの帯域幅

次の表に示すように、インテル® Optane™ DC パーシステント・メモリー・モジュールのアクセス特性は、DRAM と比べてリードおよびライトの帯域幅が異なります。

表 12-2 インテル® Optane™ DC パーシステント・メモリー・モジュールと DRAM の DIMM ごとの帯域幅

DIMM ごとの帯域幅	インテル® Optane™ DC パーシステント・メモリー・モジュール	DRAM
シーケンシャル・リード	~7.6GB/s	~15GB/s
ランダムリード	~2.4GB/s	~15GB/s
シーケンシャル・ライト	~2.3GB/s	~15GB/s
ランダムライト	~0.5GB/s	~15GB/s

上記から次のことが分かります。

1. リードとライトは非対称的であり、リード帯域幅はライト帯域幅よりも大きくなっています。これはソフトウェアの設計者にとって重要な考慮事項であり、判断材料に含める必要があります。たとえば、ランダムライトと高い書き

込み帯域幅を持つデータ構造は、インテル® Optane™ DC パーシステント・メモリー・モジュールには適していません。

2. シーケンシャルとランダムアクセスの特性は大きく異なります。これも、DRAM とインテル® Optane™ DC パーシステント・メモリー・モジュール向けのデータ構造設計と配置を選択する上で考慮すべきことです。シーケンシャル・アクセスの利点を得るには、256B 単位での局所性を重視します。

帯域幅は、これらの DIMM を使用する際の一番の制約であり、図 12-3 の赤丸で示すように、DIMM の限界能力付近の帯域幅で操作しないようにすることが重要です。

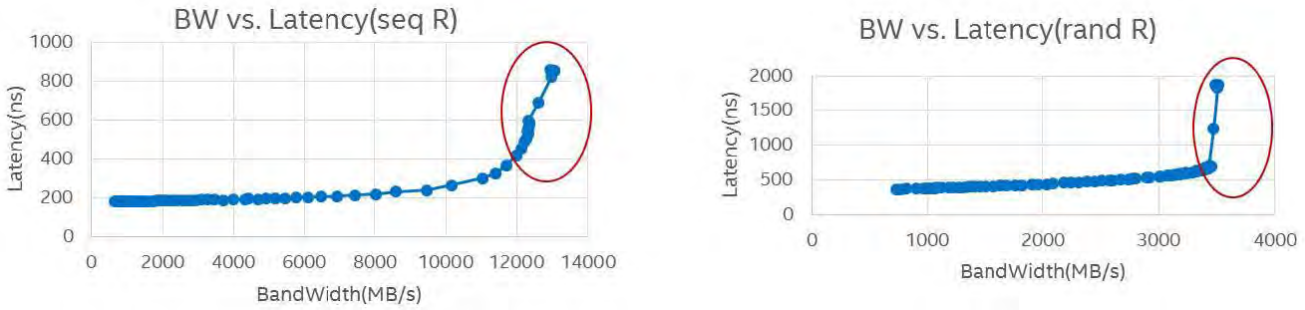


図 12-3 単独のインテル® Optane™ DC パーシステント・メモリー・モジュール DIMM のロード・レイテンシー曲線: シーケンシャル・トラフィック (左) とランダム・トラフィック (右)

メモリー帯域幅が飽和状態に近づくと、レイテンシーが極端に高くなり、アプリケーションのパフォーマンスが低下する傾向があります。帯域幅に対する要求は、通常、メモリーアクセスを行うコア数、およびアクセスの性質 (シーケンシャル・アクセス・パターンとランダム・アクセス・パターン、リードとライトの混在) の関数です。一方、プラットフォームの帯域幅能力は、利用可能なチャネル数と DIMM 数の関数です。したがって、リードとライトのトラフィックとシステムの能力、例えば、インテル® Optane™ DC パーシステント・メモリー・モジュールに対するリードとライトのスレッド数、および使用されているメモリーチャネル数のバランスをとることが重要です。

インテル® Optane™ DC パーシステント・メモリー・モジュールへの書き込み中は、帯域幅が DRAM よりも制限されるため、書き込まれたデータの再利用が期待できない場合、または大きなバッファーへの書き込みは、通常のストアに代わって非テンポラルストアを使用することを推奨します。詳細は、「9.4.1.2」セクションを参照してください。

12.2.3 最適な帯域幅のスレッド数

前述のように、インテル® Optane™ DC パーシステント・メモリー・モジュール DIMM は、256B の粒度でデータをバッファリングして結合します。これは、インテル® Optane™ DC パーシステント・メモリー・モジュールのメモリーにアクセスするスレッドの数に影響する可能性があります。多数のスレッドがインテル® Optane™ DC パーシステント・メモリー・モジュール上のメモリーに同時書き込みを試みる場合、書き込みが他のスレッドによって妨害され空間の局所性が失われると、書き込み結合 (ライトコンバイン) と 256B の局所性の利点が失われます。その結果、各スレッドがシーケンシャルに書き込みを行っても、DIMM レベルでトラフィックはランダムに見えるようになります。そのため、インテル® Optane™ DC パーシステント・メモリー・モジュールへの書き込みを行うスレッド数がしきい値を超えると、シーケンシャル・アクセス帯域幅の代わりにランダムアクセス帯域幅が観測されるようになります。

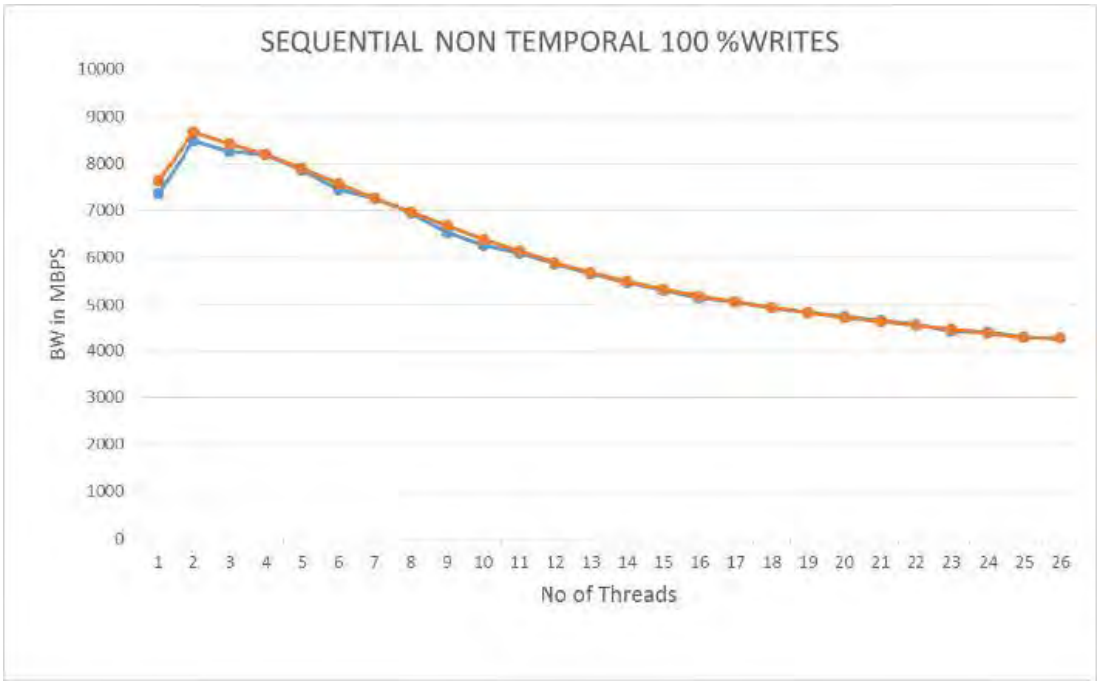


図 12-4 スレッド数と帯域幅[1]

注意:

1. スレッド数が増加すると帯域幅は増加し、その後減少します。減少の理由は、アクセスが本質的にシーケンシャルであっても、メモリー・サブシステムにアクセスを要求するスレッドが増えるため (特に前述の 256B 単位での)、ライト・コンバイン・バッファー数の観点からは「連続性」が失われるためです。

図 12-5、図 12-6、および図 12-7 は、256B の局所性での結合の違いと、これに対するインテル® Optane™ DC パーシステント・メモリー・モジュールを参照するスレッド数の影響を示しています。データ構造の選択、およびインテル® Optane™ DC パーシステント・メモリー・モジュールへの同時アクセスでは、256B の局所性を考慮することが重要です。

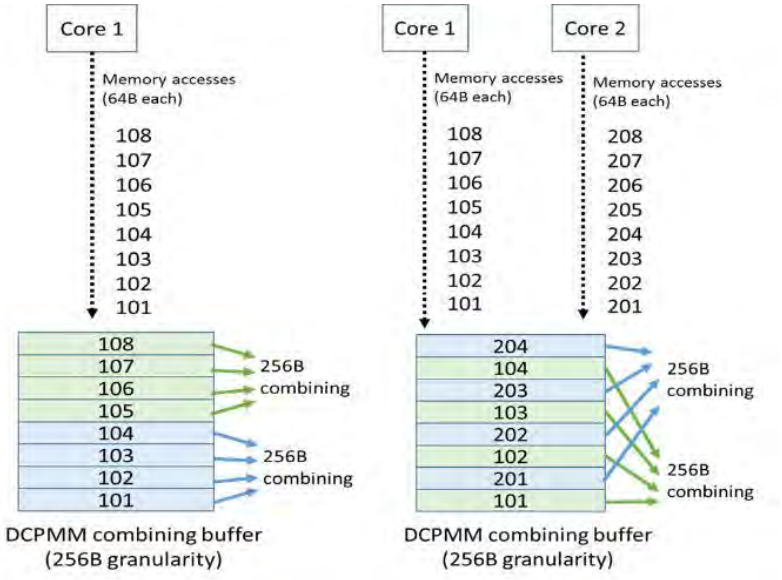


図 12-5 2 つのコアを結合[1]

注意:

1. 101、102 などは、コア 1 からの 64B アクセスを指し、他のコアでも同様です。2 コアおよび 8 個のサンプルバッファー (これは、図説を目的としたバッファー数であることに注意してください) に対し、256B の粒度でバッファー内が 100% 結合されていることが分かります。これにより、メモリーシステムの視点からアクセスは 100% シーケンシャルになります。

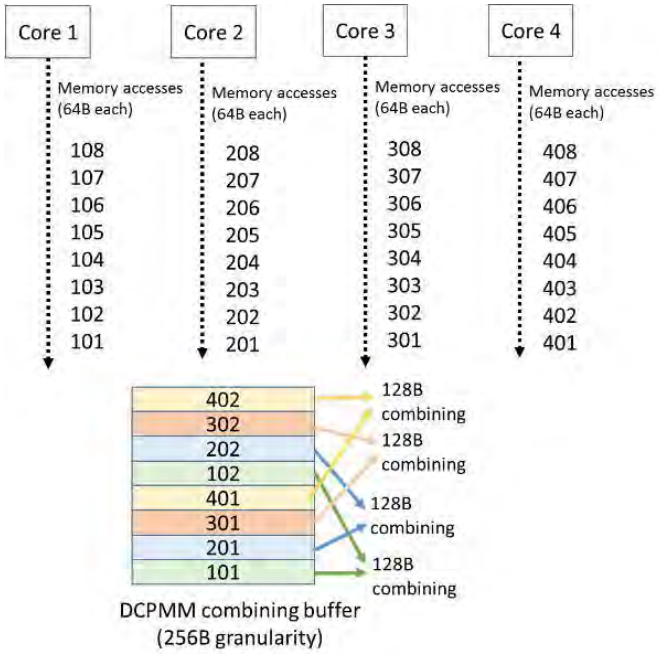


図 12-6 4 つのコアを結合[1]

注意:

1. 101、102 などは、コア 1 からの 64B アクセスを指し、他のコアでも同様です。4 コアおよび 8 個のサンプルバッファーでは、256B の粒度でバッファー内の 50% が結合されます (バッファーが一杯になり、処理を続行するには排出する必要があるため、128B のみ結合できます)。これにより、メモリーシステムの視点からアクセスは 50% シーケンシャルになります。

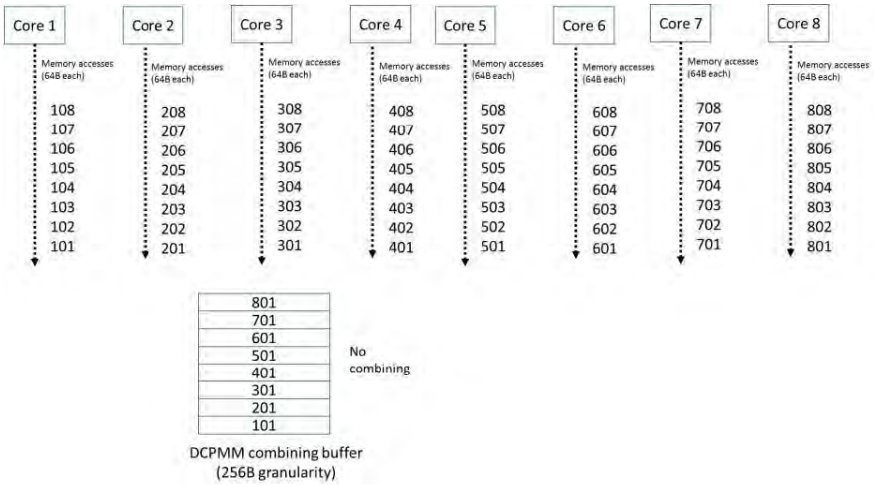


図 12-7 8 つののコアを結合[1]

注意:

1. 8 コアでは結合されず、メモリーシステムの視点からアクセスはランダムになります。

12.3 2 つ目のタイプのメモリーを扱うことによるプラットフォームへの影響

12.3.1 マルチプロセッサーのキャッシュー貫性

複数のプロセッサーを搭載したシステムでは、キャッシュの一貫性を保持するためディレクトリーが使用されます。このディレクトリーは、分散インメモリー・ディレクトリーとして実装され、各キャッシュラインの一貫性状態はメモリー内のライン自身のメタデータとしてストアされます。この実装は、スヌープベースのメカニズムよりも優れています。スヌープベースのメカニズムでは、それぞれのメモリーアクセスに対してプロセッサーは、そのラインの一貫性状態を把握するため、常に他のプロセッサーのキャッシュをチェックします (キャッシュラインが他の場所にあると、アクセスのレイテンシーは増加します)。

ディレクトリー・ベースのプロトコルでは、ディレクトリーは一貫性状態が変更されたかどうかを追跡します。例えば、他のプロセッサーから読み取られたメモリーは、メモリー内のメタデータとして記録されます。このディレクトリーの更新は、一貫性状態の変化を記録するためメモリー (メタデータ) に書き込まれます。異なるプロセッサーのコアが、インテル® Optane™ DC パーシステント・メモリー・モジュールの同じラインセットを繰り返し読み取ると、一貫性状態の変化を記録するためその都度インテル® Optane™ DC パーシステント・メモリー・モジュールへの書き込みが行われます。この書き込みは、「ディレクトリー書き込み」と呼ばれ、基本的にランダムとなる傾向があります。その結果、本来アプリケーションが利用できるインテル® Optane™ DC パーシステント・メモリー・モジュールの帯域幅が、ディレクトリー書き込みにより低下する可能性があります。ソフトウェアの観点から、インテル® Optane™ DC パーシステント・メモリー・モジュールへの異なるスレッドによるアクセス方法を考慮する価値があります。この種のパターンが観察される場合、システム全体でディレクトリーを無効化することでインテル® Optane™ DC パーシステント・メモリー・モジュール領域の一貫性プロトロルをディレクトリー・ベースからスヌープベースに変更すべきです。

12.3.2 メモリー階層の共有キュー

Cascade Lake† 製品ベースのプロセッサーでは、DRAM とインテル® Optane™ DC パーシステント・メモリー・モジュールへのアクセスは大部分が独立しています。ただし、DRAM とインテル® Optane™ DC パーシステント・メモリー・モジュールの両方のメモリー参照がメモリー・サブシステムの同じパスを通過すると、相互に影響することがあります。例えば、DRAM とインテル® Optane™ DC パーシステント・メモリー・モジュール間には共通のプロセッサー・キューがあります。このキューを調整するには BIOS の QoS 設定を使用します。同様に、DRAM とインテル® Optane™ DC パーシステント・メモリー・モジュールは、同じメモリーチャネルを共有できます (チャネルごとに個別のキューがありますが、チャネルは共有されます)。これら個別のキューの切り替えを制御する設定があります。細かい粒度で切り替えるとレイテンシーが最適化され、荒い粒度で切り替えるとバーストが増加し帯域幅が最適化されます。これらは BIOS 設定で提供されます。詳細については、BIOS の最適化ガイドを参照してください。

12.4 メモリー永続性の実装

App Direct モードでは、ソフトウェアが永続性を使用する際に、メモリーストアが永続的であることを明示的に制御したいことがあります。しかし、プロセッサー・コアがライト命令を発行すると、データはフィルバッファーで結合され、更新されたキャッシュラインはプロセッサーの揮発性キャッシュにストアされます。永続性を保持するため、ソフトウェアは更新されたプロセッサー・キャッシュを明示的に排出する必要があるかもしれません。これは、キャッシュライン・フラッシュ命令 (CLFLUSH/CLFLUSHOPT/CLWB) を使用して行われます。通常、9.4.6 節「CLFLUSH 命令」と 9.4.7 節「CLFLUSHOPT 命令」で示すように、CLFLASH よりも CLFLUSHOPT の使用が推奨されます。

フラッシュされたデータを再利用するには、CLWB を使用することを推奨します。CLWB は、永続化のためフラッシュされたキャッシュラインのコピーを保持します。その結果、後続のアクセス (データの再利用) でラインがキャッシュにヒットし、アクセス・レイテンシーが短縮されます。大きなメモリー範囲のわずかな領域だけが実際に更新される場合、更新された領域を追跡してそれらのみをフラッシュするのが有効です。特に、オペレーティング・システムが、書き込まれたページ (汚染された) を追跡して、そのページをのみをフラッシュできれば、少ない範囲の書き込みで済むた

めさらに効率的です。ほぼ範囲全体が書き込まれる場合、ユーザー空間で最適化されたコードでフラッシュを実装する方が効果的である可能性があります。

図 12-8 は、Linux* 上でわずかなファイルな汚染されるケースの msync の効率を示しています。一方、ほとんどのファイルが汚染されるケースでは、ユーザー空間で CLFLUSHOPT 命令を使用する方が効率的です。

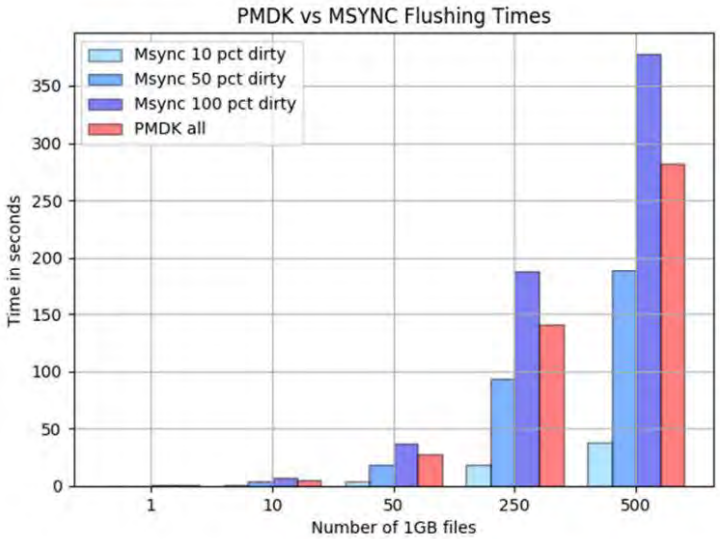


図 12-8 PMDK と MSYNC のフラッシュ時間[1]

注意:

1. ファイルの 10% または 50% が汚染されている場合、ソフトウェアの観点からはユーザー空間でフラッシュせず msync を使用する方が適切です。ファイルの 100% が汚染されていると、ユーザー空間で実装する (PMDK と示される) 方が効率的です。

12.5 消費電力

一般に、インテル® Optane™ DC パーシステント・メモリー・モジュールの帯域幅は、次の図に示すように消費電力によって制限されます。RAPL (Runtime Average Power Limiting) などの技術を使用することでソフトウェアによって電力が制御される場合、全体の帯域幅は適切に制限されます。

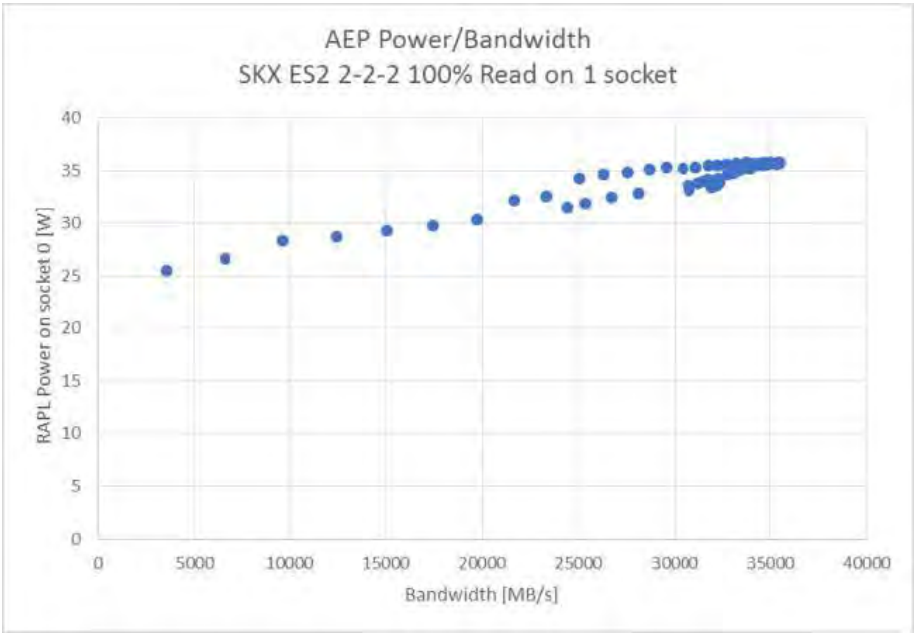


図 12-9 帯域幅と消費電力

12.5.1 リード - ライトの等価性

インテル® Optane™ DC パーシステント・メモリー・モジュールでは、リードよりもライトにはるかに大きな電力コストがかかります。一般的な指標として、一度の書き込みで 3 回分の読み取りと同等の電力を消費します。これは、リードとライトの消費電力がほとんど変わらない DRAM とは対照的です。したがって、書き込みを控えめにすべきことは明らかです。これは、インテル® Optane™ DC パーシステント・メモリー・モジュールのアクセスにおいて、ソフトウェアによる書き込みの増加は、読み取りの増加よりもはるかに高価であることを意味します。データ構造を設計する際にこれを考慮する必要があります。たとえば、何度か再調整する必要があるツリー構造では、ルックアップのため読み取りが多く挿入の書き込みが少ないデータ構造と比較して、大量の書き込みが発生する可能性があります。

利用可能な電力バジェットにより、リードとライトの比率が最大帯域幅を決定します。例えば、複数のスレッドがインテル® Optane™ DC パーシステント・メモリー・モジュールへの書き込みを行う場合、書き込みを行うスレッドが消費する電力がプラットフォームで利用可能な総電力から差し引かれます。次の図は、合計電力バジェット内でのインテル® Optane™ DC パーシステント・メモリー・モジュール DIMM の読み取りと書き込みの等価性を示します。

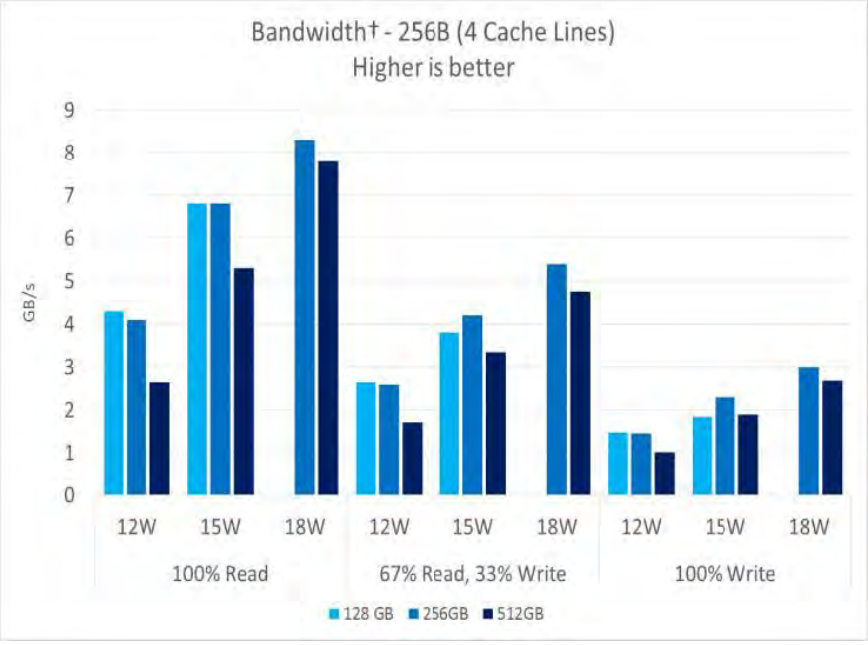


図 12-10 異なる電力バジェット[1] でのインテル® Optane™ DC パーシステント・メモリー・モジュール DIMM のリード/ライトの等価性

注意:

1. 左のバーは 100% 読み取りを示しています。このシナリオでは、15W の電力バジェットでおよそ 6.9GB/秒の読み取りが可能です。ただし、同じ 15W の電力バジェットでも書き込みは 2.1GB/秒のみが可能です。

12.5.2 空間と時間の局所性

消費電力を最適化することに加え、256B 粒度でのデータアクセスと組み合わせることの影響を考慮します。次の図に示すように、アクセスに局所性がない場合、アプリケーションで利用可能な帯域幅はかなり制限されます。

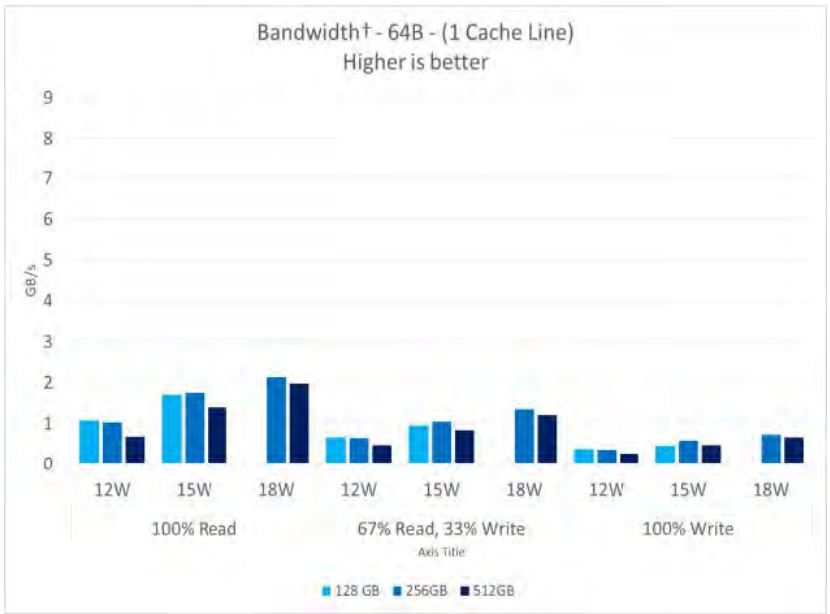


図 12-11 256B 単位の局所性がない場合の帯域幅

図 12-11 から、帯域幅の視点から電力バジェットを利用するには、256B 以内のアクセス局所性を持つデータ構造を選択することが重要であることが分かります。具体的には、図 12-10 と図 12-11 を比較すると、256B ウィンドウのアクセス局所性の導入により、帯域幅が最大 3 ~ 4 倍向上しています。

13.1 はじめに

この章では、64 ビット・モードで動作するように作成されたアプリケーション・ソフトウェア向けのコーディング・ガイドラインについて説明します。コーディングの推奨事項の一部は、第 3 章で触れています。この章のガイドラインは、第 3 章から第 11 章で説明したコーディング・ガイドラインへの補遺であると言えます。

互換モードまたはレガシーの非 64 ビット・モードで実行されるソフトウェアは、第 3 章から第 11 章で説明するガイドラインに従わなければなりません。

13.2 64 ビット・モードに影響するコーディング規則

13.2.1 データサイズが 32 ビットの場合はレガシーの 32 ビット命令を使用

64 ビット・モードでは、アプリケーションは 16 個の汎用 64 ビット・レジスターを利用できます。アプリケーションのデータサイズが 32 ビットである場合、64 ビット・レジスターや 64 ビット演算を使用する必要はありません。

ほとんどの命令では、デフォルトのオペランドサイズは 32 ビットです。このような命令では、上位 32 ビットをすべてゼロにします。例えば、レジスターをゼロにする場合、以下の 2 つの命令は同じように動作しますが、32 ビット版では 1 つの命令バイトのみ保存されます。

32 ビット版
xor eax, eax; 下位 32 ビットが xor され、上位 32 ビットにはゼロを設定

64 ビット版
xor rax, rax; 64 ビットすべてに xor を実行

この最適化は、8 つの汎用レジスター (EAX、ECX、EBX、EDX、ESP、EBP、ESI、EDI) に適用されます。レジスター R9 ~ R15 のデータをアクセスするには、REX プリフィクスが必要です。この場合、32 ビット形式を使用してもコードサイズは減少しません。

アセンブリー/コンパイラー・コーディング規則 57 (影響 H、一般性 M): 64 ビット・モードでは、64 ビット・データや追加レジスターへのアクセスに 64 ビット版の命令が必要な場合を除き、32 ビット版の命令を使うことでコードサイズを削減します。

13.2.2 追加のレジスターを使用してレジスターへの負荷を削減

64 ビット・モードでは、アプリケーションは 8 つの追加 64 ビット汎用レジスターと 8 つの追加 XMM レジスターを利用できます。追加されたレジスターにアクセスするには、1 バイトの REX プリフィクスが必要です。8 つの追加レジスターを使用することで、コンパイラーは値をスタックに退避させる必要がなくなります。

REX プリフィクスによってコードサイズが増加すると、キャッシュミスが増える可能性があるので注意が必要です。これは、追加のレジスターを使ってデータにアクセスするメリットに悪影響をもたらす恐れがあります。アルゴリズムで使用するレジスターが 8 つで十分なら、REX プリフィクスが必要なレジスターは使用してはいけません。これにより、コードサイズが小さくなります。

アセンブリー/コンパイラー・コーディング規則 58 (影響 H、一般性 MH): レジスターへの負荷削減に必要であれば、整数コードには 8 つの追加汎用レジスターと、浮動小数点および SIMD コードには 8 つの追加 XMM レジスターを使用します。

13.2.3 64 ビット値同士の乗算を有効活用

64 ビット・オペランド同士の整数乗算は、128 ビット幅の結果を生成する可能性があります。128 ビットの結果の上位 64 ビットは、下位 64 ビットよりも計算結果を生成するのにさらに数サイクル分必要な場合があります。128 ビットよりも幅が広い整数の加算の依存関係チェーンでは、最適なソフトウェア・パイプライン処理を行う上で、乗算結果の上位 64 ビットへのアクセスが、下位 64 ビットと比べて遅くなります。

コンパイラーは、コンパイル時に乗算結果が 64 ビットを超えないことを判断できる場合、64 ビットの結果を生じる乗算命令を生成すべきです。乗算結果が 64 ビットよりも小さくなることをコンパイラーやアセンブリー・プログラマーが判断できない場合、128 ビットの結果を生成する乗算が必要です。

アセンブリー/コンパイラー・コーディング規則 59 (影響 ML、一般性 M): 128 ビットの結果を生成する乗算よりも、64 ビットの結果を生成する 64 ビット同士の整数乗算を優先します。

アセンブリー/コンパイラー・コーディング規則 60 (影響 ML、一般性 M): 128 ビット乗算の下位 64 ビットの結果にアクセスした後に、上位 64 ビットの結果にアクセスしないようにします。

Sandy Bridge† マイクロアーキテクチャーでは、128 ビットの乗算の下位 64 ビットの結果は 3 サイクル後に使用できるのに対し、上位 64 ビットの結果は下位 64 ビットの結果から 1 サイクル遅れて使用可能になります。これにより、大きな整数の整数乗算および除算の計算速度を向上できます。

13.2.4 128 ビット整数除算を 128 ビット乗算で置き換え

現在のコンパイラーは、定数除数を含む高級言語コードの整数除算数式において、IDIV/DIV 命令を IMUL/MUL 命令に置換してアセンブリー・シーケンスを生成できます。通常、コンパイラーが除数値を置換するのは、この除数値が 32 ビットの範囲内であり、かつコンパイル時に既知である場合です。除数値がコンパイル時に不明である場合、または除数が 32 ビット表現よりも大きい場合は、DIV または IDIV が生成されます。

128 ビットの被除数を含む DIV 命令のレイテンシーはかなり長くなります。被除数値が 64 ビットよりも大きい場合のレイテンシーは 70～ 90 サイクルとなります。

整数除算を 128 ビットの整数乗算に変換する際にコンパイラーが利用する基本的な手法は、モジュラー計算の合同原則に基づきます。これは大きな除数値の処理まで簡単に拡張でき、128 ビットの高速な IMUL/MUL 操作を利用することができます。

整数式:
被除数= Q * 除数 + R
または
Q = 下限(被除数/除数)、R = 被除数 - Q * 除数

実数ドメインに変換:
下限(被除数/除数)= 被除数/ 除数 - R/除数、は以下と等しくなります
Q * C2 = 被除数 *(C2 /除数) + R * C2/除数

最後の項の丸めを制御するため、C2 = 2^N を選択できます。この場合、次のようになります。
Q = ((被除数 * (C2 /除数)) >> N) + ((R * C2/除数) >> N)

「除数」がコンパイル時に既知である場合、(C2/ 除数) は合同定数 Cx = 上限(C2/ 除数) として事前に計算でき、商は整数倍数によって算出した後にシフトできます。
Q = (被除数 * Cx) >> N;
R = 被除数 - ((被除数 * Cx)>> N) * 除数

128 ビットの IDIV/DIV 命令は、数値例外の発生を避けるため、除数、商、剰余の範囲を 64 ビット内に制限します。これは、この 3 つの値のいずれかが 64 ビットの上限に近い場合、および被除数値が 128 ビットの上限に近い場合に問題となります。

この問題は、大きなシフトカウント N を選択し、(被除数 * Cx) 演算を 128 ビット範囲から次に計算効率の良い範囲に拡張することで解決できます。例えば、(被除数 * Cx) が 128 ビットよりも大きく、N が 63 ビットよりも大きい場合、192 ビットの乗算を実装することなく、128 ビットの MUL を使用して、192 ビットの結果の計算ビット 191:64 を利用できます。

合同定数 Cx を選択する方法は以下のとおりです。

- 被除数の範囲が 64 ビット内の場合: Nmin ~ BSR(除数) + 63
- 除数の範囲と比べたときに商/剰余の動的な範囲が異なる場合、商/剰余が効率的に算出できるよう、適切に N を生成します。

除数が 10^16 で、符号なしの被除数が 64 ビットの範囲に近い商/剰余の計算について考えてみます。例 13-1 に、「MUL r64」命令を使用した、64 ビットの除数による 64 ビットの被除数の処理を示します。

例 13-1 64 ビットの除数による 64 ビットの商および剰余の計算

```
_Cx10to16: ; シフトカウント N=117 の合同定数 10^16
   DD 0c44de15ch ; floor ( (2^117 / 10^16) + 1)
   DD 0e69594beh ; 128 ビットの倍数で減るように Cx の長さを最適化
_tento16: ; 10^16
   DD 6fc10000h
   DD 002386f2h

   mov r9, qword ptr [rcx] ; 64 ビットの被除数をロード
   mov rax, r9
   mov rsi, _Cx10to16 ; シフトカウント N=117 の合同定数 10^16
   mul [rsi] ; 128 ビット乗算
   mov r10, qword ptr 8[rsi] ; 除数 10^16 のロード
   shr rdx, 53; ;
   mov r8, rdx

   mov rax, r8
   mul r10 ; 128 ビット乗算
   sub r9, rax; ;
   jae remain
   sub r8, 1 ; 丸めによって 1 つずれているかもしれない
   mov rax, r8
   mul r10 ; 128 ビット乗算
   sub r9, rax; ;
remain:
   mov rdx, r8 ; 商
   mov rax, r9 ; 余り
```

例 13-2 に、同様の手法を使用した 64 ビットの除数による 128 ビットの被除数の処理を示します。

例 13-2 64 ビットの除数による 128 ビットの商および剰余の計算

```
   mov rax, qword ptr [rcx] ; メモリーから 128 ビット被除数の 63:0 ビットをロード
   mov rsi, _Cx10to16 ; シフトカウント N=117 の合同定数 10^16
   mov r9, qword ptr [rsi] ; 合同定数をロード
   mul r9 ; 128 ビット乗算
   xor r11, r11 ; 積算器のクリア
   mov rax, qword ptr 8[rcx] ; 128 ビット被除数の 127:64 ビットをロード
   shr rdx, 53; ;
   mov r10, rdx ; 192 ビット結果の 127:64 ビットを初期化
   mul r9 ; 191:128 ビットを積算
   add rax, r10; ;
   adc rdx, r11; ;
   shr rax, 53; ;
   shl rdx, 11; ;
   or rdx, rax; ;
   mov r8, qword ptr 8[rsi] ; 除数 10^16 をロード
   mov r9, rdx; ; おおよその商、1 つずれているかもしれない
   mov rax, r8
   mul r9 ; 商* 除数 > 被除数 ?
   sub rdx, qword ptr 8[rcx] ;
   sbb rax, qword ptr [rcx] ;

   jb remain
   sub r9, 1 ; これは丸めによって 1 つずれているかもしれない
   mov rax, r8 ; 除数 10^16 の復元
   mul r9 ; 最終的な商 * 除数
   sub rax, qword ptr [rcx] ;
   sbb rdx, qword ptr 8[rcx] ;
remain:
   mov rdx, r9 ; 商
   neg rax ; 余り
```

例 13-1 および例 13-2 に示す手法により、128 ビットの被除数の剰余/商の計算速度を、32 ビットの整数除算のコストと同じかそれ以下に向上させることができます。

上記の手法を 64 ビットよりも大きい除数の処理に拡張するのは比較的容易です。シフトカウント N > 128 ビットを選択することは、考慮に値する 1 つの最適化です。これによって、(被除数 * Cx) の関連する上位ビットを計算するために必要となる 128 ビットの MUL の数を減らすことができます。

13.2.5 完全な 64 ビットへの符号拡張

Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサーは、64 ビット・モードで単一のマイクロオペレーション (uop) により 64 ビットへの符号拡張を行うことができます。64 ビット・モードでは、デスティネーションが 32 ビットの場合、上位 32 ビットをゼロにする必要があります。

上位 32 ビットをゼロにするには、追加のマイクロオペレーション (uop) が必要になり、64 ビットへの符号拡張よりも効率が悪くなります。64 ビットに符号拡張すると、命令が 1 バイト長くなりますが、Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサーでは、トレースキャッシュにストアしなければならないマイクロオペレーション (uop) が減少するので、パフォーマンスが向上します。

例えば、1 バイトを ESI に符号拡張するには、以下を使用します。

```
movsx rsi, BYTE PTR[rax]
```

以下は使用すべきではありません。

```
movsx esi, BYTE PTR[rax]
```

次の命令で 32 ビット形式の ESI レジスターを使用する場合でも、結果は同じです。この最適化は、予期しない依存関係を解消するのにも利用できます。例えば、プログラムが 16 ビット値をレジスターに書き込んでから、8 ビット値をそのレジスターに書き込む場合、デスティネーションのビット 15:8 が不要であれば、符号拡張を伴う書き込みを利用します (利用可能であれば)。

次に例を示します。

```
mov r8w, r9w; ビット 63:15 をマージして保存する場合
mov r8b, r10b; ビット 63:8 をマージして保存する場合
```

これは以下に置き換えられます。

```
movsx r8, r9w ; ビット 63:8 を保存する必要がない場合
movsx r8, r10b ; ビット 63:8 を保存する必要がない場合
```

上記の例では、R8W への移動および R8B への移動は、マージによってレジスター内の残りのビットを保持する必要があります。「MOV R8W, R9W」と「MOV R8B, R10B」の間には、R8 に対する暗黙的な依存関係があります。MOVSX を使用すると、実際の依存関係が解消され、出力の依存関係のみが残ります。この依存関係は、プロセッサーがリネームによって排除できます。

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、上位 32 ビットをゼロにすることは、64 ビットへの符号拡張よりも高速です。Nehalem† マイクロアーキテクチャー・ベースのプロセッサーでは、上位ビットをゼロにすること、または符号拡張することは単一のマイクロオペレーション (uop) で行われます。

13.3 64 ビット・モード向けの代替えコーディング規則

13.3.1 64 ビット演算結果では 2 つの 32 ビット・レジスターの代わりに 64 ビット・レジスターを使用

レガシーの 32 ビット・モードでは、拡張精度整数演算 (64 ビット演算など) をサポートする機能が提供されています。ただし、64 ビット・モードでは、64 ビット演算をネイティブにサポートしています。64 ビットの整数が必要な場合は、64 ビット形式の演算命令を使用します。

32 ビットのレガシーモードでは、32 ビット同士の整数乗算から 64 ビットの結果を得るのに 3 つのレジスターが必要となり、結果は、EDX:EAX のペアにストアされます。64 ビット・モードの命令が利用可能な場合、64 ビットの結果が必要なときに 32 ビット命令を使用するのは最適な実装とはいえません。拡張レジスターを使用します。

例えば、以下のコードシーケンスでは、64 ビット・レジスターに符号拡張された 32 ビット値をロードして、乗算を行います。

```
movsx rax, DWORD PTR[x]
movsx rcx, DWORD PTR[y]
imul rax, rcx
```

上記の 64 ビット版の方が、以下の 32 ビット版よりも効率的です。

```
mov eax, DWORD PTR[x]
mov ecx, DWORD PTR[y]
imul ecx
```

上記の 32 ビット版では、EAX がソースである必要があります。結果は、単一の 64 ビット・レジスターではなく、EDX:EAX のペアに格納されます。

アセンブリー/コンパイラー・コーディング規則 61 (影響 ML、一般性 M): 64 ビットの結果が必要な 32 ビットの整数乗算には、64 ビット版の乗算を使用します。

32 ビットのレガシーモードで 2 つの 64 ビット値を加算するには、add 命令の後に adc 命令を続けて使用します。例えば、2 つの 64 ビット変数 (X および Y) を加算するには、以下の 4 つの命令を使用できます。

```
mov eax, DWORD PTR[X]
mov edx, DWORD PTR[X+4]
add eax, DWORD PTR[Y]
adc edx, DWORD PTR[Y+4]
```

結果は、2 つのレジスターペア EDX:EAX に格納されます。

64 ビット・モードの場合、上記のシーケンスを以下のように短縮できます。

```
mov rax, QWORD PTR[X]
add rax, QWORD PTR[Y]
```

結果は RAX に格納されます。必要なレジスターは、2 つではなく 1 つです。

アセンブリー/コンパイラー・コーディング規則 62 (影響 ML、一般性 M): 64 ビットの加算には、64 ビット版の加算を使用します。

13.3.2 ソフトウェア・プリフェッチの使用

ソフトウェア・プリフェッチを使用せず、ハードウェア・プリフェッチを利用できるようにデータ・アクセス・パターンを編成する手法を検討する際は、第 3 章と第 9 章の推奨事項に従うことを推奨します。

アセンブリー/コンパイラー・コーディング規則 63 (影響 L、一般性 L): ソフトウェア・プリフェッチ命令が必要な場合は、インテル® SSE 命令セットで提供されているプリフェッチ命令を使用します。

第 14 章
テキスト処理/字句解析/構文解析向けのインテル® SSE4.2 と SIMD プログラミング

インテル® SSE4.2 で提供される文字列/テキスト処理は、従来の SIMD 整数ベクトル処理とは異なる手法が要求される分野にも及んでいます。従来の文字列/テキスト・アルゴリズムの多くは文字ベースであり、この場合の文字はバイトサイズが固定または可変のエンコード (コードポイント) によって表現されました。テキストデータは大量のローデータからなり、通常は文脈情報が含まれています。ロー・テキスト・データに含まれる文脈情報には、文字の値、文字の位置、文字のエンコード形式、文字セットのサブセット、明示的または暗黙的な長さの文字列、トークン、区切り文字など、広範な属性に対応したアルゴリズムが要求されます。文脈オブジェクトは、事前に定義された文字サブセット内の連続文字 (10 進値の文字列など) によって表現できます。テキストストリームには、異なる文脈オブジェクトを区切る組込み型の状態遷移 (タグ区切りフィールドなど) を含めることができます。

従来の SIMD 整数ベクトル命令でも、一部の簡単な文字列処理関数を高速化できます。インテル® SSE4.2 では、構造化されたまたは構造化されていないテキストデータの文字列/テキスト処理、字句解析、構文解析を目的とした演算アルゴリズムを強化する 4 つの新しい命令が追加されました。

14.1 インテル® SSE4.2 の文字列/テキスト命令

インテル® SSE4.2 には、PCMPESTRI、PCMPESTRM、PCMPISTRI、PCMPISTRM の 4 つの命令が用意されています。SIMD プログラミングの効率とこの 4 つの命令に組み込まれた字句プリミティブを組み合わせることによって、文字列/テキスト処理を高速化できます。これらの命令の簡単な使用例としては、文字列長の判断、文字列の直接比較、文字列の大文字/小文字の処理、区切り文字/トークンの処理、ワード境界の検索、大規模なテキストブロックにおけるサブ文字列の一致検索が挙げられます。インテル® SSE4.2 を効果的に利用するアプリケーションでは、XML の構文解析とスキーマの検証を高速化できます。

プロセッサーがインテル® SSE4.2 をサポートしているかどうかは、EAX の入力値を 1 に設定して CPUID 命令を実行した後に ECX [bit 20] に返される機能フラグの値によって示されます。CPUID.01H:ECX.SSE4_2 [bit 20] = 1 の場合にインテル® SSE4.2 がサポートされていることが分かります。ソフトウェアは、この 4 つの命令を使用する前に、CPUID.01H:ECX.SSE4_2 [bit 20] がセットされていることを確認する必要があります(CPUID.01H:ECX.SSE4_2 = 1 の確認は、PCMPGTQ または CRC32 の使用前にも必要です。CPUID.01H:ECX.POPCNT[Bit 23] = 1 の確認は、POPCNT 命令の使用前に必要です)。

これらの文字列/テキスト処理命令は、テキスト・フラグメントに対して最大 256 の比較演算を実行することによって行われます。各テキスト・フラグメントの最大サイズは、16 バイトです。命令は、バイト要素またはワード要素を含むさまざまな形式のフラグメントを処理できます。4 つの命令はどれも、2 つのテキスト・フラグメントに対して 4 種類の並列比較演算を実行するように設定できます。

2 つのテキスト・フラグメントに対する並列比較の中間結果の合計は、バイト要素の処理では 16 ビット、ワード要素の処理では 8 ビットのビットパターンになります。各命令は高い柔軟性を備えており、命令構文中の即値オペランドのビットフィールドを使用して、最初の中間結果の単項変換 (極性) を設定できます。

また、命令の即値オペランドでは出力選択の制御が可能であり、命令から得られる最終結果の柔軟性をさらに高めます。図 14-1 に、これらの命令の高い設定の可能性を示します。

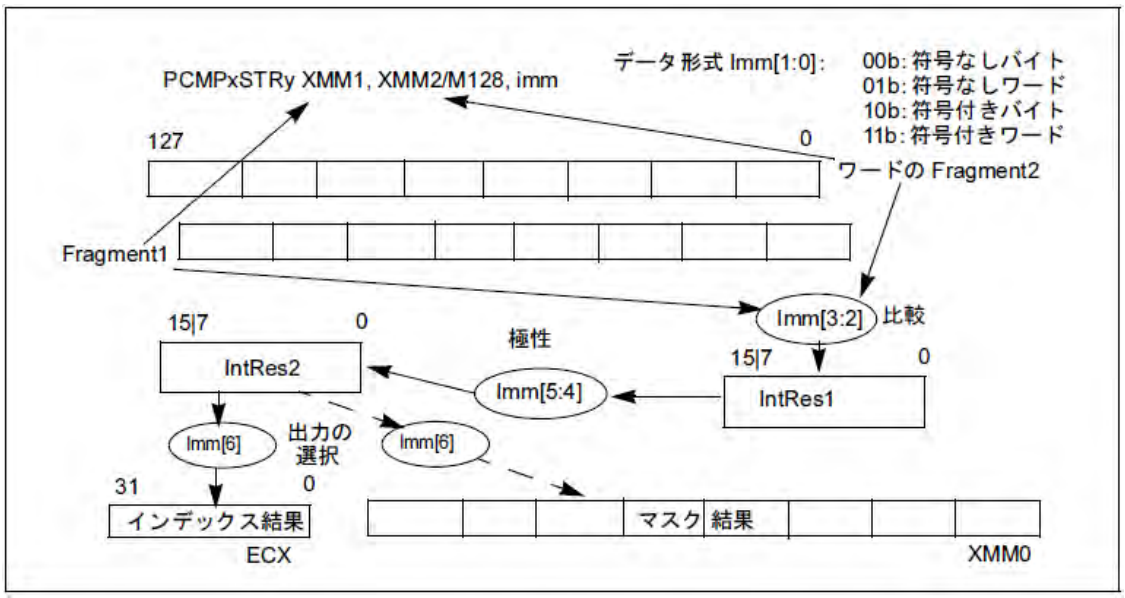


図 14-1 インテル® SSE4.2 の文字列/テキスト命令における即値オペランドの制御

PCMPxSTRI 命令は、最終結果を整数インデックスとして ECX に生成し、PCMPxSTRM 命令は、最終結果をビットマスクとして XMM0 レジスターに生成します。PCMPISTRy 命令では、NULL 終端文字による暗黙的なレングス制御を利用した文字列/テキスト・フラグメント処理をサポートして、サイズが不明な文字列/テキストを扱います。また、PCMPESTRy 命令では、EDX：EAX レジスターペアによる明示的なレングス制御により、ソースオペランド中のテキスト・フラグメントのレングスを指定できます。

最初の中間結果 IntRes1 は、ビットフィールド imm[3:2] のエンコードに従って、各テキスト・フラグメント中のデータ要素ペアに対する並列比較演算で得られたビットパターンの合計結果となります。表 14-1 を参照してください。

表 14-1 インテル® SSE4.2 の文字列/テキスト命令における N 要素の比較演算

imm[3:2]	**名称**	**以下の場合に IntRes1[i] が真**	**考えられる用法**
00B	Equal Any (いずれかと等しい)	fragment2 の要素 i が、fragment1 のいずれかの要素 j に一致する	トークン化、XML パーサー
01B	range (範囲)	fragment2 の要素 i が、fragment1 で指定されたいずれかの範囲ペア内に収まる	サブセット、大文字/小文字の処理、XML パーサー、スキーマの検証
10B	Equal Each (それぞれ等しい)	fragment2 の要素 i が、fragment1 の要素 i に一致する	Strcmp()
11B	Equal Ordered (等しい並び)	fragment2 の要素 i と後続の連続する有効要素が、要素 0 から始まる fragment1 と完全または部分的に一致する	サブ文字列の検索、KMP、strstr( )

imm[1:0] を利用した入力データ要素形式の選択では、符号付きまたは符号なしのバイト/ワード要素をサポートします。ビットフィールド imm[5:4] を利用すると、IntRes1 に対して単項変換を適用できます。表 14-2 を参照してください。

表 14-2 インテル® SSE4.2 の文字列/テキスト命令における IntRes1 の単項変換

Imm[5:4]	名称	IntRes2[i] =	考えられる用法
00B	No Change (変更なし)	IntRes1[i]	
01B	Invert (反転)	-IntRes1[i]	
10B	No Change (変更なし)	IntRes1[i]	
11B	Mask Negative (マスク否定)	fragment2 の要素 i が無効の場合は IntRes1[i]。それ以外は -IntRes1[i]。	

出力選択フィールド imm[6] については、表 14-3 で説明します。

表 14-3 インテル® SSE4.2 の文字列/テキスト命令における出力選択フィールド imm[6]

imm[6]	命令	最終結果	考えられる用法
00B	PCMPxSTRI	IntRes2 != 0 の場合は、ECX = IntRes2 にセットされた最下位ビットのオフセット。 それ以外の場合は、ECX = 16 バイトごとのデータ要素の数。	
01B	PCMPxSTRM	XMM0 = ZeroExtend(IntRes2);	
10B	PCMPxSTRI	IntRes2 != 0 の場合は、ECX = IntRes2 にセットされた最上位ビットのオフセット。 それ以外の場合は、ECX = 16 バイトごとのデータ要素の数。	
11B	PCMPxSTRM	XMM0 のデータ要素 i = SignExtend(IntRes2[i]);	

各データ要素ペアに対する比較演算について表 14-4 に定義するとおりです。表 14-4 には、有効なデータ要素間の比較演算の種類 (表 14-4 の最終行) と、ソースオペランド中のフラグメントに無効なデータ要素が含まれる場合の境界条件 (表 14-4 の 1 ~ 3 行目)が定義されています。算術比較は、表 14-4 の 4 行目に示すように、fragment1 と fragment2 のデータ要素がいずれも有効な要素の場合のみ実行されます。

表 14-4 インテル® SSE4.2 の文字列/テキスト命令における要素ペアの比較の定義

Fragment1 の要素	Fragment2 の要素	Imm[3:2]=00B, Equal Any	Imm[3:2]=01B, Ranges	Imm[3:2]=10B, Equal Each	Imm[3:2]=11B, Equal Ordered
無効	無効	偽を強制	偽を強制	真を強制	真を強制
無効	有効	偽を強制	偽を強制	偽を強制	真を強制
有効	無効	偽を強制	偽を強制	偽を強制	偽を強制
有効	有効	比較	比較	比較	比較

文字列/テキスト処理命令には、文字列終了 (EOS) 状態に対応する補助手段が複数用意されています (表 14-5 を参照)。また、テキスト処理の要件を簡素化するため、PCMPxSTRy 命令は 16 バイト・アライメントが不要なように設計されています。

表 14-5 インテル® SSE4.2 の文字列/テキスト命令と EFLAG の関係

EFLAG	説明	考えられる用法
CF	IntRes2 = 0 ならリセット; でなければセット	CF = 0 の場合は ECX = 次にスキャンするデータ要素の数
ZF	fragment2 の 16 バイト全体が有効な場合はリセット	文字列終了の可能性
SF	fragment1 の 16 バイト全体が有効な場合はリセット	
OF	IntRes2[0];	

14.1.1 CRC32

CRC32 命令は、バイト/ワード/ダブルワード/クワッドワードのデータストリームの 32 ビットの巡回冗長チェックサム・シグネチャーを計算します。また、ハッシュ関数としても使用できます。例えば、辞書はハッシュ・インデックスを使用して、文字列を逆参照します。CRC32 命令は、このような状況で容易に応用できます。

例 14-1 に示すのは、ハッシュテーブルに登録する文字列のハッシュ・インデックスの評価に使用できる簡単なハッシュ関数です。ハッシュ・インデックスは通常、ハッシュテーブルのサイズを除数としてハッシュ値の剰余を求めることにより得られます。

例 14-1 ハッシュ関数の例

```
unsigned int hash_str(unsigned char* pStr)
{  unsigned int hVal = (unsigned int)(*pStr++);
   while (*pStr)
   { hVal = (hashVal * CONST_A) + (hVal >> 24) + (unsigned int)(*pStr++);
   }
   return hVal;
}
```

CRC32 命令を使用すると、別のハッシュ関数を作成できます。例 14-2 では、32 ビット単位の CRC32 命令を利用して入力データストリームのシグネチャー値を更新します。小～中規模の文字列では、ハードウェアでアクセラレートされた CRC32 を利用すると、例 14-1 を 2 倍高速化できます。

例 14-2 CRC32 を使用するハッシュ関数

```
static unsigned cn_7e = 0x7efefeff, Cn_81 = 0x81010100;

unsigned int hash_str_32_crc32x(unsigned char* pStr)
{  unsigned *pDW = (unsigned *) &pStr[1];
   unsigned short *pWd = (unsigned short *) &pStr[1];
   unsigned int tmp, hVal = (unsigned int)(*pStr);
   if( !pStr[1]) ;
   else {
      tmp = ((pDW[0] +cn_7e ) ^(pDW[0]^ -1)) & Cn_81;
      while ( !tmp ) // *pDW のバイトが 0x00 までループ
      {
         hVal = _mm_crc32_u32 (hVal, *pDW ++);
         tmp = ((pDW[0] +cn_7e ) ^(pDW[0]^ -1)) & Cn_81;
      };
      if(!pDW[0]);
      else if(pDW[0] < 0x100) { // 最後のバイトが非ゼロで終了
         hVal = _mm_crc32_u8 (hVal, pDW[0]);
      }
      else if(pDW[0] < 0x10000) { // 最後の 2 バイトが非ゼロで終了
         hVal = _mm_crc32_u16 (hVal, pDW[0]);
      }
      else if(pDW[0] < 0x10000) { // 最後の 3 バイトが非ゼロで終了
         hVal = _mm_crc32_u32 (hVal, pDW[0]);
      }
   }
   return hVal;
}
```

14.2 インテル® SSE4.2 の文字列/テキスト命令を使用

高水準言語やシステム・ライブラリーの一部として提供されている文字列ライブラリーは、各種のアプリケーションやシステム・ソフトウェアのさまざまな状況で使用されます。このような文字列処理は、PCMPESTRI、PCMPESTRM、PCMPISTRI、PCMPISTRM を実装した文字列ライブラリーに置き換えることで高速化できます。

システムで提供される文字列ライブラリーには、標準化された文字列処理機能とインターフェイスが用意されていますが、構造化されたドキュメントを処理する大半では、システム提供の文字列ライブラリーにはない精巧さ、最適化、サービスが数多く必要になります。例えば、構造化されたドキュメントを処理するソフトウェアでは多くの場合、さまざまなクラス・オブジェクトを用意して、アプリケーション固有のニーズを満たすビルディング・ブロック機能を提供します。アプリケーションは通常、同等の文字列ライブラリー・サービスを個別のクラス (文字列、レクサー、パーサー) に分割するか、メモリー管理機能を文字列処理/字句解析/構文解析オブジェクトに統合するかを選択します。

PCMPESTRI、PCMPESTRM、PCMPISTRI、PCMPISTRM の各命令は汎用プリミティブであり、ソフトウェアが置換可能な文字列ライブラリーを構築したり、構造化ドキュメント処理用の字句解析/構文解析サービスを提供するためにクラス階層を構築する際に利用できます。後者の例としては、XML の構文解析やスキーマの検証が挙げられます。

構造化されていないローテキスト/文字列データは文字で構成され、自然なアライメントは考慮されていません。したがって、PCMPESTRI、PCMPESTRM、PCMPISTRI、PCMPISTRM 命令は、ほかの 128 ビット SIMD 整数ベクトル処理命令が持つ 16 バイト・アライメントの制限が不要なように設計されています。

メモリー・アライメントに関しては、PCMPESTRI、PCMPESTRM、PCMPISTRI、PCMPISTRM は、ほかのアライメントの必要がない 128 ビット・メモリー・アクセス命令 (MOVDQU など) と同様に、アライメントされていないメモリー・ロードをサポートしています。

アライメントされていないメモリーアクセスでは、リング 3 アプリケーション空間や特権空間で実行されるコードによっては、ほかのコーディング手法が必要となる特殊な状況が発生することがあります。具体的には、アライメントされていない 16 バイト・ロードがページ境界をまたぐことがあります。14.2.1 節では、アプリケーション・コードで利用可能な手法について説明しています。14.2.2 節では、文字列ライブラリー関数で対処が必要な状況について説明します。14.3 節では、アプリケーション・コードがメモリーバッファー割り当てを制御する状況で、PCMPESTRI、PCMPESTRM、PCMPISTRI、PCMPISTRM 命令を使用して、文字列ライブラリー関数と同等の機能を実装する例を紹介します。

14.2.1 アライメントされていないメモリーアクセスとバッファーサイズの管理

アプリケーション・コードでは、メモリーバッファー割り当てのサイズに関して、アライメントされていない SIMD メモリーのセマンティクスとアプリケーションの用途を考慮する必要があります。

一部のアプリケーションでは、バッファーの有効範囲と有効なアプリケーション・データ・サイズ (ビデオフレームなど) を区別することが望ましいことがあります。その場合、前者は後者と同じかそれよりも大きくなければなりません。

アライメントされていない 128 ビット SIMD メモリーアクセスが必要なアルゴリズムをサポートするには、呼び出し先関数がアライメントされていない 128 ビット SIMD メモリー操作でアドレスポインターを安全に使用できるように、呼び出し元関数によるメモリーバッファー割り当てでパディング領域の追加を検討する必要があります。

パディングの最小サイズは、アライメントされていない SIMD メモリーアクセスで使用される SIMD レジスターの幅となります。

14.2.2 アライメントされていないメモリーアクセスと文字列ライブラリー

アプリケーション・コードや特権コードでは、文字列ライブラリー関数を使用できます。ただし文字列ライブラリー関数については、メモリーアクセス権を違反しないように注意しなければなりません。したがって、アライメントされていない SIMD アクセスを利用する文字列ライブラリーに置き換える場合、メモリーアクセス違反が発生しない特別な手法を用いる必要があります。

14.3.6 節では、インテル® SSE4.2 で実装された代替文字列ライブラリー関数の例を紹介し、ページ境界をまたぐことなくアライメントされていない 128 ビット・メモリー・アクセスを利用する手法について説明します。

14.3 インテル® SSE4.2 のアプリケーション・コーディングのガイドラインと例

インテル® SSE4.2 命令が実装されたソフトウェアは、CPUID 機能フラグのメカニズムを利用して、プロセッサーがインテル® SSE4.2 をサポートしていることを確認しなければなりません。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 12 章と、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の第 3 章の CPUID を参照してください。

以降では、複雑な文字列/テキスト処理の例をいくつか紹介し、SIMD アプローチを応用してインテル® SSE4.2 の PCMPxSTRy 命令を用いた文字列/テキスト処理を実現する基本的な手法を説明します。簡潔にするために、呼び出し元関数によって十分なバッファーサイズが割り当てられた状況でのバイトデータ形式の文字列/テキストについて検討します。バッファーサイズが十分であれば、ページ境界をまたぐことなく、アライメントされていないメモリーからの 128 ビット SIMD ロードをサポートできます。

14.3.1 NULL 文字の識別 (strlen と同等)

最も頻繁に使用される文字列関数は strlen() でしょう。strlen() の字句解析要件は、サイズが不明なテキストブロック中の NULL 文字 (EOS 条件) を識別することです。総当たり方式のバイト単位コードでは、一度に 1 バイトずつロードするため、データのフェッチが非効率です。

汎用命令を使用する最適化されたコードでは、32 ビット環境ではダブルワード操作 (64 ビット環境ではクワッドワード操作) を利用して、反復回数を削減できます。

例 14-3 に、strlen() の 32 ビット・アセンブリー・コードを示します。EOS 条件を処理するピーク時の実行スループットは、メインループ中の 8 個の ALU 命令によって決定されます。

例 14-3 汎用命令を使用した strlen()

```
int strlen_asm(const char* s1)
{int len = 0;
_asm{
   mov ecx, s1
   test ecx, 3 ; アドレスが dword にアラインされているかテスト
   je short _main_loop1 ; dword にアラインされたロードは高速
_malign_str1:
   mov al, byte ptr [ecx] ; 1 バイトずつ読み込み
   add ecx, 1
   test al, al ; NULL を見つけたら、レングスを計算
   je short _byte3a
   test ecx, 3; 再度アドレスが dword にアラインされているかテスト
   jne short _malign_str1; アラインされていなければ繰り返し
   align16
   (続く)
```

```
_main_loop1:; 4 バイトのブロックを読み込み、dword に NULL があるかをチェック
   mov eax, [ecx]; ループ回数を減らすため 4 バイトを読み込み
   mov edx, 7efefeffh
   add edx, eax
   xor eax, -1
   xor eax, edx
   add ecx, 4; アドレスポインターを 4 つ増加
   test eax, 81010100h ; 4 バイト内に NULL がなければ、次の 4 バイトへ
   je short _main_loop1
   ; 読み込んだ 4 バイト内に NULL があった場合、
   ; すでにポインター ecx は 4 つカウントされており、dword を失う
   mov eax, [ecx -4]; NULL を含む dword を再度読み込み
   test al, al ; byte0 が NULL の場合
   je short _byte0a; バイトが NULL
   test ah, ah ; byte1 が NULL の場合
   je short _byte1a
   test eax, 00ff0000h; byte2 が NULLの場合
   je short _byte2a
   test eax, 00ff000000h; byte3 が NULL の場合
   je short _byte3a
   jmp short _main_loop1
_byte3a:
   ; NULL を検出したが、ポインターはすでに 1 つ先にいる
   lea eax, [ecx-1]; NULL に相当する有効アドレスをロード
   mov ecx, s1
   sub eax, ecx; NULL と開始アドレスの差
   jmp short _resulta
_byte2a:
   lea eax, [ecx-2]
   mov ecx, s1
   sub eax, ecx
   jmp short _resulta
_byte1a:
   lea eax, [ecx-3]
   mov ecx, s1
   sub eax, ecx
   jmp short _resulta
_byte0a:
   lea eax, [ecx-4]
   mov ecx, s1
   sub eax, ecx
_resulta:
   mov len, eax; 結果をストア
   }
   return len;
}
```

同等の EOS 識別機能は、PCMPISTRI を使用して実装できます。例 14-4 は単純なインテル® SSE4.2 コードであり、16 バイト・テキスト・フラグメントをロードしてテキストブロックをスキャンし、NULL 終端文字を検索しています。例 14-5 は最適化されたインテル® SSE4.2 コードであり、メモリー・ディスアンビゲーションを利用して命令レベルの並列性を高めることの重要性を実証しています。

例 14-4 最適化前の PCMPISTRI コードによる EOS 処理

```
static char ssch2[16]= {0x1, 0xff, 0x00, }; // 非 NULL 文字値の範囲
int strlen_un_optimized(const char* s1)
{int len = 0;
   _asm{
   mov eax, s1
   movdquxmm2, ssch2 ; 範囲内の文字列ペアをロード (0x01 から 0xff)
   xor ecx, ecx ; 初期オフセットを 0 に設定
_loopc:
   add eax, ecx ; テキストの開始位置へポインターを移動
   pcmpistri xmm2, [eax], 14h; 符号なしバイト、範囲、反転、lsb インデックスは ecx へ返る
      ; [eax] で示される 16 バイト範囲に NULL 文字があれば、zf が設定される
      ; 16 バイト範囲に NULL 文字がなければ、ecx に 16 が返る
   jnz short _loopc; xmm1 には NULL は含まれず、ecx は 16、検索を続行
   ; xmm1 に NULL 文字が含まれていると、ecx にはそのオフセット i が設定される
   add eax, ecx ; 最後の文字列2/xmm1 のアドレスを ecx に加算
   mov edx, s1; 入力文字列のアドレスを戻す
   sub eax, edx; 文字列のレングス
   mov len, eax; 結果をストア
   }
   return len;
}
```

例 14-4 に示したコードシーケンスには、3 個の命令で構成されたループがあります。パフォーマンス・チューニングの観点から見ると、前のループ反復の結果 (ECX 値) に基づいてアドレス更新が行われるため、ループ反復にはループ伝搬依存があります。このループ伝搬依存は、アウトオブオーダー・エンジンの機能を妨げ、命令シーケンスの複数の反復を順方向に進行させます。ループ伝搬依存が存在すると、アウトオブオーダー実行ではメモリーロードのレイテンシー、各命令のレイテンシー、バイパス遅延を吸収できません。

例 14-5 に、ループ伝搬依存を排除する簡単な最適化手法を示します。

メモリー・ディスアンビゲーション手法を利用してループ伝搬依存を排除すると、例 14-5 に示す 3 個の命令で構成されるシーケンス中のレイテンシーの累積は、複数の反復を実行する過程で吸収され、各反復を実行 (16 バイトを処理) する純コストは 3 サイクル未満となります。また、例 14-3 に示すように 8 個の ALU 命令を用いて 4 バイトの文字列データを処理しても、反復ごとに 3 サイクル未満です。一方、例 14-4 に示したコードシーケンスの各反復は、ループ伝搬依存が原因で 10 サイクル以上かかります。

例 14-5 ループ伝搬依存がない PCMPISTRI を使用した strlen()

```
int strlen_sse4_2(const char* s1)
{int len = 0;
   _asm{
   mov eax, s1
   movdquxmm2, ssch2 ; 範囲内の文字列ペアをロード (0x01 から 0xff)
   xor ecx, ecx ; 初期オフセットを 0 に設定
   sub eax, 16 ; 追加の命令と分岐を防ぐためアドレスを計算
_loopc:
   add eax, 16 ; アドレスポインターと各反復のあいまいなロードアドレスを調整する
   pcmpistri xmm2, [eax], 14h; 符号なしバイト、範囲、反転、lsb インデックスは ecx へ返る
       ; [eax] で示される 16 バイト範囲に NULL文字があれば、zf が設定される
       ; 16 バイト範囲に NULL 文字がなければ、ecx に 16 が返る
   jnz short _loopc ; [eax] に NULL 文字がなければ、ecx には 16 が設定される、これはあいまいさの解
   消
_endofstring:
   add eax, ecx ; 最後の文字列2/xmm1 のアドレスを ecx に加算
   mov edx, s1; 入力文字列のアドレスを戻す
   sub eax, edx; 文字列のレングス
   mov len, eax; 結果をストア
   }
   return len;
}
```

インテル® SSE4.2 コーディング規則 5 (影響 H、一般性 H): アドレス調整を PCMPESTRI、PCMPESTRM、PCMPISTRI、PCMPISTRM の ECX 結果に依存しているループ伝搬依存は、最小限に抑えなければなりません。メモリー・ディスアンビゲーション・ハードウェアを利用するには、ECX 結果が 16 (バイト) または 8 (ワード) になると予期されるコードパスを特定し、これらの ECX 値をアドレス調整式の定数に置き換えます。

14.3.2 スペース類の文字の識別

文字単位のテキスト処理アルゴリズムでは、文字単位のアプローチを効率化する課題に向けて、固有のタスクに対応する手法が開発されてきました。そうした手法の 1 つとして、文字サブセットの分類におけるルックアップ・テーブルの使用があげられます。例えば、一部のアプリケーションでは、英数文字とスペース類の文字を区別する場合に、複数の文字をスペース文字として扱うことがあります。

例 14-6 は、スペース以外の連続した文字の開始と終了をマークするため、スペース類の文字を識別する簡単な例を示しています。

例 14-6 C コードとバイト･スキャニング手法を使用した wordCnt( )

```
// スペース以外の連続した文字をカウント
// ソフトウェアごとに独自のスペース文字セットのマッピングを選択する
// この例では紹介のため単純な定義を利用する:
// スペース以外の文字は次を考慮:  A-Z, a-z, 0-9, そして '(アポストロフィー)
// この例では文字をビットパターンにマップする簡単な方法を利用する
// これで簡単に文字数をカウントできる
static char alphnrange[16]= {0x27, 0x27, 0x30, 0x39, 0x41, 0x5a, 0x61, 0x7a,
0x0};

static char alp_map8[32] = {0x0, 0x0, 0x0, 0x0, 0x80,0x0,0xff, 0x3,0xfe, 0xff,
0xff, 0x7, 0xfe, 0xff, 0xff, 0x7}; // 32
バイトのルックアップ･テーブル、最初に英数字のビットパターンをマップする
int wordcnt_c(const char* s1)
{int i, j, cnt = 0;
char cc, cc2;
char flg[3]; // 文字列の区切りを検出
   cc2 = cc = s1[0];
   // 一度の検索で複数の比較ができるよう、圧縮されたビットパターンを利用する
   if( alp_map8[cc>>3] & ( 1<< ( cc & 7) ) )
   { flg[1] = 1; } // ワード中の非スペース文字
   // 's' に続く単語を別々にカウントするのは適切でないため、
   // この例ではアポストロフィーを含めています。
   else
   { flg[1] = 0; } // 0: 空白、単語の一部とはみなさない
   i = 1; // 残りのブロックをスキャンする準備ができている
   // 単語の語数をカウントするように、ビットパターンの立下りエッジを検出する。
   // これは句読点を処理し、2 つの単語を接続するハイフン、
   // 連続した空白に対処できる
   while (cc2 )
   {  cc2 = s1[i];
      if( alp_map8[cc2>>3] & ( 1<< ( cc2 & 7) ) )
      { flg[2] = 1;} // スペースではない
      else
      { flg[2] = 0;} // スペース類
      if( !flg[2] && flg[1] )
      { cnt ++; }// 区切りを検出
      flg[1] = flg[2];
      i++;
   }
   return cnt;
}
```

例 14-6 では、ASCII コード値 0x0 ~ 0xff を表す 32 バイトのルックアップ･テーブルを作成し、指定した文字サブセットに一致する各ビットに 1 を割り当てて分割しています。このビット･ルックアップ手法では比較演算を簡素化できますが、データフェッチは依然として 1 バイト単位のままです。

例 14-7 に、PCMPISTRM を使用して単語をカウントする同等のコードを示します。ここではループ反復は、1 バイト単位ではなく 16 バイト単位で実行されます。また、範囲値のペアを使用して文字セットのサブセットを容易に表現でき、PCMPISTRI を 1 回実行するだけで範囲値とテキスト･フラグメント中の各バイトとの並列比較を行うことができます。

例 14-7 PCMPISTRM を使用した wordCnt( )

```
// インテル® SSE4.2 の例では、文字列セット {A-Z, a-z, 0-9, '} (空白でない)を
// 定義して単語をカウントする。各テキストの断片 (16バイトまで) は、
// 1 つ以上の立下りエッジを含む 16 ビット・パターンにマップされる。
// ビットごとのスキャンは効率が悪く、SIMD プログラミングに適さない。
// 立下りエッジの前には立ち上がりエッジがなくてはならず、ここでは popcnt を利用し、
// 2 つに分割することで、立ち上がりと立下りエッジを 2 つのビットパルスにマップし、
// 2 ビット・パルス中のビットをカウントする差分手法を行う。
int wdcnt_sse4_2(const char* s1)
{int len = 0;
   _asm{
   mov eax, s1
   movdquxmm3, alphnrange ; 非スペースコードを見つけるためレンジ値のペアをロード
   xor ecx, ecx
   xor esi, esi
   xor edx, edx
   movdquxmm1, [eax]
   pcmpistrm xmm3, xmm1, 04h ; スペース類文字は xmm0[15:0] で 0 で表現される
   movdqa xmm4, xmm0
   movdqa xmm1, xmm0
   psrld xmm4, 15 ; 次の反復のため MSB をセーブ
   movdqa xmm5, xmm1
   psllw xmm5, 1; 事実上 LSB はスペースにマップされる
   pxor xmm5, xmm0; 最初のエッジを作成する
   pextrd edi, xmm5, 0
   jz _lastfragment; xmm1 が NULL を含めば、ZF がセットされる
   popcnt edi, edi; 最初の断片はおそらく立ち上がりエッジを含む
   add esi, edi
   mov ecx, 16
_loopc:
   add eax, ecx ; アドレスポインターを進める
   movdquxmm1, [eax]
   pcmpistrm xmm3, xmm1, 04h ; スペース類文字は xmm0[15:0] で 0 で表現される
   movdqa xmm5, xmm4 ; 最後の反復からマスクの MSB を探す
   movdqa xmm4, xmm0
   psrld xmm4, 15 ; 次の反復のため、現在の反復の MSB をセーブ
   movdqa xmm1, xmm0
   psllw xmm1, 1
   por xmm5, xmm1 ; 最後の反復の MSB と残りの反復を組み合わせる
   pxor xmm5, xmm0; エッジパターンとバイナリー波形を区別する
   pextrdedi, xmm5, 0 ; エッジパターンは(最後から 1 ビット、この範囲から 15 ビット)
   jz _lastfragment; xmm1 が NULL を含めば、ZF がセットされる
   mov ecx, 16; xmm1 が NULL を含まなければ、16 バイト進める
   popcntedi, edi; 立ち上がりと立下りエッジをカウント
   add esi, edi; 両方のエッジのカウントを続行
   jmp short _loopc
_lastfragment:
   popcntedi, edi; 立ち上がりと立下りエッジをカウント
   add esi, edi; 両方のエッジのカウントを続行
   shr esi, 1 ; ワードカウントは立下りエッジの数に相当
   mov len, esi
   }
   return len;
}
```

14.3.3 サブ文字列の検索

strstr() は、標準文字列ライブラリーで頻繁に使用される関数です。ライブラリーは通常、参照文字列と 1 回の文字列比較に使用するターゲット文字列中のサブセットとの間で反復的な比較を行う、バイト単位の総当たり方式で strstr(sTarg, sRef) を実装します。ターゲットサブ文字列と参照文字列の先頭文字が異なる場合、以降のターゲットサブ文字列の比較ではターゲット文字列中の次のバイトから行うことができるため、バイト単位の総当たり方式は妥当な効率を示します。

文字列比較で複数の文字の部分一致 (すなわち、サブ文字列検索で参照文字列の先頭から部分一致) が検出され、その部分一致が偽の一致であると判定された場合、総当たり方式の検索プロセスは前に戻って、以前の文字列比較演算で処理済みの位置から文字列比較を再開する必要があります。これは、総当たり方式のサブ文字列検索アルゴリズムにおける再トレースの非効率性として知られています。図 14-2 を参照してください。

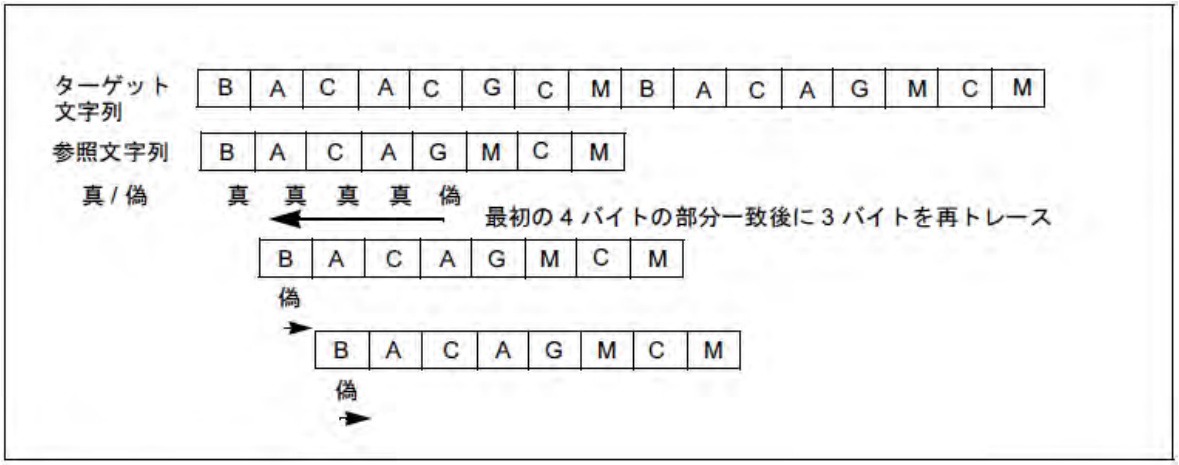


図 14-2 総当たり方式のバイト単位検索における再トレースの非効率性

Knuth, Morris, Pratt (KMP) 法[20] は、総当たり方式のサブ文字列検索における再トレースの非効率性を解決する洗練された実装といえます。KMP 法は、オーバーラップ・テーブルを使用して部分一致が偽の一致である場合の再トレース距離を管理しているため、大量のドキュメントからキーワードが含まれる関連事項を検索するアプリケーションに極めて有効です。

例 14-8 に、KMP サブ文字列検索を使用した C コードの例を示します。

20 Donald E. Knuth 氏、James H. Morris 氏、Vaughan R. Pratt 氏。SIAM J. Comput. 第 6 巻、第 2 号、323 ~ 350 ページ (1977 年)

例 14-8 C コードの KMP サブ文字列検索

```
// s1 は、レングス cnt1 のターゲット文字列
// s2 は、レングス cnt2 の参照文字列
// j は、文字列比較を開始するターゲット文字列 s1 のオフセット
// i は、バイト単位の比較を行う参照文字列 s2 のオフセット
int str_kmp_c(const char* s1, int cnt1, const char* s2, int cnt2 )
{  int i, j;
   i = 0; j = 0;
   while ( i+j < cnt1) {
       if( s2[i] == s1[i+j]) {
          i++;
          if( i == cnt2) break; // 完全に一致
       }
       else {
          j = j+i - ovrlap_tbl[i]; // 次の文字列比較のため、s1 オフセットを更新
          if( i > 0) {
              i = ovrlap_tbl[i]; // 次の文字列比較の開始位置のため s2 オフセットを更新
          }
       }
   };
   return j;
}

void kmp_precalc(const char * s2, int cnt2)
{  int i = 2;
   char nch = 0;
   ovrlap_tbl[0] = -1; ovrlap_tbl[1] = 0;
      // KMP テーブルの事前計算
   while( i < cnt2) {
      if( s2[i-1] == s2[nch]) {
         ovrlap_tbl[i] = nch +1;
         i++; nch++;
      }
      else if ( nch > 0) nch = ovrlap_tbl[nch];
      else {
         ovrlap_tbl[i] = 0;
         i++;
      }
   };
   ovrlap_tbl[cnt2] = 0;
}
```

例 14-8 では、KMP オーバーラップ・テーブルの計算も行われています。KMP 法では通常、同じ参照文字列が複数回呼び出されるため、オーバーラップ・テーブルを事前計算する際のオーバーヘッドを容易に吸収できます。参照文字列のオフセット i が偽の一致であると判定された場合、オーバーラップ・テーブルは、オフセット j を更新して次回の文字列比較を開始すべき位置と、バイト単位の文字比較をレジューム/再開すべき参照文字列中のオフセットを予測します。

KMP 法では総当たり方式のバイト単位サブ文字列検索に比べて効率が改善されていますが、最大パフォーマンスは依然としてバイト単位の演算数により制限されています。PCMPISTRI の汎用性と組込み型字句解析機能を示すため、総当たり方式の 16 バイト単位アプローチを使用したサブ文字列検索のインテル® SSE4.2 コード (例 14-9) と、PCMPISTRI を使用したサブ文字列検索に KMP オーバーラップ・テーブルを組み合わせたもの (例 14-10) を示します。

例 14-9 PCMPISTRI の組込み関数を使用した総当たり方式のサブ文字列検索

```
int strsubs_sse4_2i(const char* s1, int cnt1, const char* s2, int cnt2 )
{ int kpm_i=0, idx;
int ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;
int cmp, cmp2, cmp_s;
__m128i *pt = NULL;
   if( cnt2 > cnt1 || !cnt1) return -1;
   frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
   frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード
   while(rcnt1 > 0)
   { cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)?
   ln1: rcnt1, 0x0c);
     cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1:
     rcnt1, 0x0c);
     if( !cmp) { // 解析の継続を必要とする部分的一致がある
     if( cmp_s) { // s2 を終えているなら
     if( pt)
     {idx = (int) ((char *) pt - (char *) s1) ; }
     else
     {idx = (int) ((char *) p1 - (char *) s1) ; }
     return idx;
     }
     // s2 の最後まで完全な一致を検証するため、文字列の比較を行う
     if( pt == NULL) pt = p1;
     cmp2 = 16;
     rcnt2 = cnt2 - 16 -(int) ((char *)p2-(char *)s2);
     while( cmp2 == 16 && rcnt2) { // 各 16 バイト断片の一致
       rcnt1 = cnt1 - 16 -(int) ((char *)p1-(char *)s1);
       rcnt2 = cnt2 - 16 -(int) ((char *)p2-(char *)s2);
       if( rcnt1 <=0 || rcnt2 <= 0 ) break;
       p1 = (__m128i *)(((char *)p1) + 16);
       p2 = (__m128i *)(((char *)p2) + 16);
       frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
       frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード
       cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)?
       ln1: rcnt1, 0x18); // lsb, eq each
     };
     (続く)

     if( !rcnt2 || rcnt2 == cmp2) {
         idx = (int) ((char *) pt - (char *) s1) ;
         return idx;
     }
     else if ( rcnt1 <= 0) { // cmp2 < 16, 一致しない
         if( cmp2 == 16 && ((rcnt1 + 16) >= (rcnt2+16) ) )
         {idx = (int) ((char *) pt - (char *) s1) ;
         return idx;
         }
         else return -1;
     }
     else { // ターゲット文字列 s1 のオフセット断片を 1 つ進める
```

```
      p1 = (__m128i *)(((char *)pt) + 1); // kmp の利点を活用しない
      rcnt1 = cnt1 -(int) ((char *)p1-(char *)s1);
      pt = NULL;
      p2 = (__m128i *)((char *)s2) ;
      rcnt2 = cnt2 -(int) ((char *)p2-(char *)s2);
      frag1 = _mm_loadu_si128(p1);// s1 から次の断片をロード
      frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード
      }
   }
   else{
      if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
      else p1 = (__m128i *)(((char *)p1) + cmp);
      rcnt1 = cnt1 -(int) ((char *)p1-(char *)s1);
      if( pt && cmp ) pt = NULL;
      frag1 = _mm_loadu_si128(p1);// s1 から次の断片をロード
     }
   }
   return idx;
}
```

例 14-9 では、定数を使用してループ伝搬依存を最小限に抑えるアドレス調整が以下の 2 個所で行われています。

- 完全一致であるか偽の一致であるかを判定する文字列比較の内側 while ループ内 (cmp2 の結果は、依存関係を回避するためのアドレス調整には使用されません)。
- 外側ループが PCMPISTRI を実行して、ターゲット・フラグメントと参照文字列の最初の 16 バイト・フラグメントとの 16 セットの順序付き比較を行い、16 セットの順序付き比較演算すべてで偽の結果が生じた (値が 16 の cmp を生成した場合)、最後のコードブロック内。

例 14-10 に、インテル® SSE4.2 と KMP オーバーラップ・テーブルを使用してサブ文字列を検索する同等の組込み関数コードを示します。文字列比較が内側ループで偽の一致と判定された場合、KMP オーバーラップ・テーブルを参照して、ターゲット文字列フラグメントと参照文字列フラグメントのアドレスオフセットを判断することにより、再トレースを最小限に抑えます。

例 14-9 に示した総当たり方式のインテル® SSE4.2 コードでも、再トレース距離が 15 バイト未満の再トレースの大部分は回避される点に注意してください。これは、PCMPISTRI の順序付き比較プリミティブによるものです。「順序付き比較」では 16 セットの文字列フラグメント比較が行われ、PCMPISTRI を実行する同じ反復内では、部分一致が 15 バイト未満の偽の一致の多くをフィルターで排除できます。

例 14-9 では、再トレース距離が 15 バイトを超えると、フィルターでは排除できません。例 14-10 では、KMP オーバーラップ・テーブルを参照することにより、15 バイトを超える再トレースを回避できます。

例 14-10 PCMPISTRI と KMP オーバーラップ·テーブルを使用したサブ文字列検索

```
int strkmp_sse4_2(const char* s1, int cnt1, const char* s2, int cnt2 )
{ int kpm_i=0, idx;
int ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;
int cmp, cmp2, cmp_s;
__m128i *pt = NULL;
   if( cnt2 > cnt1 || !cnt1) return -1;
   frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
   frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード

   while(rcnt1 > 0)
   { cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)?
   ln1: rcnt1, 0x0c);
     cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1:
     rcnt1, 0x0c);
     if( !cmp) { // 解析の継続を必要とする部分的一致がある
       if( cmp_s) { // もし s2 の最後に到達したら
           if( pt)
           {idx = (int) ((char *) pt - (char *) s1) ; }
           else
           {idx = (int) ((char *) p1 - (char *) s1) ; }
           return idx;
       }
       // s2 の最後まで完全な一致を検証するため、文字列の比較を行う
       if( pt == NULL) pt = p1;
       cmp2 = 16;
       rcnt2 = cnt2 - 16 -(int) ((char *)p2-(char *)s2);
       while( cmp2 == 16 && rcnt2) { // 各 16 バイト断片が一致
           rcnt1 = cnt1 - 16 -(int) ((char *)p1-(char *)s1);
           rcnt2 = cnt2 - 16 -(int) ((char *)p2-(char *)s2);
           if( rcnt1 <=0 || rcnt2 <= 0 ) break;
           p1 = (__m128i *)(((char *)p1) + 16);
           p2 = (__m128i *)(((char *)p2) + 16);
           frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
           frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード
           cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1,
           (rcnt1>ln1)? ln1: rcnt1, 0x18); // lsb, eq each
       };
       if( !rcnt2 || rcnt2 == cmp2) {
           idx = (int) ((char *) pt - (char *) s1) ;
           return idx;
       }
       else if ( rcnt1 <= 0 ) { // cmp2 < 16, 一致しない
          return -1;
       }
       else { // 部分的な一致は誤った一致を招き、アドレス調整のため KMP オーバーラップ·テーブルを
       利用
          kpm_i = (int) ((char *)p1 - (char *)pt)+ cmp2 ;
          p1 = (__m128i *)(((char *)pt) + (kpm_i - ovrlap_tbl[kpm_i])); // スキップ
          を取り消すため kmp を利用
          rcnt1 = cnt1 -(int) ((char *)p1-(char *)s1);
```

```
          pt = NULL;
          p2 = (__m128i *)(((char *)s2) + (ovrlap_tbl[kpm_i]));
          rcnt2 = cnt2 -(int) ((char *)p2-(char *)s2);
          frag1 = _mm_loadu_si128(p1);// s1 から次の断片をロード
          frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード
        }
      }
      else{
        if( kpm_i && ovrlap_tbl[kpm_i]) {
        p2 = (__m128i *)(((char *)s2) );
        frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード
        //p1 = (__m128i *)(((char *)p1) );

        //rcnt1 = cnt1 -(int) ((char *)p1-(char *)s1);
        if( pt && cmp ) pt = NULL;
        rcnt2 = cnt2 ;
        //frag1 = _mm_loadu_si128(p1);// s1 から次の断片をロード
        frag2 = _mm_loadu_si128(p2);// 最大 16 バイトの断片をロード
        kpm_i = 0;
        }
        else { // サブ断片の同じ並びの比較結果もしくは不一致
        if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
        else p1 = (__m128i *)(((char *)p1) + cmp);
        rcnt1 = cnt1 -(int) ((char *)p1-(char *)s1);
        if( pt && cmp ) pt = NULL;
        frag1 = _mm_loadu_si128(p1);// s1 から次の断片をロード
        }
      }
    }
    return idx;
}
```

総当たり方式のバイト単位サブ文字列検索と、それと比較したバイト単位 KMP、総当たり方式のインテル® SSE4.2、および KMP オーバーラップ·テーブルを使用したインテル® SSE4.2 について、相対的なスピードアップをグラフに示します。このグラフでは、55 バイト長の参照文字列の再トレース率に対する相対的なスピードアップを示しています。グラフ中の再トレース率が 40% である場合、最初の 22 文字が部分一致した後で、偽の一致であると判定されたことを意味します。

総当たり方式のバイト単位の実装が再トレースを必要とする場合でも、それ以外の 3 つの実装では、以下の理由により再トレースを回避できることがあります。

- 例 14-8 では、KMP オーバーラップ·テーブルを使用して、部分一致/偽の一致後の次回の文字列比較演算の開始オフセットを予測できます。ただし、先頭文字の偽の一致後に行われる順方向の進行は依然としてバイト単位です。
- 例 14-9 では、15 バイト未満の再トレースは回避できますが、参照文字列の 22 バイト目で部分一致/偽の一致が生じると 21 バイトの再トレースが行われます。順序付き比較での偽の一致の後ごとに行われる順方向の進行は 16 バイト単位です。
- 例 14-10 では、部分一致/偽の一致後の 21 バイトの再トレースは回避できますが、KMP オーバーラップ·テーブルのルックアップで多少のオーバーヘッドが生じます。順序付き比較での偽の一致の後ごとに行われる順方向の進行は 16 バイト単位です。

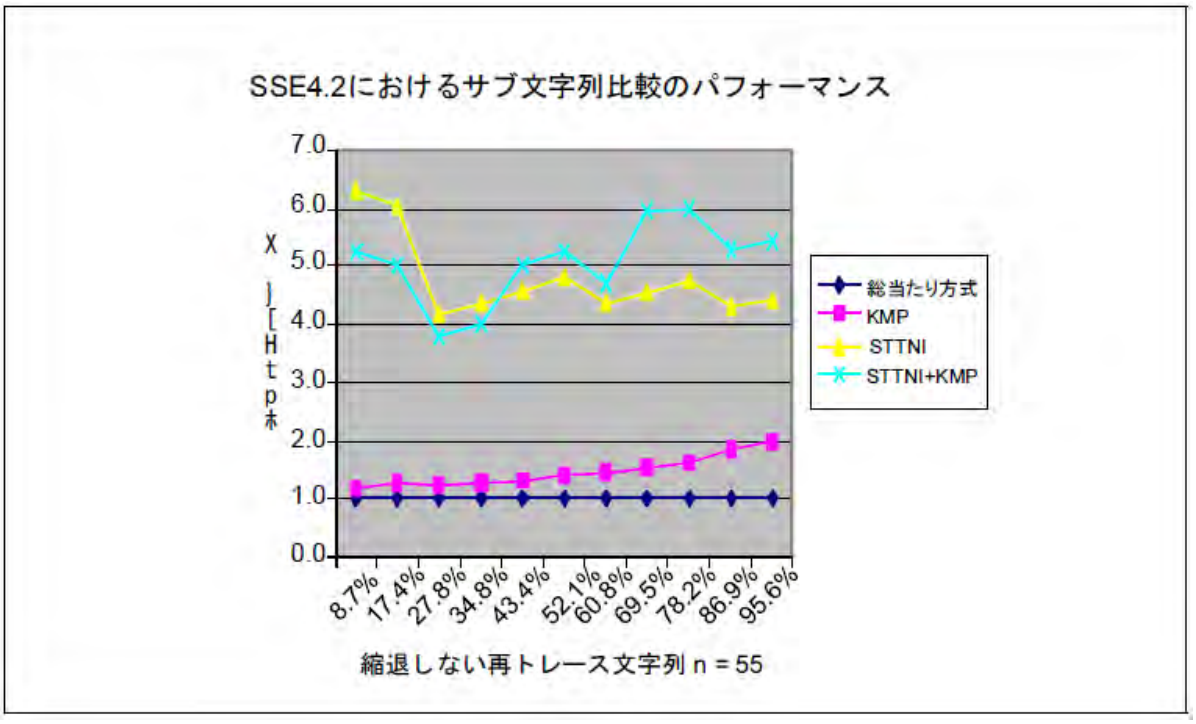


図 14-3 インテル® SSE4.2 におけるサブ文字列検索の高速化

14.3.4 文字列トークンの抽出と大文字/小文字の処理

トークンの抽出は、テキスト/文字列処理で一般的なタスクです。これは、より高度なレクサー/パーサー・オブジェクトを実装する上での基盤の 1 つとなります。インデックス・サービスも、トークン化プリミティブに基づいてストリームからのテキストデータをソートします。

トークン化には、区切り文字の配列を使用する柔軟性が求められます。

ライブラリー関数の strtok_s() では、テーブル・ルックアップ手法を利用して、区切り文字の連続的な比較を単一の比較に統合できます (例 14-6 と同様)。例 14-11 に、組込み関数を使用して strtok_s() と同等の機能を実装したインテル® SSE4.2 コードを示します。

例 14-11 PCMPISTRI の組込み関数を使用した strtok_s( ) と同等のコード

```
char ws_map8[32]; // 区切り文字のためのパックされたビットのルックアップ・テーブル
char * strtok_sse4_2i(char* s1, char *sdlm, char ** pCtxt)
{
__m128i *p1 = (__m128i *) s1;
__m128ifrag1, stmpz, stmp1;
int cmp_z, jj =0;
int start, endtok, s_idx, ldx;
   if (sdlm == NULL || pCtxt == NULL) return NULL;
   if( p1 == NULL && *pCtxt == NULL) return NULL;
   if( s1 == NULL) {
      if( *pCtxt[0] == 0 ) { return NULL; }
      p1 = (__m128i *) *pCtxt;
      s1 = *pCtxt;
   }
   else p1 = (__m128i *) s1;
   memset(&ws_map8[0], 0, 32);
   (続く)
   while (sdlm[jj] ) {
```

```
    ws_map8[ (sdlm[jj] >> 3) ] |= (1 << (sdlm[jj] & 7) ); jj ++
}
frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
stmpz = _mm_loadu_si128((__m128i *)sdelimiter);
// 最初の文字が区切り文字でなければ、非区切り文字のチェックを続行
// そうでなければ先導する区切り文字をスキップする必要がある
if( ws_map8[s1[0]>>3] & (1 << (s1[0]&7)) ) {
start = s_idx = _mm_cmpistri(stmpz, frag1, 0x10);// 符号なしバイト/いずれも等しい、
反転、LSB
}
else start = s_idx = 0;
```

```
// 16 バイト以下の短い文字列を扱うかチェック
cmp_z = _mm_cmpistrz(stmpz, frag1, 0x10);
if( cmp_z) { // 最後の断片
    if( !start) {
        endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);
        if( endtok == 16) { // NULL まで区切り文字が見つからない
            // どこに NULL があるか検出
            *pCtxt = s1+ 1+ _mm_cmpistri(frag1, frag1, 0x40);
            return s1;
        }
        else { // この単語を終了させる区切り文字を検出
        s1[start+endtok] = 0;
        *pCtxt = s1+start+endtok+1;
        }
    }
    else {
        if(!s1[start] ) {
            *pCtxt = s1 + start +1;
            return NULL;
        }
        p1 = (__m128i *)(((char *)p1) + start);
        frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
        endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);// 符号なしバイト/ いずれ
        も等しい、反転、LSB
        if( endtok == 16) { // 区切り文字を探したが、見つからなかった
            *pCtxt = (char *)p1 + 1+ _mm_cmpistri(frag1, frag1, 0x40);
            return s1+start;
        }
        else { // NULL バイトの前に区切り文字を検出
            s1[start+endtok] = 0;
            *pCtxt = s1+start+endtok+1;
        }
    }
}
```

```
else
{ while ( !cmp_z && s_idx == 16) {
    p1 = (__m128i *)(((char *)p1) + 16);
    frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
    s_idx = _mm_cmpistri(stmpz, frag1, 0x10);// 符号なしバイト/ いずれも等しい、反
    転、LSB
    cmp_z = _mm_cmpistrz(stmpz, frag1, 0x10);
    (続く)
  }
```

```
    if(s_idx != 16) start = ((char *) p1 -s1) + s_idx;
    else { // 区切り文字を探して最後まで達したが、区切り文字は見つからなかった
        *pCtxt = (char *)p1 +1+ _mm_cmpistri(frag1, frag1, 0x40);
        return NULL;
    }
    if( !s1[start] ) { // 区切り文字の後に NULL 文字が続く場合
        *pCtxt = s1 + start+1;
        return NULL;
    }
    // 現在いくつの非区切り文字があるか検出
    p1 = (__m128i *)(((char *)p1) + s_idx);
    frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
    endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);// 符号なしバイト/ いずれも等し
    い、反転、LSB
    cmp_z = 0;
    while ( !cmp_z && ldx == 16) {
        p1 = (__m128i *)(((char *)p1) + 16);
        frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
        ldx = _mm_cmpistri(stmpz, frag1, 0x00);// 符号なしバイト/いずれも等しい、反転、
        LSB
        cmp_z = _mm_cmpistrz(stmpz, frag1, 0x00);
        if(cmp_z) { endtok += ldx; }
    }
    if( cmp_z ) { // s1 の最後に到達したら
        if( ldx < 16) // 区切り文字で検出された単語の終わり
        endtok += ldx;
        else { // NULL 文字で検出された単語の終わり
            if( s1[start+endtok]) // NULL 文字で始まらないことを確実にする
            endtok += 1+ _mm_cmpistri(frag1, frag1, 0x40);
        }
      }
     *pCtxt = s1+start+endtok+1;
     s1[start+endtok] = 0;
   }
   return (char *) (s1+ start);
}
```

例 14-12 に、組込み関数を使用して strupr() と同等の機能を実装したインテル® SSE4.2 コードを示します。

例 14-12 PCMPISTRM の組込み関数を使用した strupr( ) と同等のコード

```
static char uldelta[16]= {0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20};
static char ranglc[6]= {0x61, 0x7a, 0x00, 0x00, 0x00, 0x00};
char * strup_sse4_2i( char* s1)
{int len = 0, res = 0;
__m128i *p1 = (__m128i *) s1;
__m128ifrag1, ranglo, rmsk, stmpz, stmp1;
int cmp_c, cmp_z, cmp_s;
   if( !s1[0]) return (char *) s1;
   frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
   ranglo = _mm_loadu_si128((__m128i *)ranglc);// 最大 16 バイトの断片をロード
   stmpz = _mm_loadu_si128((__m128i *)uldelta);
   (続く)
```

```
   cmp_z = _mm_cmpistrz(ranglo, frag1, 0x44);// 比較範囲、バイト･マスク生成
```

```
    while (!cmp_z)
    {
        rmsk = _mm_cmpistrm(ranglo, frag1, 0x44); // バイト・マスク生成
        stmp1 = _mm_blendv_epi8(stmpz, frag1, rmsk); // lc のバイトは保存され、その他のバイトは const で置き換えられる
        stmp1 =_mm_sub_epi8(stmp1, stmpz); // lc バイトは uc になり、その他のバイトはゼロになる
        stmp1 = _mm_blendv_epi8(frag1, stmp1, rmsk); // lc バイトは uc に置き換わり、その他のバイトは変わらない
        _mm_storeu_si128(p1, stmp1);//
        p1 = (__m128i *)(((char *)p1) + 16);
        frag1 = _mm_loadu_si128(p1);// 最大 16 バイトの断片をロード
        cmp_z = _mm_cmpistrz(ranglo, frag1, 0x44);
    }
    if( *(char *)p1 == 0) return (char *) s1;
    rmsk = _mm_cmpistrm(ranglo, frag1, 0x44);// バイト・マスク、有効な lc バイトは 1、その他すべては 0
    stmp1 = _mm_blendv_epi8(stmpz, frag1, rmsk); // lc のバイトは継続され、その他のバイトは const で置き換えられる
    stmp1 =_mm_sub_epi8(stmp1, stmpz); // lc バイトは uc になり、その他のバイトはゼロになる
    stmp1 = _mm_blendv_epi8(frag1, stmp1, rmsk); // lc バイトは uc に置き換わり、その他のバイトは変わらない
    rmsk = _mm_cmpistrm(frag1, frag1, 0x44);// バイト・マスク、有効なバイトは 1、その他すべては 0
    _mm_maskmoveu_si128(stmp1, rmsk, (char *) p1);//
    return (char *) s1;
}
```

14.3.5 Unicode* 処理と PCMPxSTRy

ソフトウェアをローカライズする際には、文字列/テキストデータの Unicode* 表現が必要になります。ローカライズされたコンテンツの一般的なエンコード方式は、UTF-16 です。UTF-16 表現の場合、各文字はコードポイントによって表されます。コードポイントには、16 ビット・コードポイントと 32 ビット・コードポイントの 2 種類があります。32 ビット・コードポイントは、指定された値範囲内の 16 ビット・コードポイントのペアからなり、代替ペアとも呼ばれます。

Unicode* 処理における一般的な手法では、テーブル・ルックアップ手法が使用され、これには分岐が排除されるメリットがあります。事例として、適切にエンコードされた UTF-16 文字列のレングスを判断する同様の処理に対し、テーブル・ルックアップ使用の汎用コードを用いた場合とインテル® SSE4.2 を用いた場合について比較します。

例 14-13 に示す C コードシーケンスでは、Unicode* テキストブロック中の適切にエンコードされた UTF-16 コードポイント (16 ビットまたは 32 ビット) の数を判断しています。このコードでは、不適切にエンコードされた代替ペアがテキストブロック中に存在するかどうかも確認しています。

例 14-13 C コードとテーブル・ルックアップ手法を使用した UTF16 の verstrlen()

```
// この例では、代替ペア(32 ビット・コードポイント)とテキストブロックの
// 16 ビットと 32 ビットのコードポイント数を記録する
// 引数: s1 は UTF16 テキストブロックへのポインター
// pLen: UTF16 コードポイントのストア数
// 代替範囲でエンコードされた 16 ビットのコードポイント数を返すが、
// 適切にエンコードされた代替ペアを生成してはならない。
// 戻り値が 0 なら、s1 は適切にエンコードされた UTF16 ブロックであり、
// 0 より大きければ、s1 は無効な代替エンコードを含む。

int u16vstrlen_c(const short* s1, unsigned * pLen)
{int i, j, cnt = 0, cnt_invl = 0, spcnt= 0;
unsigned short cc, cc2;
char flg[3];
cc2 = cc = s1[0];
// s1 内の各ワードをテーブル・ルックアップを使用して 0、1、2 のビットパターンに割り当てる
// 2 として割り当てられる D8000-DBFF 間のエンコードの代替ペアの前半
// 1 として割り当てられる DC000-DFFF 間のエンコードの代替ペアの後半
// 0 に割り当てられる通常の 16 ビット・エンコード、3 に割り当てられた NULL コードを除く
flg[1] = utf16map[cc];
flg[0] = flg[1];
if(!flg[1]) cnt ++;
i = 1;
while (cc2 ) // それぞれの非 NULL ワードのエンコードを調査
{  cc2 = s1[i];
   flg[2] = utf16map[cc2];
   if( (flg[1] && flg[2] && (flg[1]-flg[2] == 1) ) )
   { spcnt ++; }// 代替ペアを検出
   else if(flg[1] == 2 && flg[2] != 1)
   { cnt_invl += 1; } // 前半単独
   else if( !flg[1] && flg[2] == 1)
   { cnt_invl += 1; } // 後半単独
   else
   { if(!flg[2]) cnt ++;// 通常の非 NULL コード 16 ビット・コードポイント
      else ;
   }
   flg[0] = flg[1];// 次の反復のためペアシーケンスをストア
   flg[1] = flg[2];
   i++;
}
*pLen = cnt + spcnt;
return cnt_invl;
}
```

UTF-16 でエンコードされたテキストブロック用の verstrlen() 関数は、インテル® SSE4.2 を使用して実装できます。

例 14-14 にインテル® SSE4.2 のアセンブリー・コードを、例 14-15 にインテル® SSE4.2 を使用した verstrlen() の組込み関数のコードを示します。

例 14-14 PCMPISTRI を使用した UTF16 の verstrlen() のアセンブリー・コード

```
// 32 ビットの UTF-16 コードポイントのいずれか半分を検出する範囲値の補足
static short ssch0[16]= {0x1, 0xd7ff, 0xe000, 0xffff, 0, 0};
// 32 ビットの UTF-16 コードポイントの前半を検出する範囲値の補足
static short ssch1[16]= {0x1, 0xd7ff, 0xdc00, 0xffff, 0, 0};
// 32 ビットの UTF-16 コードポイントの後半を検出する範囲値の補足
static short ssch2[16]= {0x1, 0xdbff, 0xe000, 0xffff, 0, 0};

int utf16slen_sse4_2a(const short* s1, unsigned * pLen)
{int len = 0, res = 0;
   _asm{
   mov eax, s1
   movdquxmm2, ssch0 ; 前半か後半か識別するため範囲値をロード
   movdquxmm3, ssch1 ; 前半を識別するため範囲値をロード (0xd800から0xdbff)
   movdquxmm4, ssch2 ; 後半を識別するため範囲値をロード (0xdc00から0xdfff)
   xor ecx, ecx
   xor edx, edx; 32ビット・コードポイント数をストア (隣接するペア)
   xor ebx, ebx; NULL 以外の 16 ビット・コードポイント数をストア
   xor edi, edi ; 無効なワードエンコード数をストア
   (続く)
_loopc:
   shl ecx, 1; ワード処理を行う PCMPISTRI 命令は、ワード単位の ECX を返す。2倍してバイト・オフセットを取得。
   add eax, ecx
   movdquxmm1, [eax] ; 最大 8 ワードの文字列の断片をロード
   pcmpistri xmm2, xmm1, 15h; 符号なしワード、範囲、反転、LSB インデックスを ECX に返す
      ; XMM1 に UTF-16 NULL wchar があれば、ZF がセットされる
      ; 8 つのワード比較がすべて一致したなら、
      ; 中間結果ではビットがセットされず、反転後にセットされ、
      ; ECX には 8 が返る
   jz short _lstfrag; NULL ならば最後の断片を扱う
   ; ECX < 8 ならば、ECX は 32 ビット・コードポイントの前半か後半のいずれかのワードを指す
   cmp ecx, 8
   jne _chksp
   add ebx, ecx ; 16 ビット非 NULL コードポイント数を計算
   mov ecx, 8 ; ループ伝搬の依存性を避けるため、ここで ECX は 8 でなければならない。
   jmp _loopc
```

```
_chksp:; この断片は、隣接する値でワードエンコードされる
   add ebx, ecx ; 16 ビットのコードポイント
   shl ecx, 1; ワード処理を行う PCMPISTRI 命令は、ワード単位の ECX を返す。2 倍してバイト・オフセットを取得。
   add eax, ecx
   movdqu xmm1, [eax] ; ワードエンコードされた断片は、前半か後半のいずれかで始まることを確実にする
   pcmpistri xmm3, xmm1, 15h; 符号なしワード、範囲、反転、LSB インデックスを ECX に返す
   jz short _lstfrag2; NULL コードなら、最後の断片を扱う
   cmp ecx, 0 ; 適切にエンコードされた32 ビット・コードポイントは、前半から始まる
   jg _invalidsp; この断片にはいくつかの無効な S-P コードがある
   pcmpistri xmm4, xmm1, 15h; 符号なしワード、範囲、反転、LSB インデックスを ECX に返す
   cmp ecx, 1 ; 後半は前半に続く必要がある
   jne _invalidsp
   add edx, 1; 有効な隣接ペア数を計算
   add ecx, 1 ; 2 ワード分進める
   jmp _loopc
```

```
_invalidsp:; この断片の最初のワードは、ワードの後半か異なるペアの前半
   add edi, 1 ; 無効なコードペアを検出 (隣接しないペア)
   mov ecx, 1 ; 1 つワードを進め、32 ビット・コードポイントのスキャンを続行
   jmp _loopc
_lstfrag:
   add ebx, ecx ; NULL でない 16 ビットのコードポイント
_morept:
   shl ecx, 1; ワード処理を行う PCMPISTRI 命令は、ワード単位の ECX を返す。2 倍してバイト・オフ
   セットを取得。
   add eax, ecx
   mov si, [eax] ; NULL コードがあるかどうかをチェックする必要がある
   cmp si, 0
   je _final
   movdqu xmm1, [eax] ; 前半/後半いずれかの残りのワードをロード
   pcmpistri xmm3, xmm1, 15h; 符号なしワード、範囲、反転、LSB インデックスを ECX に返す
_lstfrag2:
   cmp ecx, 0 ; 有効な 32 ビットのコードポイントは前半から始まらなければいけない
   jne _invalidsp2
   pcmpistri xmm4, xmm1, 15h; 符号なしワード、範囲、反転、LSB インデックスを ECX に返す
   cmp ecx, 1
   jne _invalidsp2
   add edx, 1
   mov ecx, 2
   jmp _morept
_invalidsp2:
   add edi, 1
   mov ecx, 1
   jmp _morept
_final:
   add edx, ebx; 16 ビットと 32 ビット・コードポイント数を加算
   mov ecx, pLen; 呼び出し元によって提供されたアドレスポインターを復旧
   mov [ecx], edx; 文字列長をメモリーへストア
   mov res, edi
   }
   return res;
}
```

例 14-15 PCMPISTRI を使用した UTF16 の verstrlen() の組込み関数コード

```
int utf16slen_i(const short* s1, unsigned * pLen)
{int len = 0, res = 0;
__m128i *pF = (__m128i *) s1;
__m128iu32 =_mm_loadu_si128((__m128i *)ssch0);
__m128i u32a =_mm_loadu_si128((__m128i *)ssch1);
__m128i u32b =_mm_loadu_si128((__m128i *)ssch2);
__m128ifrag1;
int offset1 = 0, cmp, cmp_1, cmp_2;
intcnt_16 = 0, cnt_sp=0, cnt_invl= 0;
short *ps;
while (1) {
   pF = (__m128i *)(((short *)pF) + offset1);
   frag1 = _mm_loadu_si128(pF);// 最大 8 バイトの断片をロード
   // frag1 は、32 ビット UTF-16 コードポイントのどちらか半分を含むか?
   cmp = _mm_cmpistri(u32, frag1, 0x15);// 符号なしバイト、等しい並び、LSB インデックスを
   ECX に返す
```

```
   if (_mm_cmpistrz(u32, frag1, 0x15))// frag1 に NULL がある
   { cnt_16 += cmp;
       ps = (((short *)pF) + cmp);
       while (ps[0])
       {  frag1 = _mm_loadu_si128( (__m128i *)ps);
          cmp_1 = _mm_cmpistri(u32a, frag1, 0x15);
          if(!cmp_1)
          { cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
             if( cmp_2 ==1) { cnt_sp++; offset1 = 2;}
             (続く)
             else {cnt_invl++; offset1= 1;}
          }
          else
          { cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
            if(!cmp_2) {cnt_invl ++; offset1 = 1;}
            else {cnt_16 ++; offset1 = 1; }
          }
          ps = (((short *)ps) + offset1);
       }
       break;
   }
```

```
   if(cmp != 8) // 32 ビット UTF-16 コードポイントの半分はある
   {  cnt_16 += cmp; // 通常の 16 ビット UTF-16 コードポイント
      pF = (__m128i *)(((short *)pF) + cmp);
      frag1 = _mm_loadu_si128(pF);
      cmp_1 = _mm_cmpistri(u32a, frag1, 0x15);
      if(!cmp_1)
      { cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
          if( cmp_2 ==1) { cnt_sp++; offset1 = 2;}
          else {cnt_invl++; offset1= 1;}
      }
      else
      { cnt_invl ++;
         offset1 = 1;
      }
   }
   else {
       offset1 = 8; // 次の断片を処理するためアドレスを 16 バイト進める
       cnt_16+= 8;
   }
  };
 *pLen = cnt_16 + cnt_sp;
 return cnt_invl;
}
```

14.3.6 インテル® SSE4.2 を使用した文字列ライブラリー関数の置き換え

アライメントされていない 128 ビット SIMD メモリーアクセスでは、ページ境界をまたいでデータをフェッチできます。システム・ソフトウェアはページ単位でメモリーアクセス権を管理するため、SIMD 命令を使用して文字列ライブラリー関数を実装する際は、メモリーアクセス違反を発生させてはなりません。この要件は、各文字列フラグメントのメモリーアドレスをチェックする少量のコードを追加することで解決できます。メモリーアドレスが次のページ境界から 16 バイト以内であると判明した場合、文字列処理アルゴリズムはバイト単位の手法に戻ることができます。

例 14-16 に、標準ツールで用意されているバイト単位コードを置き換えるインテル® SSE4.2 実装の strcmp() コードを示します。

例 14-16 インテル® SSE4.2 を使用した文字列ライブラリーの strcmp 置き換え

```
// 文字列が等しければ 0 を、大きければ 1、小さければ -1 を返す
int strcmp_sse4_2(const char *src1, const char *src2)
{
   int val;
   __asm{
      mov esi, src1 ;
      mov edi, src2
      mov edx, -16 ; 文字列ポインターベースに対する相対インデックス
      xor eax, eax
topofloop:
      add edx, 16 ; ループ伝搬依存を排除
next:
      lea ecx, [esi+edx] ; ロードする断片のアドレス
      and ecx, 0x0fff ; ページ境界のアドレスの最下位 12 ビットをチェック
      cmp ecx, 0x0ff0
      jg too_close_pgb ; 16 バイト境界以内ならバイト単位へ分岐
      lea ecx, [edi+edx] ; 2 番目の文字列の断片に同じチェックを行う
      and ecx, 0x0fff
      cmp ecx, 0x0ff0
      jg too_close_pgb
      movdqu xmm2, BYTE PTR[esi+edx]
      movdqu xmm1, BYTE PTR[edi+edx]
      pcmpistri xmm2, xmm1, 0x18 ; それぞれ等しい
      ja topofloop
      jnc ret_tag
      add edx, ecx ; ECX は異なるバイト·オフセットを示す
not_equal:
      movzx eax, BYTE PTR[esi+edx]
      movzx edx, BYTE PTR[edi+edx]
      cmp eax, edx
      cmova eax, ONE
      cmovb eax, NEG_ONE
      jmp ret_tag
too_close_pgb:
      add edx, 1 ; バイト単位で比較
      movzx ecx, BYTE PTR[esi+edx-1]
      movzx ebx, BYTE PTR[edi+edx-1]
      cmp ecx, ebx
      jne inequality
      add ebx, ecx
      jnz next
      jmp ret_tag
inequality:
      cmovb eax, NEG_ONE
      cmova eax, ONE
ret_tag:
      mov [val], eax
   }
   return(val);
}
```

例 14-16 では、「next」ラベルの後に 8 個の命令が追加され、2 つの文字列フラグメントをレジスターにロードするのに使用されるアドレスに対して 4KB 境界のチェックを行っています。いずれかのアドレスが次のページ境界から 16 バイト以内であることが判明した場合、コードは「too_close_pgb」ラベルの後のバイト単位比較パスに分岐します。

例 14-16 の戻り値は、CMOV を用いて 0、+1、-1 を返す形式が採用されています。命令を少し変更して、0、正の整数、負の整数を返す形式を実装するのは容易です。

14.4 インテル® SSE4.2 で可能となる数値および字句計算

インテル® SSE4.2 では、SIMD プログラミング手法を有効にして、SIMD 命令を使用する際の候補としてふさわしくないとみなされたバイト単位の計算問題を調査できます。一般的なライブラリー関数 atol() を完全な 64 ビット方式で使用し、データ型 __int64 で表現可能な範囲内に変換する方法を検討します。

このような文字列から整数への変換には次のようないくつかの特性があり、インテル® SSE4.2 以前の SIMD 命令セットを使用して、文字列から整数への変換における数値計算を高速化するのは困難です。

- 文字サブセット検証: 入力ストリーム内の各文字は、文字サブセット定義の観点で検証され、スペース、符号、数字のデータ表現規則に適合する必要があります。インテル® SSE4.2 には、文字サブセット検証する完全な機能が用意されています。
- 文字検証における状態依存の性質: SIMD 計算命令では「10 で乗算して加算」という算術演算を高速化できますが、算術演算の対象となる入力バイトストリームは数値のみで構成されている必要があります。例えば、数値、スペース、および符号の有無の検証は中間ストリームで行う必要があります。インテル® SSE4.2 では、プリミティブの柔軟性により、これらの状態依存の検証を適切に処理できます。
- さらに、無効な文字が原因で、算術演算を終了させる終了条件が中間ストリームで発生することもあります。また、データ型の表現可能範囲が有限 (int64 では ~10^19、int32 は先頭が 0 でない 10 桁以下の数字) であるため、ショートバーストで構成されるこのタイプのデータストリームは SIMD 命令セットによる調査には適しておらず、バイト単位のソリューションで十分な可能性があります。

文字サブセット検証および状態依存の性質により、標準ライブラリー関数のバイト単位のソリューションは初期コストが高くなり (例えば、1 つの数字を整数に変換するのに 50 または 60 サイクルかかることがある)、スループットが低くなる傾向にあります (入力文字ストリーム内の追加の各数字を処理するのにバイトあたり 6 ~ 8 サイクルかかることがある)。

例 14-17 に、標準ライブラリーに含まれる atol() を置き換える疑似フローを示します。

例 14-17 文字列変換用の文字サブセット検証のフローの概要

1. Early_Out 終了条件 (先頭バイトが有効でない場合など) をチェックする。
2. 先頭バイトがスペースであるかどうかをチェックし、先頭にスペースが追加されている場合はそれをスキップする。
3. 符号バイトの存在をチェックする。
4. 残りのバイトストリームの有効性をチェックする (それらが数字である場合)。
5. バイトストリームが「0」で始まる場合、先頭が 0 であるすべての数字をスキップする。
6. 先頭が 0 でない有効な数字の数を確認する。
7. 先頭が 0 でない最大 16 桁の数字を int64 値に変換する。
8. 最大で次の 16 バイトまでを安全にロードし、結果の数字をチェックする。
9. 先頭の 16 桁から変換された int64 値を残りの桁数に従って正規化する。
10. 正規化した中間 int64 値の許容範囲外の結果をチェックする。
11. 残りの桁を int64 値に変換し、正規化した中間結果に追加する。
12. 許容範囲外の最終結果をチェックする。

例 14-18 に、int64 出力範囲を生成し、atol() と同等の機能を持つコードリストを示します。例 14-19 に、補助関数とデータ定数を示します。

例 14-18 PCMPISTRI を使用した atol() 組込み関数コード

```
__int64 sse4i_atol(const char* s1)
{ char *p = ( char *) s1;
  int NegSgn = 0;
  __m128i mask0;
  __m128i value0, value1;
  __m128i w1, w1_l8, w1_u8, w2, w3 = _mm_setzero_si128();
  __int64 xxi;
  int index, cflag, sflag, zflag, oob=0;
  // ルックアップで最初の文字が正当かチェック
  if ( (BtMLValDecInt[ *p >> 3] & (1 << ((*p) & 7)) ) == 0) return 0;
  // 最初がスペースなら残りのスペースをスキップ
  if (BtMLws[*p >>3] & (1 <<((*p) & 7)) )
  { p ++;
    value0 = _mm_loadu_si128 ((__m128i *) listws);
  skip_more_ws:
    mask0 = __m128i_strloadu_page_boundary (p);
    /* 最初のスペース以外の文字を探す */
    index = _mm_cmpistri (value0, mask0, 0x10);
    cflag = _mm_cmpistrc (value0, mask0, 0x10);
    sflag = _mm_cmpistrs (value0, mask0, 0x10);
    if( !sflag && !cflag)
    { p = ( char *) ((size_t) p + 16);
      goto skip_more_ws;
    }
    else p = ( char *) ((size_t) p + index);
  }
  if( *p == '-')
  { p++;
    NegSgn = 1;
  }
  else if( *p == '+') p++;

  /* 最大 16 バイトを安全にロードし、何桁の有効な数値を SIMD 演算できるかチェック */
  value0 = _mm_loadu_si128 ((__m128i *) rangenumint);
  mask0 = __m128i_strloadu_page_boundary (p);
  index = _mm_cmpistri (value0, mask0, 0x14);
  zflag = _mm_cmpistrz (value0, mask0, 0x14);
```

```
  /* 有効な数値ではない最初の桁へのポインター */
  if( !index) return 0;
  else if (index == 16)
  {  if( *p == '0') /* もし 16 バイトすべてが数値なら */
     { /* 先行するゼロをスキップ */
       value1 = _mm_loadu_si128 ((__m128i *) rangenumintzr);
       index = _mm_cmpistri (value1, mask0, 0x14);
       zflag = _mm_cmpistrz (value1, mask0, 0x14);
       while(index == 16 && !zflag )
       {  p = ( char *) ((size_t) p + 16);
          mask0 = __m128i_strloadu_page_boundary (p);
          index = _mm_cmpistri (value1, mask0, 0x14);
          zflag = _mm_cmpistrz (value1, mask0, 0x14);
       }
       (続く)
```

```
      /* 最初の数値はゼロ以外、最大 16 バイトをロードし、インデックスを更新 */
      if( index < 16)
      p = ( char *) ((size_t) p + index);
      /* 最大 16 バイトの非ゼロ数値をロード */
      mask0 = __m128i_strloadu_page_boundary (p);
      /* 非数値文字へのポインターを更新、もしくは 16 バイト以上あることを通知 */
      index = _mm_cmpistri (value0, mask0, 0x14);
    }
  }
  if( index == 0) return 0;
  else if( index == 1) return (NegSgn? (long long) -(p[0]-48): (long long) (p[0]-
  48));
  // xmm の入力桁の並びは反転している。出力 LS 桁は、EOS と並んでいる
  // 再下位数値桁はバイト 15 にアライメントされ、それぞれ ACSII コードから 0x30 を減算する
  mask0 = ShfLAlnLSByte( mask0, 16 -index);
  w1_u8 = _mm_slli_si128 ( mask0, 1);
  w1 = _mm_add_epi8( mask0, _mm_slli_epi16 (w1_u8, 3)); /* 8 を乗算し加算 */
  w1 = _mm_add_epi8( w1, _mm_slli_epi16 (w1_u8, 1)); /* バイト 0, 2, 4, 6, 8,
  10, 12, 14 中のバイトごとの 7 つの LS ビット */
  w1 = _mm_srli_epi16( w1, 8); /* 各ワードの上位ビットをクリア */
  w2 = _mm_madd_epi16(w1, _mm_loadu_si128( (__m128i *) &MulplyPairBaseP2[0]) ); /*
  base^2 を乗算し、調整ワードを加算 */
  w1_u8 = _mm_packus_epi32 ( w2, w2); /* 各 DWORD の 4 つの下位ワードを 63:0 にパック */
  w1 = _mm_madd_epi16(w1_u8, _mm_loadu_si128( (__m128i *) &MulplyPairBaseP4[0]) );
  /* base^4 を乗算し、調整ワードを加算 */
  w1 = _mm_cvtepu32_epi64( w1); /* 63:0 の変換された DWORD を QWORD に拡張 */
  w1_l8 = _mm_mul_epu32(w1, _mm_setr_epi32( 100000000, 0, 0, 0) );
  w2 = _mm_add_epi64(w1_l8, _mm_srli_si128 (w1, 8) );
  if( index < 16)
  { xxi = _mm_extract_epi64(w2, 0);
    return (NegSgn? (long long) -xxi: (long long) xxi);
  }
  /* 64 ビット整数は非ゼロ数値を最大 20 桁許容 */
  /* 各 16 桁の断片を計算 */
  w3 = _mm_add_epi64(w3, w2);
  /* 最大 16 桁の次のバッチを処理、64 ビット整数は 4 桁のみ許容 */
  p = ( char *) ((size_t) p + 16);
  if( *p == 0)
  { xxi = _mm_extract_epi64(w2, 0);
    return (NegSgn? (long long) -xxi: (long long) xxi);
  }
  mask0 = __m128i_strloadu_page_boundary (p);
  /* 最初の非数値へのポインター */
  index = _mm_cmpistri (value0, mask0, 0x14);
  zflag = _mm_cmpistrz (value0, mask0, 0x14);
  if( index == 0) /* 最初の文字は無効な数値 */
  { xxi = _mm_extract_epi64(w2, 0);
    return (NegSgn? (long long) -xxi: (long long) xxi);
  }
  if ( index > 3) return (NegSgn? (long long) RINT64VALNEG: (long long)
  RINT64VALPOS);
  /* base^index で qword の下位を乗算 */
  w1 = _mm_mul_epu32( _mm_shuffle_epi32( w2, 0x50), _mm_setr_epi32( MulplyByBaseExpN
  [index - 1] , 0, MulplyByBaseExpN[index-1], 0));
```

```
  w3 = _mm_add_epi64(w1, _mm_slli_epi64 (_mm_srli_si128(w1, 8), 32 ) );
  mask0 = ShfLAlnLSByte( mask0, 16 -index);
  // xmm の上位 8 バイトを変換: 再下位のみ。出力の 4 桁が前の 16 桁に追加される
  w1_u8 = _mm_cvtepi8_epi16(_mm_srli_si128 ( mask0, 8));
  /* 一度に 2 桁の乗数を dword にマージ */
  w1_u8 = _mm_madd_epi16(w1_u8, _mm_loadu_si128( (__m128i *) &MulplyQuadBaseExp3To0
  [ 0]));
  /* ビット 63:0 は 2 つの dword 整数を持つ。ビット 63:32 は出力の LS dword。ビット 127:64 は必要な
  い */
  w1_u8 = _mm_cvtepu32_epi64( _mm_hadd_epi32(w1_u8, w1_u8) );
  w3 = _mm_add_epi64(w3, _mm_srli_si128( w1_u8, 8) );
  xxi = _mm_extract_epi64(w3, 0);
  if( xxi >> 63 )
    return (NegSgn? (long long) RINT64VALNEG: (long long) RINT64VALPOS);
  else return (NegSgn? (long long) -xxi: (long long) xxi);
}
```

インテル® SSE4.2 の拡張版 atol() 置換の一般的なパフォーマンス特性により、C コードから生成されるバイト単位の実装よりも初期コストが多少低くなります。

例 14-19 sse4i_atol() コードで使用される補助ルーチンとデータ定数

```
// 10 進文字列、スペース、記号、数値変換のための有効な ASCII コードのルックアップ・テーブル。
static char BtMLValDecInt[32] = {0x0, 0x3e, 0x0, 0x0, 0x1, 0x28, 0xff, 0x03,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};

// ビット・ルックアップ・テーブル、スペースのみ
static char BtMLws[32] = {0x0, 0x3e, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0};

// STTNI で利用するスペースのリスト
static char listws[16] =
{0x20, 0x9, 0xa, 0xb, 0xc, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
// STTNI で利用する数値のリスト
static char rangenumint[16] =
{0x30, 0x39, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
static char rangenumintzr[16] =
{0x30, 0x30, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};

// 隣接した short 整数のペアに PMADDWD を利用、これは 2 つの整数をマージする 2 番目のステップ
static short MulplyPairBaseP2[8] =
{ 100, 1, 100, 1, 100, 1, 100, 1};

// 2 つの隣接する short 整数ペアのための乗数のペア。これは 4 つの整数ペアをマージする 3 番目のステップ
static short MulplyPairBaseP4[8] =
{ 10000, 1, 10000, 1, 10000, 1, 10000, 1 };

// 16 桁の整数を正規化する PMULLD のための乗数
static int MulplyByBaseExpN[8] =
{ 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000};

static short MulplyQuadBaseExp3To0[8] =
```

```
{ 1000, 100, 10, 1, 1000, 100, 10, 1};
__m128i __m128i_shift_right (__m128i value, int offset)
{  switch (offset)
   {
   case 1: value = _mm_srli_si128 (value, 1); break;
   case 2: value = _mm_srli_si128 (value, 2); break;
   case 3: value = _mm_srli_si128 (value, 3); break;
   case 4: value = _mm_srli_si128 (value, 4); break;
   case 5: value = _mm_srli_si128 (value, 5); break;
   case 6: value = _mm_srli_si128 (value, 6); break;
   case 7: value = _mm_srli_si128 (value, 7); break;
   case 8: value = _mm_srli_si128 (value, 8); break;
   case 9: value = _mm_srli_si128 (value, 9); break;
   case 10: value = _mm_srli_si128 (value, 10); break;
   case 11: value = _mm_srli_si128 (value, 11); break;
   case 12: value = _mm_srli_si128 (value, 12); break;
   case 13: value = _mm_srli_si128 (value, 13); break;
   case 14: value = _mm_srli_si128 (value, 14); break;
   case 15: value = _mm_srli_si128 (value, 15); break;
   }
   return value;
}
/* 安全にページ境界近くの S で文字列をロード*/
__m128i __m128i_strloadu_page_boundary (const char *s)
{
   int offset = ((size_t) s & (16 - 1));
   if (offset)
   {
      __m128i v = _mm_load_si128 ((__m128i *) (s - offset));
      __m128i zero = _mm_setzero_si128 ();
      int bmsk = _mm_movemask_epi8 (_mm_cmpeq_epi8 (v, zero));
      if ( (bmsk >> offset) != 0 ) return __m128i_shift_right (v, offset);
   }
   return _mm_loadu_si128 ((__m128i *) s);
}
__m128i ShfLAlnLSByte( __m128i value, int offset)
{
   /* 定数バイアスを排除し、各バイト要素は符号なし整数 */
   value = _mm_sub_epi8(value, _mm_setr_epi32(0x30303030, 0x30303030, 0x30303030,
   0x30303030));
   switch (offset)
   {
   case 1:
   value = _mm_slli_si128 (value, 1); break;
   case 2:
   value = _mm_slli_si128 (value, 2); break;
   case 3:
   value = _mm_slli_si128 (value, 3); break;
   case 4:
   value = _mm_slli_si128 (value, 4); break;
   case 5:
   value = _mm_slli_si128 (value, 5); break;
   case 6:
   value = _mm_slli_si128 (value, 6); break;
```

```
   case 7:
   value = _mm_slli_si128 (value, 7); break;
   case 8:
   value = _mm_slli_si128 (value, 8); break;
   case 9:
   value = _mm_slli_si128 (value, 9); break;
   case 10:
   value = _mm_slli_si128 (value, 10); break;
   case 11:
   value = _mm_slli_si128 (value, 11); break;
   case 12:
   value = _mm_slli_si128 (value, 12); break;
   case 13:
   value = _mm_slli_si128 (value, 13); break;
   case 14:
   value = _mm_slli_si128 (value, 14); break;
   case 15:
   value = _mm_slli_si128 (value, 15); break;
   }
   return value;
}
```

先頭が 0 でない 16 桁以下の数字で構成される入力バイトストリームでは、一定のパフォーマンスが得られます。先頭が 0 でない 16 バイトを超える数字で構成される入力文字列は 100 サイクル以下で処理できます (バイト単位のソリューションでは約 200 サイクル必要になります)。先頭が 0 でない 9 桁の数字で構成される短い入力文字列の場合でも、インテル® SSE4.2 の拡張版置換ではバイト単位のソリューションと比較して 2 倍のパフォーマンスを実現できます。

14.5 ASCII 形式への数値データ変換

バイナリー整数データから ASCII 形式への変換は、C ライブラリー関数から金融計算まで多様な状況で使用されます。C ライブラリーで提供される関数 (itoa や ltoa など) は変換機能をエクスポートし、他のライブラリーは、標準出力関数のデータ形式をサポートするため同等の内部機能を実装しています。最も一般的なバイナリー整数から ASCII への変換は、基数 10 の変換です。例 14-20 は、64 ビット整数の ASCII への基数 10 の変換において、多くのライブラリーで実装される基本的な手法です。簡略化のため、この例では小文字の出力を生成しています。

例 14-20 64 ビット整数から ASCII への変換

```
// 64 ビット符号付き整数から ASCII 小文字形式への変換

static char lc_digits[]= "0123456789abcdefghijklmnopqrstuvwxyz";

int lltoa_cref( __int64 x, char* out)
{const char *digits = &lc_digits[0];
char lbuf[32] // 64 ビットの符号付き整数の基数 10 の変換は 21 桁を必要とする
char * p_bkwd = &lbuf[2];
__int64 y ;
unsigned int base = 10, len = 0, r, cnt;
   if( x < 0)
   {  y = -x;
      while (y > 0)
      { r = (int) (y % base); // 最下位桁から 1 桁ずつ
         y = y /base;
         * --p_bkwd = digits[r];
         len ++;
```

```
      }
      *out++ = '-';
      cnt = len +1;
      while( len--) *out++ = p_bkwd++; // 変換された数値をコピー
   } else
   {
      y = x;
      while (y > 0)
      { r = (int) (y % base); // 最下位桁から 1 桁ずつ
         y = y /base;
         * --p_bkwd = digits[r];
         len ++;
      }
      cnt = len;;
      while( len--) *out++ = p_bkwd++; // 変換された数値をコピー
   }
   out[cnt] = 0;
   return (int) cnt;
}
```

例 14-20 では、ハードウェアのネイティブ整数除算命令により、1 桁ずつ処理する繰り返しシーケンスを導入しています。整数除算は、第 13 章で示す固定小数点乗算に置き換えることができます。この様子を例 14-21 に示します。

例 14-21 整数除算なしの 64 ビット整数から ASCII への変換

```
// 64 ビット符号付き整数から ASCII 小文字形式への変換し、
// 整数除算を固定小数点の乗算に置き換える
;__int64 umul_64x64(__int64* p128, __int64 u, __int64 v)
   umul_64x64 PROC
   mov rax, rdx ; 2 番目の引数
   mul r8 ; u * v
   mov qword ptr [rcx], rax
   mov qword ptr [rcx+8], rdx
   ret 0
umul_64x64 ENDP
#define cg_10_pms3 0xcccccccccccccccdull
static char lc_digits[]= "0123456789";

int lltoa_cref( __int64 x, char* out)
{const char *digits = &lc_digits[0];
char lbuf[32] // 64 ビットの符号付き整数の基数 10 の変換は 21 桁を必要とする
char * p_bkwd = &lbuf[2];
__int64 y, z128[2];
unsigned __int64 q;
unsigned int base = 10, len = 0, r, cnt;
```

```
   if( x < 0)
   { y = -x;
       while (y > 0)
       {  umul_64x64( &z128[0], y, cg_10_pms3);
          q = z128[1] >> 3;
          q = (y < q * (unsigned __int64) base)? q-1: q;
          r = (int) (y - q * (unsigned __int64) base); // 最下位桁から 1 桁ずつ
          y =q;
          * --p_bkwd = digits[r];
```

```
            len ++;
        }
        *out++ = '-';
        cnt = len +1;
        while( len--) *out++ = p_bkwd++; // 変換された数値をコピー
    } else
    {
        y = x;
        while (y > 0)
        { umul_64x64( &z128[0], y, cg_10_pms3);
            q = z128[1] >> 3;
            q = (y < q * (unsigned __int64) base)? q-1: q;
            r = (int) (y - q * (unsigned __int64) base); // 最下位桁から 1 桁ずつ
            y =q;
            * --p_bkwd = digits[r];
            len ++;
        }
        cnt = len;;
        while( len--) *out++ = p_bkwd++; // 変換された数値をコピー
    }
    out[cnt] = 0;
    return cnt;
}
```

例 14-21 のコードは、整数除算への依存性を排除することで、劇的なパフォーマンス向上をもたらします。しかし、1 桁ずつ処理する依存性チェーンによる値の書式変換の問題はまだ残っています。

符号なし 64 ビット整数は最大 20 桁に拡張されることに注意して、このような整数値の変換処理には SIMD 命令を使用する手法を適用できます。このような動的範囲は次の多項式で表現できます。

$a_0 + a_1 *10^4 + a_2 *10^8 + a_3 *10^12 + a_4 *10^16$
ここで、a_i の動的範囲は [0, 9999] となります。

符号なし 64 ビット整数の最大 5 つの縮小範囲への削減は、固定小数点乗算を使用して計算できます。動的範囲が 4 桁以下になると、SIMD 手法を適用して ASCII 変換を並列に計算できます。

図 14-4 に、入力動的範囲 [0, 9999] の基数 10 で符号なし 16 ビット整数に変換する SIMD 手法を示します。この手法は、16 未満の基数 2 の他の非累乗に適用するため一般化できます。

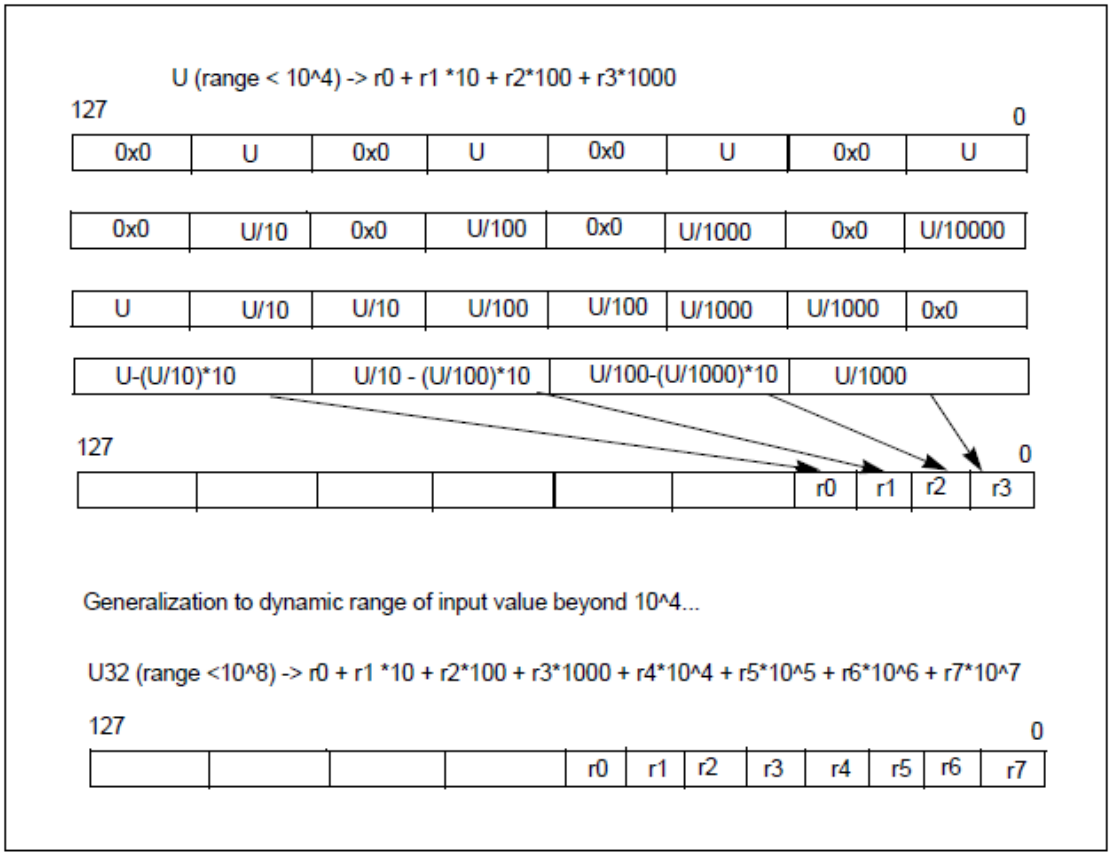


図 14-4 符号なし short 整数の残り 4 つを並列に計算

より大きな動的範囲を入力するため、入力が複数の符号なし short 整数に短縮され、連続して変換されます。最上位桁の U16 変換が最初に行われ、次に 4 つの有効桁が変換されます。

例 14-22 は、最大 19 桁までの 64 ビット整数変換にインテル® SSE4 命令を使用した、並列リマインダー計算と固定小数点乗算の組み合わせを示します。

例 14-22 インテル® SSE4 を使用した 64 ビット整数から ASCII への変換

```
#include <smmintrin.h>
#include <stdio.h>
#define QWCG10to8 0xabcc77118461cefdull
#define QWCONST10to8 100000000ull

/* short 整数 "hi4" の入力引数を出力変数 "X3" (__m128i) に変換するマクロ;
入力値 "hi4" は 10^4 未満と想定;
出力は 0-9 間の 4 つの単一桁の整数で、各 dword の下位バイトに配置され、最低位 DW の最上位桁。
暗黙のオーバーライト: ローカルに割り当てられた __m128i 変数 "x0", "x2"
*/

#define __ParMod10to4SSSE3( x3, hi4 ) ¥
{ ¥
   x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128( (hi4)), 0); ¥
   x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d));¥
   x2 = _mm_srli_epi32( _mm_madd_epi16( x2, _mm_loadu_si128( (__m128i *)
   quo4digComp_mulplr_d)), 10); ¥
   (x3) = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) (hi4), 1); ¥
   (x3) = _mm_or_si128(x2, (x3));¥
   (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;¥
```

```
}

/* 3 番目の dword 要素 "t5" の入力引数 (__m128i 型) を出力変数 "x3" (__m128i) に変換するマクロ
3 番目の dword 要素 "t5" は 10^4 未満と想定され、4 番目の dword は 0 である必要がある;
出力は 0-9 の間の 4 つの単一桁の整数で、各 dword の下位バイトに配置され、LS DW の MS 桁。
暗黙のオーバーライト: ローカルに割り当てられた __m128i 変数 "x0", "x2"
*/

#define __ParMod10to4SSSE3v( x3, t5 ) ¥
{ ¥
   x0 = _mm_shuffle_epi32( t5, 0xaa); ¥
   x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d));¥
   x2 = _mm_srli_epi32( _mm_madd_epi16( x2, _mm_loadu_si128( (__m128i *)
   quo4digComp_mulplr_d)), 10); ¥
   (x3) = _mm_or_si128(_mm_slli_si128(x2, 6), _mm_srli_si128(t5, 6)); ¥
   (x3) = _mm_or_si128(x2, (x3));¥
   (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;¥
}
```

```
static __attribute__ ((aligned(16)) ) short quo4digComp_mulplr_d[8] =
{ 1024, 0, 64, 0, 8, 0, 0, 0};
static __attribute__ ((aligned(16)) ) short quoTenThsn_mulplr_d[8] =
{ 0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static __attribute__ ((aligned(16)) ) short mten_mulplr_d[8] =
{ -10, 1, -10, 1, -10, 1, -10, 1};
static __attribute__ ((aligned(16)) ) unsigned short bcstpklodw[8] =
{0x080c, 0x0004, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) ) unsigned short bcstpkdw1[8] =
{0x8080, 0x8080, 0x080c, 0x0004, 0x8080, 0x8080, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) ) unsigned short bcstpkdw2[8] =
{0x8080, 0x8080, 0x8080, 0x8080, 0x080c, 0x0004, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) ) unsigned short bcstpkdw3[8] =
{0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x080c, 0x0004};
static __attribute__ ((aligned(16)) ) int asc0bias[4] =
{0x30, 0x30, 0x30, 0x30};
static __attribute__ ((aligned(16)) ) int asc0reversebias[4] =
{0xd0d0d0d0, 0xd0d0d0d0, 0xd0d0d0d0, 0xd0d0d0d0};
static __attribute__ ((aligned(16)) ) int pr_cg_10to4[4] =
{ 0x68db8db, 0 , 0x68db8db, 0};
static __attribute__ ((aligned(16)) ) int pr_1_m10to4[4] =
{ -10000, 0 , 1, 0};

/* 入力値 "xx" は 2^63-1 未満 */
/* __int128_t のバイナリー整数変換をサポートしない環境では、asm ルーチンとしてこのアシストが可能
*/
__inline __int64_t u64mod10to8( __int64_t * pLo, __int64_t xx)
{  __int128_t t, b = (__int128_t)QWCG10to8;
   __int64_t q;
   t = b * (__int128_t)xx;
   q = t>>(64 +26); // QWCG10to8 に関連するシフトカウント
   *pLo = xx - QWCONST10to8 * q;
   return q;
}

/* 2^63-1 と 0 間の整数値を ASCII 文字列に変換 */
```

```
int sse4i_q2a_u63 ( __int64_t xx, char *ps)
{int j, tmp, idx=0, cnt;
__int64_t lo8, hi8, abv16, temp;
__m128i x0, m0, x1, x2, x3, x4, x5, x6, m1;
long long w, u;
   if ( xx < 10000 )
   {  j = ubs_Lt10k_2s_i2 ( (unsigned ) xx, ps);
      ps[j] = 0; return j;
   }
   if (xx < 100000000 ) // xx のダイナミック･レンジは 32 ビット未満
   {  m0 = _mm_cvtsi32_si128( xx);
      x1 = _mm_shuffle_epi32(m0, 0x44); // dw0 と dw2 へブロードキャスト
      x3 = _mm_mul_epu32(x1, _mm_loadu_si128( (__m128i *) pr_cg_10to4 ));
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
      *)pr_1_m10to4));
      m0 = _mm_add_epi32( _mm_srli_si128( x1, 8), x3); // 商は dw2、余剰は dw0
      __ParMod10to4SSSE3v( x3, m0); // 各 dword から dw0 へ値をパック
      x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
      __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8)); // 最初に余剰を dw2 へ移動
      x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
      x4 = _mm_or_si128(x4, x5); //  0-7 バイトを先頭 0 でパック
      cnt = 8;
   }
   else
   { hi8 = u64mod10to8(&lo8, xx);
      if ( hi8 < 10000) // lo8 の dword を商と余剰  mod 10^4 に分解
      {  m0 = _mm_cvtsi32_si128( lo8);
         x2 = _mm_shuffle_epi32(m0, 0x44);
         x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
         x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
         *)pr_1_m10to4));
         m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3); // 商は dw0
         __ParMod10to4SSSE3( x3, hi8); // 最初に 11:8 桁を処理
         x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
         __ParMod10to4SSSE3v( x3, m0); // 7:4 桁を処理
         x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
         x4 = _mm_or_si128(x4, x5);
         __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));
         x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw2) );
         x4 = _mm_or_si128(x4, x5); // 0-11 バイトを先頭 0 でパック
         cnt = 12;
   }
   else
   {  cnt = 0;
      if ( hi8 >= 100000000) // 10^16 以上の入力を処理
      { abv16 = u64mod10to8(&temp, (__int64_t)hi8);
          hi8 = temp;
          __ParMod10to4SSSE3( x3, abv16);
          x6 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
          cnt = 4;
      } // 15:12 桁の処理を開始
      m0 = _mm_cvtsi32_si128( hi8);
      x2 = _mm_shuffle_epi32(m0, 0x44);
      x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
```

```
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
      *)pr_1_m10to4));
      m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
      m1 = _mm_cvtsi32_si128( lo8);
      x2 = _mm_shuffle_epi32(m1, 0x44);
      x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
      *)pr_1_m10to4));
      m1 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
      __ParMod10to4SSSE3v( x3, m0);
      x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
      __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));
      x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
      x4 = _mm_or_si128(x4, x5);
      __ParMod10to4SSSE3v( x3, m1);
      x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw2) );
      x4 = _mm_or_si128(x4, x5);
      __ParMod10to4SSSE3v( x3, _mm_slli_si128(m1, 8));
      x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw3) );
      x4 = _mm_or_si128(x4, x5);
      cnt += 16;
      }
   }
m0 = _mm_loadu_si128( (__m128i *) asc0reversebias);
if( cnt > 16)
{ tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(x6,_mm_setzero_si128()) );
   x6 = _mm_sub_epi8(x6, m0);
} else {
   tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(x4,_mm_setzero_si128()) );
}
```

```
#ifndef __USE_GCC__
  __asm__ ("bsfl %1, %%ecx; movl %%ecx, %0;" :"=r"(idx) :"r"(tmp) : "%ecx");
#else
  _BitScanForward(&idx, tmp);
#endif
   x4 = _mm_sub_epi8(x4, m0);
   cnt -= idx;
   w = _mm_cvtsi128_si64(x4);
   switch(cnt)
   { case5:*ps++ = (char) (w >>24); *(unsigned *) ps = (w >>32);
   break;
   case6:*(short *)ps = (short) (w >>16); *(unsigned *) (&ps[2]) = (w >>32);
   break;
   case7:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
   *(unsigned *) (&ps[3]) = (w >>32);
   break;
   case 8: *(long long *)ps = w;
   break;
   case9:*ps++ = (char) (w >>24);
   *(long long *) (&ps[0]) = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
   break;
   case10:*(short *)ps = (short) (w >>16);
   *(long long *) (&ps[2]) = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
   break;
```

```
   case11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
   *(long long *) (&ps[3]) = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
   break;
   case 12: *(unsigned *)ps = w;
   *(long long *) (&ps[4]) = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
   break;
   case13:*ps++ = (char) (w >>24); *(unsigned *) ps = (w >>32);
   *(long long *) (&ps[4]) = _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
   break;
   case14:*(short *)ps = (short) (w >>16); *(unsigned *) (&ps[2]) = (w >>32);
   *(long long *) (&ps[6]) = _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
   break;
   case15: *ps = (char) (w >>8);
   *(short *) (&ps[1]) = (short) (w >>16); *(unsigned *) (&ps[3]) = (w >>32);
   *(long long *) (&ps[7]) = _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
   break;
   case 16: _mm_storeu_si128( (__m128i *) ps, x4);
   break;
   case17:u = _mm_cvtsi128_si64(x6); *ps++ = (char) (u >>24);
   _mm_storeu_si128( (__m128i *) &ps[0], x4);
   break;
   case18:u = _mm_cvtsi128_si64(x6); *(short *)ps = (short) (u >>16);
   _mm_storeu_si128( (__m128i *) &ps[2], x4);
   break;
   case19:u = _mm_cvtsi128_si64(x6); *ps = (char) (u >>8);
   *(short *) (&ps[1]) = (short) (u >>16);
   _mm_storeu_si128( (__m128i *) &ps[3], x4);
   break;
   case20:u = _mm_cvtsi128_si64(x6); *(unsigned *)ps = (short) (u);
   _mm_storeu_si128( (__m128i *) &ps[4], x4);
   break;
   }
   return cnt;
}
/* それぞれの dword 要素を並列固定小数点演算によって、入力値を 4 桁の数値に変換し、
各数値を低位の dword 要素にパックして、先頭のスペースなしでバッファーに書き込み:
入力値は 10000 未満で 10 以上であること
*/
__inline int ubs_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
__m128i x0, m0, x2, x3, x4, compv;
// 一連のスケーリング定数を使用して、1 要素あたりのシフトカウントの不足を補います
compv = _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d);
// 入力値をそれぞれの dword 要素にブロードキャスト
x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128( x_Lt10k), 0);
// x0 の下位から上位への dword: u16, u16, u16, u16
m0 = _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d); // 4 つの一致する定数をロード
x2 = _mm_mulhi_epu16(x0, m0); // 基数 10,100, 1000, 10000 向けの並列固定小数点乗算
x2 = _mm_srli_epi32( _mm_madd_epi16( x2, compv), 10);
// x2 の dword の内容: u16/10, u16/100, u16/1000, u16/10000
x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
// x3 の word の内容: 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
x4 = _mm_or_si128(x2, x3);
// 4 つのバイアスされていない 1 桁の数値結果を求めるため、それぞれ word ペアの残りの操作を実行
```

```
x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;
x2 = _mm_add_epi32( x4, _mm_loadu_si128( (__m128i *) asc0bias) ) ;
// dword から下位 dword 要素までそれぞれ ASCII バイアスされた数値をパック
x3 = _mm_shuffle_epi8(x2, _mm_loadu_si128( (__m128i *) bcstpklodw) );
// 先頭のスペースなしで ASCII 結果をバッファーへストア
if (x_Lt10k > 999 )
{ *(int *) ps = _mm_cvtsi128_si32( x3);
return 4;
}
else if (x_Lt10k > 99 )
{ tmp = _mm_cvtsi128_si32( x3);
*ps = (char ) (tmp >>8);
*((short *) (++ps)) = (short ) (tmp >>16);
return 3;
}
else if ( x_Lt10k > 9) // 分岐を減らすため >9 に低減されたダイナミック·レンジを利用
{ *((short *) ps) = (short ) _mm_extract_epi16( x3, 1);
return 2;
}
*ps = '0' + x_Lt10k;
return 1;
}

char lower_digits[] = "0123456789";

int ltoa_sse4 (const long long s1, char * buf)
{long long temp ;
int j = 1, len = 0;
const char *digits = &lower_digits[0];
   if( s1 < 0) {
       temp = -s1;
       len ++;
       beg[0] = '-';
       if( temp < 10) beg[1] = digits[ (int) temp];
       else len += sse4i_q2a_u63( temp, &buf[ 1]); // 4 桁単位の操作での並列変換
   }
   else {
   if( s1 < 10) beg[ 0 ] = digits[(int)s1];
   else len += sse4i_q2a_u63( s1, &buf[ 1] );
   }
   buf[len] = 0;
   return len;
}
```

ltoa() にようなユーティリティー関数実装がネイティブ IDIV 命令を実行して一度に 1 桁を変換する場合、桁あたりおよそ 45-50 サイクルのスピードで出力を生成できます。(例 13-21 のように) IDIV を固定小数点に置き換えると、桁あたり 10-15 サイクルに軽減できます。128 ビット SIMD 手法を使用して並列固定小数点計算を行うと、Sandy Bridge† マイクロアーキテクチャーや Nehalem† マイクロアーキテクチャーでは、出力スピードが桁あたり 4-5 サイクルに改善されます。

例 14-22 に示す範囲削減手法では、最大 19 レベルの依存性チェーンを 5 階層まで減少させ、SIMD 手法による 4 桁幅の数値変換を可能にします。この手法はインテル® SSSE3 命令でもスピードの改善はほぼ同等に実装できます。

例 14-23 で示すコードを使用することで、ワイド文字列の変換サポートも容易に行えます。

例 14-23 インテル® SSE4 を使用した 64 ビット整数からワイド文字列への変換

```
static __attribute__ ((aligned(16)) ) int asc0bias[4] =
{0x30, 0x30, 0x30, 0x30};

// exponent_x は < 10000 および > 9 でなければならない
__inline int ubs_Lt10k_2wcs_i2(int x_Lt10k, wchar_t *ps)
{
__m128i x0, m0, x2, x3, x4, compv;
   compv = _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d);
   x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128( x_Lt10k), 0); // 低位から上位の dw:
   u16, u16, u16, u16
   m0 = _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d);
   // u16, 0, u16, 0, u16, 0, u16, 0
   x2 = _mm_mulhi_epu16(x0, m0);
   x2 = _mm_srli_epi32( _mm_madd_epi16( x2, compv), 10); // u16/10, u16/100,
   u16/1000, u16/10000
   x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1); // 0, u16,
   0, u16/10, 0, u16/100, 0, u16/1000
   x4 = _mm_or_si128(x2, x3);
   x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;
   x2 = _mm_add_epi32( x4, _mm_loadu_si128( (__m128i *) asc0bias) ) ;
   x2 = _mm_shuffle_epi32(x2, 0x1b); // switch シーケンス
   if (x_Lt10k > 999 ) {
      _mm_storeu_si128( (__m128i *) ps, x2);
      return 4;
   }
   else if (x_Lt10k > 99 ) {
      *ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x2, 4));
      *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x2, 8));
      return 3;
   }
   else if ( x_Lt10k > 9){ // 分岐を減らすため >9 に低減されたダイナミック･レンジを利用
      *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x2, 8));
      return 2;
   }
   *ps = L'0' + x_Lt10k;
   return 1;
}
```

```
long long sse4i_q2wcs_u63 ( __int64_t xx, wchar_t *ps)
{int j, tmp, idx=0, cnt;
__int64_t lo8, hi8, abv16, temp;
__m128i x0, m0, x1, x2, x3, x4, x5, x6, x7, m1;

if ( xx < 10000 ) {
   j = ubs_Lt10k_2wcs_i2 ( (unsigned ) xx, ps); ps[j] = 0; return j;
}
if (xx < 100000000 ) { // xx のダイナミック･レンジは 32 ビット未満
   m0 = _mm_cvtsi32_si128( xx);
   x1 = _mm_shuffle_epi32(m0, 0x44); // dw0 と dw2 へブロードキャスト
   x3 = _mm_mul_epu32(x1, _mm_loadu_si128( (__m128i *) pr_cg_10to4 ));
   x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
   *)pr_1_m10to4));
```

```
    m0 = _mm_add_epi32( _mm_srli_si128( x1, 8), x3); // 商は dw2、余剰は dw0
    __ParMod10to4SSSE3v( x3, m0);
    //x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
    x3 = _mm_shuffle_epi32(x3, 0x1b);
    __ParMod10to4SSSE3v( x4, _mm_slli_si128(m0, 8)); // 最初に余剰を dw2 へ移動
    x4 = _mm_shuffle_epi32(x4, 0x1b);
    cnt = 8;
} else {
    hi8 = u64mod10to8(&lo8, xx);
    if( hi8 < 10000) {
    m0 = _mm_cvtsi32_si128( lo8);
    x2 = _mm_shuffle_epi32(m0, 0x44);
    x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
    x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
    *)pr_1_m10to4));
    m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
    __ParMod10to4SSSE3( x3, hi8);
    x3 = _mm_shuffle_epi32(x3, 0x1b);
    __ParMod10to4SSSE3v( x4, m0);
    x4 = _mm_shuffle_epi32(x4, 0x1b);
    __ParMod10to4SSSE3v( x5, _mm_slli_si128(m0, 8));
    x5 = _mm_shuffle_epi32(x5, 0x1b);
    cnt = 12;
    } else {
    cnt = 0;
    if ( hi8 > 100000000) {
    abv16 = u64mod10to8(&temp, (__int64_t)hi8);
    hi8 = temp;
    __ParMod10to4SSSE3( x7, abv16);
    x7 = _mm_shuffle_epi32(x7, 0x1b);
    cnt = 4;
    }
    m0 = _mm_cvtsi32_si128( hi8);
    x2 = _mm_shuffle_epi32(m0, 0x44);
    x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
    x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
    *)pr_1_m10to4));
    m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
    m1 = _mm_cvtsi32_si128( lo8);
    x2 = _mm_shuffle_epi32(m1, 0x44);
    x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
    x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i
    *)pr_1_m10to4));
    m1 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
    __ParMod10to4SSSE3v( x3, m0);
    x3 = _mm_shuffle_epi32(x3, 0x1b);
    __ParMod10to4SSSE3v( x4, _mm_slli_si128(m0, 8));
    x4 = _mm_shuffle_epi32(x4, 0x1b);
    __ParMod10to4SSSE3v( x5, m1);
    x5 = _mm_shuffle_epi32(x5, 0x1b);
    __ParMod10to4SSSE3v( x6, _mm_slli_si128(m1, 8));
    x6 = _mm_shuffle_epi32(x6, 0x1b);
    cnt += 16;
    }
```

```
}
m0 = _mm_loadu_si128( (__m128i *) asc0bias);
if( cnt > 16) {
tmp = _mm_movemask_epi8( _mm_cmpgt_epi32(x7,_mm_setzero_si128()) ) ;
//x7 = _mm_add_epi32(x7, m0);
} else {
tmp = _mm_movemask_epi8( _mm_cmpgt_epi32(x3,_mm_setzero_si128()) );
}
#ifndef __USE_GCC__
__asm__ ("bsfl %1, %%ecx; movl %%ecx, %0;" :"=r"(idx) :"r"(tmp) : "%ecx");
#else
_BitScanForward(&idx, tmp);
#endif
x3 = _mm_add_epi32(x3, m0);
cnt -= (idx >>2);
x4 = _mm_add_epi32(x4, m0);
switch(cnt) {
case5:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
   _mm_storeu_si128( (__m128i *) ps, x4);
break;
case6:*(long long *)ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
   _mm_storeu_si128( (__m128i *) &ps[2], x4);
break;
(続く)
case7:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
   *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
   _mm_storeu_si128( (__m128i *) &ps[2], x4);
break;
case 8: _mm_storeu_si128( (__m128i *) &ps[0], x3);
   _mm_storeu_si128( (__m128i *) &ps[4], x4);
break;
case9:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) ps, x4);
   _mm_storeu_si128( (__m128i *) &ps[4], x5);
break;
case10:*(long long *)ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[2], x4);
   _mm_storeu_si128( (__m128i *) &ps[6], x5);
break;
```

```
case11:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
   *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[2], x4);
   _mm_storeu_si128( (__m128i *) &ps[6], x5);
break;
case 12: _mm_storeu_si128( (__m128i *) &ps[0], x3);
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[4], x4);
   _mm_storeu_si128( (__m128i *) &ps[8], x5);
break;
case13:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
   x5 = _mm_add_epi32(x5, m0);
```

```
   _mm_storeu_si128( (__m128i *) ps, x4);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[4], x5);
   _mm_storeu_si128( (__m128i *) &ps[8], x6);
break;
case14:*(long long *)ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[2], x4);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[6], x5);
   _mm_storeu_si128( (__m128i *) &ps[10], x6);
break;
case15:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
   *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[2], x4);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[6], x5);
   _mm_storeu_si128( (__m128i *) &ps[10], x6);
break;
case 16: _mm_storeu_si128( (__m128i *) &ps[0], x3);
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[4], x4);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[8], x5);
   _mm_storeu_si128( (__m128i *) &ps[12], x6);
break;
case17:x7 = _mm_add_epi32(x7, m0);
   *ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x7, 12));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) ps, x3);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[4], x4);
   _mm_storeu_si128( (__m128i *) &ps[8], x5);
   _mm_storeu_si128( (__m128i *) &ps[12], x6);
break;
case18:x7 = _mm_add_epi32(x7, m0);
   *(long long *)ps = _mm_cvtsi128_si64( _mm_srli_si128( x7, 8));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[2], x3);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[6], x4);
   _mm_storeu_si128( (__m128i *) &ps[10], x5);
   _mm_storeu_si128( (__m128i *) &ps[14], x6);
break;
case19:x7 = _mm_add_epi32(x7, m0);
   *ps++ = (wchar_t) _mm_cvtsi128_si64( _mm_srli_si128( x7, 4));
   *(long long *)ps = _mm_cvtsi128_si64( _mm_srli_si128( x7, 8));
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[2], x3);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[6], x4);
   _mm_storeu_si128( (__m128i *) &ps[10], x5);
   _mm_storeu_si128( (__m128i *) &ps[14], x6);
```

```
break;
case20:x7 = _mm_add_epi32(x7, m0);
   _mm_storeu_si128( (__m128i *) &ps[0], x7);
   x5 = _mm_add_epi32(x5, m0);
   _mm_storeu_si128( (__m128i *) &ps[4], x3);
   x6 = _mm_add_epi32(x6, m0);
   _mm_storeu_si128( (__m128i *) &ps[8], x4);
   _mm_storeu_si128( (__m128i *) &ps[12], x5);
   _mm_storeu_si128( (__m128i *) &ps[16], x6);
break;
}
return cnt;
}
```

14.5.1 大きな整数値の計算

14.5.1.1 MULX 命令と大きな整数値の計算

MULX 命令は MUL 命令に似ていますが、算術フラグを読み書きせず、デスティネーション・オペランドのレジスター割り当てにおける柔軟性を高めます。これらの拡張は、ハードウェアによる適切なアウトオブオーダー実行を可能にし、ソフトウェアはキャリーチェーンを破壊することなく加算キャリー命令を混在できます。

大きな整数値の計算 (2048 ビット RSA キーなど) では、MULX は MUL/ADC チェーンシーケンスに基づく手法のパフォーマンスをかなり向上できます (詳細は、http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf を参照してください)。インテル® AVX2 は効率良い手法を構築する際に使用できます。15.16.2 節を参照してください。

例 14-24 は、64 ビットよりも大きな整数値のキャリーチェーン計算を改善するため、MULX を使用する例を示します。

例 14-24 大きな整数値の MULX とキャリーチェーン

```
mov rax, [rsi+8*1]
mul rbp ; rdx:rax = rax * rbp
mov r8, rdx
add r9, rax
adc r8, 0
add r9, rbx
adc r8, 0
```

```
mulx rbx, r8, [rsi+8*1] ; rbx:r8 = rdx
* [rsi+8*1]
add r8, r9
adc rbx, 0
add r8, rbp
adc rbx, 0
```

MULX を使用した 128 ビット整数出力の実装は、atof/strtod から中間仮数の計算、128 ビット 2 進-10 進浮動小数点の仮数/指数の正規化まで、ライブラリー関数を実装するビルディング・ブロックとなります。例 14-25 では、128 ビットの 2 進-10 進浮動小数点命令を使用したビルディング・ブロック・マクロの例を示します。これは、MULX による 128 から 256 ビット幅整数の他倍長精度の中間結果を計算する利点を得られます。2 進-整数-10 進 (BID) 浮動小数点形式の詳細と、BID 操作を実装するライブラリーについては、http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library (英語) を参照してください。

例 14-25 2 進-10 進浮動小数点操作で使用されるビルディング·ブロック·マクロ

```
// 32 ビット word 単位の操作を使用した 64x64 ビット積のポータブルな C マクロ
// 出力: BID_UINT128 P128
#define __mul_64x64_to_128MACH(P128, CX64, CY64) ¥
{ ¥
   BID_UINT64 CXH,CXL,CYH,CYL,PL,PH,PM,PM2; ¥
   CXH = (CX64) >> 32; ¥
   CXL = (BID_UINT32)(CX64); ¥
   CYH = (CY64) >> 32; ¥
   CYL = (BID_UINT32)(CY64); ¥
   PM = CXH*CYL; ¥
   PH = CXH*CYH; ¥
   PL = CXL*CYL; ¥
   PM2 = CXL*CYH; ¥
   PH += (PM>>32); ¥
   PM = (BID_UINT64)((BID_UINT32)PM)+PM2+(PL>>32); ¥
   (P128).w[1] = PH + (PM>>32); ¥
   (P128).w[0] = (PM<<32)+(BID_UINT32)PL; ¥
}
```

```
// 64 ビット·モードで 128 ビット出力を生成する組込み関数を使用した 64x64 ビット積
// 出力: BID_UINT128 P128
#define __mul_64x64_to_128MACH_x64(P128, CX64, CY64) ¥
{ ¥
   (P128).w[0] = mulx_u64(CX64, CY64, &( (P128).w[1]) ); ¥
}
```

第 15 章
インテル® AVX、FMA およびインテル® AVX2 向けの最適化

インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) は、インテル® アーキテクチャーへの重要な拡張機能です。これまでの世代の 128 ビットのインテル® SSE ベクトル命令の機能を拡張し、256 ビット操作をサポートするようにベクトルレジスター幅を拡大しています。インテル® AVX 命令セット・アーキテクチャー (ISA) 拡張は浮動小数点命令に注目します。一部の 256 ビット整数ベクトルは、浮動小数点から整数および整数から浮動小数点への変換を行うことでサポートされます。

Sandy Bridge† マイクロアーキテクチャーにおけるインテル® AVX 命令は、ほとんどが 256 ビット・ハードウェアで実装されています。したがって、それぞれのコアは 256 ビット浮動小数点加算と乗算ユニットを備えています。除算と平方根ユニットは 256 ビットへ強化されていません。それゆえ、インテル® AVX 命令は、128 ビット・ハードウェアを使用した 2 つのステップで 256 ビット操作を完了します。

これまでの世代のインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) の大部分は 2 オペランド構文です (オペランドの 1 つがソースとデスティネーションとして機能)。インテル® AVX 命令は、ベクトル長をエンコードするビットフィールドを含む VEX プリフィクス付きでエンコードされ、3 オペランド構文をサポートします。ほとんどの命令は 2 つのソースと 1 つのデスティネーションを持ちます。VBLENDVPS や VBLENDVPD のような 4 オペランド命令もあります。追加されたオペランドにより非破壊ソース (NDS) が可能となり、MOVAPS 命令を使用してレジスターを複製する必要がなくなります。インテル® MMX® 命令を除く、ほとんどすべてのレガシーインテル® SSE 128 ビット命令には、3 オペランド構文をサポートする等価なインテル® AVX 命令があります。256 ビットのインテル® AVX 命令は 3 オペランド (いくつかは 4 オペランド構文) を採用しています。

256 ビット・ベクトル・レジスター YMM は、128 ビットの XMM レジスターを 256 ビットに拡張したものです。そのため YMM の下位 128 ビットは、レガシー XMM レジスターにエイリアスされます。256 ビットのインテル® AVX 命令は 256 ビットの結果を YMM に書き込むのに対し、128 ビットのインテル® AVX 命令は 128 ビットの結果を XMM に書き込んで YMM の上位 128 ビットにはゼロを設定します。64 ビット・モードでは 16 個のベクトルレジスターが利用できます。64 ビット・モード以外 (32 ビット・モード) では、下位 8 個のベクトルレジスターのみが利用できます。

ソフトウェアは、レガシーインテル® SSE コード、128 ビットのインテル® AVX コード、そして 256 ビットのインテル® AVX コードをどのような組み合わせでも使用できます。この節では、ベクトル長が混在するコードモジュールがインテル® SSE とインテル® AVX コード間の移行の遅延を被ることなく最適なパフォーマンスを達成するガイドラインをカバーしています。128 ビットのインテル® AVX コードと 256 ビットのインテル® AVX コードの混在では移行の遅延は発生しません。

メモリーに保存されるインテル® AVX ベクトルの最適なメモリー・アライメントは、32 バイトです。256 ビットのインテル® AVX 命令のデータ移動のいくつかは 32 バイト・アライメントを強制します。メモリーオペランドが適切にアライメントされていないと #GP フォルトが発生します。ほとんどの 256 ビットのインテル® AVX 命令はアドレス・アライメントを要求しません。これらの命令では一般にロードと計算操作が組み合わされており、命令はどのようなアライメントのメモリーアドレスも使用できます。

最高のパフォーマンスを達成するには、ソフトウェアは可能な限り 32 バイトにアライメントされたロードとストアを使用すべきです。

インテル® AVX 命令とレガシーインテル® SSE 命令を使用する上での主な違いを表 15-1 に示します。

表 15-1 256 ビットのインテル® AVX、128 ビットのインテル® AVX、およびレガシーインテル® SSE 拡張の機能比較

機能	**256 ビットのインテル® AVX**	**128 ビットのインテル® AVX**	**レガシーインテル® SSE - インテル® AES-NI**
機能範囲	浮動小数点演算、 データ移動	レガシー SIMD ISA と一致 (インテル® MMX® 命令を除く)	128 ビット FP と整数 SIMD ISA
レジスターオペランド	YMM	XMM	XMM
オペランド・シンタックス	4 つまで、 ソースを非破壊	4 つまで、 ソースを非破壊	2 オペランド・シンタックス、 ソースを破壊
メモリー・アライメント	ロード操作のセマンティクスでアライメントは不要	ロード操作のセマンティクスでアライメントは不要	常に 16B アライメントを強制
アライメントされた移動命令	32 バイト・アライメント	16 バイト・アライメント	16 バイト・アライメント
非破壊ソースオペランド	はい	はい	いいえ
レジスターストールの扱い	ビット 255:0 を更新	127:0 を更新、 128 以上はゼロビット	127:0 を更新、 128 以上は変更しない
組込み関数サポート	新しい 256 ビット・データ型 プロモートされた機能のための _mm256 プレフィクス 新しい機能のための新しい組込み関数	既存のデータ型 既存の機能のための同じプロトタイプの継続 新しい VEX-128 の機能には “_mm” プリフィクスを使用	ベースラインのデータ型とプロトタイプの定義
128 ビット・レーン	ほとんどの 256 ビット操作に適用	1 つの 128 ビット・レーン	1 つの 128 ビット・レーン
混在したコードの扱い	移行のペナルティーを避けるため VZEROUPPER を使用	移行のペナルティーなし	256 ビットのインテル® AVX コード実行後の移行のペナルティー

15.1 インテル® AVX 組込み関数のコーディング

256 ビットのインテル® AVX 命令では新しい組込み関数が定義されています。特に、既存のインテル® SSE の機能を 256 ビット・ベクトルへ拡張した 256 ビットのインテル® AVX 命令では、プロトタイプ化された “_mm” プリフィクスの代わりに “_mm256” プリフィクスによって、256 ビット操作向けに定義された新しいデータ型を使用します。256 ビットのインテル® AVX 命令の新しい機能には、全く新しいプロトタイプが定義されています。

レガシー SIMD ISA を継続する 128 ビットのインテル® AVX 命令には、これまでと同じプロトタイプを使用します。256 ビットと 128 ビットのインテル® AVX 命令の新しい機能は、それぞれ “_mm256” と “_mm” プリフィクス付きのプロトタイプです。

そのため、組込み関数で記述されたレガシー SIMD コードを 256 ビットのインテル® AVX へ移植する労力はそれほど掛かりません。

次のガイドラインは、インテル® SSE コードのシーケンスからインテル® AVX へ簡単に組込み関数を書き換える方法を示しています。

インテル® AVX:

- 静的および動的に割り当てられているバッファーを 32 バイトにアライメントします。
- バッファーサイズは倍にする必要があるでしょう。
- 「__mm_組込み関数名」のプリフィクスを __mm256 に変更します。
- 変数のデータ型名を __m128 から __m256 へ変更します。
- ループの反復回数を半分にします (または倍のストライド長に)。

以下に示す例 (デカルト座標変換) は、インテル® AVX 命令形式、32 バイト YMM レジスター、32 バイト・データ境界での静的および動的メモリーの割り当て、および YMM レジスター内の 8 つの浮動小数点を表現する C のデータ型を使用しています。

例 15-1 組込み関数を使用したデカルト座標変換

```
// インテル® SSE 組込み関数を使用
#include "wmmintrin.h"
int main()
{ int len = 3200;
// 16 バイト境界で動的にメモリーを割り当て
//
float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 16);
float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 16);
//データの初期化
for(int i=0; i<len; i++) pInVector[i] = 1;
float cos_teta = 0.8660254037;
float sin_teta = 0.5;
// 4 つの float を 16 バイトのアライメントで静的にメモリー割り当て
__declspec(align(16)) float cos_sin_teta_vec[4] =
{cos_teta, sin_teta, cos_teta, sin_teta};
__declspec(align(16)) float sin_cos_teta_vec[4] =
{sin_teta, cos_teta, sin_teta, cos_teta};
// __m128 データ型は、xmm レジスターの
// 4 つの float 要素を表します
__m128 Xmm_cos_sin =
_mm_load_ps(cos_sin_teta_vec);
// インテル® SSE の 128 ビットのパックド単精度ロード
__m128 Xmm_sin_cos =
_mm_load_ps(sin_cos_teta_vec);
__m128 Xmm0, Xmm1, Xmm2, Xmm3
// 2 回分アンロールされたループで 8 つの要素を処理
for(int i=0; i<len; i+=8)
{
  Xmm0 = _mm_load_ps(pInVector+i);
  Xmm1 = _mm_moveldup_ps(Xmm0);
  Xmm2 = _mm_movehdup_ps(Xmm0);
  Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);
  Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);
  Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);
  _mm_store_ps(pOutVector + i, Xmm3);
  Xmm0 = _mm_load_ps(pInVector+i+4);
  Xmm1 = _mm_moveldup_ps(Xmm0);
  Xmm2 = _mm_movehdup_ps(Xmm0);
  Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);
  Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);
  Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);
  _mm_store_ps(pOutVector+i+4, Xmm3);
}
_mm_free(pInVector);
_mm_free(pOutVector);
return 0;
  }
```

```
// インテル® AVX 組込み関数を使用
#include "immintrin.h"
int main()
{ int len = 3200;
// 32 バイト境界で動的にメモリーを割り当て
//
float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 32);
float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 32);
//データの初期化
for(int i=0; i<len; i++) pInVector[i] = 1;
float cos_teta = 0.8660254037;
float sin_teta = 0.5;
// 8 つの float を 32 バイトのアライメントで静的にメモリー割り当て
__declspec(align(32)) float cos_sin_teta_vec[8] =
{cos_teta, sin_teta, cos_teta, sin_teta,
cos_teta, sin_teta, cos_teta, sin_teta};
__declspec(align(32)) float sin_cos_teta_vec[8] =
{sin_teta, cos_teta, sin_teta, cos_teta,
sin_teta, cos_teta, sin_teta, cos_teta };
// __m256 データ型は 8 つの float 要素を格納
__m256 Ymm_cos_sin =
_mm256_load_ps(cos_sin_teta_vec);
// インテル® AVX の 256 ビットのパックド単精度ロード
__m256 Ymm_sin_cos =
_mm256_load_ps(sin_cos_teta_vec);
__m256 Ymm0, Ymm1, Ymm2, Ymm3;
// 2 回分アンロールされたループで 8 つの要素を処理
for(int i=0; i<len; i+=16)
{
   Ymm0 = _mm256_load_ps(pInVector+i);
   Ymm1 = _mm256_moveldup_ps(Ymm0);
   Ymm2 = _mm256_movehdup_ps(Ymm0);
   Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);
   Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
   Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);
   _mm256_store_ps(pOutVector + i, Ymm3);
   Ymm0 = _mm256_load_ps(pInVector+i+8);
   Ymm1 = _mm256_moveldup_ps(Ymm0);
   Ymm2 = _mm256_movehdup_ps(Ymm0);
   Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);
   Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
   Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);
   _mm256_store_ps(pOutVector+i+8, Ymm3);
}
_mm_free(pInVector);
_mm_free(pOutVector);
return 0;
   }
```

15.1.1 インテル® AVX アセンブリーのコーディング

組込み関数の移植のガイドラインと同様に、アセンブリーで記述されたコードを移植するガイドラインを以下に示します。

- 静的および動的に割り当てられているバッファーを 32 バイトにアライメントします。
- 必要であれば補足のバッファーサイズを 2 倍にします。
- 命令ニーモニックの前に “v” プリフィクスを追加します。
- レジスター名を xmm から ymm へ変更します。
- インテル® AVX 演算命令にデスティネーション・レジスターを追加します。
- ループの反復回数を半分にします (またはストライド長を倍にします)。

例 15-2 アセンブリーを使用したデカルト座標変換

```
// インテル® SSE アセンブリーを使用
int main()
{
  int len = 3200;
// 16 バイト境界で動的にメモリーを割り当て
//
float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 16);
float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 16);
//データの初期化
for(int i=0; i<len; i++)
  pInVector[i] = 1;
// 4 つの float を 16 バイトのアライメントで静的にメモリー割り当て

  float cos_teta = 0.8660254037;
  float sin_teta = 0.5;
  __declspec(align(16)) float cos_sin_teta_vec[4]
= {cos_teta, sin_teta, cos_teta, sin_teta};
  __declspec(align(16)) float sin_cos_teta_vec[4]
= {sin_teta, cos_teta, sin_teta, cos_teta};
// 2 回分アンロールされたループで 8 つの要素を処理
  __asm
  {
  mov rax, pInVector
  mov rbx, pOutVector
// ymm レジスターに 16 バイトをロード
  movups xmm3,
    xmmword ptr[cos_sin_teta_vec]
  movups xmm4,
    xmmword ptr[sin_cos_teta_vec]
  mov edx, len
  shl rdx, 2  // 入力配列のサイズ (バイト)
  xor ecx, ecx
loop1:
  movsldup xmm0, [eax+ecx]
  movshdup xmm1, [eax+ecx]
// 例: mulps は 2 オペランド
  mulps xmm0, xmm3
  mulps xmm1, xmm4
```

```
// インテル® AVX アセンブリーを使用
int main()
{
int len = 3200;
// 32 バイト境界で動的にメモリーを割り当て
//
float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 32);
float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 32);
//データの初期化
for(int i=0; i<len; i++)
  pInVector[i] = 1;
// 8 つの float を 32 バイト境界で静的にメモリー割り当て


  float cos_teta = 0.8660254037;
  float sin_teta = 0.5;
  __declspec(align(32)) float cos_sin_teta_vec[8]
= {cos_teta, sin_teta, cos_teta, sin_teta,
cos_teta, sin_teta, cos_teta, sin_teta};
  __declspec(align(32)) float sin_cos_teta_vec[8]
= {sin_teta, cos_teta, sin_teta, cos_teta,
sin_teta, cos_teta, sin_teta, cos_teta};
// 2 回分アンロールされたループで 16 つの要素を処理
  __asm
  {
  mov rax, pInVector
  mov rbx, pOutVector
// ymm レジスターに 32 バイトをロード
  vmovups ymm3,
    ymmword ptr[cos_sin_teta_vec]
  vmovups ymm4,
    ymmword ptr[sin_cos_teta_vec]
  mov edx, len
  shl rdx, 2  // 入力配列のサイズ (バイト)
  xor ecx, ecx
loop1:
  vmovsldup ymm0, [eax+ecx]
  vmovshdup ymm1, [eax+ecx]
// 例: vmulps は 3 オペランド
```

```
  addsubps xmm0, xmm1
// xmm レジスターから 16 バイトのストア
  movaps [ebx+ecx], xmm0

  movsldup xmm0, [eax+ecx+16]
  movshdup xmm1, [eax+ecx+16]
  mulps xmm0, xmm3
  mulps xmm1, xmm4
  addsubps xmm0, xmm1
// 前のストアからの 16 バイトのオフセット
  movaps [ebx+ecx+16], xmm0

// このループで処理された 32 バイト
// (コードは 2 回アンロール)
  add ecx, 32
  cmp ecx, edx
  jl loop1
  }
_mm_free(pInVector);
_mm_free(pOutVector);
return 0;
}
```

```
  vmulps ymm0, ymm0, ymm3
  vmulps ymm1, ymm1, ymm4
  vaddsubps ymm0, ymm0, ymm1
// ymm レジスターから 32 バイトのストア
  vmovaps [ebx+ecx], ymm0

  vmovsldup ymm0, [eax+ecx+32]
  vmovshdup ymm1, [eax+ecx+32]
  vmulps ymm0, ymm0, ymm3
  vmulps ymm1, ymm1, ymm4
  vaddsubps ymm0, ymm0, ymm1
// 前のストアからの 32 バイトのオフセット
  vmovaps [ebx+ecx+32], ymm0

// このループで処理された 64 バイト
// (コードは 2 回アンロール)
  add ecx, 64
  cmp ecx, edx
  jl loop1
  }
_mm_free(pInVector);
_mm_free(pOutVector);
return 0;
}
```

15.2 非破壊ソース (NDS)

ほとんどのインテル® AVX 命令は 3 つのオペランドで構成されています。一般的な命令は 2 つのソースと 1 つのデスティネーションを持ち、ソースオペランドは命令によって変更されることはありません。この節では NDS の機能を使用して、レジスターの複製を回避し、命令数と uop 数を減らしてパフォーマンスを向上する方法を説明します。この例では、インテル® AVX のコードはインテル® SSE のコードよりも 2 倍高速です。

次の例はベクトル化された多項式 A^3+A^2+A の計算を行っています。多項式計算の疑似コードを示します。

```
While (i<len)
{
  B[i] := A[i]^3 + A[i]^2 + A[i]
  i++
}
```

例 15-3 の左側のリストはインテル® SSE アセンブリーを使用したベクトル化の実装を示しています。このコードでは、A は追加のロード命令によってメモリーからレジスターへコピーされ、A2 はレジスター間の割り当てによってコピーされています。コードは 4 つの要素を処理するため 10 uop を要します。

中央の例では 128 ビットのインテル® AVX 命令を使用し NDS の利点を活用しています。追加のロードとレジスターのコピーは排除されています。このコードは 4 つの要素を 8 uop で処理し、左のベースラインよりも 30% ほど高速化されています。

右側の例は 256 ビットのインテル® AVX 命令を使用しています。8 つの要素を処理するため 8 uop を要します。NDS の機能と 2 倍のベクトル幅により、ベースラインよりも 2 倍以上スピードアップされています。

例 15-3 直接多項式の計算

インテル® SSE コード (ベースライン)	128 ビットのインテル® AVX	256 ビットのインテル® AVX

float* pA = InputBuffer;
float* pB = OutputBuffer;
int len = miBufferWidth-4;
__asm
{
mov rax, pA
mov rbx, pB
movsxd r8, len
loop1:
// ロード A
movups xmm0, [rax+r8*4]
// コピー A
movups xmm1, [rax+r8*4]
//A^2
mulps xmm1, xmm1
//コピー A^2
movups xmm2, xmm1
//A^3
mulps xmm2, xmm0
//A + A^2
addps xmm0, xmm1
//A + A^2 + A^3
addps xmm0, xmm2
// 結果をストア
movups[rbx+r8*4], xmm0
sub r8, 4
jge loop1
}
	float* pA = InputBuffer1;
	float* pB = OutputBuffer1;
	int len = miBufferWidth-4;
	__asm
	{
	mov rax, pA
	mov rbx, pB
	movsxd r8, len
	loop1:
	// ロード A
	vmovups xmm0, [rax+r8*4]
	
	//A^2
	vmulps xmm1, xmm0, xmm0
	
	//A^3
	vmulps xmm2, xmm1, xmm0
	//A+A^2
	vaddps xmm0, xmm0, xmm1
	//A+A^2+A^3
	vaddps xmm0, xmm0, xmm2
	// 結果をストア
	vmovups [rbx+r8*4], xmm0
	sub r8, 4
	jge loop1
	}
		float* pA = InputBuffer1;
		float* pB = OutputBuffer1;
		int len = miBufferWidth-8;
		__asm
		{
		mov rax, pA
		mov rbx, pB
		movsxd r8, len
		loop1:
		// ロード A
		vmovups ymm0, [rax+r8*4]
		
		//A^2
		vmulps ymm1, ymm0, ymm0
		
		//A^3
		vmulps ymm2, ymm1, ymm0
		//A+A^2
		vaddps ymm0, ymm0, ymm1
		//A+A^2+A^3
		vaddps ymm0, ymm0, ymm2
		// 結果をストア
		vmovups [rbx+r8*4], ymm0
		sub r8, 8
		jge loop1
		}

15.3 インテル® AVX コードとインテル® SSE コードの混在

インテル® AVX 命令セット・アーキテクチャーは、プログラマーが大規模なコードベースを段階的に移植することを可能にしますが、その結果インテル® AVX コードとインテル® SSE コードが混在することになります。コードにインテル® AVX とインテル® SSE が含まれる場合、次のことを考慮してください。

- インテル® SSE コードをインテル® コンパイラーで “/QxAVX” (Windows*) または “-xAVX” (Linux*) オプションを指定して再コンパイルします。これにより、すべてのインテル® SSE 命令は 128 ビットのインテル® AVX 命令へ自動的に変換されます。これは、アセンブリーと組込み関数コードに適用されます。”GCC -c -mAVX” はインテル® AVX コード (アセンブリー・ファイルを含む) を生成します。GCC アセンブラーもまた、インテル® SSE からインテル® AVX コードを生成する “-msse2avx” オプションをサポートしています。
- インテル® AVX とインテル® SSE コードは共存して実行することができます。インテル® SSE コードを含むサードパーティー・ライブラリーを使用するアプリケーションが、インテル® SSE コードを実行する他のモジュールを呼び出すインテル® AVX を実装する新しい DLL を展開する場合や、アプリケーション全体を再コンパイルできない場合が該当するでしょう。この場合、インテル® AVX コードは VZEROUPPER 命令を使用して インテル® AVX/インテル® SSE 間の移行のペナルティーを避ける必要があります。

インテル® AVX 命令は YMM レジスターの上位ビットを常に更新しますが、インテル® SSE 命令はこの上位ビットを変更しません。ハードウェアの観点からは、YMM レジスターの上位ビットは次の 3 つのステートのいずれかであると考えられます。

- クリーン (Clean): YMM の上位ビットはゼロ。これはプロセッサーが RESET から開始されたステートです。

- 変更後未保存 (Modified and Unsaved) (表 13-2 では M/U と省略されます): インテル® AVX 命令 (256 ビットまたは 128 ビット) の実行により、YMM デスティネーションの上位ビットが変更されています。これは YMM の上位がダーティーなステートと呼ばれます。このステートでは、YMM レジスターのビット 255:128 とビット 127:0 は、そのレジスターを操作した最後の (256 ビットまたは 128 ビット) インテル® AVX 命令に関連付けられています。
- 保存された/Non_INIT 上位ステート (Preserved/Non_INIT Upper State) (表 13-2 では P/N と省略されます): YMM の上位ビットは非ゼロです。YMM レジスターの上位 128 ビットと下位 128 ビットは、XRSTOR によって YMM の上位がダーティーなイメージがセーブされたため、最後に実行されたインテル® AVX 命令に関連していない可能性があります。

ソフトウェアが適切に VZEROUPPER を使用せずにインテル® AVX/インテル® SSE 命令を交互に混在させると、インテル® AVX/インテル® SSE 移行のペナルティーを被ることになります。インテル® SSE、インテル® AVX、または XSAVE/XRSTOR/VZEROUPPER/VZEROALL を使用した YMM ステート管理による実行の様子を図 15-1 に示します。移行またはプロセッサー・ステート “変更後未保存” に関連するペナルティーは、マイクロアーキテクチャーに依存する実装固有のものです。

図 15-1 は、Broadwell† マイクロアーキテクチャーを含むインテル® AVX をサポートする最近の世代のマイクロアーキテクチャーにおける、ステート移行のペナルティーが生じる可能性を示しています。A と B の移行のペナルティーは、移行を引き起こす命令が実行された時に発生します。これは、YMM ステートの全体を内部ストレージへコピーする大きなコストです。

“保存された/Non_INIT 上位ステート” に関連する YMM のステート移行の発生を最小限にするため、YMM のステートをセーブ/リストアする命令 XSAVE/XRSTOR を使用するソフトウェアは、メモリーの XSAVE 領域に上位が “クリーン” な YMM ステートを書き込む必要があります。メモリーからダーティーな YMM イメージを YMM レジスターにリストアするとペナルティーを被ります。この様子を図 14 -1 に示します。

Skylake† マイクロアーキテクチャーでは、インテル® SSE とインテル® AVX 命令の混在による YMM ステートの移行を管理するため、前の世代とは異なるステートマシンを実装しています。”変更後未保存 (Modified and Unsaved)” ステートでインテル® SSE 命令が実行されると、すべての YMM ステートの上位をセーブせず、個々のレジスターの上位ビットのみをセーブします。その結果、インテル® SSE とインテル® AVX 命令を混在させると、デスティネーション・レジスターの部分レジスターの依存性に関係するペナルティーと、デスティネーション・レジスターの上位ビットを操作する追加の混合命令が引き起こされます。図 15-2 に Skylake† マイクロアーキテクチャーに適用される移行のペナルティーを示します。

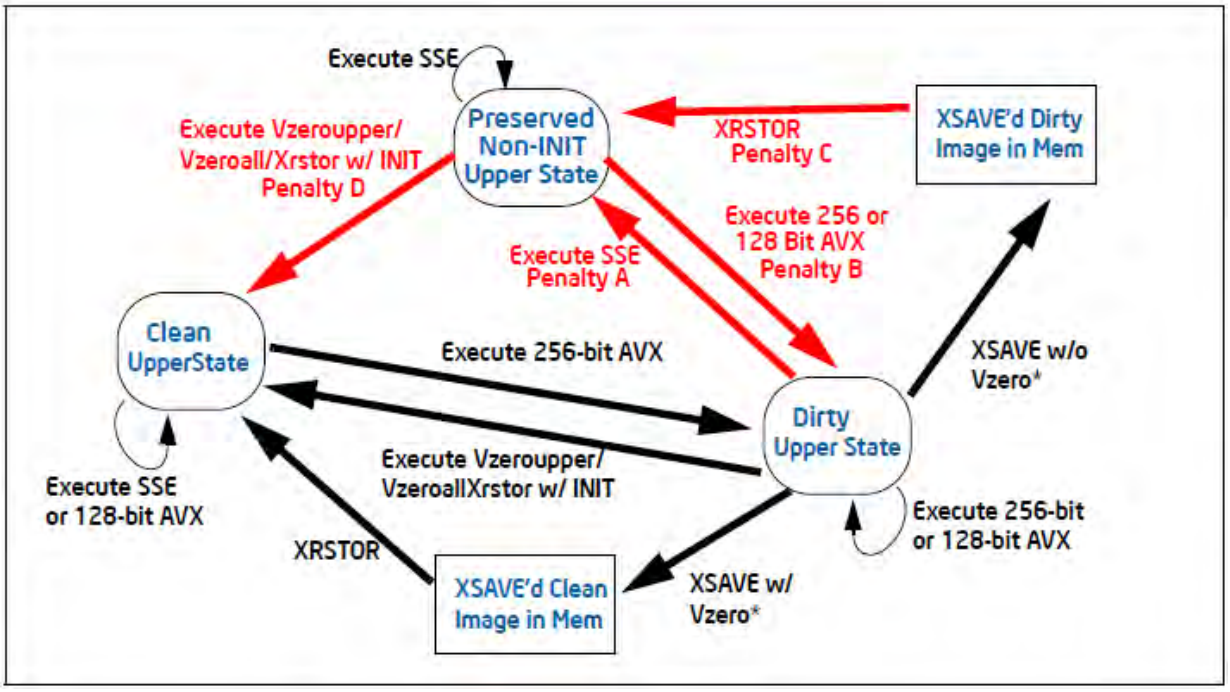


図 15-1 Broadwell† およびそれ以前のマイクロアーキテクチャーのインテル® AVX-インテル® SSE の移行

表 15-2 に、インテル® AVX とインテル® SSE コードが混在する場合の影響を示します。最下部の行は、最初の YMM ステート ('開始' と示される行) に依存して引き起こされる可能性があるペナルティーのタイプを示しています。また、表 15-2 はメモリーにストアされたダーティーな YMM イメージに関連する移行のペナルティー (タイプ C と D) の影響を示します。

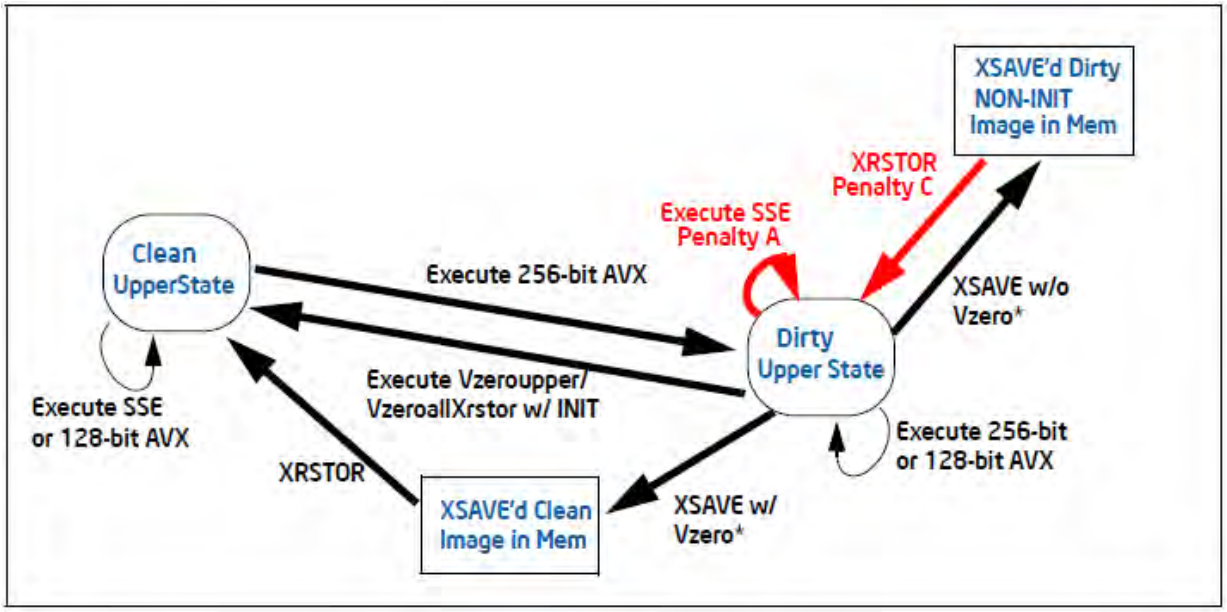


図 15-2 Skylake† マイクロアーキテクチャーにおけるインテル® AVX-インテル® SSE の移行

表 15-2 インテル® AVX とインテル® SSE コードが混在したステート移行

	インテル® SSE を実行			**128 ビットのインテル® AVX を実行**			**256 ビットのインテル® AVX を実行**			**VZeroupper**		**XRSTOR**
開始	クリーン	M/U	P/N	クリーン	M/U	P/S	クリーン	P/N	P/N	P/N	ダーティーイメージ	クリーンイメージ
終了	クリーン	P/N	P/N	クリーン	M/U	M/U	M/U	M/U	M/U	クリーン	P/N	クリーン
ペナルティー	なし	A	なし	なし	なし	B	なし	なし	B	D	C	なし

それぞれの移行ペナルティーのタイプの影響度は、マイクロアーキテクチャーによって異なります。Skylake† マイクロアーキテクチャーではいくつかの移行のペナルティーが軽減されています。図 15-2 に移行のダイアグラムと関連するペナルティーを示します。表 15-3 に、最近のマイクロアーキテクチャーにおける異なる移行ペナルティー・タイプの大まかな影響度を示しています。

表 15-3 異なるマイクロアーキテクチャーにおけるインテル® AVX-インテル® SSE 移行ペナルティーの影響度概算

タイプ	**Haswell† マイクロアーキテクチャー**	**Broadwell† マイクロアーキテクチャー**	**Skylake† マイクロアーキテクチャー**	**Ice Lake† Client マイクロアーキテクチャー**
A	~XSAVE	~XSAVE	部分的なレジスター依存性 + ブレンド	~XSAVE
B	~XSAVE	~XSAVE	N/A	~XSAVE
C	~XSAVE の仮数	~XSAVE の仮数	~XSAVE	~XSAVE の仮数
D	~XSAVE	~XSAVE	N/A	~XSAVE

256 ビットのインテル® AVX とインテル® SSE コードブロック間の高速な移行を可能にするには、インテル® SSE コードの実行との切り替えが必要なインテル® AVX コードブロックの前後で VZEROUPPER 命令を使用します。VZEROUPPER 命令はインテル® AVX レジスターの上位 128 ビットをリセットします。この命令のレイテンシーはゼロサイクルです。さらに、プロセッサーはインテル® SSE 命令やインテル® AVX 命令の実行後、クリーンステートに戻ります。今後の世代のマイクロアーキテクチャーでは移行のペナルティーはなくなります。Skylake† マイクロアーキテクチャーでは、インテル® SSE コードブロックはクリーンステートから上位ビットの依存性とブレンド操作のペナルティーなしで実行されます。

128 ビットのインテル® AVX 命令のデスティネーション・レジスターの上位 128 ビットは常にゼロです。そのため、128 ビットと 256 ビットのインテル® AVX 命令はペナルティーなしで混在できます。

アセンブリー/コンパイラー・コーディング規則 64 (影響 H、一般性 H): 256 ビットのインテル® AVX コードブロックと 128 ビットのインテル® SSE コードブロックが連続して実行する場合は、次のコードブロックを実行する “クリーン” ステートへの移行を容易にするため常に VZEROUPPER 命令を使用します。

15.3.1 関数呼び出しでインテル® AVX とインテル® SSE を混在させる

インテル® AVX とインテル® SSE 間の移行は、関数呼び出しや関数からのリターンにおいて意図せずに発生する可能性があります。例えば、256 ビットのインテル® AVX 関数がほかの関数を呼び出す場合、呼び出される関数がインテル® SSE コードを使用していることがあります。同様に、256 ビットのインテル® AVX 関数からリターンする場合に、呼び出し元がインテル® SSE コードを実行していることも考えられます。

アセンブリー/コンパイラー・コーディング規則 65 (影響 H、一般性 H): 256 ビットのインテル® AVX 命令が実行された後と、インテル® SSE コードが実行される関数呼び出しの前には、VZEROUPPER 命令を追加します。さらに、256 ビットのインテル® AVX 命令を使用するすべての関数の終わりに VZEROUPPER 命令を追加します。

例 15-4 関数呼び出しとインテル® AVX/インテル® SSE の移行

```
__attribute__((noinline)) void SSE_function()
{
__asm addps xmm1, xmm2
__asm xorps xmm3, xmm4
}
__attribute__((noinline)) void AVX_function_no_zeroupper()
{
__asm vaddps ymm1, ymm2, ymm3
__asm vxorps ymm4, ymm5, ymm6
}
__attribute__((noinline)) void AVX_function_with_zeroupper()
{
__asm vaddps ymm1, ymm2, ymm3
__asm vxorps ymm4, ymm5, ymm6
// インテル® AVX 関数からリターンする時は vzeroupper を追加
__asm vzeroupper
}
```

```
// コードは移行のペナルティーを被る
__asm vaddps ymm1, ymm2, ymm3
..
// ペナルティー
SSE_function();
AVX_function_no_zeroupper();
// ペナルティー
__asm addps xmm1, xmm2
```

```
// コードの移行ペナルティーを軽減
__asm vaddps ymm1, ymm2, ymm3
// インテル® AVX コードからインテル® SSE 関数を
// 呼び出す前に vzeroupper を追加
__asm vzeroupper //no penalty
SSE_function();
AVX_function_with_zeroupper();
// ペナルティーなし
__asm addps xmm1, xmm2
```

表 15-4 は、インテル® AVX コード実装とインテル® SSE コード間を移行する関数呼び出しにおいて VZEROUPPER を使用する場合と、使用しない場合のパフォーマンスへの影響のヒューリスティックをまとめたものです。

表 15-4 インテル® AVX とインテル® SSE コードの関数間の呼び出しにおける VZEROUPPER の影響

関数間の呼び出し	以前のマイクロアーキテクチャー	Skylake† マイクロアーキテクチャー
VZEROUPPER を使用	1X (ベースライン)	~1
VZEROUPPER なし	< 0.1X	ベースラインの仮数

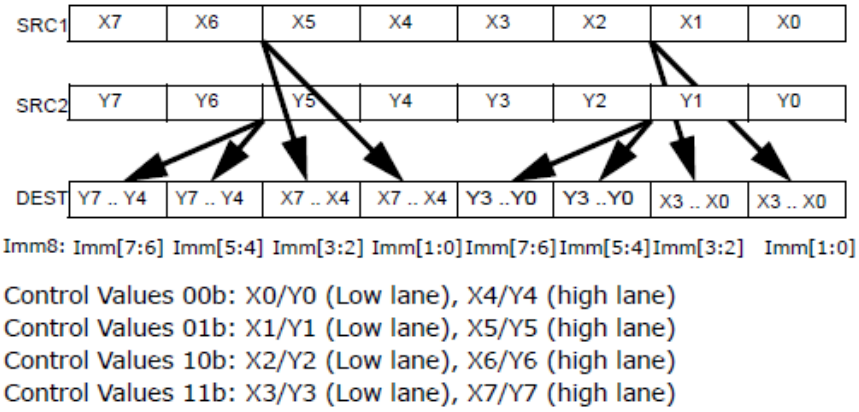
15.4 128 ビット・レーン操作とインテル® AVX

インテル® AVX における 256 ビット演算は、通常 128 ビット・レーンを 2 つ使用して行われます。ほとんどの 256 ビットのインテル® AVX 命令はレーンとして定義されます。各レーンのデスティネーション要素は、同じレーンのソース要素のみを使用して計算されます。以下で説明するように、レーン間で操作を行う命令もわずかにあります。

インテル® SSE 演算命令の大部分は、各データ要素の垂直位置に沿って計算を行います。128 ビット・レーンは、128 ビット・コードを 256 ビットのインテル® AVX コードへ移植する際には影響しません。VADDPS 命令はこの例の 1 つです。

多くの 128 ビットのインテル® SSE 命令はデータ要素を水平方向に移動します。例えば、SHUFPS は imm8 バイトを使用してデータ要素の水平移動を制御します。

インテル® AVX は、下位 128 ビット・レーンと上位 128 ビット・レーン内で同じ制御フィールドを使用することで、水平方向の 128 ビット SIMD 命令のレーンを 256 ビット操作に適用します。例えば 256 ビット VSHUFPS 命令は、128 ビット・レーン内の各デスティネーション要素のソースの位置を選択するため、4 つの制御値を含む制御バイトを使用します。以下にその様子を示します。



15.4.1 レーンの概念とプログラミング

レーンの概念を導入しインテル® SSE 命令セットを実装したアルゴリズムは、256 ビットのインテル® AVX に容易に変換できます。反復 0 から n を実行するインテル® SSE のアルゴリズムは、反復 i を下位のレーンで計算し反復 i + k を上位のレーンで計算するように変換できます。連続する反復 k は 1 に相当します。

インテル® SSE 命令で実装されたいくつかのベクトル・アルゴリズムには、前述した簡単な変換が適用できないものがあります。例えば、16 バイト内で要素を移動するシャッフルは、32 バイト・シャッフルではレーンをまたげないため、32 バイトでシャッフルするようにそのまま変換することはできません。

レーンを操作するビルディング・ブロックとして、次の命令を使用できます。

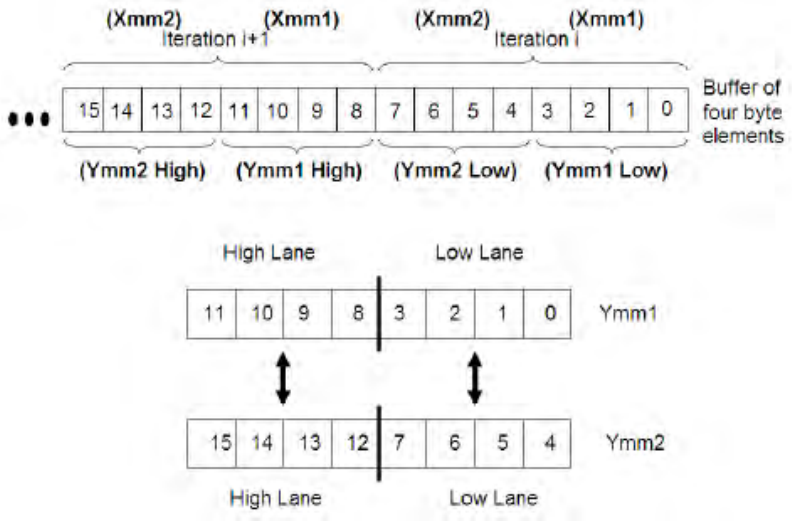
- VINSERTF128 - パックド浮動小数点値を挿入。
- VEXTRACTF128 - パックド浮動小数点値を抽出。
- VPERM2F128 - 浮動小数点値の並べ替え (permute)。
- VBROADCAST - ブロードキャストをロード。

次の節では 2 つの手法を示します: ストライドロードとレジスター間のオーバーラップ。これらの方法は前述のレーン内のデータ配置を実装し、レーンを交差する計算を必要とする多くのアルゴリズムで役立ちます。

15.4.2 ストライドロードの手法

ストライドロードの手法は、インテル® AVX 命令を使用するプログラミング方式であり、サポートされないレーン間のシャッフルを伴うアルゴリズムに適しています。

レーン間のシャッフルを避けてデータを配置する方法を示します。考え方としては、対応するインテル® SSE アルゴリズムを模倣する方法で 128 ビット・ロードを使用するため、下位レーンのループ反復 i と上位レーンの反復 i + k を実行する 256 ビットのインテル® AVX 命令を有効にします。次の例において k は 1 に相当します。

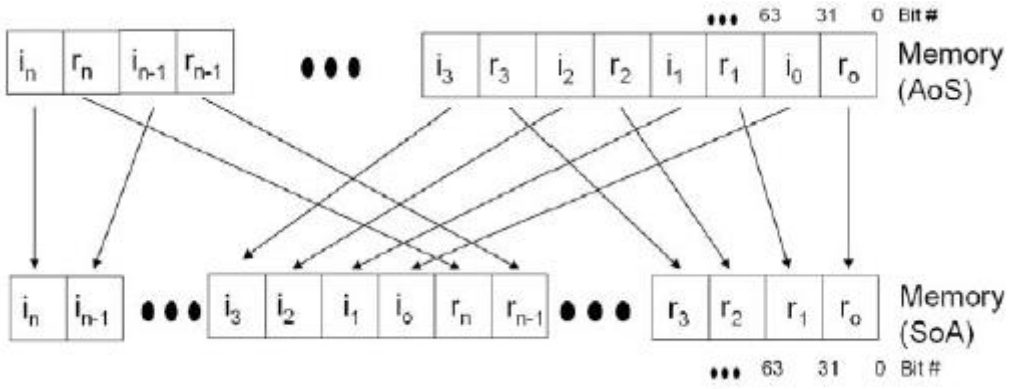


図中の Ymm1 と Ymm2 の下位レーンの値は、インテル® SSE 実装の反復 i に相当します。同様に、Ymm1 と Ymm2 の上位レーンの値は、反復 i + 1 に相当します。

以下に、構造体配列 (AoS) から配列構造体 (SoA) への変換におけるストライドロード方式の例を示します。この例では、入力バッファーには AoS 形式の複素数値が含まれています。それぞれの複素数は実数と float 値の虚数で構成されます。出力バッファーは SoA として配置されます。複素数のすべての実数コンポーネントは、出力バッファーの前半に配置され、すべての虚数コンポーネントはバッファーの後半に格納されます。次の疑似コードと図は変換の様子を示しています。

例 15-5 C コードにおける複素数の AoS-SoA 変換

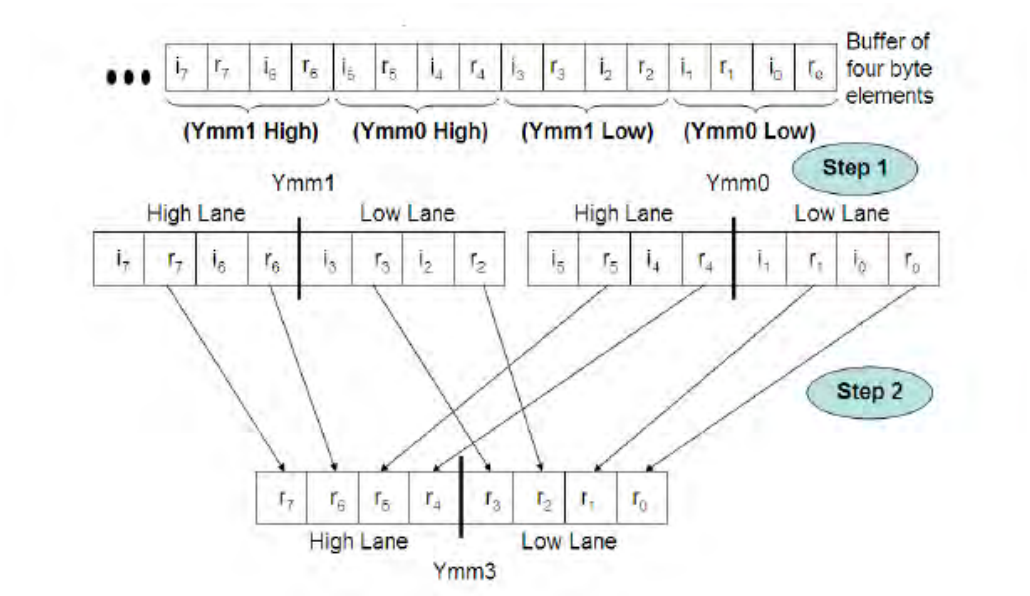
```
for (i=0; i < N; i++)
{
    Real[i] = Complex[i].Real;
    Imaginary[i] = Complex[i].Imaginary;
}
```



以下の図に示すように、インテル® SSE アルゴリズムの 16 バイトから 32 バイトへの簡単な操作の拡張では、レーンを交差するデータの移動が必要になります。しかし、これはインテル® AVX 命令セット・アーキテクチャーでは不可能であるため、異なる手法が必要とされます。

レーンを交差するシャッフルの課題は、インテル® AVX による AoS から SoA への変換で解決できます。VINSERTF128 を使用して YMM レジスターの適切なレーンへ 16 バイトをロードすることで、レーンを交差する

シャッフルの必要性を排除できます。ステップ 1 で YMM レジスターにデータを適切に配置されると、ステップ 2 でレーン内のデータを移動するため 32 バイト VSHUFPS を使用できます。



次のコードは、インテル® SSE 実装と 256 ビットのインテル® AVX 実装の AoS から SoA への変換の比較とパフォーマンス上のゲインを示しています。

例 15-6 インテル® AVX を使用した複素数の AoS から SoA への変換

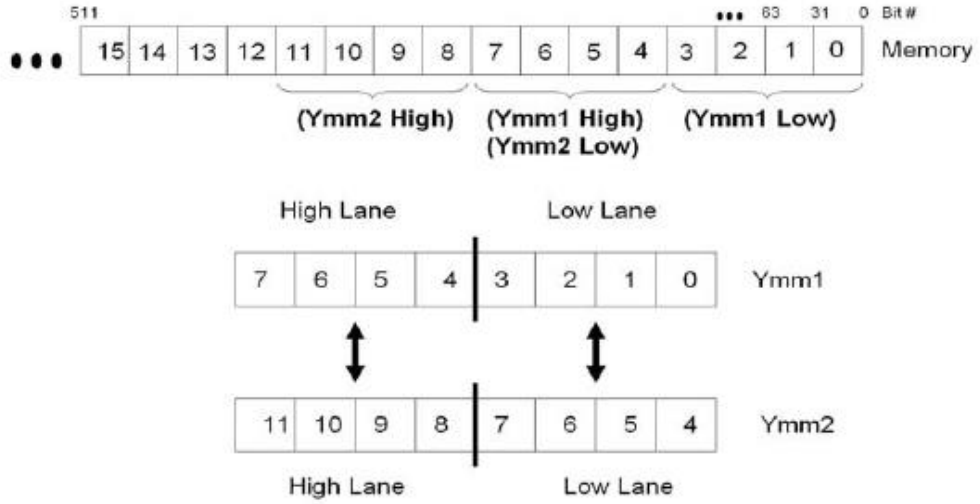
インテル® SSE コード	インテル® AVX コード

xor rbx, rbx
xor rdx, rdx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr1
mov rax, outPtr2
loop1:
movups xmm0, [rdi+rbx]
//i1 r1 i0 r0
movups xmm1, [rdi+rbx+16]
// i3 r3 i2 r2
movups xmm2, xmm0
 	
shufps xmm0, xmm1, 0xdd
//i3 i2 i1 i0
shufps xmm2, xmm1, 0x88
//r3 r2 r1 r0
 	
movups [rax+rdx], xmm0
movups [rsi+rdx], xmm2
add rdx, 16
add rbx, 32
cmp rcx, rbx
jnz loop1
	xor rbx, rbx
	xor rdx, rdx
	mov rcx, len
	mov rdi, inPtr
	mov rsi, outPtr1
	mov rax, outPtr2
	loop1:
	vmovups xmm0, [rdi+rbx]
	//i1 r1 i0 r0
	vmovups xmm1, [rdi+rbx+16]
	// i3 r3 i2 r2
	vinsertf128 ymm0, ymm0, [rdi+rbx+32] , 1
	//i5 r5 i4 r4; i1 r1 i0 r0
	vinsertf128 ymm1, ymm1, [rdi+rbx+48] , 1
	//i7 r7 i6 r6; i3 r3 i2 r2
	vshufps ymm2, ymm0, ymm1, 0xdd
	//i7 i6 i5 i4; i3 i2 i1 i0
	vshufps ymm3, ymm0, ymm1, 0x88
	//r7 r6 r5 r4; r3 r2 r1 r0
	vmovups [rax+rdx], ymm2
	vmovups [rsi+rdx], ymm3
	add rdx, 32
	add rbx, 64
	cmp rcx, rbx
	jnz loop1

15.4.3 レジスター・オーバーラップの手法

レジスター・オーバーラップの手法は、シャッフルを使用するアルゴリズムにおいて有効です。ストライドロードの手法と同様に、レジスター・オーバーラップの手法はレーンを交差するシャッフルを避けてデータを配置します。

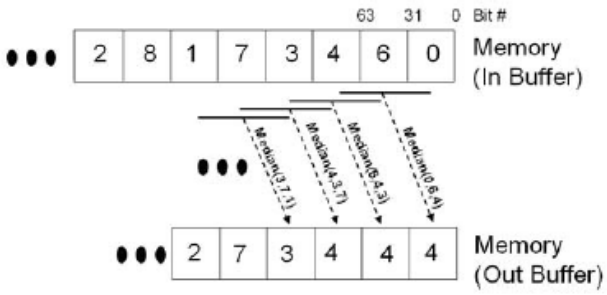
この手法は、シーケンシャルな反復によって部分的に共有される連続したデータを処理するアルゴリズムに役立ちます。以下の図は期待されるデー配置を示します。これは、256 ビット・ロードをオーバーラップするか、VPERM2F128 命令を使用することで可能になります。



次の Median3 のサンプルコードは、レジスター・オーバーラップの手法を示しています。Median3 の手法では、ベクトル中の 3 つの連続した要素ごとの中央値を計算します。

```
Y[i] = Median( X[i], X[i+1], X[i+2] )
```

ここで Y は出力ベクトル、X は入力ベクトル。次の図はメディアン・アルゴリズムによる計算の様子を示しています。



Median3 アルゴリズムの 3 つの実装例を示します。リスト 1 はインテル® SSE の実装で、リスト 2 と 3 の実装はレジスター・オーバーラップの手法を使用した 2 つの方法です。リスト 2 では、入力バッファーから YMM レジスターへ、オーバーラップした 256 ビットのロード命令を使用してデータをロードしています。リスト 3 は、入力バッファーから YMM レジスターへ、256 ビットのロードと VPERM2F128 命令を使用してデータをロードしています。リスト 2 と 3 は広いベクトル幅によるパフォーマンスの利点を得ています。

例 15-7 3 つの数値の中央値をもとめるレジスター・オーバーラップの方法

1: インテル® SSE コード	2: オーバーラップ・ロードと 256 ビットのインテル® AVX	3: VPERM2F128 と 256 ビットのインテル® AVX

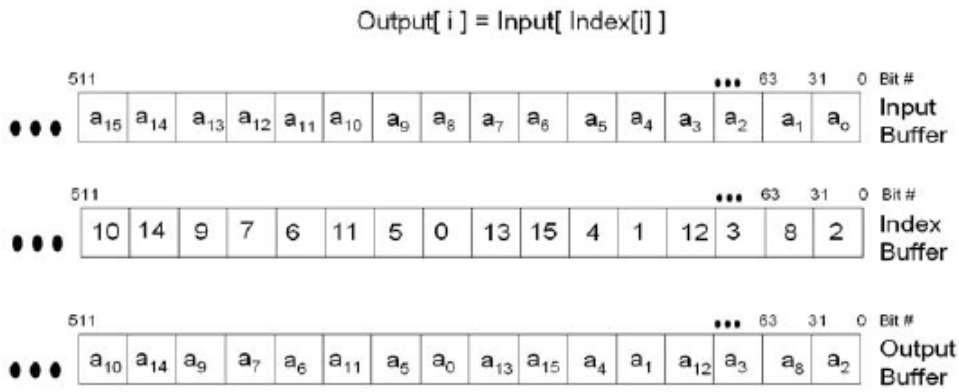
xor ebx, ebx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
movaps xmm0, [rdi]
 	
loop_start:
movaps xmm4, [rdi+16]
movaps xmm2, [rdi]
movaps xmm1, [rdi]
movaps xmm3, [rdi]
 	
add rdi, 16
add rbx, 4
shufps xmm2, xmm4, 0x4e
shufps xmm1, xmm2, 0x99
minps xmm3, xmm1
maxps xmm0, xmm1
minps xmm0, xmm2
maxps xmm0 ,xmm3
movaps [rsi], xmm0
movaps xmm0, xmm4
add rsi, 16
cmp rbx, rcx
jl loop_start
	xor ebx, ebx
	mov rcx, len
	mov rdi, inPtr
	mov rsi, outPtr
	vmovaps ymm0, [rdi]
	
	loop_start:
	vshufps ymm2, ymm0,
	[rdi+16], 0x4E
	vshufps ymm1, ymm0,
	ymm2, 0x99
	
	add rbx, 8
	add rdi, 32
	
	vminps ymm4, ymm0 ,ymm1
	vmaxps ymm0, ymm0, ymm1
	vminps ymm3, ymm0, ymm2
	vmaxps ymm5, ymm3, ymm4
	vmovaps [rsi], ymm5
	add rsi, 32
	vmovaps ymm0, [rdi]
	cmp rbx, rcx
	jl loop_start
		xor ebx, ebx
		mov rcx, len
		mov rdi, inPtr
		mov rsi, outPtr
		vmovaps ymm0, [rdi]
		
		loop_start:
		add rdi, 32
		vmovaps ymm6, [rdi]
		vperm2f128 ymm1, ymm0, ymm6, 0x21
		vshufps ymm3, ymm0, ymm1, 0x4E
		
		vshufps ymm2,ymm0,ymm3,0x99
		add rbx, 8
		vminps ymm5, ymm0, ymm2
		vmaxps ymm0, ymm0, ymm2
		vminps ymm4, ymm0, ymm3
		vmaxps ymm7, ymm4, ymm5
		vmovaps ymm0, ymm6
		vmovaps [rsi], ymm7
		add rsi, 32
		cmp rbx, rcx
		jl loop_start

15.5 データのギャザーとスキャッター

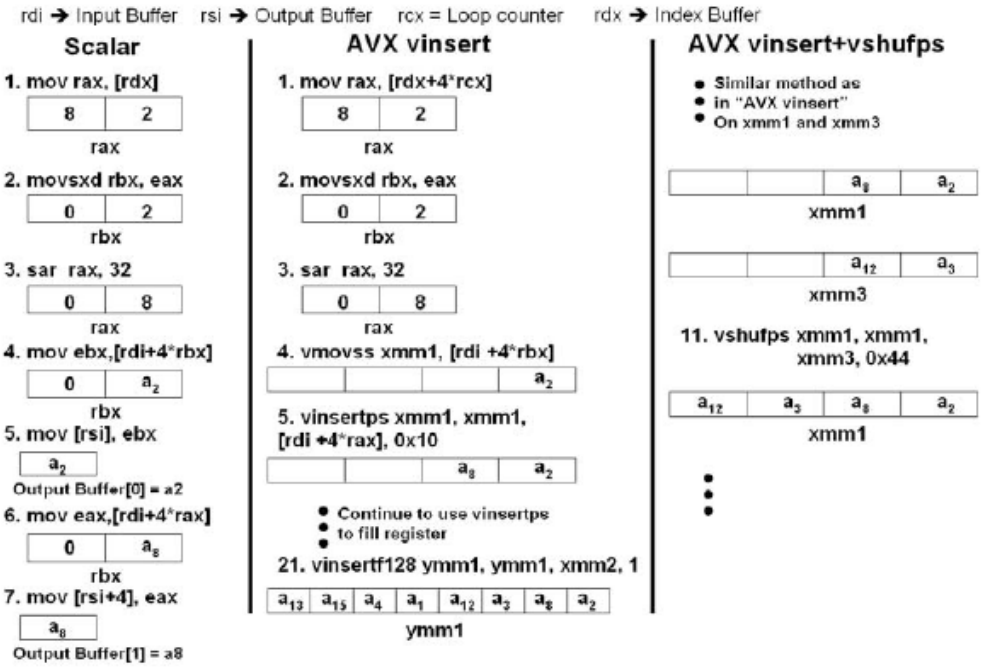
この節では、インテル® AVX 命令を使用したデータのギャザー (集約) とスキャッター (分散) の実装手法を説明します。

15.5.1 データギャザー

ギャザー操作はインデックス・バッファーで指定されたインデックスに基づいて入力バッファーから要素を読み取ります。ギャザーされた要素は出力バッファーへ書き込まれます。次の図はギャザー操作の様子を示しています。



次の 3 つの実装は 4 バイト要素の配列からのギャザー操作を行っています。リスト 1 は、汎用レジスターを使用したスカラー実装です。リスト 2 と 3 ではインテル® AVX 命令を使用しています。図には、前の図でデータの最初の反復を実行すると想定される、コードの一部を示しています (例 15-8 から抜粋)。



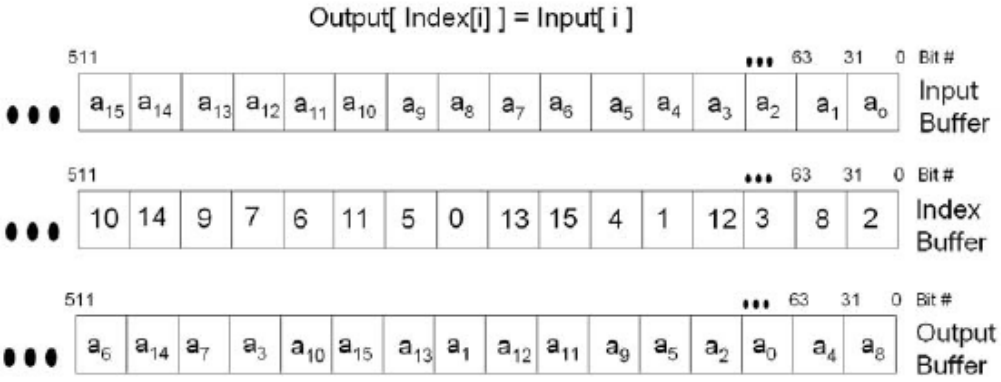
インテル® AVX 例のパフォーマンスは、対応するインテル® SSE 実装のパフォーマンスと同等です。次の表に 3 つのギャザーの実装を示します。

例 15-8 データギャザー – インテル® AVX とスカラーコード

1: スカラーコード	2: VINSERTPS を使用したインテル® AVX	3: VINSERTPS +VSHUFPS
mov rdi, InBuf	mov rdi, InBuf	mov rdi, InBuf
mov rsi, OutBuf	mov rsi, OutBuf	mov rsi, OutBuf
mov rdx, Index	mov rdx, Index	mov rdx, Index
xor rcx, rcx	xor rcx, rcx	xor rcx, rcx
loop1:	loop1:	loop1:
mov rax, [rdx]	mov rax, [rdx+4*rcx]	mov rax, [rdx+4*rcx]
movsxd rbx, eax	movsxd rbx, eax	movsxd rbx, eax
sar rax, 32	sar rax, 32	sar rax, 32
mov ebx,[rdi+4*rbx]	vmovss xmm1, [rdi +4*rbx]	vmovss xmm1, [rdi +4*rbx]
mov [rsi], ebx	vinsertps xmm1, xmm1,	vinsertps xmm1, xmm1,
mov eax,[rdi+4*rax]	[rdi +4*rax], 0x10	[rdi +4*rax], 0x10
mov [rsi+4], eax		
mov rax, [rdx+8]	mov rax, [rdx + 8+4*rcx]	mov rax, [rdx + 8+4*rcx]
movsxd rbx, eax	movsxd rbx, eax	movsxd rbx, eax
sar rax, 32	sar rax, 32	sar rax, 32
mov ebx, [rdi+4*rbx]	vinsertps xmm1, xmm1,	vmovss xmm3, [rdi +4*rbx]
mov [rsi+8], ebx	[rdi +4*rbx], 0x20	vinsertps xmm3, xmm3,
mov eax,[rdi+4*rax]	vinsertps xmm1, xmm1,	[rdi +4*rax], 0x10
mov [rsi+12], eax	[rdi +4*rax], 0x30	vshufps xmm1, xmm1,xmm3, 0x44
mov rax, [rdx+16]	mov rax, [rdx + 16+4*rcx]	mov rax, [rdx + 16+4*rcx]
movsxd rbx, eax	movsxd rbx, eax	movsxd rbx, eax
sar rax, 32	sar rax, 32	sar rax, 32
mov ebx, [rdi+4*rbx]	vmovss xmm2, [rdi +4*rbx]	vmovss xmm2, [rdi +4*rbx]
mov [rsi+16], ebx	vinsertps xmm2, xmm2,	vinsertps xmm2, xmm2,
mov eax, [rdi+4*rax]	[rdi +4*rax], 0x10	[rdi +4*rax], 0x10
mov [rsi+20], eax		
mov rax, [rdx+24]	mov rax,[rdx + 24+4*rcx]	mov rax,[rdx + 24+4*rcx]
movsxd rbx, eax	movsxd rbx, eax	movsxd rbx, eax
sar rax, 32	sar rax, 32	sar rax, 32
mov ebx, [rdi+4*rbx]	vinsertps xmm2, xmm2,	vmovss xmm4, [rdi +4*rbx]
mov [rsi+24], ebx	[rdi +4*rbx], 0x20	vinsertps xmm4, xmm4,
mov eax, [rdi+4*rax]	vinsertps xmm2, xmm2,	[rdi +4*rax], 0x10
mov [rsi+28], eax	[rdi +4*rax], 0x30	vshufps xmm2, xmm2,
	vinsertf128 ymm1, ymm1,	xmm4, 0x44
add rsi, 32	xmm2, 1	vinsertf128 ymm1, ymm1,
add rdx, 32	vmovaps [rsi+4*rcx], ymm1	xmm2, 1
add rcx, 8	add rcx, 8	vmovaps [rsi+4*rcx], ymm1
cmp rcx, len	cmp rcx, len	add rcx, 8
jl loop1	jl loop1	cmp rcx, len
		jl loop1

15.5.2 データ・スキャッター

スキャッター操作は入力バッファーから要素をシーケンシャルに読み取ります。そして、インデックス・バッファーで指定されたインデックスに基づいて出力バッファーへ書き込みます。次の図はスキャッター操作の様子を示しています。



次の例には、スカラー実装とインテル® AVX によるスキャッター・シーケンスの実装が示されています。インテル® AVX の例は、主に 128 ビットのインテル® AVX 命令で構成されます。インテル® AVX 例のパフォーマンスは、対応するインテル® SSE 実装のパフォーマンスと同等です。

例 15-9 インテル® AVX を使用したスキャッター操作

スカラーコード	インテル® AVX コード

mov rdi, InBuf
mov rsi, OutBuf
mov rdx, Index
xor cx, rcx
 	
loop1:
movsxd rax, [rdx]
mov ebx, [rdi]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 4]
mov ebx, [rdi + 4]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 8]
 	
mov ebx, [rdi + 8]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 12]
mov ebx, [rdi + 12]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 16]
mov ebx, [rdi + 16]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 20]
mov ebx, [rdi + 20]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 24]
mov ebx, [rdi + 24]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 28]
mov ebx, [rdi + 28]
mov [rsi + 4*rax], ebx
add rdi, 32
add rdx, 32
add rcx, 8
cmp rcx, len
jl loop1
	mov rdi, InBuf
	mov rsi, OutBuf
	mov rdx, Index
	xor cx, rcx
	
	loop1:
	vmovaps ymm0, [rdi + 4*rcx]
	movsxd rax, [rdx + 4*rcx]
	movsxd rbx, [rdx + 4*rcx + 4]
	vmovss [rsi + 4*rax], xmm0
	movsxd rax, [rdx + 4*rcx + 8]
	vpalignr xmm1, xmm0, xmm0, 4
	
	vmovss [rsi + 4*rbx], xmm1
	movsxd rbx, [rdx + 4*rcx + 12]
	vpalignr xmm2, xmm0, xmm0, 8
	vmovss [rsi + 4*rax], xmm2
	movsxd rax, [rdx + 4*rcx + 16]
	vpalignr xmm3, xmm0, xmm0, 12
	vmovss [rsi + 4*rbx], xmm3
	movsxd rbx, [rdx + 4*rcx + 20]
	vextractf128 xmm0, ymm0, 1
	vmovss [rsi + 4*rax], xmm0
	movsxd rax, [rdx + 4*rcx + 24]
	vpalignr xmm1, xmm0, xmm0, 4
	vmovss [rsi + 4*rbx], xmm1
	movsxd rbx, [rdx + 4*rcx + 28]
	vpalignr xmm2, xmm0, xmm0, 8
	vmovss [rsi + 4*rax], xmm2
	vpalignr xmm3, xmm0, xmm0, 12
	vmovss [rsi + 4*rbx], xmm3
	add rcx, 8
	cmp rcx, len
	jl loop1

15.6 インテル® AVX 向けのデータ・アライメント

この節では、インテル® AVX 命令におけるアライメントされたデータの利点を説明し、アライメントができない場合にパフォーマンスを改善するいくつかの方法を紹介します。この節では、いくつかの SAXPY カーネル例を使用して説明を行います。SAXPY はスカラーの「Alpha * X + Y」アルゴリズムです。

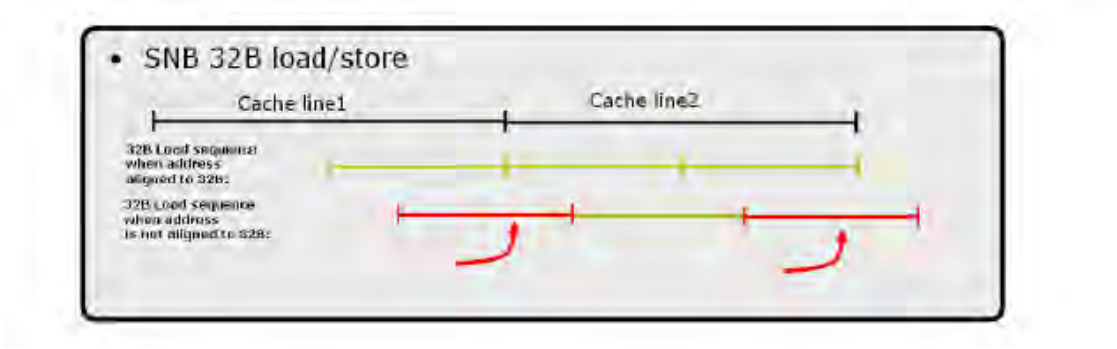
以下の C コードは SAXPY の C 実装です。

```
for (int i = 0; i < n; i++)
{ c[i] = alpha * a[i] + b[i]; }
```

15.6.1 32 バイトにデータをアライメント

ベクトル長に合わせてデータをアライメントすることを推奨します。16 バイト SIMD 命令を使用する場合、ロードされるデータは 16 バイトにアライメントされているべきです。同様に、インテル® AVX 命令で 32 バイト・レジスターを使用する場合の最良の結果は、32 バイトにアライメントされたデータから得られます。

インテル® AVX でアライメントされていない 32 バイト・ベクトルを使用すると、2 回に一度のロードがキャッシュライン分割を引き起こします (キャッシュラインが 64 バイトであるため)。16 バイトのベクトルを使用するインテル® SSE と比べて、キャッシュライン分割の頻度は倍になります。Nehalem† マイクロアーキテクチャー以降、キャッシュライン分割のペナルティーは劇的に少なくなりましたが、キャッシュライン分割の頻度が高いとメモリー集約型のコードではパフォーマンスの低下の原因となります。



例 15-10 インテル® AVX を使用した SAXPY

```
   mov rax, src1
   mov rbx, src2
   mov rcx, dst
   mov rdx, len
   xor rdi, rdi
   vbroadcastss ymm0, alpha

start_loop:
   vmovups    ymm1, [ rax + rdi ]
   vmulps     ymm1, ymm1, ymm0
   vmovups    ymm2, [ rbx + rdi ]
   vaddps     ymm1, ymm1, ymm2
   vmovups    [ rcx + rdi ], ymm1
   vmovups    ymm1, [ rax + rdi + 32 ]
   vmulps     ymm1, ymm1, ymm0
   vmovups    ymm2, [ rbx + rdi + 32]
   vaddps     ymm1, ymm1, ymm2
   vmovups   [ rcx + rdi + 32 ], ymm1
   add        rdi, 64
   cmp        rdi, rdx
   jl         start_loop
```

SAXPY は、データ・アライメントの重要性が高いメモリー集約型のカーネルです。最適なパフォーマンスを達成するには、両方のソースアドレスとデスティネーション・アドレスが 32 バイトにアライメントされる必要があります。3 つのアドレスのうち 1 つでも 32 バイト境界にアライメントされていないと、パフォーマンスは半減します。また、3 つのアドレスがすべて 32 バイトにアライメントされていなければ、パフォーマンスはさらに低下します。いくつかのケースでは、アライメントされていないインテル® AVX コードによるアクセスのパフォーマンスはインテル® SSE よりも低くなることがあります。他のインテル® AVX カーネルは、通常より計算集約型であるためデータ・アライメントのペナルティーの影響を軽減できます。

アセンブリー/コンパイラー・コーディング規則 66 (影響 H、一般性 M): 可能な限りデータを 32 バイトにアライメントします。ロードのアライメントよりもストアのアライメントの方が重要です。

インテル® コンパイラーで提供される _mm_malloc 組込み関数や MIcrosoft* コンパイラーの _aligned_malloc を使用して動的なデータ割り当てをアライメントできます。次に例を示します。

```
// 2048 float 要素のバッファーを 32 バイトのアライメントで動的に割り当て。
InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 32);
```

__declspec(align(32)) を使用して静的なデータ割り当てをアライメントできます。次に例を示します。

```
// 2048 float 要素のバッファーを 32 バイトのアライメントで静的に割り当て。
__declspec(align(32)) float InputBuffer[2048];
```

15.6.2 メモリーがアライメントされていない場合に 16 バイトのメモリーアクセスを考慮する

インテル® AVX の 32 バイト・ロードを使用して最良の結果を得るには、データを 32 バイト境界にアライメントします。しかし、データをアライメントできない場合や、データのアライメントが不明であることもあります。この状況は、ライブラリー関数を記述していて、入力データのアライメントが不明である場合に起こりえます。この場合、16 バイト・メモリー・アクセスを使用するのが最も良い手段であると考えられます。次の方法では、32 バイト YMM レジスターの恩恵を得ながら 16 バイト・ロードを使用します。

注意

Skylake† マイクロアーキテクチャから、この最適化は不要になりました。16 バイト・ロードが効率良く行われる唯一のケースは、データが 16 バイトにアライメントされており、32 バイトではアライメントされていない場合です。この場合、キャッシュライン分割メモリーアクセスが発行されないため、16 バイトのロードが有効な場合があります。

VMOVUPS、VINSERTF128 および VEXTRACTF128 命令の組み合わせを使用して、アライメントされていない 32 バイト・メモリー・アクセスを置き換えることを検討してください。

例 15-11 アライメントされていない 32 バイト・メモリー操作向けに使用する 16 バイト・メモリー操作

```
32 バイト・ロードを次のように置き換えます。
    vmovups ymm0, mem -> vmovups xmm0, mem
                         vinsertf128 ymm0, ymm0, mem+16, 1
32 バイト・ストアを次のように置き換えます。
    vmovups mem, ymm0 -> vmovups mem, xmm0
                         vextractf128 mem+16, ymm0, 1
次の組込み関数は、16 バイト・メモリー・アクセスを使用してアライメントされていない 32 バイト・メモリー操作を扱うことができます。
    _mm256_loadu2_m128 ( float const * addr_hi, float const * addr_lo);
    _mm256_loadu2_m128d ( double const * addr_hi, double const * addr_lo);
    _mm256_loadu2_m128 i( __m128i const * addr_hi, __m128i const * addr_lo);
    _mm256_storeu2_m128 ( float * addr_hi, float * addr_lo, __m256 a);
    _mm256_storeu2_m128d ( double * addr_hi, double * addr_lo, __m256d a);
    _mm256_storeu2_m128 i( __m128i * addr_hi, __m128i * addr_lo, __m256i a);
```

例 15-12 は、アライメントされていないアドレスによる SAXPY 向けの 2 つの実装を示しています。リスト 1 では 32 バイト・ロードを使用し、リスト 2 では 16 バイト・ロードを使用しています。これらのコードサンプルは、32 バイト境界からの 4 バイトのオフセットを持つ 2 つのソースバッファー (src1、src2) と、32 バイトにアライメントされたデスティネーション・バッファー (DST) を使用して実行します。32 バイトのメモリーアクセスの代わりに、2 つの 16 バイト・メモリー操作を行う方が高速です。[21]

例 15-12 アライメントされていないデータアクセス向けの SAXPY 実装

インテル® AVX (32 バイト・メモリー操作)	インテル® AVX (2 つの 16 バイト・メモリー操作)

mov rax, src1
mov rbx, src2
mov rcx, dst
mov rdx, len
xor rdi, rdi
vbroadcastss ymm0, alpha
start_loop:
vmovups ymm1, [rax + rdi]
vmulps ymm1, ymm1, ymm0
vmovups ymm2, [rbx + rdi]
vaddps ymm1, ymm1, ymm2
vmovups [rcx + rdi], ymm1
 	
vmovups ymm1, [rax+rdi+32]
vmulps ymm1, ymm1, ymm0
vmovups ymm2, [rbx+rdi+32]
vaddps ymm1, ymm1, ymm2
vmovups [rcx+rdi+32], ymm1
add rdi, 64
cmp rdi, rdx
jl start_loop
	mov rax, src1
	mov rbx, src2
	mov rcx, dst
	mov rdx, len
	xor rdi, rdi
	vbroadcastss ymm0, alpha
	start_loop:
	vmovups xmm2, [rax+rdi]
	vinsertf128 ymm2, ymm2, [rax+rdi+16], 1
	vmulps ymm1, ymm0, ymm2
	vmovups xmm2, [rbx + rdi]
	vinsertf128 ymm2, ymm2, [rbx+rdi+16], 1
	vaddps ymm1, ymm1, ymm2
	vaddps ymm1, ymm1, ymm2
	vmovups [rcx+rdi], ymm1
	vmovups xmm2, [rax+rdi+32]
	vinsertf128 ymm2, ymm2, [rax+rdi+48], 1
	vmulps ymm1, ymm0, ymm2
	vmovups xmm2, [rbx+rdi+32]
	vinsertf128 ymm2, ymm2, [rbx+rdi+48], 1
	vaddps ymm1, ymm1, ymm2
	vmovups [rcx+rdi+32], ymm1
	add rdi, 64
	cmp rdi, rdx
	jl start_loop

アセンブリー/コンパイラー・コーディング規則 67 (影響 M、一般性 H): 可能な限りデータを 32 バイトにアライメントします。ロードとストアの両方をアライメントできない場合、ロードよりもストアのアライメントを優先します。

15.6.3 ロードのアライメントよりもストアのアライメントが重要

処理されたデータバッファーの一部分だけが、アライメントされていることがあります。そのような場合、ストア操作で使用されるアライメントされたバッファーは、ロード操作がアライメントされたバッファーを使用するよりも良いパフォーマンスをもたらします。

アライメントされていないストアは、ページをまたぐキャッシュライン分割では非常に高いペナルティーを被るため、アライメントされていないロードに比べパフォーマンスは大幅に低下します。このペナルティーは、150 サイクルと推測されます。ページ境界をまたぐストアはリタイアメント時に実行されます。例 14-12 において、アライメントされていないストアアドレスを使用すると、3 つのアライメントされていないアドレスの SAXPY のパフォーマンスは、アライメントされた場合の約 4 分の 1 になります。

[21] Haswell† マイクロアーキテクチャー以降では、32 バイトまたは 64 バイト (インテル® AVX-512 を利用できる場合) レジスター全体を読み取ることを推奨します。

15.7 L1D キャッシュラインの置き換え

注意

Haswell† マイクロアーキテクチャー以降では、キャッシュラインの置き換えは問題となりません。

L1D キャッシュのロードミスが発生すると、要求されたデータを含むキャッシュラインが上位メモリー階層レベルから転送されます。L1D キャッシュが常にアクティブであるメモリー集約型のコードでは、L1D キャッシュのキャッシュライン入れ替えは他のロードを遅延させる恐れがあります。Sandy Bridge† マイクロアーキテクチャーと Ivy Bridge† マイクロアーキテクチャーでは、32 バイト・ロードのペナルティーは 16 バイト・ロードのペナルティーよりも大きくなります。そのため、32 バイト・ロードと L1D キャッシュよりも大きなデータセットを使用するメモリー集約型のインテル® AVX コードは、同等の 16 バイト・ロードのコードよりも低速になります。

例 15-12 においてデータセットが L2 キャッシュに収まり、16 バイト・メモリー・アクセスが実装されていると、32 バイト・メモリー操作を行う場合に比べ劇的に高速化されます。

16 バイト・メモリー・アクセスと 32 バイト・メモリー・アクセスの相対的なパフォーマンスの差は、マイクロアーキテクチャーの世代による実装固有であることに注意してください。

Haswell† マイクロアーキテクチャーでは、L1D キャッシュは各サイクルで 2 つの 32 バイト・フェッチをサポートするため、ここで述べたキャッシュラインの入れ換えの問題は適用されません。

15.8 4K エイリアシング

4K バイト・メモリー・エイリアシングは、コードがメモリー位置にデータをストアし、その直後に 4K バイト・オフセット離れたメモリー位置からロードを行うような場合に発生します。例えば、リニアアドレス 0x400020 からのロードが、リニアアドレス 0x401020 へのストアに続く場合、ロードとストアアドレスの 5-11 ビットは同じ値であり、アクセスされるバイト・オフセットは部分的、または完全に一致します。

4K エイリアシングはロードのレイテンシーに 5 サイクルのペナルティーを加算します。4K エイリアシングが繰り返し発生し、ロードがクリティカル・パスにある場合、このペナルティーは重大である可能性があります。ロードが 2 つのキャッシュラインにまたがっている場合、衝突するストアがキャッシュにコミットされるまでロードは遅延されます。そのため、アライメントされていないインテル® AVX のロードで繰り返し 4K エイリアシングが発生すると、高いパフォーマンス・ペナルティーを被ることになります。

4K エイリアシングを検出するには LD_BLOCKS_PARTIAL.ADDRESS_ALIAS イベントを使用します。このイベントはインテル® AVX のロードが 4K エイリアシングによってブロックされた回数をカウントします。

4K エイリアシングを回避するには、次の手法を順番に試してください。

- データを 32 バイトにアライメントします。
- 可能であれば、入力と出力バッファー間のオフセットを変更します。
- Sandy Bridge† マイクロアーキテクチャーと Ivy Bridge† マイクロアーキテクチャーでは 32 バイトにアライメントされていないメモリーには、16 バイトのメモリーアクセスを使用します。

15.9 条件付き SIMD パックドロードとストア

VMASKMOV 命令は、各データ要素に関連付けられたマスクビットに応じて、メモリー間との条件付きパックドデータの移動を行います。各データ要素のマスクビットは、マスクレジスター内の対応する要素の最上位ビットです。

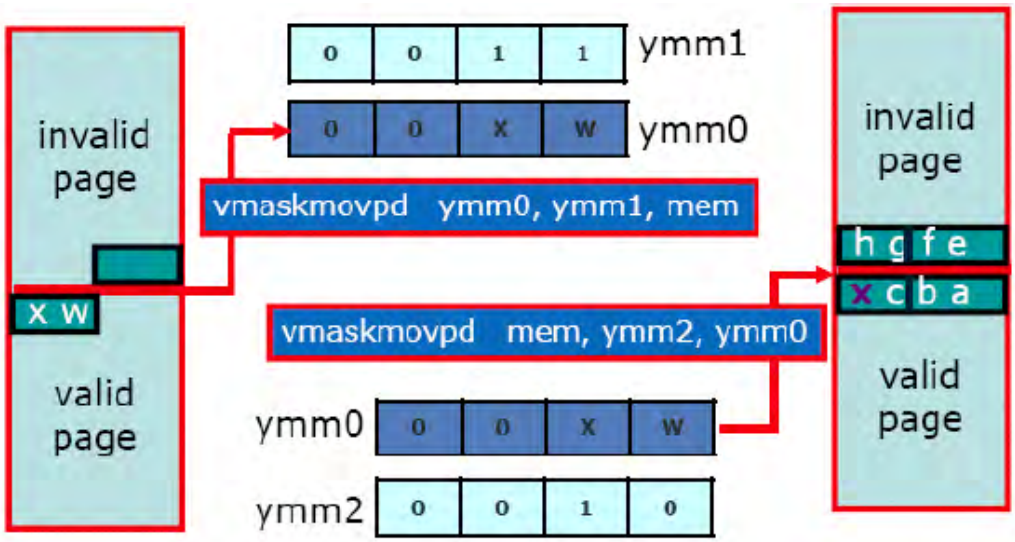
マスク付きロードを行う場合、マスク値 0 に対応する要素には 0 が返されます。マスク付きストアは、マスク値が 1 に対応する要素のみをメモリーに書き込みます。マスク値が 0 に対応するメモリー上の要素は保持されます。メモリーアクセスがマスクされるとフォルトが発生する可能性があります。メモリー位置に対応するマスクビットがゼ

ロである場合、メモリー位置を参照することでフォルトは発生しません。例えば、マスクビットがすべてゼロであればフォルトは検出されません。

次の図は、フォルトを発生しないマスク付きロードとストアの例を示しています。この例では、ロード操作のためのマスクレジスターは ymm1 であり、ストア操作のマスクレジスターは ymm2 です。

マスク付きロードやストアを使用する場合次のことを留意してください。

- Skylake† マイクロアーキテクチャー以前のプロセッサーでは、VMASKMOV ストアのアドレスは、マスクが判明した後にのみ解決されると考えられるべきです。マスク付きストアに続くロードは、マスク値が判明するまでブロックされることがあります (メモリー・ディスアンビグーションによって解決されない限り)。
- マスクがすべて 1 または 0 でないロードはマスク付きストアに依存し、ストアデータがキャッシュに書き込まれるまで待機します。マスクがすべて 1 のデータは、マスク付きストアから依存関係のあるロードへフォワードされます。マスクがすべて 0 のロードはマスク付きストアと依存関係はありません。



不正なアドレス範囲を含むマスク付きロードは、範囲がゼロ以下のマスク値であれば例外を発生しません。しかし、プロセッサーは不正な範囲のどの部分にもマスク値 1 がないことを決定するため、数百サイクルの “アシスト” を必要とすることがあります。このアシストは、マスクが “ゼロ” であってもプログラマーにとってロードを実行すべきでないことが明白な場合に発生する可能性があります。

VMASKMOV を使用する際には次のことを考慮してください。

- VMOVUPS が使用できない場合にのみ VMASKMOV を使用します。
- 可能であれば、32 バイトにアライメントされたアドレスで VMASKMOV を使用します。
- 不正な部分がゼロでマスクされていても、可能な限り有効な範囲でマスク付きロードを使用します。
- できるだけ早くマスクを決定します。
- できれば VMASKMOV ストアより前でロードを実行することで生じるストアフォワードの問題を回避します。
- マスク値によって VMASKMOV 命令がアシストを必要とすることがあることに注意してください (アシストが必要になるとデータをロードする VMASKMOV のレイテンシーは劇的に増加します)。
  - 不正なアドレスから 0 個の要素を選択するマスク値で VMASKMOV を使用するロードは、アシストを必要とします。
  - 不正なアドレスから特定のアドレス形式 ([base+index] や disp[base+index]) で 0 個の要素を選択するマスク値で VMASKMOV を使用するロードは、アシストを必要とします。

Skylake† マイクロアーキテクチャー・ベースのプロセッサーでは、VMASKMOV 命令のパフォーマンス特性に次のような注目すべきことがあります。

- マスク付きストアに続くロードは、マスク値が不明であってもブロックされません。
- 不正なアドレスへ 0 個の要素を書き込むマスク値で VMASKMOV を使用するストアデータは、アシストを必要とします。

15.9.1 条件付きループ

VMASKMOV は条件式を含むループのベクトル化を可能にします。スカラー実装で VMASKMOV を使用すると 2 つの利点があります。

- VMASKMOV コードはベクトル化されます。
- 分岐予測ミスは排除されます。

次に条件付きループの C のコードを示します。

例 15-13 条件式とループ

```
for(int i = 0; i < miBufferWidth; i++)
{    if(A[i]>0)
     {        B[i] = (E[i]*C[i]);
     }
     else
     {        B[i] = (E[i]*D[i]);
     }
}
```

例 15-14 VMASKMOV でループ条件を処理

スカラー	VMASKMOV を使用したインテル® AVX

float* pA = A;
float* pB = B;
float* pC = C;
float* pD = D;
float* pE = E;
uint64 len = (uint64) (miBuffer-
Width)*sizeof(float);
__asm
{
  mov rax, pA
  mov rbx, pB
  mov rcx, pC
  mov rdx, pD
  mov rsi, pE
  mov r8, len
//xmm8 はすべてゼロ
  vxorps xmm8, xmm8, xmm8
  xor r9,r9
loop1:
  vmovss xmm1, [rax+r9]
  vcomiss xmm1, xmm8
  jbe a_le
a_gt:
  vmovss xmm4, [rcx+r9]
	float* pA = A;
	float* pB = B;
	float* pC = C;
	float* pD = D;
	float* pE = E;
	uint64 len = (uint64)
	(miBufferWidth)*sizeof(float);
	__asm
	{
	  mov rax, pA
	  mov rbx, pB
	  mov rcx, pC
	  mov rdx, pD
	  mov rsi, pE
	  mov r8, len
	//ymm8 はすべてゼロ
	  vxorps ymm8, ymm8, ymm8
	//ymm9 はすべて 1
	  vcmpps ymm9, ymm8, ymm8, 0
	  xor r9,r9
	loop1:
	  vmovups ymm1, [rax+r9]
	  vcmpps ymm2, ymm8, ymm1, 1
	  vmaskmovps ymm4, ymm2, [rcx+r9]

```
   jmp      mul
a_le:
   vmovss   xmm4, [rdx+r9]
mul:
   vmulss   xmm4, xmm4, [rsi+r9]
   vmovss   [rbx+r9], xmm4
   add      r9, 4
   cmp      r9, r8
   jl       loop1
}
```

```
   vxorps      ymm2, ymm2, ymm9
   vmaskmovps  ymm5, ymm2, [rdx+r9]
   vorps       ymm4, ymm4, ymm5
   vmulps      ymm4,ymm4, [rsi+r9]
   vmovups     [rbx+r9], ymm4
   add         9, 32
   cmp         r9, r8
   jl          loop1
}
```

例 15-14 の左のリストのパフォーマンスは分岐予測ミスに影響され、右のデータ依存の分岐がない VMASKMOV の例より 1 桁遅くなることがあります。

15.10 整数と浮動小数点コードの混在

インテル® AVX 命令の整数 SIMD の機能性は 128 ビットに制限されます。整数 SIMD と浮動小数点 SIMD 命令を混在するアルゴリズムもあります。そのようなレガシー 128 ビット・コードを 256 ビットのインテル® AVX コードへ移植するには、特別な配慮が必要です。

例えば、PALINGR (Packed Align Right) は、整数と浮動小数点コードでデータ要素の配置を行う際に役立つ整数 SIMD 命令ですが、インテル® AVX には対応する 256 ビットの VPALINGR 命令がありません。

大半が浮動小数点で一部が整数操作を含むレガシーコードを 256 ビットのインテル® AVX コードへ移植する場合、検討すべき 3 つの手法があります。

- 重要な 128 ビットの整数 SIMD 命令に代わる 256 ビットのインテル® AVX 命令 (存在する場合) を見つけます。これは、データ要素を再配置する整数 SIMD 命令に当てはまる傾向があります。
- 128 ビットのインテル® AVX 命令と 256 ビットのインテル® AVX 命令を混在させます。
- インテル® AVX2 命令を使用します。

この 2 つの手法による性能の向上は変動する可能性があります。256 ビットのベクトル幅をフルに利用するには、できる限り最初の方法を使用します。

コードの大半が整数の場合は、128 ビットのインテル® SSE 命令から 128 ビットのインテル® AVX 命令にコードを変換し、非破壊ソース(NDS) から利点を得ることを検討します。

例 15-15 C コードでの 3 タップフィルター

```
for(int i = 0; i < len -2; i++)
{
    pOut[i] = A[i]*coeff[0]+A[i+1]*coeff[1]+A[i+2]*coeff[2];
}
```

例 15-16 128 ビットの整数と FP が混在した SIMD の 3 タップフィルター

```
   xor ebx, ebx
   mov rcx, len
   mov rdi, inPtr
   mov rsi, outPtr
   mov r15, coeffs
   movss xmm2, [r15] // coeff 0 をロード
   shufps xmm2, xmm2, 0 // coeff 0 をブロードキャスト
   movss xmm1, [r15+4] // coeff 1 をロード
   shufps xmm1, xmm1, 0 // coeff 1 をブロードキャスト
   movss xmm0, [r15+8] // coeff 2 をロード
   shufps xmm0, xmm0, 0 // coeff 2 をブロードキャスト
   movaps xmm5, [rdi] //xmm5={A[n+3],A[n+2],A[n+1],A[n]}
loop_start:
   movaps xmm6, [rdi+16] //xmm6={A[n+7],A[n+6],A[n+5],A[n+4]}
   movaps xmm7, xmm6
   movaps xmm8, xmm6
   add rdi, 16 //inPtr+=16
   add rbx, 4 //ループカウンター
   palignr xmm7, xmm5, 4 //xmm7={A[n+4],A[n+3],A[n+2],A[n+1]}
   palignr xmm8, xmm5, 8 //xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
   mulps xmm5, xmm2 //xmm5={C0*A[n+3],C0*A[n+2],C0*A[n+1], C0*A[n]}


   mulps xmm7, xmm1 //xmm7={C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
   mulps xmm8, xmm0 //xmm8={C2*A[n+5],C2*A[n+4] C2*A[n+3],C2*A[n+2]}
   addps xmm7 ,xmm5
   addps xmm7, xmm8
   movaps [rsi], xmm7
   movaps xmm5, xmm6
   add rsi, 16 //outPtr+=16
   cmp rbx, rcx
   jl loop_start
```

例 15-17 VSHUFPS を使用した 256 ビットのインテル® AVX の 3 タップフィルター

```
   xor ebx, ebx
   mov rcx, len
   mov rdi, inPtr
   mov rsi, outPtr
   mov r15, coeffs
   vbroadcastss ymm2, [r15] //coeff 0 のロードとブロードキャスト
   vbroadcastss ymm1, [r15+4] //coeff 1 のロードとブロードキャスト
   vbroadcastss ymm0, [r15+8] //coeff 2 のロードとブロードキャスト
loop_start:
   vmovaps ymm5, [rdi] // Ymm5={A[n+7],A[n+6],A[n+5],A[n+4];
                       // A[n+3],A[n+2],A[n+1] , A[n]}
   vshufps ymm6,ymm5,[rdi+16],0x4e // ymm6={A[n+9],A[n+8],A[n+7],A[n+6];
                                   // A[n+5],A[n+4],A[n+3],A[n+2]}
   vshufps ymm7,ymm5,ymm6,0x99 // ymm7={A[n+8],A[n+7],A[n+6],A[n+5];
                               // A[n+4],A[n+3],A[n+2],A[n+1]}
   vmulps ymm3,ymm5,ymm2       // ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5],C0*A[n+4];
                               // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
   vmulps ymm9,ymm7,ymm1 // ymm9={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                         // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
   vmulps ymm4,ymm6,ymm0 // ymm4={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                         // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
   vaddps ymm8 ,ymm3,ymm4
   vaddps ymm10, ymm8, ymm9
   vmovaps [rsi], ymm10
   add rdi, 32   //inPtr+=32
   add rbx, 8    //ループカウンター
   add rsi, 32 //outPtr+=32
   cmp rbx, rcx
   jl loop_start
```

例 15-18 256 ビットのインテル® AVX と128 ビットのインテル® AVX コードが混在した 3 タップフィルター

```
   xor ebx, ebx
   mov rcx, len
   mov rdi, inPtr
   mov rsi, outPtr
   mov r15, coeffs
   vbroadcastss ymm2, [r15] //coeff 0 のロードとブロードキャスト
   vbroadcastss ymm1, [r15+4] //coeff 1 のロードとブロードキャスト
   vbroadcastss ymm0, [r15+8] //coeff 2 のロードとブロードキャスト
   vmovaps xmm3, [rdi] //xmm3={A[n+3],A[n+2],A[n+1],A[n]}
   vmovaps xmm4, [rdi+16] //xmm4={A[n+7],A[n+6],A[n+5],A[n+4]}
   vmovaps xmm5, [rdi+32] //xmm5={A[n+11], A[n+10],A[n+9],A[n+8]}
loop_start:
   vmovaps xmm4, [rdi+16] //xmm4={A[n+7],A[n+6],A[n+5],A[n+4]}
   vmovaps xmm5, [rdi+32] //xmm5={A[n+11], A[n+10],A[n+9],A[n+8]}
   vinsertf128 ymm3, ymm3, xmm4, 1 // ymm3={A[n+7],A[n+6],A[n+5],A[n+4];
                                   // A[n+3], A[n+2],A[n+1],A[n]}
   vpalignr xmm6, xmm4, xmm3, 4 // xmm6={A[n+4],A[n+3],A[n+2],A[n+1]}
   vpalignr xmm7, xmm5, xmm4, 4 // xmm7={A[n+8],A[n+7],A[n+6],A[n+5]}
   vinsertf128 ymm6,ymm6,xmm7,1 // ymm6={A[n+8],A[n+7],A[n+6],A[n+5];
                                // A[n+4],A[n+3],A[n+2],A[n+1]}
   vpalignr xmm8,xmm4,xmm3,8 // xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
   vpalignr xmm9, xmm5, xmm4, 8 // xmm9={A[n+9],A[n+8],A[n+7],A[n+6]}
   vinsertf128 ymm8, ymm8, xmm9,1 // ymm8={A[n+9],A[n+8],A[n+7],A[n+6];
                                  // A[n+5],A[n+4],A[n+3],A[n+2]}
   vmulps ymm3,ymm3,ymm2 // Ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5], C0*A[n+4];
                         // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
   vmulps ymm6,ymm6,ymm1 // Ymm6={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                          // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
   vmulps ymm8,ymm8,ymm0 // Ymm8={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                         // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
   vaddps ymm3 ,ymm3,ymm6
   vaddps ymm3, ymm3, ymm8
   vmovaps [rsi], ymm3
   vmovaps xmm3, xmm5
   add rdi, 32   //inPtr+=32
   add rbx, 8    //ループカウンター
   add rsi, 32 //outPtr+=32
   cmp rbx, rcx
   jl loop_start
```

例 15-17 では、256 ビットの VSHUFPS で 128 ビットのインテル® SSE が混在したコード内の PALIGNR を置換しています。これにより、例 15-16 の 128 ビットのインテル® SSE が混在したコードよりもおよそ 70% 速度が向上し、例 15-18 を多少上回ります。

整数命令を含み、256 ビットのインテル® AVX 命令で記述されたコードでは、整数命令を、同等の機能とパフォーマンスを持つ浮動小数点命令に置き換えます。同等の浮動小数点命令がない場合は、128 ビットのインテル® AVX 命令を使用して、必要な整数操作を行うことを検討してください。

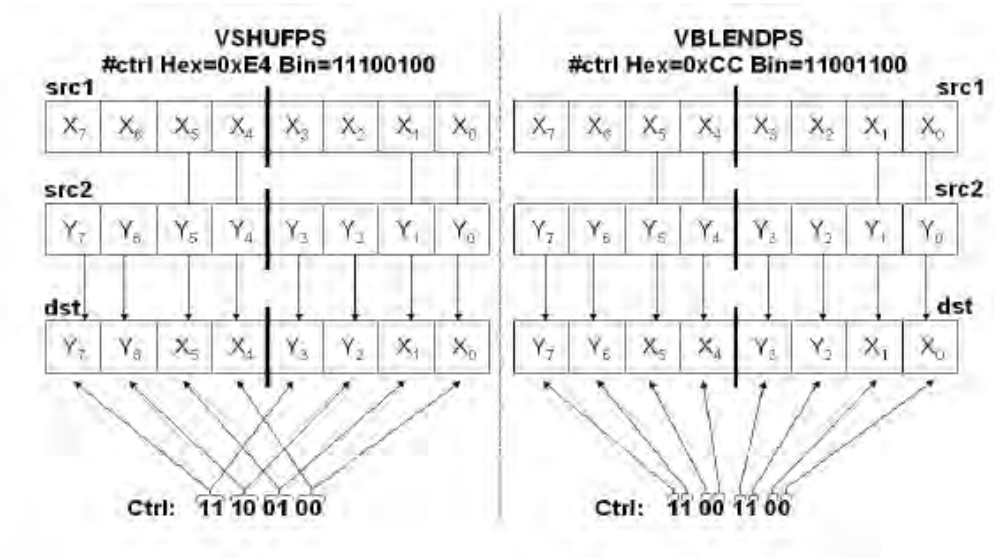
15.11 ポート 5 への負荷の考慮

Sandy Bridge† マイクロアーキテクチャーのポート 5 にはシャッフルユニットが含まれますが、これがしばしばパフォーマンスのボトルネックになることがあります。Ice Lake† Client マイクロアーキテクチャーでは、ポート 1 に限定された機能を持つレーン内シャッフルユニットが追加され、負荷の一部を軽減できるようになりました。レーン内で

動作する再構成可能なシャッフル操作は、このユニットの恩恵を得られます。ポート 5 にのみディスパッチされるシャッフル命令を別の命令に置き換えて、ポート 5 の負荷を軽減することでパフォーマンスを向上できる場合があります。詳細は、表 E-11 を参照してください。

15.11.1 シャッフルをブレンドに置き換える

VSHUFPS や VPERM2F128 などのシャッフルをブレンド命令に置き換えられる場合があります。インテル® AVX のシャッフルはポート 5 でしか実行できないのに対し、ブレンド命令はポート 0 でも実行できます。そのため、シャッフルをブレンド命令に置き換えるとポート 5 の負荷を軽減できます。以下の図に、VBLENDPS を使用した VSHUFPS 実装の置き換え方法を示します。



次の例は、8x8 行列転置の 2 つの実装を示しています。どちらのケースでも、ボトルネックはポート 5 への負荷です。リスト 1 は、ポート 5 でしか実行できない 12 個の vshufps 命令を使用しています。リスト 2 では、この vshufps 命令を、ポート 0 でも実行できる vblendps 命令に置き換えています。

例 15-19 8x8 行列転置 - シャッフルをブレンドに置き換え

1) VSHUFPS を使用する 256 ビットのインテル® AVX	2) VSHUFPS を VBLENDPS で置き換えたインテル® AVX
movrcx, inpBuf movrdx, outBuf movr10, NumOfLoops loop1: vmovaps ymm9, [rcx] vmovaps ymm10, [rcx+32] vmovaps ymm11, [rcx+64] vmovaps ymm12, [rcx+96] vmovaps ymm13, [rcx+128] vmovaps ymm14, [rcx+160] vmovaps ymm15, [rcx+192] vmovaps ymm2, [rcx+224] vunpcklps ymm6, ymm9, ymm10 vunpcklps ymm1, ymm11, ymm12 vunpckhps ymm8, ymm9, ymm10 vunpcklps ymm0, ymm13, ymm14 vunpcklps ymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vshufps ymm10, ymm6, ymm3, 0xE4 vshufps ymm6, ymm0, ymm9, 0x4E vunpckhps ymm7, ymm11, ymm12 vshufps ymm11, ymm0, ymm6, 0xE4 vshufps ymm12, ymm3, ymm1, 0xE4 vperm2f128 ymm3, ymm10, ymm11, 0x20 vmovaps [rdx], ymm3 vunpckhps ymm5, ymm13, ymm14 vshufps ymm13, ymm6, ymm9, 0xE4 vunpckhps ymm4, ymm15, ymm2 vperm2f128 ymm2, ymm12, ymm13, 0x20 vmovaps 32[rdx], ymm2 vshufps ymm14, ymm8, ymm7, 0x4E vshufps ymm15, ymm14, ymm7, 0xE4 vshufps ymm7, ymm5, ymm4, 0x4E vshufps ymm8, ymm8, ymm14, 0xE4 vshufps ymm5, ymm5, ymm7, 0xE4 vperm2f128 ymm6, ymm8, ymm5, 0x20 vmovaps 64[rdx], ymm6 vshufps ymm4, ymm7, ymm4, 0xE4 vperm2f128 ymm7, ymm15, ymm4, 0x20 vmovaps 96[rdx], ymm7 vperm2f128 ymm1, ymm10, ymm11, 0x31 vperm2f128 ymm0, ymm12, ymm13, 0x31 vmovaps 128[rdx], ymm1 vperm2f128 ymm5, ymm8, ymm5, 0x31 vperm2f128 ymm4, ymm15, ymm4, 0x31 vmovaps 160[rdx], ymm0 vmovaps 192[rdx], ymm5 vmovaps 224[rdx], ymm4 dec r10 jnz loop1	movrcx, inpBuf movrdx, outBuf movr10, NumOfLoops loop1: vmovaps ymm9, [rcx] vmovaps ymm10, [rcx+32] vmovaps ymm11, [rcx+64] vmovaps ymm12, [rcx+96] vmovaps ymm13, [rcx+128] vmovaps ymm14, [rcx+160] vmovaps ymm15, [rcx+192] vmovaps ymm2, [rcx+224] vunpcklps ymm6, ymm9, ymm10 vunpcklps ymm1, ymm11, ymm12 vunpckhps ymm8, ymm9, ymm10 vunpcklps ymm0, ymm13, ymm14 vunpcklps ymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vblendps ymm10, ymm6, ymm3, 0xCC vshufps ymm6, ymm0, ymm9, 0x4E vunpckhps ymm7, ymm11, ymm12 vblendps ymm11, ymm0, ymm6, 0xCC vblendps ymm12, ymm3, ymm1, 0xCC vperm2f128 ymm3, ymm10, ymm11, 0x20 vmovaps [rdx], ymm3 vunpckhps ymm5, ymm13, ymm14 vblendps ymm13, ymm6, ymm9, 0xCC vunpckhps ymm4, ymm15, ymm2 vperm2f128 ymm2, ymm12, ymm13, 0x20 vmovaps 32[rdx], ymm2 vshufps ymm14, ymm8, ymm7, 0x4E vblendps ymm15, ymm14, ymm7, 0xCC vshufps ymm7, ymm5, ymm4, 0x4E vblendps ymm8, ymm8, ymm14, 0xCC vblendps ymm5, ymm5, ymm7, 0xCC vperm2f128 ymm6, ymm8, ymm5, 0x20 vmovaps 64[rdx], ymm6 vblendps ymm4, ymm7, ymm4, 0xCC vperm2f128 ymm7, ymm15, ymm4, 0x20 vmovaps 96[rdx], ymm7 vperm2f128 ymm1, ymm10, ymm11, 0x31 vperm2f128 ymm0, ymm12, ymm13, 0x31 vmovaps 128[rdx], ymm1 vperm2f128 ymm5, ymm8, ymm5, 0x31 vperm2f128 ymm4, ymm15, ymm4, 0x31 vmovaps 160[rdx], ymm0 vmovaps 192[rdx], ymm5 vmovaps 224[rdx], ymm4 dec r10 jnz loop1

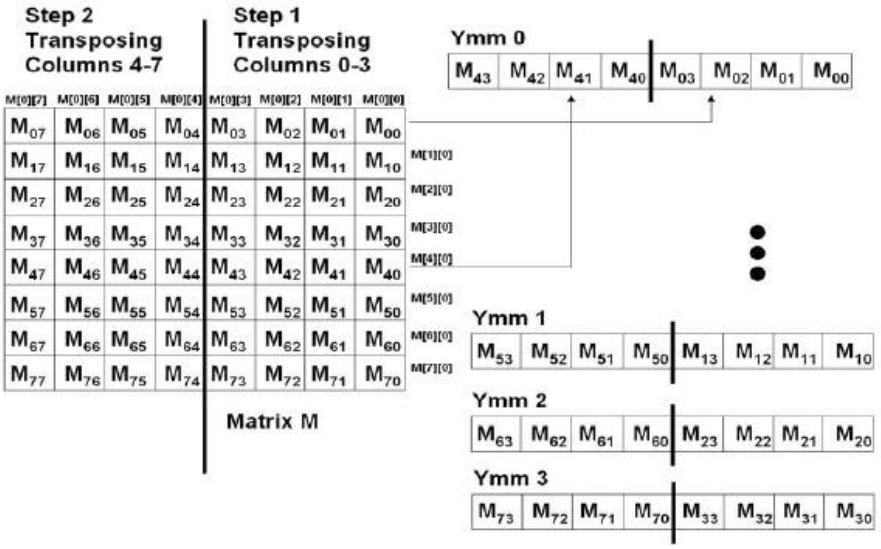
例 15-19 では、VSHUFPS を VBLENDPS に置き換えることでポート 5 の負荷を軽減し、およそ 40% スピードアップを達成しています。

アセンブリー/コンパイラー・コーディング規則 68 (影響 M、一般性 M): インテル® AVX のシャッフル命令の代わりに、可能であればブレンド命令を使用します。

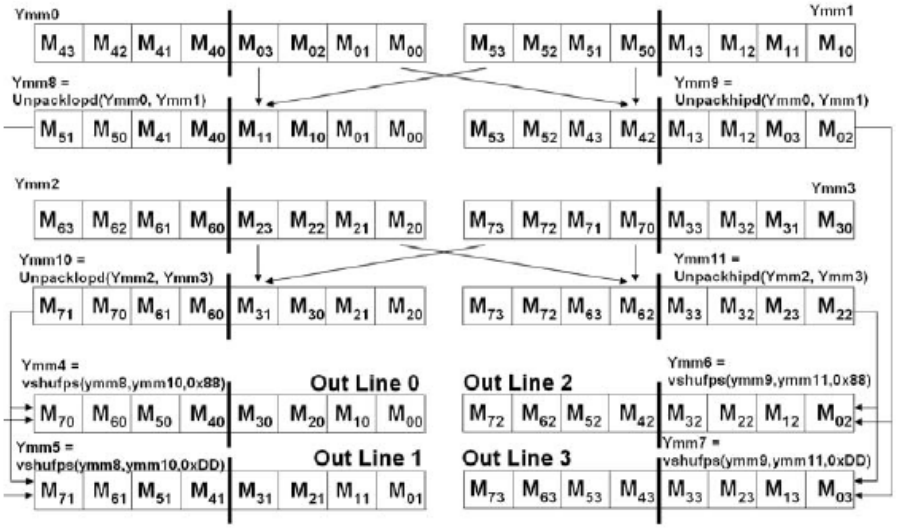
15.11.2 シャッフルの数を減らしたアルゴリズムの設計

アルゴリズムで使用するシャッフルの数を減らすことで、ポート 5 の負荷を軽減できる場合があります。下の図では、転置によって、0 ~ 3 行目の要素すべてが下位レーンに移動し、4 ~ 7 行目の要素すべてが上位レーンに移動しています。そのため、アルゴリズムの最初で 256 ビットのロードを使用するには、レーン間の要素を入れ替える VPERM2F128 を使用する必要があります。プロセッサーは、ポート 5 でのみ VPERM2F128 命令を実行します。

例 15-19 では、8 つの 256 ビット・ロードと 8 つの VPERM2F128 命令を使用しました。この 256 ビット・ロードと 8 つの VPERM2F128 の代わりに、VINSERTF128 を使用して、同じ 8x8 行列転置を実装できます。メモリーからの VINSERTF128 は、ロードポートとポート 0 またはポート 5 で実行されます。当初の方法では、ロードポートで実行されるロードとポート 5 でのみ実行さる VPERM2F128 が必要でした。そのため、VINSERTF128 を使用するようにアルゴリズムを設計し直すことで、ポート 5 の負荷が軽減されパフォーマンスが向上します。



次の図に、vinsertf128 を使用した 8x8 行列転置のステップ 1 を示します。ステップ 2 では、異なる列で同じ操作を行います。



例 15-20 VINSERTPS 使用した 8x8 行列転置

```
   mov rcx, inpBuf
   mov rdx, outBuf

   mov r10, NumOfLoops
loop1:
   vmovaps xmm0, [rcx]
   vinsertf128 ymm0, ymm0, [rcx + 128], 1
   vmovaps xmm1, [rcx + 32]
   vinsertf128 ymm1, ymm1, [rcx + 160], 1

   vunpcklpd ymm8, ymm0, ymm1
   vunpckhpd ymm9, ymm0, ymm1
   vmovaps xmm2, [rcx+64]
   vinsertf128 ymm2, ymm2, [rcx + 192], 1
   vmovaps xmm3, [rcx+96]
   vinsertf128 ymm3, ymm3, [rcx + 224], 1

   vunpcklpd ymm10, ymm2, ymm3
   vunpckhpd ymm11, ymm2, ymm3
   vshufps ymm4, ymm8, ymm10, 0x88
   vmovaps [rdx], ymm4
   vshufps ymm5, ymm8, ymm10, 0xDD
   vmovaps [rdx+32], ymm5
   vshufps ymm6, ymm9, ymm11, 0x88
   vmovaps [rdx+64], ymm6
   vshufps ymm7, ymm9, ymm11, 0xDD
   vmovaps [rdx+96], ymm7
   vmovaps xmm0, [rcx+16]
   vinsertf128 ymm0, ymm0, [rcx + 144], 1
   vmovaps xmm1, [rcx + 48]
   vinsertf128 ymm1, ymm1, [rcx + 176], 1

   vunpcklpd ymm8, ymm0, ymm1
   vunpckhpd ymm9, ymm0, ymm1

   vmovaps xmm2, [rcx+80]
```

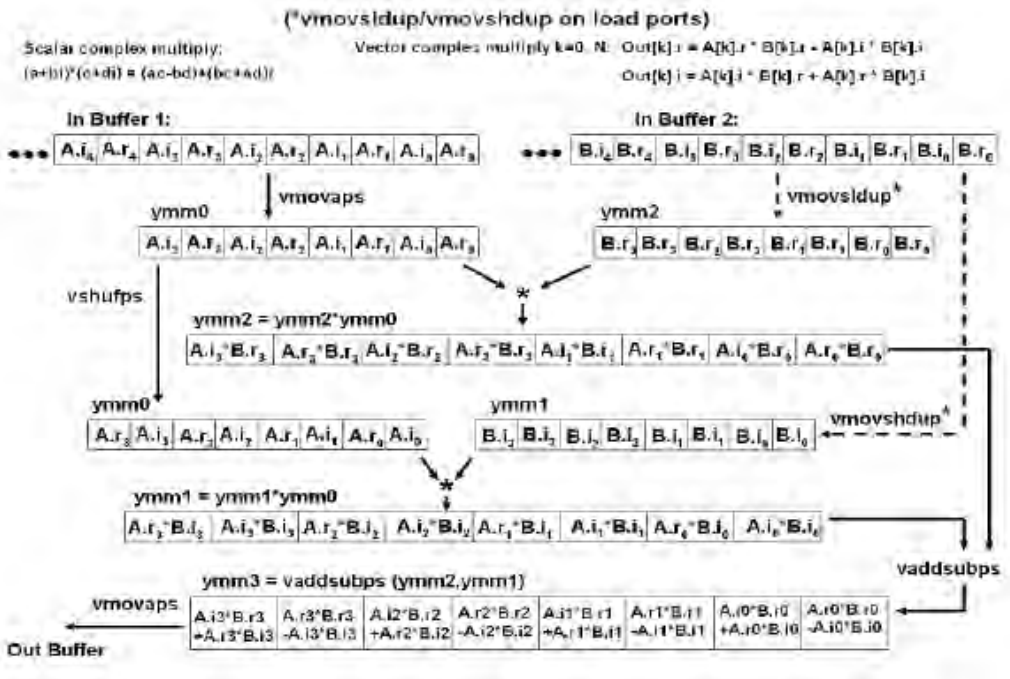
```
vinsertf128 ymm2, ymm2, [rcx + 208], 1
vmovaps xmm3, [rcx+112]
vinsertf128 ymm3, ymm3, [rcx + 240], 1

vunpcklpd ymm10, ymm2, ymm3
vunpckhpd ymm11, ymm2, ymm3

vshufps ymm4, ymm8, ymm10, 0x88
vmovaps [rdx+128], ymm4
vshufps ymm5, ymm8, ymm10, 0xDD
vmovaps [rdx+160], ymm5
vshufps ymm6, ymm9, ymm11, 0x88
vmovaps [rdx+192], ymm6
vshufps ymm7, ymm9, ymm11, 0xDD
vmovaps [rdx+224], ymm7
dec r10
jnz loop1
```

例 15-20 では、例 15-19 の VSHUFPS と VBLENDPS の組み合わせよりも、さらにポート 5 の負荷が軽減されます。例 15-19 の VSHUFPS だけを利用した場合と比べ、70% のスピードアップを達成できます。

15.11.3 ロードポートで基本的なシャッフルを実行

ソースがメモリーである場合、一部のシャッフルはロードポート (ポート 2、ポート 3) で実行できます。以下に、一部のシャッフル (vmovsldup/vmovshdup) をポート 5 からロードポートに移動することで、いかに大幅なパフォーマンス向上が得られるかを示します。

次の図は、vmovsldup/vmovshdup がロードポートにある複素数乗算アルゴリズムのインテル® AVX の実装を示しています。



例 15-21 には、2 つのバージョンの複素数乗算が含まれます。どちらのバージョンも 2 回アンロールされています。リスト 1 は、レジスター内のすべてのデータをシャッフルしています。リスト 2 は、メモリーからデータをロードしながらシャッフルします。

例 15-21 ポート 5 に対するロード・ポート・シャッフル

1) レジスター内でデータをシャッフル

```
   mov rax, inPtr1
   mov rbx, inPtr2
   mov rdx, outPtr
   mov r8, len
   xor rcx, rcx

loop1:
   vmovaps ymm0, [rax +8*rcx]
   vmovaps ymm4, [rax +8*rcx +32]
   vmovaps ymm3, [rbx +8*rcx]
   vmovsldup ymm2, ymm3
   vmulps ymm2, ymm2, ymm0
   vshufps ymm0, ymm0, ymm0, 177
   vmovshdup ymm1, ymm3
   vmulps ymm1, ymm1, ymm0
   vmovaps ymm7, [rbx +8*rcx +32]
   vmovsldup ymm6, ymm7
   vmulps ymm6, ymm6, ymm4
   vaddsubps ymm2, ymm2, ymm1
   vmovshdup ymm5, ymm7
   vmovaps [rdx+8*rcx], ymm2
   vshufps ymm4, ymm4, ymm4, 177
   vmulps ymm5, ymm5, ymm4
   vaddsubps ymm6, ymm6, ymm5
   vmovaps [rdx+8*rcx+32], ymm6

   add rcx, 8
   cmp rcx, r8
   jl loop1
```

2) ロードされたデータをシャッフル

```
  mov rax, inPtr1
  mov rbx, inPtr2
  mov rdx, outPtr
  mov r8, len
  xor rcx, rcx

loop1:
  vmovaps ymm0, [rax +8*rcx]
  vmovaps ymm4, [rax +8*rcx +32]

  vmovsldup ymm2, [rbx +8*rcx]
  vmulps ymm2, ymm2, ymm0
  vshufps ymm0, ymm0, ymm0, 177
  vmovshdup ymm1, [rbx +8*rcx]
  vmulps ymm1, ymm1, ymm0
  vmovsldup ymm6, [rbx +8*rcx +32]
  vmulps ymm6, ymm6, ymm4
  vaddsubps ymm3, ymm2, ymm1
  vmovshdup ymm5, [rbx +8*rcx +32]

  vmovaps [rdx +8*rcx], ymm3
  vshufps ymm4, ymm4, ymm4, 177
  vmulps ymm5, ymm5, ymm4
  vaddsubps ymm7, ymm6, ymm5
  vmovaps [rdx +8*rcx +32], ymm7

  add rcx, 8
  cmp rcx, r8
  jl loop1
```

15.12 除算と平方根命令

Skylake† マイクロアーキテクチャーより前は、インテル® SSE の除算命令 DIVPS および平方根命令 SQRTPS はレイテンシーが 14 サイクルで、それらはパイプライン構造になっていません。つまり、これらの命令のスループットは 14 サイクルにつき 1 回になります。128 ビットのデータパスで実行される 256 ビットのインテル® AVX 命令の VDIVPS および VSQRTPS は、レイテンシーが 28 サイクルで、同じくパイプライン構造になっていません。そのため、Sandy Bridge† マイクロアーキテクチャーでは、インテル® SSE の除算命令と平方根命令のパフォーマンスは、インテル® AVX の 256 ビット命令と同程度です。

Skylake† マイクロアーキテクチャーでは、256 ビットと 128 ビットの (V)DIVPS/(V)SQRTPS は、256 ビットのデータパスで実行されるため同じレイテンシーです。レイテンシーは改善され、パイプラインで実行できるようになったことでスループットも大幅に向上しました。付録 D「命令レイテンシーとスループット」を参照してください。

高いレイテンシーと低いスループットの DIVPS/SQRTPS を提供するマイクロアーキテクチャーでは、(V)RSQRTPS 命令および(V)RCPPS 命令を使用して単精度の除算計算および平方根計算をスピードアップできます。例えば、128 ビットの RCPPS/RSQRTPS は 5 サイクルのレイテンシーで 1 サイクルのスループット、また 256 ビットの同命令では 7 サイクルのレイテンシーと 2 サイクルのスループットです。1 回のニュートンラフソン反復法

またはテイラー近似式は、(V)DIVPS 命令および(V)SQRTPS 命令とほぼ同じ精度を実現できます。これらの命令の詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』を参照してください。

除算演算や平方根演算が、これらの演算のレイテンシーの一部を隠蔽する大きなアルゴリズムに含まれる場合、ニュートンラフソンと近似法によって実行速度が低下することがあります。これは、命令が増加したことによって、マイクロオペレーション(uop)が増え、パイプが一杯になるためです。

Skylake† マイクロアーキテクチャーでは、簡単な代数計算の最適なパフォーマンスのため近似逆数命令に対する DIVPS/SQRTPS の選択は、いくつかの要因に依存します。表 15-5 に、いくつかの代数式の異なる数値精度交差の実装におけるスループットの比較を示しています。各行の 24 ビット精度実装は IEEE に準拠しており、128 ビットまたは 256 ビット ISA をそれぞれ使用しています。22 ビットと 11 ビットの精度実装の列は、それぞれの命令セットの逆数近似命令を使用しています。

表 15-5 Skylake† マイクロアーキテクチャーにおける線形代数の数値代替手段の比較

アルゴリズム	命令タイプ	24 ビット精度	22 ビット精度	11 ビット精度
Z = X/Y	SSE	1X	0.9X	1.3X
	256 ビット AVX	1X	1.5X	2.6X
$Z = X^{0.5}$	SSE	1X	0.7X	2X
	256 ビット AVX	1X	1.4X	3.4X
$Z = X^{-0.5}$	SSE	1X	1.7X	4.3X
	256 ビット AVX	1X	3X	7.7X
$Z = (X*Y + Y*Y)^{0.5}$	SSE	1X	0.75X	0.85X
	256 ビット AVX	1X	1.1X	1.6X
Z = (X+2Y+3)/(Z-2Y-3)	SSE	1X	0.85X	1X
	256 ビット AVX	1X	0.8X	1X

ターゲット・プロセッサーが Skylake† マイクロアーキテクチャーをベースとしている場合、表 15-5 のように要約できます。

- アルゴリズムが除算ユニットで実行される操作のみを含む場合、Skylake† マイクロアーキテクチャー上の 256 ビットのインテル® AVX コードではニュートンラフソン近似が有益です。しかし、単精度除算や平方根操作が長い計算の一部として含まれる場合、低レイテンシーの DIVPS や SQRTPS 命令は全体のパフォーマンス改善に役立ちます。
- インテル® SSE や 128 ビットのインテル® AVX 実装では、11 ビットより高い精度を必要としないアルゴリズムや、除算ユニットで実行される複数の処理を含むアルゴリズム向けにのみ、近似の除算と平方根命令を使用することを検討してください。

表 15-6 に、最近の世代のインテル® マイクロアーキテクチャーで必要とされる精度レベルで、単精度命令を使用する演算精度除算や平方根の推奨される計算方法を示します。

表 15-6 単精度除算と平方根の代替

演算	許容される精度	推奨
除算	24 ビット (IEEE)	DIVPS
	~ 22 ビット	Skylake†: 表 15-5 を参照。 以前の uarch: RCPPS + 1 ニュートンラフソン反復 + MULPS
	~ 11 ビット	RCPPS + MULPS
逆数平方根	24 ビット (IEEE)	SQRTPS + DIVPS
	~ 22 ビット	RSQRTPS + 1 ニュートンラフソン反復
	~ 11 ビット	RSQRTPS
平方根	24 ビット (IEEE)	SQRTPS
	~ 22 ビット	Skylake†: 表 15-5 を参照。 以前の uarch: RSQRTPS + 1 ニュートンラフソン反復 + MULPS
	~ 11 ビット	RSQRTPS + RCPPS

15.12.1 単精度除算

次の式を計算:

Z[i]=A[i]/B[i]

単精度数の大きなベクトルでは、Z[i] は除算演算により、または 1/B[i] に A[i] を掛けることで算出できます。

B[i] を N で表すと、(V)RCPPS 命令を使用して 1/N を計算でき、ほぼ 11 ビットの精度を実現できます。

精度を向上させるには、1 回のニュートンラフソン反復法を使用します。

X_(0) ~= 1/N ; 初期推測、rcp(N)
X_(0) = 1/N*(1-E)
E=1-N*X_0 ; E ~= 2^(-11)
X_1=X_0*(1+E)=1/N*(1-E^2) ; E^2 ~= 2^(-22)
X_1=X_0*(1+1-N*X_0)= 2 *X_0 - N*X_0^2

X_1 は、ほぼ 22 ビット精度の 1/N の近似値です。

例 15-22 24 ビット精度で DIVPS を使用した除算

DIVPS を使用したインテル® SSE コード	VDIVPS を使用

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
 	
loop1:
movups xmm0, [rax+rdx*1]
movups xmm1, [rbx+rdx*1]
divps xmm0, xmm1
movups [rcx+rdx*1], xmm0
add rdx, 16
cmp rdx, rsi
jl loop1
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx
	
	loop1:
	vmovups ymm0, [rax+rdx*1]
	vmovups ymm1, [rbx+rdx*1]
	vdivps ymm0, ymm0, ymm1
	vmovups [rcx+rdx*1], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

例 15-23 11 ビット近似で RCPPS を使用した除算

RCPPS を使用したインテル® SSE コード	VRCPPS を使用

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
 	
loop1:
movups xmm0,[rax+rdx*1]
movups xmm1,[rbx+rdx*1]
rcpps xmm1,xmm1
mulps xmm0,xmm1
movups [rcx+rdx*1],xmm0
add rdx, 16
cmp rdx, rsi
jl loop1
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx
	
	loop1:
	vmovups ymm0, [rax+rdx]
	vmovups ymm1, [rbx+rdx]
	vrcpps ymm1, ymm1
	vmulps ymm0, ymm0, ymm1
	vmovups [rcx+rdx], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

例 15-24 RCPPS とニュートンラフソン反復を使用した除算

RCPPS + MULPS ~ 22 ビット精度	VRCPPS + VMULPS ~ 22 ビット精度

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
 	
loop1:
movups xmm0, [rax+rdx*1]
movups xmm1, [rbx+rdx*1]
rcpps xmm3, xmm1
movaps xmm2, xmm3
addps xmm3, xmm2
mulps xmm2, xmm2
mulps xmm2, xmm1
subps xmm3, xmm2
mulps xmm0, xmm3
movups [rcx+rdx*1], xmm0
add rdx, 16
cmp rdx, rsi
jl loop1
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx
	
	loop1:
	vmovups ymm0, [rax+rdx]
	vmovups ymm1, [rbx+rdx]
	vrcpps ymm3, ymm1
	
	vaddps ymm2, ymm3, ymm3
	vmulps ymm3, ymm3, ymm3
	vmulps ymm3, ymm3, ymm1
	vsubps ymm2, ymm2, ymm3
	vmulps ymm0, ymm0, ymm2
	vmovups [rcx+rdx], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

15.12.2 単精度逆数平方根

単精度数の大きなベクトルで Z[i]=1/ (A[i]) ^0.5 を計算するには、A[i] を N で表すと、(V)RSQRTPS 命令を使用して 1/N を計算できます。

精度を向上させるには、1 回のニュートンラフソン反復法を使用します。

X_0 ~=1/N ; 初期推測、RCP(N)
E=1-N*X_0^2
X_0= (1/N)^0.5 * ((1-E)^0.5) = (1/N)^0.5 * (1-E/2) ; E/2~= 2^(-11)
X_1=X_0*(1+E/2) ~= (1/N)^0.5 * (1-E^2/4) ; E^2/4?2^(-22)
X_1=X_0*(1+1/2-1/2*N*X_0^2)= 1/2*X_0*(3-N*X_0^2)

X1 は、ほぼ 22 ビット精度の (1/N)^0.5 の近似値です。

例 15-25 24 ビット精度で DIVPS + SQRTPS を使用した逆数平方根

SQRTPS、DIVPS を使用	VSQRTPS、VDIVPS を使用

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
sqrtps xmm0, xmm1
divps xmm0, xmm1
movups [rbx+rdx], xmm0
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	loop1:
	vmovups ymm1, [rax+rdx]
	vsqrtps ymm0,ymm1
	vdivps ymm0, ymm0, ymm1
	vmovups [rbx+rdx], ymm0
	add rdx, 32
	cmp rdx, rcx
	jl loop1

例 15-26 11 ビット近似で RSQRTPS を使用した逆数平方根

RSQRTPS を使用したインテル® SSE コード	VRSQRTPS を使用

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
rsqrtps xmm0, [rax+rdx]
movups [rbx+rdx], xmm0
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	loop1:
	vrsqrtps ymm0, [rax+rdx]
	vmovups [rbx+rdx], ymm0
	add rdx, 32
	cmp rdx, rcx
	jl loop1

例 15-27 RSQRTPS とニュートンラフソン反復を使用した逆数平方根

RSQRTPS + MULPS ~ 22 ビット精度	VRSQRTPS + VMULPS ~ 22 ビット精度

`__declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5};`
`__declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};`
`__asm`
`{`
`  mov rax, pIn`
`  mov rbx, pOut`
`  mov rcx, iLen`
`  xor rdx, rdx`
`  movups xmm3, [three]`
`  movups xmm4, [minus_half]`
`loop1:`
`  movups xmm5, [rax+rdx]`
`  rsqrtps xmm0, xmm5`
`  movaps xmm2, xmm0`
`  mulps xmm0, xmm0`
`  mulps xmm0, xmm5`
`  subps xmm0, xmm3`
`  mulps xmm0, xmm2`
`  mulps xmm0, xmm4`
`  movups [rbx+rdx], xmm0`
`  add rdx, 16`
`  cmp rdx, rcx`
`  jl loop1`
`}`
	`__declspec(align(32)) float half[8] = {0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5};`
	`__declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};`
	`__asm`
	`{`
	`  mov rax, pIn`
	`  mov rbx, pOut`
	`  mov rcx, iLen`
	`  xor rdx, rdx`
	`  vmovups ymm3, [three]`
	`  vmovups ymm4, [half]`
	`loop1:`
	`  vmovups ymm5, [rax+rdx]`
	`  vrsqrtps ymm0, ymm5`
	`  vmulps ymm2, ymm0, ymm0`
	`  vmulps ymm2, ymm2, ymm5`
	`  vsubps ymm2, ymm3, ymm2`
	`  vmulps ymm0, ymm0, ymm2`
	`  vmulps ymm0, ymm0, ymm4`
	
	`  vmovups [rbx+rdx], ymm0`
	`  add rdx, 32`
	`  cmp rdx, rcx`
	`  jl loop1`
	`}`

15.12.3 単精度平方根

単精度数の大きなベクトルで Z[i]= (A[i])^0.5 を計算するには、A[i] を N で表すと、N^0.5 の近似値は N に (1/N) ^0.5 を掛けた値となります ((1/N)^0.5 の近似値については、直前の節で説明しました)。

N^0.5 でほぼ 22 ビット精度を実現するには、以下の計算式を使用します。

N^0.5 = X_1*N = 1/2*N*X_0*(3-N*X_0^2)

例 15-28 24 ビット精度で SQRTPS を使用した平方根

SQRTPS を使用	VSQRTPS を使用

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
sqrtps xmm1, xmm1
movups [rbx+rdx], xmm1
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	loop1:
	vmovups ymm1, [rax+rdx]
	vsqrtps ymm1,ymm1
	vmovups [rbx+rdx], ymm1
	add rdx, 32
	cmp rdx, rcx
	jl loop1

例 15-29 11 ビット近似で RSQRTPS を使用した平方根

RSQRTPS を使用したインテル® SSE コード	VRSQRTPS を使用

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
xorps xmm8, xmm8
cmpneqps xmm8, xmm1
rsqrtps xmm1, xmm1
rcpps xmm1, xmm1
andps xmm1, xmm8
movups [rbx+rdx], xmm1
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	vxorps ymm8, ymm8, ymm8
	loop1:
	vmovups ymm1, [rax+rdx]
	vcmpneqps ymm9, ymm8, ymm1
	vrsqrtps ymm1, ymm1
	vrcpps ymm1, ymm1
	vandps ymm1, ymm1, ymm9
	vmovups [rbx+rdx], ymm1
	add rdx, 32
	cmp rdx, rcx
	jl loop1

例 15-30 RSQRTPS と1 つのテイラーシリーズ展開を使用した平方根

RSQRTPS + Taylor ~ 22 ビット精度	VRSQRTPS + Taylor ~ 22 ビット精度

`__declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5};`
`__declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};`
 	
`__asm`
`{`
`  mov rax, pIn`
`  mov rbx, pOut`
`  mov rcx, iLen`
`  xor rdx, rdx`
`  movups xmm6, [three]`
`  movups xmm7, [minus_half]`
 	
`loop1:`
`  movups xmm3, [rax+rdx]`
`  rsqrtps xmm1, xmm3`
`  xorps xmm8, xmm8`
`  cmpneqps xmm8, xmm3`
`  andps xmm1, xmm8`
`  movaps xmm4, xmm1`
`  mulps xmm1, xmm3`
`  movaps xmm5, xmm1`
`  mulps xmm1, xmm4`
`  subps xmm1, xmm6`
`  mulps xmm1, xmm5`
`  mulps xmm1, xmm7`
`  movups [rbx+rdx], xmm1`
`  add rdx, 16`
`  cmp rdx, rcx`
`  jl loop1`
`}`
	`__declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};`
	`__declspec(align(32)) float minus_half[8] = {-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5};`
	
	`__asm`
	`{`
	`  mov rax, pIn`
	`  mov rbx, pOut`
	`  mov rcx, iLen`
	`  xor rdx, rdx`
	`  vmovups ymm6, [three]`
	`  vmovups ymm7, [minus_half]`
	`  vxorps ymm8, ymm8, ymm8`
	`loop1:`
	`  vmovups ymm3, [rax+rdx]`
	`  vrsqrtps ymm4, ymm3`
	`  vcmpneqps ymm9, ymm8, ymm3`
	`  vandps ymm4, ymm4, ymm9`
	`  vmulps ymm1,ymm4, ymm3`
	`  vmulps ymm2, ymm1, ymm4`
	`  vsubps ymm2, ymm2, ymm6`
	`  vmulps ymm1, ymm1, ymm2`
	`  vmulps ymm1, ymm1, ymm7`
	`  vmovups [rbx+rdx], ymm1`
	`  add rdx, 32`
	`  cmp rdx, rcx`
	`  jl loop1`
	`}`

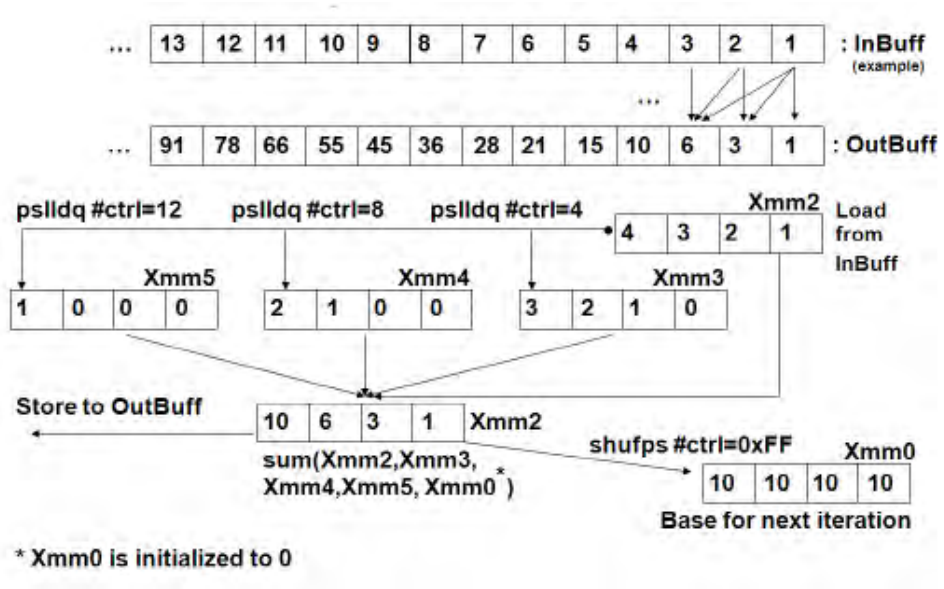
15.13 ARRAY SUB SUM の最適化例

この節では、インテル® SSE 実装の Array Sub Sum アルゴリズムをインテル® AVX 実装に変換する方法を示します。

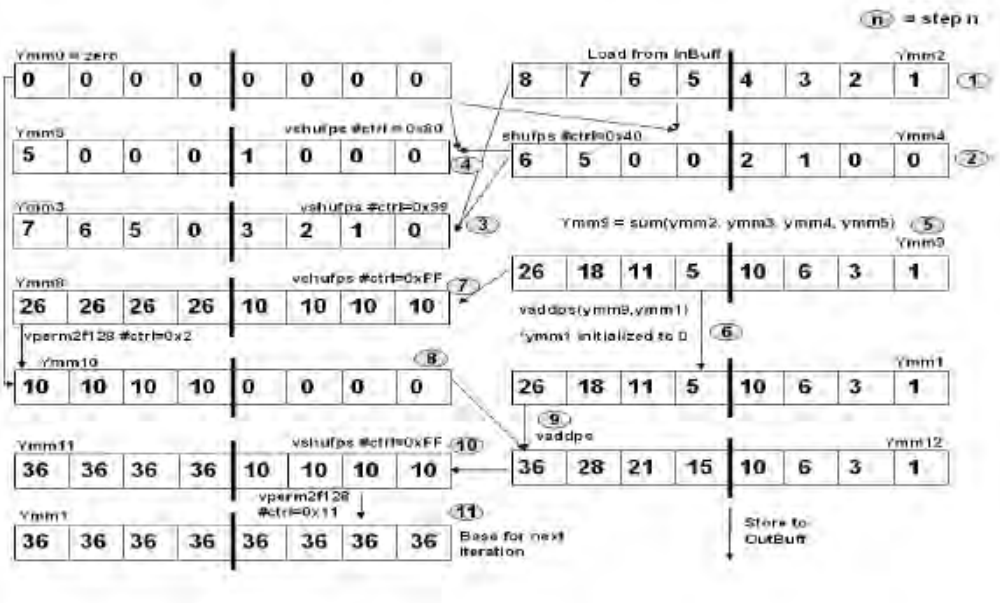
Array Sub Sum アルゴリズムは以下のとおりです。

$Y_{[i]}$ = Sum of k from 0 to i ($X_{[k]}$) = $X_{[0]}$ + $X_{[1]}$ + .. + $X_{[i]}$

以下の図はインテル® SSE 実装を示します。



以下に、Array Sub Sums アルゴリズムのインテル® AVX 実装を示します。PSLLDQ は、インテル® AVX が相当する命令を持たない整数 SIMD 命令です。これは VSHUFPS で置き換えることができます。



例 15-31 Array Sub Sum アルゴリズム

インテル® SSE コード	インテル® AVX コード

mov rax, InBuff
mov rbx, OutBuff
mov rdx, len
xor rcx, rcx
xorps xmm0, xmm0
 	
loop1:
movaps xmm2, [rax+4*rcx]
movaps xmm3, xmm2
movaps xmm4, xmm2
movaps xmm5, xmm2
pslldq xmm3, 4
pslldq xmm4, 8
pslldq xmm5, 12
addps xmm2, xmm3
addps xmm4, xmm5
addps xmm2, xmm4
addps xmm2, xmm0
movaps xmm0, xmm2
shufps xmm0, xmm2, 0xFF
movaps [rbx+4*rcx], xmm2
add rcx, 4
cmp rcx, rdx
jl loop1
	mov rax, InBuff
	mov rbx, OutBuff
	mov rdx, len
	xor rcx, rcx
	vxorps ymm0, ymm0, ymm0
	vxorps ymm1, ymm1, ymm1
	loop1:
	vmovaps ymm2, [rax+4*rcx]
	vshufps ymm4, ymm0, ymm2, 0x40
	vshufps ymm3, ymm4, ymm2, 0x99
	vshufps ymm5, ymm0, ymm4, 0x80
	vaddps ymm6, ymm2, ymm3
	vaddps ymm7, ymm4, ymm5
	vaddps ymm9, ymm6, ymm7
	vaddps ymm1, ymm9, ymm1
	vshufps ymm8, ymm9, ymm9, 0xff
	vperm2f128 ymm10, ymm8, ymm0, 0x2
	vaddps ymm12, ymm1, ymm10
	vshufps ymm11, ymm12, ymm12, 0xff
	vperm2f128 ymm1, ymm11, ymm11, 0x11
	vmovaps [rbx+4*rcx], ymm12
	add rcx, 8
	cmp rcx, rdx
	jl loop1

例 15-31 に Array Sub Sum のインテル® SSE 実装とインテル® AVX 実装を示します。インテル® AVX コードの方が、およそ 40% 高速です。

15.14 半精度浮動小数点変換

16 ビット浮動小数点形式の数値範囲と精度のみを必要とする浮動小数点アプリケーションでは、16 ビットでエンコードされた浮動小数点データを保存することでメモリー容量と帯域幅の軽減に大きな利点がもたらされます。これは、グラフィックスやイメージ処理でよく見かけられます。

半精度浮動小数点数のエンコード形式は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 4 章で確認できます。

パックド半精度浮動小数点数とパックド単精度浮動小数点数間の変換命令については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 14 章「Programming with AVX, FMA and AVX2」と、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2B』のリファレンス・ページを参照してください。

半精度の浮動小数点データ (16 ビット FP データ要素) を計算するには、最初に単精度浮動小数点形式に変換して、必要に応じて単精度の結果を半精度形式に変換して書き戻す必要があります。256 ビットの命令を使用する 8 データ要素の変換は非常に高速で、デノーマル数、無限大、ゼロおよび NaN を適切に処理することができます。

15.14.1 パックド単精度から半精度への変換

単精度浮動小数点形式から半精度形式への変換には VCVTPS2PH などのハードウェアのサポートを必要とせず、プログラマーは次のことを行う必要があります。

- 各データ要素の範囲を許容するように指数部のバイアスを調整します。
- 各データ要素の仮数部をシフトして丸めます。
- 各要素の 15 ビット目を符号ビットへコピーします。
- 半精度の範囲外の数値に注意してください。
- 各データ要素を半分のサイズのレジスターへパックします。

例 15-32 では、単精度から半精度への 2 つの浮動小数点変換の実装を比較しています。左のコードは 128 ビット SIMD 命令セットのみとパックド整数シフト命令を使用しています。右のコードでは 2 回アンロールして VCVTPS2PH 命令を使用しています。

例 15-32 単精度から半精度への変換

インテル® AVX-128 コード	VCVTPS2PH コード

__asm
mov rax, pIn
mov rbx, pOut
mov rcx, bufferSize
add rcx, rax
vmovdqu xmm0,SignMask16
vmovdqu xmm1,ExpBiasFixAndRound
vmovdqu xmm4,SignMaskNot32
vmovdqu xmm5,MaxConvertibleFloat
vmovdqu xmm6,MinFloat
loop:
vmovdqu xmm2, [rax]
vmovdqu xmm3, [rax+16]
vpaddd xmm7, xmm2, xmm1
vpaddd xmm9, xmm3, xmm1
vpand xmm7, xmm7, xmm4
vpand xmm9, xmm9, xmm4
add rax, 32
vminps xmm7, xmm7, xmm5
vminps xmm9, xmm9, xmm5
vpcmpgtd xmm8, xmm7, xmm6
vpcmpgtd xmm10, xmm9, xmm6
vpand xmm7, xmm8, xmm7
vpand xmm9, xmm10, xmm9
vpackssdw xmm2, xmm3, xmm2
vpsrad xmm7, xmm7, 13
vpsrad xmm8, xmm9, 13
vpand xmm2, xmm2, xmm0
vpackssdw xmm3, xmm7, xmm9
vpaddw xmm3, xmm3, xmm2
vmovdqu [rbx], xmm3
add rbx, 16
cmp rax, rcx
jl loop
}
	__asm {
	mov rax, pIn
	mov rbx, pOut
	mov rcx, bufferSize
	add rcx, rax
	loop:
	vmovups ymm0,[rax]
	vmovups ymm1,[rax+32]
	add rax, 64
	vcvtps2ph [rbx],ymm0, roundingCtrl
	vcvtps2ph [rbx+16],ymm1,roundingCtrl
	add rbx, 32
	cmp rax, rcx
	jl loop
	}

VCVTPS2PH を使用するコードは、128 ビットのインテル® AVX シーケンスよりも約 4 倍高速です。これは 8 つのデータ要素を一度でロードできること (256 ビットのインテル® AVX) から可能なことですが、要素ごとの変換の大部分は 256 ビット拡張を持たないパックド整数命令で行われています。VCVTPS2PH は高速なだけでなく、正常な半精度浮動小数点値にエンコードされない特殊なケースを扱うことができます。

15.14.2 パックド半精度から単精度への変換

例 15-33 は、128 ビットのインテル® AVX コードと VCVTPH2PS をともに使用した 2 つの実装を比較しています。

半精度から単精度浮動小数点形式への変換は容易に実装できますが、VCVTPH2PS 命令を使用することで 128 ビットのインテル® AVX コードよりも約 2.5 倍高速に実行できます。

例 15-33 半精度から単精度への変換

128 ビットのインテル® AVX コード

```
__asm {
   mov rax, pIn
   mov rbx, pOut
   mov rcx, bufferSize
   add rcx, rax
   vmovdqu xmm0,SignMask16
   vmovdqu xmm1,ExpBiasFix16
   vmovdqu xmm2,ExpMaskMarker
loop:
   vmovdqu xmm3, [rax]
   add rax, 16
   vpandn xmm4, xmm0, xmm3
   vpand xmm5, xmm3, xmm0
   vpsrlw xmm4, xmm4, 3
   vpaddw xmm6, xmm4, xmm1
   vpcmpgtw xmm7, xmm6, xmm2
   vpand xmm6, xmm6, xmm7
   vpand xmm8, xmm3, xmm7
   vpor xmm6, xmm6, xmm5
   vpsllw xmm8, xmm8, 13
   vpunpcklwd xmm3, xmm8, xmm6
   vpunpckhwd xmm4, xmm8, xmm6
   vmovdqu [rbx], xmm3
   vmovdqu [rbx+16], xmm4
   add rbx, 32
   cmp rax, rcx
   jl loop
}
```

VCVTPS2PH コード

```
__asm {
  mov rax, pIn
  mov rbx, pOut
  mov rcx, bufferSize
  add rcx, rax
loop:
  vcvtph2ps ymm0,[rax]
  vcvtph2ps ymm1,[rax+16]
  add rax, 32
  vmovups [rbx], ymm0
  vmovups [rbx+32], ymm1
  add rbx, 64
  cmp rax, rcx
  jl loop
}
```

15.14.3 帯域幅を維持するため半精度 FP の局所性を考慮

例 15-32 と例 15-33 は、ソフトウェアが半精度と単精度データ間の変換を必要とする場合に FP16C 命令を使用するパフォーマンス上の利点を示します。半精度 FP 形式は、単精度 FP 形式よりコンパクトで帯域幅を消費しませんが、数値計算が必要とする数値範囲、精度、そして変換のオーバーヘッド追加を犠牲にしています。ソフトウェアにとって半精度データを使用することが有益であるかどうかは、ワークロードの局所性に大きく依存します。

この節では、水平方向の中央値のフィルタリング・アルゴリズム “Median3” をベースにした例を使用します。Median3 のアルゴリズムは、ベクトル中の 3 つの連続した要素ごとの中央値を計算します。

Y[i] = Median3(X[i], X[i+1], X[i+2])

ここで、Y は出力ベクトル、X は入力ベクトルです。

例 15-34 は、Median3 アルゴリズムの 2 つの実装を示しています。1 つは変換なしで単精度形式を使用し、一方は半精度形式と変換を使用しています。左側のリスト 1 は、256 ビット・ロード/ストア命令を使用した単精度形式で動作しそれぞれのロード/ストアは 8 つの 32 ビット数を扱います。リスト 2 は、8 つの半精度形式の 16 ビット数値のロード/ストアに 128 ビット・ロード/ストア命令、そして単精度浮動小数点形式間との変換に VCVTPH2PS/VCVTPS2PH 命令を使用します。

例 15-34 半精度と単精度を使用した Median3 のパフォーマンス比較

1) 単精度コードと変換	2) 半精度コードと変換

`xor rbx, rbx`
`mov rcx, len`
`mov rdi, inPtr`
`mov rsi, outPtr`
`vmovaps ymm0, [rdi]`
`loop:`
`add rdi, 32`
`vmovaps ymm6, [rdi]`
`vperm2f128 ymm1, ymm0, ymm6, 0x21`
`vshufps ymm3, ymm0, ymm1, 0x4E`
`vshufps ymm2, ymm0, ymm3, 0x99`
`vminps ymm5, ymm0 ,ymm2`
`vmaxps ymm0, ymm0, ymm2`
`vminps ymm4, ymm0, ymm3`
`vmaxps ymm7, ymm4, ymm5`
`vmovaps ymm0, ymm6`
`vmovaps [rsi], ymm7`
`add rsi, 32`
`add rbx, 8`
`cmp rbx, rcx`
`jl loop`
`}`
	`xor rbx, rbx`
	`mov rcx, len`
	`mov rdi, inPtr`
	`mov rsi, outPtr`
	`vcvtph2ps ymm0, [rdi]`
	`loop:`
	`add rdi,16`
	`vcvtph2ps ymm6, [rdi]`
	`vperm2f128 ymm1, ymm0, ymm6, 0x21`
	`vshufps ymm3, ymm0, ymm1, 0x4E`
	`vshufps ymm2, ymm0, ymm3, 0x99`
	`vminps ymm5, ymm0 ,ymm2`
	`vmaxps ymm0, ymm0, ymm2`
	`vminps ymm4, ymm0, ymm3`
	`vmaxps ymm7, ymm4, ymm5`
	`vmovaps ymm0, ymm6`
	`vcvtps2ph [rsi], ymm7, roundingCtrl`
	`add rsi, 16`
	`add rbx, 8`
	`cmp rbx, rcx`
	`jl loop`
	`}`

メモリー上のワーキングセットの局所性が高い場合、Ivy Bridge† マイクロアーキテクチャー・ベースのプロセッサーで半精度形式を使用すると、変換のオーバーヘッドがあるにもかかわらず単精度形式よりも約 30% 高速になります。局所性が L3 で維持される場合、半精度形式はまだ ~15% 高速です。局所性が L1 で維持されると、L1 データキャッシュのキャッシュ帯域幅により単精度形式を使用した方が高速になります。L1 データキャッシュの帯域幅は上位階層のキャッシュやメモリーよりも高く、変換のオーバーヘッドがパフォーマンス上無視できなくなります。

15.15 乗算-加算融合 (FMA) 命令のガイドライン

FMA 命令は、IEEE-754-2008 浮動小数値に準拠した “a * b + c” のベクトル操作を行います。ここで、”a * b” の乗算操作は無限精度で行われ、加算の最終結果が必要とする精度に丸められて生成されます。FMA の丸めと特殊ケースの処理の詳細については、『Intel® Architecture Instruction Set Extensions Programming Reference』の 2.3 節を参照してください。

FMA 命令は多くの FP 計算の精度を改善しスピードアップします。Haswell† マイクロアーキテクチャーは、実行ユニットのポート 0 とポート 1 および 256 ビット・データパスで FMA 命令を実装します。ドット積、行列乗算、およ

び多項式評価は、FMA、256 ビット・データパス、および独立した 2 つの実行ポートを使用することから利点が得られると期待されます。各プロセッサー・コアからの FMA のピーク・スループットは、サイクルごとに 16 個の単精度または 8 個の倍精度の結果です。

FMA 命令を使用するように設計されたアルゴリズムは、(FMA 命令を使用しない) MULPD/PS と ADDPD/PS のシーケンスが生成する結果が FMA を使用する場合と大きく異なることを考慮しなければなりません。収束を判断する数値計算では、丸めの問題より完了時の意図しない結果を避けるため、中間結果の精度の違いが数値形式で縮約することを考慮する必要があります。

ユーザー/ソース・コーディング規則 33: FMA 命令を使用せずに実行される乗算/加算命令に置き換えると、精度と丸め特性が FMA 命令とは異なります。アルゴリズムが DGEMM のように実行ポートのスループットに制限される場合、FMA はパフォーマンスを向上させます。

FMA がより高いパフォーマンスをもたらさない状況も考えられます。"a * b + c * d" のベクトル操作とデータが同時に利用できる状況を考えてみてください。

3 つの命令シーケンスを考えてみます。

VADDPS (VMULPS (a,b) , VMULPS (c,b)) ;

VMULPS は同じサイクルでディスパッチされて並列に実行されますが、VADDPS (3 サイクル) にはレイテンシーがあります。アンロールすることで VADDPS のレイテンシーは相殺されるかもしれません。

2 つの命令シーケンスを使用する場合を考えます。

VFMADD213PS (c, d, VMULPS (a,b));

FMA のレイテンシー (5 サイクル) は、それぞれのベクトルの結果生成で現れます。

ユーザー/ソース・コーディング規則 34: FP add 命令を FMA 命令で置き換える場合、FP add と FMA 命令のレイテンシーは結果依存です。

15.15.1 FMA と浮動小数点 Add/Mul におけるスループットの最適化

Skylake† マイクロアーキテクチャーには、FMA、ベクトル FP 乗算、および FP ADD 命令をサポートする 2 つの実行パイプがあります。これら 3 つのカテゴリーの命令はすべて 4 サイクルのレイテンシーを持ち、ポート 0 またはポート 1 へディスパッチされてサイクルごとに実行されます。

同一のレイテンシーとパイプ数の配置により、浮動小数点計算が FP 乗算に続く浮動小数点加算操作によって制限されている状況のパフォーマンスをソフトウェアが改善することを可能にします。An = C1 + C2 * An-1 のベクトル操作を考えてみます。

例 15-35 FP Mul/FP Add と FMA

1) FP Mul/FP Add シーケンス	2) FMA シーケンス

mov eax, NumOfIterations
mov rbx, pA
mov rcx, pC1
mov rdx, pC2
vmovups ymm0, ymmword ptr [rbx] // A
vmovups ymm1, ymmword ptr [rcx] // C1
vmovups ymm2, ymmword ptr [rdx] // C2
loop:
vmulps ymm4, ymm0 ,ymm2 // A * C2
vaddps ymm0, ymm1, ymm4
dec eax
jnz loop
vmovups ymmword ptr[rbx], ymm0
// An をストア
	mov eax, NumOfIterations
	mov rbx, pA
	mov rcx, pC1
	mov rdx, pC2
	vmovups ymm0, ymmword ptr [rbx] // A
	vmovups ymm1, ymmword ptr [rcx] // C1
	vmovups ymm2, ymmword ptr [rdx] // C2
	loop:
	vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2
	dec eax
	jnz loop
	vmovups ymmword ptr[rbx], ymm0
	// An をストア

反復あたりのコスト: ~ fp add レイテンシー + fp add レイテンシー	反復あたりのコスト: ~ fma レイテンシー

リスト 1 のコードシーケンスの全体的なスループットは、特定のマイクロアーキテクチャーの FP MUL と FP ADD 命令のレイテンシーの組み合わせによって制限されます。また、リスト 2 のコードシーケンスの全体的なスループットは、対応するマイクロアーキテクチャーの FMA 命令のスループットによって制限されます。

FP ADD 操作のレイテンシーがパフォーマンスに影響する一般的な状況を次の C のコードに示します。

```
for ( int 1 = 0; i < arrLenght; i ++) result += arrToSum[i];
```

例 15-36 にアンロールありとなしの 2 つの実装を示します。

例 15-36 FP Add のレイテンシーを隠匿するアンローリング

1) アンロールなし	2) 8 回アンロール

mov eax, arrLength
mov rbx, arrToSum
vmovups ymm0, ymmword ptr [rbx]
sub eax, 8
loop:
add rbx, 32
vaddps ymm0, ymm0, ymmword ptr [rbx]
sub eax, 8
jnz loop
vextractf128 xmm1, ymm0, 1
vaddps xmm0, xmm0, xmm1
vpermilps xmm1, xmm0, 0xe
vaddps xmm0, xmm0, xmm1
vpermilps xmm1, xmm0, 0x1
vaddss xmm0, xmm0, xmm1
	mov eax, arrLength
	mov rbx, arrToSum
	vmovups ymm0, ymmword ptr [rbx]
	vmovups ymm1, ymmword ptr 32[rbx]
	vmovups ymm2, ymmword ptr 64[rbx]
	vmovups ymm3, ymmword ptr 96[rbx]
	vmovups ymm4, ymmword ptr 128[rbx]
	vmovups ymm5, ymmword ptr 160[rbx]
	vmovups ymm6, ymmword ptr 192[rbx]
	vmovups ymm7, ymmword ptr 224[rbx]
	sub eax, 64
	loop:
	add rbx, 256
	vaddps ymm0, ymm0, ymmword ptr [rbx]
	vaddps ymm1, ymm1, ymmword ptr 32[rbx]
	vaddps ymm2, ymm2, ymmword ptr 64[rbx]
	vaddps ymm3, ymm3, ymmword ptr 96[rbx]
	vaddps ymm4, ymm4, ymmword ptr 128[rbx]
	vaddps ymm5, ymm5, ymmword ptr 160[rbx]
	vaddps ymm6, ymm6, ymmword ptr 192[rbx]
	vaddps ymm7, ymm7, ymmword ptr 224[rbx]
	sub eax, 64
	jnz loop
	vaddps ymm0, ymm0, ymm1
	vaddps ymm2, ymm2, ymm3
	vaddps ymm4, ymm4, ymm5
	vaddps ymm6, ymm6, ymm7
	vaddps ymm0, ymm0, ymm2
	vaddps ymm4, ymm4, ymm6
	vaddps ymm0, ymm0, ymm4

vmovss result, xmm0	vextractf128 xmm1, ymm0, 1 vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0xe vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0x1 vaddss xmm0, xmm0, xmm1 vmovss result, xmm0

(例 15-36 のリスト 1 の) アンロールなしでの 8 つの配列要素ごとの累計のコストは、ワーキングセットが L1 に収まると仮定すると FP ADD 命令のレイテンシーにほぼ比例します。アンロールを効率良く使用するため、アンロールされる操作の数は少なくとも “クリティカルな操作のレイテンシー × パイプの数” にすべきです。アンロールなしバージョンに対する最適化されたアンロールバージョンのパフォーマンス・ゲインは、特定のマイクロアーキテクチャーの “パイプの数 × FP ADD のレイテンシー” で求めることができます。

ユーザー/ソース・コーディング規則 35: FMA、FP ADD またはベクトル MUL 操作の連続した依存関係を持つループでは、アンロールの導入を検討します。アンロール数は、依存関係が連鎖するクリティカルな命令のレイテンシーと命令を実行するパイプの数から求めることができます。

15.15.2 ベクトルシフトによるスループットの最適化

Skylake† マイクロアーキテクチャーでは、ほとんどの一般的なベクトルシフト命令はポート 0 またはポート 1 のいずれかにディスパッチできます。以前の世代では 1 つのポートにのみディスパッチ可能でした。表 2-9 と表 E-2 を参照してください。

FP ADD 操作のレイテンシーがパフォーマンスに影響する一般的な状況を次の C コードに示します。

コード中の a、b、c は整数配列です。

```
for ( int 1 = 0; i < len; i ++) c[i] += 4* a[i] + b[i]/2;
```

例 15-37 FP Mul/FP Add と FMA

FP Mul/FP Add シーケンス	FMA シーケンス

mov eax, NumOfIterations
mov rbx, pA
mov rcx, pC1
mov rdx, pC2
vmovups ymm0, ymmword ptr [rbx] // A
vmovups ymm1, ymmword ptr [rcx] // C1
vmovups ymm2, ymmword ptr [rdx] // C2
loop:
vmulps ymm4, ymm0 ,ymm2 // A * C2
vaddps ymm0, ymm1, ymm4
dec eax
jnz loop
vmovups ymmword ptr[rbx], ymm0
// An をストア
	mov eax, NumOfIterations
	mov rbx, pA
	mov rcx, pC1
	mov rdx, pC2
	vmovups ymm0, ymmword ptr [rbx] // A
	vmovups ymm1, ymmword ptr [rcx] // C1
	vmovups ymm2, ymmword ptr [rdx] // C2
	loop:
	vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2
	dec eax
	jnz loop
	vmovups ymmword ptr[rbx], ymm0
	// An をストア

反復あたりのコスト: ~ fp add レイテンシー + fp add レイテンシー	反復あたりのコスト: ~ fma レイテンシー

15.16 インテル® AVX2 最適化のガイドライン

インテル® AVX2 命令は、128 ビット SIMD 整数命令のほとんどを 256 ビット YMM レジスターで動作するように拡張します。インテル® AVX2 はまた、数値計算を加速する豊富な broadcast/permute/各種シフト命令を備えています。256 ビットのインテル® AVX2 命令は、低レイテンシーの 256 ビット・データパスと高いスループットを実装する Haswell† マイクロアーキテクチャーからサポートされます。

図 15-3 のイントラ符号化 4x4 ブロックのイメージ転置[22] を考えてみます。

128 ビット SIMD 実装では、この転置を次の方法で行うことができます。

- 8 ビット・ピクセルを 16 ビットの word 要素に変換し、4 つの行ベクトルとして 2 つの 4x4 イメージブロックをフェッチします。
- 行列操作 1/128 * (B x R) は、PMADDWD、パックドシフト、および blend 命令による SIMD シーケンスを使用して、イメージブロックの行ベクトルと右側の係数行列の列ベクトルを評価できます。
- 2 つの 4x4 word 粒度は、中間結果を列ベクトルに再配置できます。
- 左側の行ベクトルの係数行列と中間ブロックの列ベクトルを (PMADDWD、shift、blend を使用して) 計算して、書き出します。

[22] C. Yeo, Y. H. Tan, Z. Li and S. Rahardja, "Mode-Dependent Fast Separable KLT for Block-based Intra Coding," JCTVC-B024, Geneva, Switzerland, Jul 2010

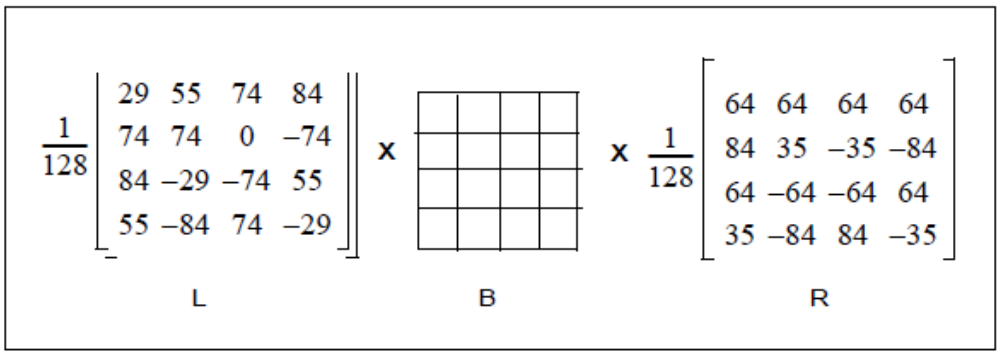


図 15-3 4x4 イメージブロックの転置

いくつかの手法はインテル® AVX2 を使用して簡単に実装できます。インテル® AVX2 のコードシーケンスを例 15-38 と例 15-39 に示します。

例 15-38 インテル® AVX2 を使用した分離可能な KLT ブロック内転置向けのマクロ

```
// b0: ワードピクセルの 4 つの連続した 4x4 イメージブロックから行ベクトルを入力
// rmc0-3: 右側の行列の列ベクトル係数、256 ビットのため 4 回繰り返す
// min32km1: 中間ピクセルを 32767 以下に制限する飽和定数ベクトル
// w0: 不明確な中間行列の行を出力、各ブロック内の要素は不明確
// 例えば、行 0 の下位 128 ビットは降順: y07, y05, y06, y04, y03, y01, y02, y00
#define __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0_256,
rmc1_256, rmc2_256, rmc3_256, min32km1)¥
{__m256i tt0, tt1, tt2, tt3, tttmp;¥
  tt0 = _mm256_madd_epi16(b0, (rmc0_256));¥
  tt1 = _mm256_madd_epi16(b0, rmc1_256);¥
  tt1 = _mm256_hadd_epi32(tt0, tt1);¥
  tttmp = _mm256_srai_epi32( tt1, 31);¥
  tttmp = _mm256_srli_epi32( tttmp, 25);¥
  tt1 = _mm256_add_epi32( tt1, tttmp);¥
  tt1 = _mm256_min_epi32(_mm256_srai_epi32( tt1, 7), min32km1);¥
  tt1 = _mm256_shuffle_epi32(tt1, 0xd8); ¥
  tt2 = _mm256_madd_epi16(b0, rmc2_256);¥
  tt3 = _mm256_madd_epi16(b0, rmc3_256);¥
  tt3 = _mm256_hadd_epi32(tt2, tt3);¥
  tttmp = _mm256_srai_epi32( tt3, 31);¥
  tttmp = _mm256_srli_epi32( tttmp, 25);¥
  tt3 = _mm256_add_epi32( tt3, tttmp);¥ tt3 = _mm256_min_epi32( _mm256_srai_epi32(tt3, 7),
  min32km1);¥
  tt3 = _mm256_shuffle_epi32(tt3, 0xd8);¥
  w0 = _mm256_blend_epi16(tt1, _mm256_slli_si256( tt3, 2), 0xaa);¥
}
(続く)
```

```
// t0-t3: 正確な中間行列 1/128 * (B x R) の256 ビット入力ベクトル
// lmr_256: 左の係数行 1 つの 256 ビット･ベクトル (4 回繰り返し)
// min32km1: 最終ピクセルを 32767 以下に制限する飽和定数ベクトル
// w0; 正しい順番で最終結果の行ベクトルを出力
#define __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr_256, min32km1)¥
{__m256itb0, tb1, tb2, tb3, tbtmp;¥
    tb0 = _mm256_madd_epi16( lmr_256, t0);¥
    tb1 = _mm256_madd_epi16( lmr_256, t1);¥
    tb1 = _mm256_hadd_epi32(tb0, tb1);¥
    tbtmp = _mm256_srai_epi32( tb1, 31);¥
    tbtmp = _mm256_srli_epi32( tbtmp, 25);¥
    tb1 = _mm256_add_epi32( tb1, tbtmp);¥
    tb1 = _mm256_min_epi32( _mm256_srai_epi32( tb1, 7), min32km1);¥
    tb1 = _mm256_shuffle_epi32(tb1, 0xd8);¥
    tb2 = _mm256_madd_epi16( lmr_256, t2);¥
    tb3 = _mm256_madd_epi16( lmr_256, t3);¥
    tb3 = _mm256_hadd_epi32(tb2, tb3);¥
    tbtmp = _mm256_srai_epi32( tb3, 31);¥
    tbtmp = _mm256_srli_epi32( tbtmp, 25);¥
    tb3 = _mm256_add_epi32( tb3, tbtmp);¥
    tb3 = _mm256_min_epi32( _mm256_srai_epi32( tb3, 7), min32km1);¥
    tb3 = _mm256_shuffle_epi32(tb3, 0xd8); ¥
    tb3 = _mm256_slli_si256( tb3, 2);¥
    tb3 = _mm256_blend_epi16(tb1, tb3, 0xaa);¥
    w0 = _mm256_shuffle_epi8(tb3, _mm256_setr_epi32( 0x5040100, 0x7060302, 0xd0c0908,
    0xf0e0b0a,
    0x5040100, 0x7060302, 0xd0c0908, 0xf0e0b0a));¥
}
```

例 15-39 では、行列 1/128 * (B xR) の乗算は、word ピクセルの 4 つの連続した 4x4 イメージブロックから最初に 4 ワイドでフェッチすることで評価されます。例 14-38 に示す最初のマクロは、各 4x4 ブロックの 2 つの中央要素間の不明瞭なシーケンスにある、それぞれの中間列結果から出力ベクトルを生成します。例 15-39 は、shuffle/unpack プリミティブの代わりに blend プリミティブを使用して、不明瞭な要素を元に戻し中間行ベクトルから列ベクトルへの転置を実装します。

Haswell† マイクロアーキテクチャーでは、shuffle/pack/unpack プリミティブはポート 5 のシャッフル実行ユニットにディスパッチされます。重い SIMD シーケンスでは、ポート 5 への負荷がパフォーマンスを決定する重要な要因となる可能性があります。

Haswell† マイクロアーキテクチャー上で実行されるポート 5 への負荷が高い 128 ビット SIMD コードは、256 ビットのインテル® AVX2 へ移植することでパフォーマンスを向上してポート 5 への負荷を軽減できます。

例 15-39 インテル® AVX2 を使用した分離可能な KLT ブロック内転置

```
short __declspec(align(16))cst_rmc0[8] = {64, 84, 64, 35, 64, 84, 64, 35};
short __declspec(align(16))cst_rmc1[8] = {64, 35, -64, -84, 64, 35, -64, -84};
short __declspec(align(16))cst_rmc2[8] = {64, -35, -64, 84, 64, -35, -64, 84};
short __declspec(align(16))cst_rmc3[8] = {64, -84, 64, -35, 64, -84, 64, -35};
short __declspec(align(16))cst_lmr0[8] = {29, 55, 74, 84, 29, 55, 74, 84};
short __declspec(align(16))cst_lmr1[8] = {74, 74, 0, -74, 74, 74, 0, -74};
short __declspec(align(16))cst_lmr2[8] = {84, -29, -74, 55, 84, -29, -74, 55};
short __declspec(align(16)) cst_lmr3[8] = {55, -84, 74, -29, 55, -84, 74, -29};

void Klt_256_d(short * Input, short * Output, int iWidth, int iHeight)
{int iX, iY;
__m256i rmc0 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *) &cst_rmc0[0]));
__m256i rmc1 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *)&cst_rmc1[0]));
__m256i rmc2 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *)&cst_rmc2[0]));
__m256i rmc3 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *)&cst_rmc3[0]));
__m256i lmr0 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *)&cst_lmr0[0]));
__m256i lmr1 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *)&cst_lmr1[0]));
__m256i lmr2 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *)&cst_lmr2[0]));
__m256i lmr3 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *)&cst_lmr3[0]));
__m256i min32km1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((_mm_setr_epi32( 0x7fff7fff,
0x7fff7fff, 0x7fff7fff, 0x7fff7fff));
__m256i b0, b1, b2, b3, t0, t1, t2, t3;
__m256i w0, w1, w2, w3;
short* pImage = Input;
short* pOutImage = Output;
int hgt = iHeight, wid= iWidth;

// 最も内側の括弧から 1/128 * (Mat_L x (1/128 * (Mat_B x Mat_R))) を実装
for( iY = 0; iY < hgt; iY+=4) {
   for( iX = 0; iX < wid; iX+=16) {
   // word ピクセルの 4 つの連続した 4x4 行列の行 0 をロード
   b0 = _mm256_loadu_si256( (__m256i *) (pImage + iY*wid+ iX)) ;
   // 右の行列係数の列ベクトルと行 0 を掛ける
   __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0, rmc1, rmc2, rmc3, min32km1);
   // 不正確な行 0 の下位 128 ビット、上位 -> 下位: y07, y05, y06, y04, y03, y01, y02, y00
   b1 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+1)*wid+ iX) );
   __MyM_KIP_PxRMC_ROW_4x4Wx4(b1, w1, rmc0, rmc1, rmc2, rmc3, min32km1);
   // hi->lo y17, y15, y16, y14, y13, y11, y12, y10
   b2 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+2)*wid+ iX) );
   __MyM_KIP_PxRMC_ROW_4x4Wx4(b2, w2, rmc0, rmc1, rmc2, rmc3, min32km1);
   b3 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+3)*wid+ iX) );
   __MyM_KIP_PxRMC_ROW_4x4Wx4(b3, w3, rmc0, rmc1, rmc2, rmc3, min32km1);
(続く)
```

```
    // 各 4x4 ブロックの不正確な中央 2 つの要素を元に戻し、
    // 不正確なベクトルへ転置: t0 は 4 つの連続した行 0 または 4 つの 4x4 の中間値を持つ
    t0 = _mm256_blend_epi16( w0, _mm256_slli_epi64(w1, 16), 0x22);
    t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w2, 32), 0x44);
    t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w3, 48), 0x88);
    t1 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 32), _mm256_srli_epi64(w1, 16), 0x22);
    t1 = _mm256_blend_epi16( t1, w2, 0x44);
    t1 = _mm256_blend_epi16( t1, _mm256_slli_epi64(w3, 16), 0x88); // column 1
    t2 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 16), w1, 0x22);
    t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w2, 16), 0x44);
    t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w3, 32), 0x88); // column 2
    t3 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 48), _mm256_srli_epi64(w1, 32), 0x22);
    t3 = _mm256_blend_epi16( t3, _mm256_srli_epi64(w2, 16), 0x44);
    t3 = _mm256_blend_epi16( t3, w3, 0x88);// column 3
    // 中間ブロックの 4 つの列ベクトルと左の係数ベクトルの行 0 を掛ける
    // 最終的な出力行は通常の並びに配置される
    __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr0, min32km1);
    _mm256_store_si256( (__m256i *) (pOutImage+iY*wid+ iX), w0) ;
    __MyM_KIP_LMRxP_ROW_4x4Wx4(w1, t0, t1, t2, t3, lmr1, min32km1);
    _mm256_store_si256( (__m256i *) (pOutImage+(iY+1)*wid+ iX), w1) ;
    __MyM_KIP_LMRxP_ROW_4x4Wx4(w2, t0, t1, t2, t3, lmr2, min32km1);
    _mm256_store_si256( (__m256i *) (pOutImage+(iY+2)*wid+ iX), w2) ;
    __MyM_KIP_LMRxP_ROW_4x4Wx4(w3, t0, t1, t2, t3, lmr3, min32km1);
    _mm256_store_si256( (__m256i *) (pOutImage+(iY+3)*wid+ iX), w3) ;
    }
  }
}
```

ここでは 128 ビットの SIMD 実装を示しませんが簡単に作成できます。

この KLT イントラ符号化転置の 128 ビット SIMD コードを、Sandy Bridge† マイクロアーキテクチャーで実行すると、2 つのシャッフルユニットによりポート 5 への負荷は軽減され、各 4x4 イメージブロックの転置のための有効スループットはおよそ 50 サイクルとなります。最適化されたスカラー実装からの相対的なスピードアップは、約 2.5 倍です。

Haswell† マイクロアーキテクチャー上で 128 ビット SIMD コードを実行すると、ポート 5 へ発行される uop はすべての uop の 50% 未満であり、これまでの世代のマイクロアーキテクチャーではおよそ 3 分の 1 であったため、パフォーマンスが約 25% 低下します。一方、インテル® AVX2 の実装では 4x4 ブロックあたり 35 サイクル未満の有効スループットを提供します。

15.16.1 マルチバッファリングとインテル® AVX2

データバッファーのストリームを操作する計算集約型のアルゴリズムも多くあります (ハッシュ、暗号化など)。データストリームは、SIMD 命令セットを利用するためパーティション化され、複数の独立したバッファーストリームとして扱われることがあります。

複数のバッファーを並列にハッシュする方法の詳細については、http://eprint.iacr.org/2012/476.pdf (英語) および http://www.scirp.org/journal/PaperInformation.aspx?paperID=23995 (英語) を参照してください。

インテル® AVX2 では、論値と数値操作向けに複数のデータ型で豊富な機能性と完全な 256 ビット SIMD 命令が提供されます。XMM レジスターとこれまでの世代のインテル® SSE 命令セットに投資されたアルゴリズムは、インテル® AVX2 の YMM を使用して高いスループットを提供するため、複数バッファリング・アルゴリズムを拡張

できます。最適化された 256 ビットのインテル® AVX2 実装では、128 ビット・バージョンと比較して最大 1.9 倍のスループットが得られます。

15.16 節で述べたイメージブロック転置の例は、4x4 ブロックの複数バッファリングの実装として解釈できます。パフォーマンスのベースラインを 2 つのシャッフルポートのマイクロアーキテクチャー (Sandy Bridge†) から、単一シャッフルポートのマイクロアーキテクチャーへ切り替えると、256 ビット幅のインテル® AVX2 は 128 ビット SIMD の実装に対して 1.9 倍のスピードアップをもたらします。

複数バッファリングの詳細については、http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-iamulti-buffer-paper.pdf (英語) のホワイトペーパーを参照してください。

15.16.2 剰余除算とインテル® AVX2

RSA 2048 などの公開鍵暗号化において重要なモジュラー冪剰余操作を効率良く実装するため、非常に大きな整数の剰余乗算はしばしば使用されます。剰余乗算のライブラリー実装では、多くの場合 MUL/ADC のチェーンシーケンスが利用されます。通常、MUL 命令は 128 ビットの中間的な整数出力を生成し、64 ビットの中間データの粒度で加算キャリーチェーンを使用する必要があります。

インテル® AVX2 において、VPMULUDQ/VPADDQ/VPSRLQ/VPSLLQ/VPBROADCASTQ/VPERMQ 命令は、RSA 1024 と RSA 2048 に対応するキー長に関連する剰余乗算/冪剰余を効率良く実装するベクトル化のアプローチを可能にします。OpenSSL における剰余乗算/冪剰余とインテル® AVX2 の実装の詳細については、http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3_9?LI=true (英語) を参照してください。

基本的なヒューリスティックは、冗長表現の 512/1024 ビット指数で大きな整数入力オペランドを再定式化することから始めます。例えば、1024 ビット整数は基数 2^29 と 36 “**桁**” を使用して表現できます。ここで各 “**桁**” は 2^29 未満です。このような冗長表現の桁は、ベクトルレジスターの dword スロットに配置できます。大きな整数のような冗長表現は、符号なし整数乗算の中間結果のハードウェアの粒度にわたってキャリー加算チェーンの要件を簡略化します。

冗長表現された**数値**を使用するインテル® AVX2 の VPMULUDQ は、十分な余地を残して 4 つの 64 ビットの中間結果を生成できます (例えば、5 つの最上位ビットが符号ビットを除いて 0)。

その場合、VPADDQ は ADC のような命令と同等の SIMD バージョンを必要とせず、加算キャリーチェーンの要件を実装するには十分です。冗長表現への変換のコスト要因や VPMULUDQ/VPADDQ チェーンの並列出力帯域幅向けの効率良いスピードアップの説明を含む、詳しい情報については前の段落で示したウェブサイトでご覧いただけます。

15.16.3 データ移動に関する考察

Haswell† マイクロアーキテクチャーは、2 つの 256 ビット・ロードと 1 つの 256 ビット・ストアの uop を各サイクルでディスパッチできます。データ移動操作の多い既存のバイナリーは、以前の世代のマイクロアーキテクチャー向けに最適化されていれば、再コンパイルなしでこの拡張と高帯域幅の L1 や L2 キャッシュから利点を得られます。例えば、256 ビット SAXPY 計算はこれまでの世代のマイクロアーキテクチャーでは、ロード/ストアポートの数で制限されていましたが、Haswell† マイクロアーキテクチャーでは直ちに利点が得られます。

状況によっては、命令セットのマイクロアーキテクチャー上の制限との複雑な相互作用があるため、議論の余地があるかもしれません。一般的に使用される 2 つのライブラリー関数 memcpy() と memset() を新しいマイクロアーキテクチャー上に実装する最適な選択肢について考えてみましょう。

Haswell† マイクロアーキテクチャー上の memcpy() で、大きなコピー長の memcpy 操作を実装するため REP MOVSB 命令を使用することは、256 ビットのストア・データパスの利点を活用してサイクルごとに 20 バイト以上のスループットを提供できます。コピー長が数百バイト以下の場合、REP MOVSB のアプローチは 15.16.3.1 節に示す 128 ビットの SIMD 手法を使用するよりも低速です。

Ice Lake† Client マイクロアーキテクチャー上の memcpy() は、インライン化された REP MOVSB を実装する memcpy() を使用すると、コンパイル時に不明な可変コピー長の 256 ビットのインテル® AVX 実装と同程度高速になります。コンパイル時にレングスが判明している場合、REP MOVSB は 128 バイトまでの短い文字列で 256 ビットのインテル® AVX とほぼ同等のパフォーマンスを示し (9 サイクル vs 3-7 サイクル)、2K バイト以上の文字列ではインテル® AVX を上回ります。このような場合、インライン化された REP MOVSB を使用することを推奨します。しかし、ソフトウェアはゼロ・バイト・コピーの場合は分岐すべきです。

15.16.3.1 memcpy() を実装する SIMD ヒューリスティック

memcpy() に 128 ビット SIMD 命令の実装を試みるため、一般的なヒューリスティックを考えることから始めます。対象とする命令セットのレジスター幅に対する 3 つの数値要因 (デスティネーション・アドレス・アライメント、ソース・アドレス・アライメント、コピーするバイト数) を中心に議論します。

memcpy のデータ移動は次のフェーズに分割できます。

- 最初のアライメントされていない 16 バイトのコピーは、ループ中で使用するデスティネーション・アドレスのポインターが 16 バイトにアライメントされることを可能にします。これにより、後続のストアは同じように 16 バイトにアライメントされたストアを使用することができます。
- コピーで残されたバイトは、(a) アンロールされた 16 バイト・コピー操作の倍数と、(b) 16 バイト未満のいくつかのコピー操作を含む残余カウントに分類できます。例えば、ループ反復のオーバーヘッドを軽減するため 8 回アンロールするには、残余カウントは 1 から 8 x 16 - 1 = 127 のケースを個々に考慮しなければなりません。
- 8 x 16 にアンロールされたメインループ内部の各 16 バイトのコピー操作では、16 バイト境界にアライメントされていないソースのポインターアドレスに対処し、16 個のデータを 16 バイトにアライメントされたデスティネーション・アドレスにストアする必要があります。繰り返し使用されるソースポインターが 16 バイトにアライメントされていない場合、最も効率良い手法は次の 3 命令シーケンスです。
  - 16 バイト・アライメントに調整されたポインターアドレスから 16 バイトのチャンクをフェッチし、このチャンクの一部を以前の 16 バイト・アライメントからフェッチした一部で補って使用します。
  - 現在のチャンクと以前のチャンクの一部の結合には PALIGNR を使用します。
  - 結合した 16 バイトの新しいデータをアライメントされたデスティネーション・アドレスへストアし、この 3 命令シーケンスを繰り返します。

  この 3 命令の手法は、それぞれの 16 バイト・コピー操作のフェッチ:ストアの命令比率が 1:1 となることを可能にします。

上記の手法 (特にメインループが数千バイトのデータをコピーする場合) は、ストア操作に 128 ビットのデータパスを持つ Sandy Bridge† マイクロアーキテクチャーと Ivy Bridge† マイクロアーキテクチャーではサイクルごとにおよそ 10 バイトのスループットを達成できます。この手法を拡張して広いデータパスの使用を試みると次の制限を受けます。

- 2 x 128 レーンを持つマイクロアーキテクチャーで 256 ビット VPALIGNR を使用するには、現在の 32 バイトにアライメントされた 256 ビット・フェッチの 2 つのチャンクの結合に、現在の 32 バイトにアライメントされたフェッチから 16 バイト・オフセットされたアドレスからの別のフェッチが必要になります。
  - 各バイト・コピー操作のフェッチ:ストアの比率は 2:1 になります。
  - 32 バイトにアライメントされていないフェッチ (16 バイトにアライメントされている) は、コピー操作の 64 バイトごとにキャッシュライン分割のペナルティーが科せられます。

256 ビット・ストア・データパスを持つマイクロアーキテクチャーの利点を生かす 256 ビット ISA の使用の試みは、4 命令シーケンスとキャッシュライン分割のペナルティーで相殺されました。

15.16.3.2 拡張された REP MOVSB を使用した memcpy() の実装

memcpy() を実装するため、拡張された REP MOVSB を使用した代替アプローチを比較するのは興味深いことです。Haswell† マイクロアーキテクチャーと Ivy Bridge† マイクロアーキテクチャーでは、REP MOVSB は最適化され、ハードウェアによる uop フローを提供します。

Ivy Bridge† マイクロアーキテクチャーでは、REP MOVSB の memcpy 実装は、数千バイトをコピーする場合 128 ビット SIMD 実装より少し良いスループットを達成します。しかし、コピー操作が数百バイト未満のサイズを扱う場合、REP MOVSB のアプローチは 15.16.3.1 節の 2 番目のセクションで説明した、明白な残余コピーの手法ほど効率良くありません。これは、残りのコピー長 1 - 127 (ジャンプテーブルや switch/case 文によりメインループの前で実施) の処理に加え、1 回もしくは 2 回の 8 x 16 バイト反復がハードウェアによる uop フローよりは少ない分岐のオーバーヘッドを被ります。memcpy() の 128 ビット SIMD 実装の詳細を知る方法の 1 つは、glibc などのオープソース・ライブラリーのソースコードを参照することです。

Haswell† マイクロアーキテクチャー上で、大きなコピー長の memcpy 操作を実装するため REP MOVSB 命令を使用することは、256 ビットのストア・データパスの利点を活用してサイクルごとに 20 バイト以上のスループットを提供できます。コピー長が数百バイト以下の場合、REP MOVSB のアプローチは 15.16.3.1 節の残余フェーズとしてコピー長を扱うよりもまだ低速です。

15.16.3.3 memset() 実装の考察

memset() のインターフェイスには、デスティネーションとして 1 つのアドレスポインターがあります。これは、アライメントされた 256 ビットのストア命令を使用する際にアドレス・アライメントの管理を簡素化します。最初のアライメントされていないストアの後、デスティネーション・ポインターを 32 バイト境界に調整して、15.16.3.1 節で説明される同じ方法で残りを処理します。これには、アンロールされた 32 バイト・アライメントのストアに応じて、最小の分岐によって残りの値を扱う大きなジャンプテーブルを使用する可能性があります。メインループでは、単純に YMM レジスターを 32 バイトにアライメントされたストア操作へサイクルごとに 30 バイト近くを送り出すことができます (長さが数千バイトの場合)。ここでの制限要因は、各 256 ビット VMOVDQA ストアがストアアドレスとストアデータの uop フローから成り立つということです。各サイクルでストアデータ uop は、ポート 4 にのみディスパッチ可能です。

memset() を実装するため REP STOSB を使用すると、memcpy() に REP MOVSB を使用する場合と同様に、SIMD 実装に対してコードサイズの利点があります。Haswell† マイクロアーキテクチャー上で REP STOSB を使用する memset() ルーチンの実装は、256 ビット・データパスと増加した L1 データキャッシュの帯域幅 (サイクルごとに最大 32 バイト) の利点を得られます。

REP STOSB と 256 ビットのインテル® AVX2 を使用した memset() 実装のパフォーマンスを比較するには、memset() の呼び出しパターンを考慮する必要があります。呼び出しパターンによって、使用するパフォーマンス計測の方法が異なります。それぞれの計測方法には副作用があることがあります。

memset() のような単純なルーチンでしばしば使用される最も一般的な計測方法は、大きなループカウント内部で memset() を呼び出し、その前後で RDTSC 命令を使用して計測を行うものです。

memset() 呼び出しの間に干渉する命令ストリームが実行されず、異なるカウント値による複数の memset() 呼び出しが連続するパターンを計測する場合、この計測方法のバリエーションを適用できます。

上記両方の memset() 呼び出しのシナリオでは、分岐予測はループ実行の総サイクルの計測に影響する重要な要因となります。そのため、RDTSC のオーバーヘッドを最小化するため、大きなループでインテル® AVX2 が実装された memset() を計測すると、そのループカウントによって予測される分岐で歪んだ結果となる可能性があります。

より現実的なソフトウェア・スタックでは、memset() 呼び出しのパターンには次のような特性があります。

- memset() 呼び出しの間で実行される干渉する命令ストリームがあると、memset() 呼び出し前の分岐予測器の状態は、memset() 実装内の分岐シーケンス向けに事前トレーニングされたものではなくなります。
- memset() のカウント値は訂正されない傾向があります。

現実的な memset() 呼び出しのシナリオを考えると、memset() のパフォーマンスを比較する適切な計測手法は、それぞれの memset() の呼び出しの前後を 2 つの RDTSC で挟み込んで計測する方法です。

呼び出しごとに RDTSC で計測する手法では、RDTSC のオーバーヘッドが発生し、計測するループ外部で事前補正および事後評価が必要となります。呼び出しごとの計測手法は、ループを包み込んで呼び出しごとに計測するため、キャッシュ・ウォーミングを考慮する必要があるかもしれません。

計測手法に関連するスキュー要因が有効である場合、カウント値が数百バイト以下の REP STOSB を使用する memset() のパフォーマンスは、一般的にインテル® AVX2 バージョンの memset() 呼び出しシナリオよりも高速です。数百ものアンロールされた memset() 呼び出しを行う極端なシナリオにおいてのみ、数百バイト以下のカウント値をすべて使用して、memset() の間にある干渉する命令ストリームの影響なしで memset() のインテル® AVX2 バージョンは分岐予測のトレーニングの利点を生かすことができます。

15.16.3.4 使用するコードの前に Memcpy/Memset を移動

memcpy/memset を呼び出して準備したデータと、それを使用するその後の命令を再配置できることもあります。

```
memcpy ( pBuf, pSrc, Cnt);          // Cnt の情報といくつかのデータをコピー。
.....                               // 後続の命令シーケンスは pBuf をすぐには消費しません。
result = compute( pBuf);            // memcpy の結果をここで使用。
```

カウント (Cnt) が少なくても 1000 バイト以上であることが明らかである場合、拡張された REP MOVSB/STOSB を使用すると非消費コードのコストを相殺できる利点が得られます。次のように Cnt = 4096 の値で memset() を使用することで、ヒューリスティックを理解できます。

- memset() の 256 ビット SIMD 実装では、非消費命令シーケンスがリタイアする前に、128 の VMOVDQA と 32 バイトのストア操作を発行/実行/リタイアする必要があります。
- ECX = 4096 の拡張 REP STOSB は、ハードウェアによる長い uop フローとしてデコードされますが、1 命令としてリタイアします。memset() の結果を使用する前には、完了しなければならない多くのストアデータ操作があります。ストアデータ操作の完了はプログラム順序のリタイアとは分離されるため、非消費コードストリームの実質的な部分がストアバッファーのリソースと競合しなければ、発行/実行/リタイアメントを通して処理できます。

拡張された REP MOVSB/STOSB を使用するソフトウェアは、CPUID.(EAX=07H, ECX=0):EBX.[ビット 9] が 1 であることを調査して命令が利用できることを確認する必要があります。

15.16.3.5 256 ビット・フェッチと 128 ビット・フェッチ

Sandy Bridge† マイクロアーキテクチャーと Ivy Bridge† マイクロアーキテクチャーでは、マイクロアーキテクチャーのメモリー・パイプラインの 128 ビット・データパスの制約から見て、16 バイトにアライメントされた 2 つのロードの使用が推奨されます。

Haswell† マイクロアーキテクチャーの 256 ビット・データパス・マイクロアーキテクチャーの利点を活用するには、256 ビット・ロードの暗黙のアライメントを考慮すべきです。メモリーから 256 ビット・データをフェッチする命令は、32 バイトのアライメントに注意する必要があります。32 バイトにアライメントされていないフェッチがキャッシュライン境界にまたがっている場合も、2 つの 16 バイト境界でアライメントされたアドレスからフェッチすることが推奨されます。

15.16.3.6 MULX とインテル® AVX2 命令の混在

MULX とインテル® AVX2 命令を組み合わせると、特定の一般的な計算タスクのパフォーマンスをさらに向上できます。例えば、64 ビット整数から ascii 形式への数値変換では、柔軟性のある MULX レジスター割り当て、広い YMM レジスター、そして並列係数/余剰計算のためのパックド・シフト・プリミティブ VPSRLVD から恩恵を受けられます。

例 15-40 に、1 または 2 つの有限範囲の unsigned short 整数を計算してそれぞれ 10 進数にするインテル® AVX2 命令のマクロシーケンスを示します。ここでは、モンゴメリー乗算とともに VPSRLVD 命令を使用しています。

例 15-40 並列に係数/余剰を計算するマクロ

```
static short quoTenThsn_mulplr_d[16] =
{ 0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0, 0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static short mten_mulplr_d[16] = { -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10,
1};
// 商 (dword 4) と余り (dword 0) を含む入力 t5 (a __m256i type) を
// 出力 y3 (__m256i) の単一桁の整数 (0-9) へ変換するマクロ、
// 両方の dword 要素 "t5" は 10^4 未満と想定され、残りの dword は 0 である必要がある。
// 出力は 8 つの単一桁の整数で、各 dword の下位バイトに配置され、dword の MS 桁は 0。
#define __ParMod10to4AVX2dw4_0( y3, t5 ) ¥
{ __m256i x0, x2; ¥
   x0 = _mm256_shuffle_epi32( t5, 0); ¥
   x2 = _mm256_mulhi_epu16(x0, _mm256_loadu_si256( (__m256i *) quoTenThsn_mulplr_d));¥
   x2 = _mm256_srlv_epi32( x2, _mm256_setr_epi32(0x0, 0x4, 0x7, 0xa, 0x0, 0x4, 0x7, 0xa) ); ¥
   (y3) = _mm256_or_si256(_mm256_slli_si256(x2, 6), _mm256_slli_si256(t5, 2) ); ¥
   (y3) = _mm256_or_si256(x2, y3);¥
   (y3) = _mm256_madd_epi16(y3, _mm256_loadu_si256( (__m256i *) mten_mulplr_d) ) ;¥
}
// dword 整数 (< 10^4) を 4 つの単一桁整数 __m128i へ並列に変換
#define __ParMod10to4AVX2dw( x3, dw32 ) ¥
{ __m128i x0, x2; ¥
   x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128( dw32)); ¥
   x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d));¥
   x2 = _mm_srlv_epi32( x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) ); ¥
   (x3) = _mm_or_si128(_mm_slli_si128(x2, 6), _mm_slli_si128(_mm_cvtsi32_si128( dw32), 2) ); ¥
   (x3) = _mm_or_si128(x2, (x3));¥
   (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;¥
}
```

例 15-41 は、ヘルパー・ユーティリティーと 64 ビット符号付き整数を、MULX を使用して 63 ビット符号なし範囲の整数の商/余剰のペアへ縮小するすべてのステップを示しています。この例は、例 15-40 と例 15-42 に関連していることに注意してください。

例 15-41 符号付き 64 ビット整数の変換ユーティリティー

```
#defineQWCG10to80xabcc77118461cefdull

static int pr_cg_10to4[8] = { 0x68db8db, 0 , 0, 0, 0x68db8db, 0, 0, 0};
static int pr_1_m10to4[8] = { -10000, 0 , 0, 0 , 1, 0 , 0, 0};
char * i64toa_avx2i( __int64 xx, char * p)
{int cnt;
   _mm256_zeroupper();
   if( xx < 0) cnt = avx2i_q2a_u63b(-xx, p);
   else cnt = avx2i_q2a_u63b(xx, p);
   p[cnt] = 0;
   return p;
}
// unsigned short (< 10^4) を ascii へ変換
__inline int ubsAvx2_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
__m128i x0, m0, x2, x3, x4;
   if( x_Lt10k < 10) { *ps = '0' + x_Lt10k; return 1; }
   x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128( x_Lt10k));
   // 除数 10、100、1000、10000 の商を計算
   m0 = _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d);
   x2 = _mm_mulhi_epu16(x0, m0);
   // u16/10, u16/100, u16/1000, u16/10000
   x2 = _mm_srlv_epi32( x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) );
   // 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
   x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
   x4 = _mm_or_si128(x2, x3);
   各 dword の下位バイトに 4 つの単一桁を生成
   x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;// ascii エンコード向けにバイア
   スを加算
   x2 = _mm_add_epi32( x4, _mm_set1_epi32( 0x30303030 ) );
   // 4 つの単一桁をパック、最上位桁から開始
   x3 = _mm_shuffle_epi8(x2, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
   if (x_Lt10k > 999 ) {*(int *) ps = _mm_cvtsi128_si32( x3); return 4;}

   tmp = _mm_cvtsi128_si32( x3);
   if (x_Lt10k > 99 ) {
   *((short *) (ps)) = (short ) (tmp >>8);
   ps[2] = (char ) (tmp >>24);
   return 3;
   }

   *((short *) ps) = (short ) (tmp >> 16);
   return 2;
}
```

例 15-42 は、ベクトル化されたモンゴメリー乗算方式による先進的なリダクション手法を使用して、63 ビットのダイナミック・レンジを ascii 形式に数値変換するステップを示しています。この例は、例 15-40 に関連してることに注意してください。

例 15-42 符号付き 63 ビット整数の変換ユーティリティー

```
unsigned avx2i_q2a_u63b (unsigned __int64 xx, char *ps)
{ __m128i v0;
   __m256i m0, x1, x3, x4, x5 ;
   unsigned __int64 xxi, xx2, lo64, hi64;
__int64 w;
   int j, cnt, abv16, tmp, idx, u;
   // 4 桁以下を変換
   if ( xx < 10000 ) {
     j = ubsAvx2_Lt10k_2s_i2 ( (unsigned ) xx, ps); return j;
   } else if (xx < 100000000 ) { // dynamic range of xx is less than 9 digits
     // 5-8 桁の変換
     x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)xx)); // broadcast to every dword
     // 減少した範囲 (< 10^4) でそれぞれ商と余りを計算
     x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
     x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
     // 商を dw4、余りを dw0
     m0 = _mm256_add_epi32( _mm256_castsi128_si256(_mm256_setzero_si256(),
                                                   _mm_cvtsi32_si128((int)xx), 0), x3);
     __ParMod10to4AVX2dw4_0( x3, m0); // 各 dw の下位バイトの 8 桁
     x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
     x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
     0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
     // 8 つの 1 桁整数をパックして最初の 8 バイトと残りはゼロに設定
     x4 = _mm256_permutevar8x32_epi32( x4, _mm256_setr_epi32(0x4, 0x0, 0x1, 0x1, 0x1, 0x1, 0x1,
   0x1) );
     tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
     _BitScanForward((unsigned long *) &idx, tmp);
     cnt = 8 -idx; // 出力へ書き出すゼロ以外の桁数
   } else { // 9-12 桁の変換
     lo64 = _mulx_u64(xx, (unsigned __int64) QWCG10to8, &hi64);
     hi64 >>= 26;
     xxi = _mulx_u64(hi64, (unsigned __int64)100000000, &xx2);
     lo64 = (unsigned __int64)xx - xxi;
     if( hi64 < 10000) { // 桁 12-9 を最初に行う
      __ParMod10to4AVX2dw(v0, (int)hi64);
      v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
      // 12 桁未満の下位 8 桁の変換を継続
      x5 = _mm256_inserti128_si256(_mm256_setzero_si256(),_mm_cvtsi32_si128((int)lo64), 0);
      x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)lo64)); // すべての dword へブロードキャスト
      x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
      x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i
      *)pr_1_m10to4));
      m0 = _mm256_add_epi32( x5, x3); // 商を dw4、余りを dw0 に設定
      __ParMod10to4AVX2dw4_0( x3, m0);
      x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
      x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
      0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
      x5 = _mm256_inserti128_si256(_mm256_setzero_si256(), _mm_shuffle_epi8(v0,
      _mm_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080)), 0);
      x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x2, 0x4, 0x0,
      0x1, 0x1, 0x1, 0x1, 0x1) );
     (続く)
```

```
      tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
      _BitScanForward((unsigned long *) &idx, tmp);
      cnt = 12 -idx;
     } else { // 12 桁以上の入力値を処理
      cnt = 0;
      if ( hi64 > 100000000) { // 入力値が 16 桁以上の場合
        xxi = _mulx_u64(hi64, (unsigned __int64) QWCG10to8, &xx2) ;
        abv16 = (int) (xx2 >>26);
        hi64 -= _mulx_u64((unsigned __int64) abv16, (unsigned __int64) 100000000, &xx2);
        __ParMod10to4AVX2dw(v0, abv16);
        v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
        v0 = _mm_shuffle_epi8(v0, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
        tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(v0, _mm_set1_epi32( 0x30303030 )) );
        _BitScanForward((unsigned long *) &idx, tmp);
        cnt = 4 -idx;
      }
      // 下位 16 個の数値を変換
      x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128( (int)hi64)); // broadcast to every dword
      x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
      x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i
      *)pr_1_m10to4));
      m0 = _mm256_add_epi32(_mm256_inserti128_si256(_mm256_setzero_si256(),
      _mm_cvtsi32_si128((int)hi64), 0), x3);
      x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
      x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
      0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
      x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)lo64)); // broadcast to every dword
      x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
      x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i
      *)pr_1_m10to4));
      m0 =
      _mm256_add_epi32(_mm256_inserti128_si256(_mm256_setzero_si256(),_mm_cvtsi32_si128((int)lo64),
      0), ), x3);
      __ParMod10to4AVX2dw4_0( x3, m0);
      x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
      x5 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080,
      0x80808080, 0x80808080, 0x0004080c, 0x80808080) );
      x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x4, 0x0, 0x6,
      0x2, 0x1, 0x1, 0x1, 0x1) );
      cnt += 16;
      if (cnt <= 16) {
       tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
       _BitScanForward((unsigned long *) &idx, tmp);
       cnt -= idx;
      }
     }
    }
    w = _mm_cvtsi128_si64( _mm256_castsi256_si128(x4));
    switch(cnt) {
       case5:*ps++ = (char) (w >>24); *(unsigned *) ps = (w >>32);
       break;
       case6:*(short *)ps = (short) (w >>16); *(unsigned *) (&ps[2]) = (w >>32);
       break;
       (続く)
```

```
      case7:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
      *(unsigned *) (&ps[3]) = (w >>32);
      break;
      case 8: *(long long *)ps = w;
      break;
      case9:*ps++ = (char) (w >>24); *(long long *) (&ps[0]) = _mm_cvtsi128_si64(
      _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
      break;
      case10:*(short *)ps = (short) (w >>16);
      *(long long *) (&ps[2]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
      break;
      case11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
      *(long long *) (&ps[3]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
      break;
      case 12: *(unsigned *)ps = (unsigned int)w; *(long long *) (&ps[4]) = _mm_cvtsi128_si64(
      _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
      break;
      case13:*ps++ = (char) (w >>24); *(unsigned *) ps = (w >>32);
      *(long long *) (&ps[4]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
      break;
      case14:*(short *)ps = (short) (w >>16); *(unsigned *) (&ps[2]) = (w >>32);
      *(long long *) (&ps[6]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
      break;
      case15:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
      *(unsigned *) (&ps[3]) = (w >>32);
      *(long long *) (&ps[7]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
      break;
      case 16: _mm_storeu_si128( (__m128i *) ps, _mm256_castsi256_si128(x4));
      break;
      case17:u = (int)_mm_cvtsi128_si64(v0); *ps++ = (char) (u >>24);
      _mm_storeu_si128( (__m128i *) &ps[0], _mm256_castsi256_si128(x4));
      break;
      case18: u = (int)_mm_cvtsi128_si64(v0); *(short *)ps = (short) (u >>16);
      _mm_storeu_si128( (__m128i *) &ps[2], _mm256_castsi256_si128(x4));
      break;
      case19:u = (int)_mm_cvtsi128_si64(v0); *ps = (char) (u >>8); *(short *) (&ps[1]) = (short) (u
      >>16);
      _mm_storeu_si128( (__m128i *) &ps[3], _mm256_castsi256_si128(x4));
      break;
      case20:u = (int)_mm_cvtsi128_si64(v0); *(unsigned *)ps = (short) (u);
      _mm_storeu_si128( (__m128i *) &ps[4], _mm256_castsi256_si128(x4));
      break;
   }
   return cnt;
}
```

3/9/17 出力桁のダイナミック･レンジにわたってインテル® AVX2 バージョンの数値変換は、標準ライブラリー実装の入力ごとの 85/260/560 サイクルと比較すると、およそ入力ごとに 23/57/54 サイクルです。

上記の手法は、バイナリー-整数-10 進 (BID) エンコード IEEE-754-2008 10 進浮動小数点形式など、他のライブラリーの数値変換にも適用できます。BID-128 形式において、係数 10^16 でモンゴメリー乗算を行うため事前に計算された 256 ビット定数を使用して別の範囲のリダクション･ステージを追加することで例 15-42 を適用できます。256 ビットの定数を構築する手法は、第 14 章「テキスト処理/字句解析/構文解析向けのインテル® SSE4.2 と SIMD プログラミング」で詳しく説明されています。

15.16.4 Gather 命令に関する考察

VGATHER 命令ファミリーは、ベースアドレスからの相対オフセットを含むベクトル・インデックス・レジスターで指定される複数のデータ要素をフェッチします。Haswell† マイクロアーキテクチャー・ベースのプロセッサーは最初に VGATHER 命令を実装し、単一命令で複数の uop が実行されます。Broadwell† マイクロアーキテクチャーでは、VGATHER 命令ファミリーのスループットがかなり改善されています (表 D-5 を参照)。

データ編成とアクセスパターンによっては、VGATHER 命令を使用することなく、より高速な少ない uop で等価なコードシーケンスを作成することができます (15.5.1 節を参照)。例 15-43 に、Haswell† マイクロアーキテクチャー上で VGATHER 命令がパフォーマンスの利点をもたらさない利用状況を示します。

例 15-43 VGATHER を使用しない手法向けのアクセスパターン

アクセスパターン	推奨される命令の選択
連続した要素	通常の SIMD ロード (MOVAPS/MOVUPS, MOVDQA/MOVDQU)
4 要素以下	通常の SIMD ロード + スロットを再配置する水平データ移動
小さなストライド 転置	隣接する要素をすべてロード + VMOVUPD YMM0, [シーケンシャルな要素] VPERMQ YMM1, YMM0, 0x08 // 偶数要素 VPERMQ YMM2, YMM0, 0x0d // 奇数要素 通常の SIMD ロード + shuffle/permute/blend で列へ転置

冗長な要素	一度のロード + shuffle/blend/logical (レジスター内でデータのベクトルを構築)
	この場合、result[i] = x[index[i]] + x[index[i+1]]、複数の VGATHER を使用するために以下の手法が望ましい
	ymm0 <- VGATHER (x[index[k]]); // 8 つの要素をフェッチ
	ymm1 <- VBLEND(VPERM (ymm0), VBROADCAST (x[indexx[k+8]]);
	ymm2 <- VPADD(ymm0, ymm1);

VGATHER 命令を使用すると、VGATHER のレイテンシーとスループットを相殺するのを妨げない手法や、フェッチ操作を消費するコードのデスティネーション・レジスターへのフェッチの前に移動することよって、コードサイズを減らし高速に実行できます。例 15-44 には、Haswell† マイクロアーキテクチャー上で VGATHER を使用することで利点を得られるいくつかのパターンを示します。

VGATHER を使用する一般的なヒント:

- VGATHER 命令でより多くの要素を収集すると、VGATHER のレイテンシーとスループットを相殺できるため、等価な VGATHER を使用しないフローよりもパフォーマンス上の利点が得られます。例えば、256 ビット VGATHER のレイテンシーは、等価な 128 ビット VGATHER の半分以下であり、128 ビット VGATHER を 2 つ使用するよりも利点があります。また、データ要素サイズよりも大きなインデックス・サイズを使用すると、レイテンシーが短くなるだけでなくレジスタースロットを半分しか利用しません。したがって、VGATHER の dword インデックス形式は、dword または float 値がフェッチされる場合 qword インデックスよりも適しています。
- 消費するコードの手前に VGATHER を配置するのは有効です。
- VGATHER は、デスティネーションにある前値と (マスクされていない) 収集された要素をマージします。そのため、デスティネーションの前値をマージする必要がない場合 (例えば、要素がマスクされていない)、デスティネーション・レジスターへの前の値の書き込みと VGATHER 命令の依存関係を排除する利点があります (VXOR 命令でレジスターをゼロにする)。

例 15-44 VGATHER 手法に適したアクセスパターン

アクセスパターン	命令の選択

不明なマスクと 4 つ以上の要素	条件付きで要素を収集するコードは、一般に VGATHER 命令なしではベクトル化できないか、データ依存の分岐予測ミスにより相対的に低パフォーマンスとなります。
	データ依存の分岐がある C コード:
	if (condition[i] > 0) { result[i] = x[index[i]] }
	インテル® AVX2 の等価なシーケンス:
	YMM0 <- VPCMPGT (condition, zeros) // ベクトルマスクを計算
	YMM2 <- VGATHER (x[YMM1], YMM0) // addr=x[YMM1], mask=YMM0

8 つの要素でベクトル化されたインデックス計算	インデックスを生成するためにベクトル化された計算には、VGATHER 命令の機能と相乗効果があります。
	C コードの断片:
	x[index1[i] + index2[i]]
	インテル® AVX2 の等価なシーケンス:
	YMM0 <- VPADD (index1, index2) // ベクトル・インデックスを計算
	YMM1 <- VGATHER (x[YMM0], mask) // addr=x[YMM0]

VGATHER 命令のパフォーマンスを等価な複数の命令で構成されるギャザー操作フローと比較すると、(1) 基本となるアルゴリズムの違い、(2) データ編成の違い、および (3) 等価なフローの有効性などによって変化します。パフォーマンスが重要なアプリケーションでは、両方の選択肢を評価してから採用を決定することを推奨します。

GATHER 命令のスループットは、Broadwell† から Skylake† マイクロアーキテクチャーでも継続して改善されています。この様子を図 15-4 に示します。

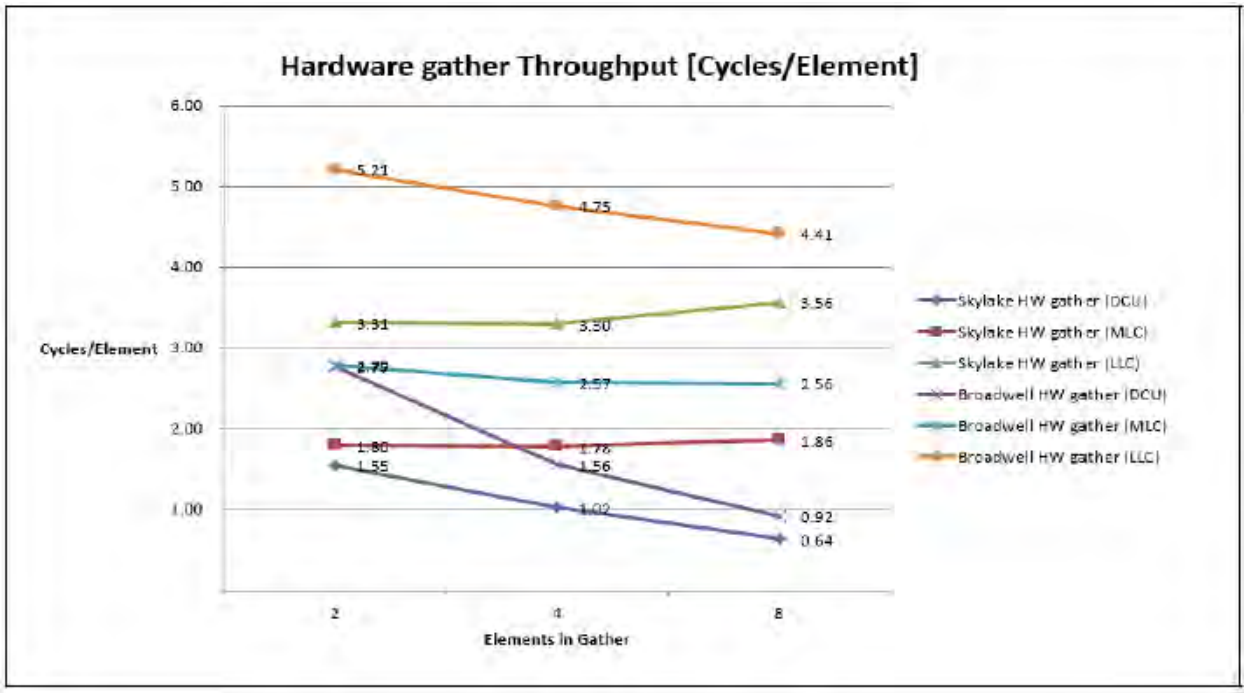


図 15-4 Gather 命令のスループットの比較

例 15-45 は、VGATHER 命令と同等なソフトウェア実装のアセンブリー・シーケンスを示しています。これは、個別要素を挿入した場合の、ハードウェアのギャザー命令とソフトウェア実装のギャザーシーケンスのトレードオフを比較するのに利用できます。

例 15-45 完全なマスクの VPGATHERD と等価なソフトウェアによるインテル® AVX シーケンス

```
mov eax, [rdi] // index0 をロード
vmovd xmm0, [rsi+4*rax] // element0 をロード
mov eax, [rdi+4] // index1 をロード
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x1 // element1 をロード
mov eax, [rdi+8] // index2 をロード
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x2 // element2 をロード
mov eax, [rdi+12] // index3 をロード
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x3 // element3 をロード
mov eax, [rdi+16] // index4 をロード
vmovd xmm1, [rsi+4*rax] // element4 をロード
mov eax, [rdi+20] // index5 をロード
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x1 // element5 をロード
mov eax, [rdi+24] // index6 をロード
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x2 // element6 をロード
mov eax, [rdi+28] // index7 をロード
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x3 // element7 をロード
vinserti128 ymm0, ymm0, xmm1, 1 // ymm0 に結果を生成
```

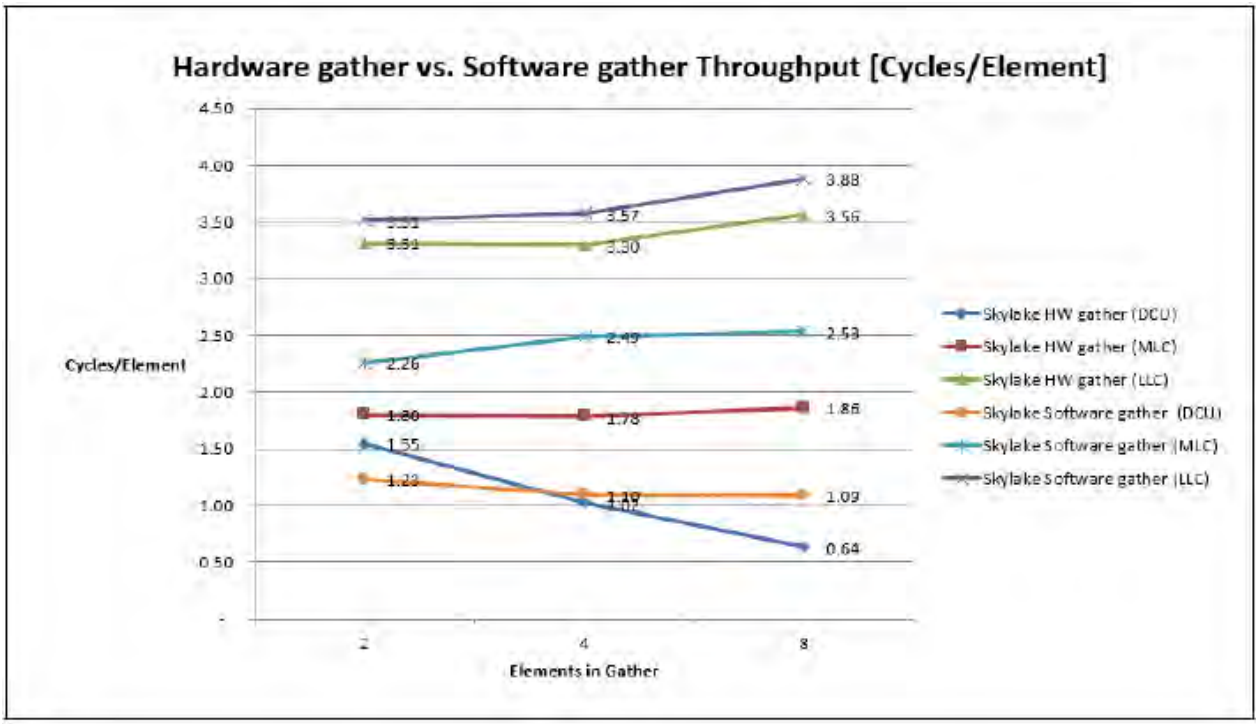


図 15-5 Skylake† マイクロアーキテクチャーにおけるハードウェア GATHER とソフトウェア・シーケンスの比較

図 15-5 は、Skylake† マイクロアーキテクチャーで、データを供給するキャッシュ局所性関数として、VPGATHERD 命令とソフトウェアによるギャザーシーケンスを使用した要素ごとのスループットを比較したものです。命令あたり 2 つのデータ要素のハードウェア GATHER を使用する場合を除き、Skylake† マイクロアーキテクチャーではギャザー命令はソフトウェア・シーケンスよりも優れています。

データがメモリーからローカルに供給される場合、ソフトウェア・シーケンスはハードウェア GATHER 命令よりもうまく動作します。

15.16.4.1 ストライドロード

この節では、構造体配列 (AOS) から配列構造体 (SOA) への転置を処理する実装とハードウェア GATHER 命令の利用を比較します。コードは、複素数配列内の実数と虚数要素を 2 つの異なる配列へ分離します。

C のコード:

```
for(int i=0;i<len;i++){
  Real_buffer[i] = Complex_buffer[i].real;
  Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

例 15-46 AOS から SOA への変換の代替

1: スカラーコード	2: VINSRT+VSHUFPS を使用したインテル® AVX	3: VPGATHERD 使用したインテル® AVX2

loop:
lea eax, [r10+r10*1]
movsxd rax, eax
inc r10d
mov r11d, dword ptr
[rsi+rax*8]
mov dword ptr [rcx+rax*4],
r11d
mov r11d, dword ptr
[rsi+rax*8+0x4]
mov dword ptr [rdx+rax*4],
r11d
mov r11d, dword ptr
[rsi+rax*8+0x8]
mov dword ptr
[rcx+rax*4+0x4],
r11d
mov r11d, dword ptr
[rsi+rax*8+0xc]
mov dword ptr
[rdx+rax*4+0x4],
r11d
cmp r10d, r8d
jl loop
	loop:
	vmovdqu xmm0, xmmword ptr
	[r10+rcx*8]
	vmovdqu xmm1, xmmword ptr
	[r10+rcx*8+0x10]
	vmovdqu xmm4, xmmword ptr
	[r10+rcx*8+0x40]
	vmovdqu xmm5, xmmword ptr
	[r10+rcx*8+0x50]
	vinserti128 ymm2, ymm0,
	xmmword
	ptr [r10+rcx*8+0x20], 0x1
	vinserti128 ymm3, ymm1,
	xmmword
	ptr [r10+rcx*8+0x30], 0x1
	vinserti128 ymm6, ymm4,
	xmmword
	ptr [r10+rcx*8+0x60], 0x1
	vinserti128 ymm7, ymm5,
	xmmword
	ptr [r10+rcx*8+0x70], 0x1
	add rcx, 0x10
	vshufps ymm0, ymm2, ymm3,
	0x88
	vshufps ymm1, ymm2, ymm3,
	0xdd
	vshufps ymm4, ymm6, ymm7,
	0x88
	vshufps ymm5, ymm6, ymm7,
	0xdd
	vmovups ymmword ptr [r9],
	ymm0
	vmovups ymmword ptr [r8],
	ymm1
	vmovups ymmword ptr
	[r9+0x20],
	ymm4
	vmovups ymmword ptr
		loop:
		lea r11, [r10+rcx*8]
		vpxor ymm5, ymm5, ymm5
		add rcx, 0x8
		vpxor ymm6, ymm6, ymm6
		vmovdqa ymm3, ymm0
		vmovdqa ymm4, ymm0
		vpgatherdd ymm5, ymmword ptr
		[r11+ymm2*4], ymm3
		vpgatherdd ymm6, ymmword ptr
		[r11+ymm1*4], ymm4
		vmovdqu ymmword ptr [r9],
		ymm5
		vmovdqu ymmword ptr [r8],ymm6
		add r9, 0x20
		add r8, 0x20
		cmp rcx, rsi
		jl loop

	[r8+0x20],ymm5
	add r9, 0x40
	add r8, 0x40
	cmp rcx, rsi
	jl loop	

ストライド・アクセス・パターンのインテル® AVX ソフトウェア・シーケンスは、複数の要素をロードしてシャッフルできるさらに最適な手法です。

表 15-10 ストライド・アクセス・パターンと AOS および SOA の比較

マイクロアーキテクチャー	スカラー	VPGATHERD	インテル® AVX の VINSRTF128/VSHUFFLEPS
Broadwell†	1X	1.7X	4.8X
Skylake†	1X	2.7X	4.9X

15.16.4.2 隣接したロード

この節では、AOS から SOA への転置を処理する各種実装とハードウェア GATHER 命令の利用を比較します。この場合、AOS データはシーケンシャルにはロードされずインデックス配列が使用されます。

C のコード:

```
for(int i=0;i<len;i++){
  Real_buffer[i] = Complex_buffer[Index_buffer[i]].real;
  Imaginary_buffer[i] = Complex_buffer[Index_buffer[i]].imag;
}
```

例 15-47 非ストライドの AOS から SOA

インテル® AVX2 の GATHERPD	インテル® AVX の VINSRTF128 /UNPACK
loop: vmovdqu ymm1, ymmword ptr [rsi+rdx*4] vpaddd ymm3, ymm1, ymm1 vpaddd ymm14, ymm13, ymm3 vxorpd ymm5, ymm5, ymm5 vmovdqa ymm2, ymm0 vxorpd ymm6, ymm6, ymm6 vmovdqa ymm4, ymm0 vxorpd ymm10, ymm10, ymm10 vmovdqa ymm7, ymm0 vxorpd ymm11, ymm11, ymm11 vmovdqa ymm9, ymm0 vextracti128 xmm12, ymm14, 0x1 vextracti128 xmm8, ymm3, 0x1 vgatherdpd ymm6, ymmword ptr[r8+xmm8*8],ymm4 vgatherdpd ymm5, ymmword ptr[r8+xmm3*8],ymm2 vmovupd ymmword ptr [rcx+rdx*8],ymm5 vmovupd ymmword ptr [rcx+rdx*8+0x20],ymm6 vgatherdpd ymm11,ymmword	loop: movsxd r10, dword ptr [rdx+rsi*4] shl r10, 0x4 movsxd r11, dword ptr [rdx+rsi*4+0x8] shl r11, 0x4 vmovupd xmm0, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0x4] shl r10, 0x4 vinsertf128 ymm2, ymm0, xmmword ptr [r9+r11*1], 0x1 vmovupd xmm1, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0xc] shl r10, 0x4 vinsertf128 ymm3, ymm1, xmmword ptr [r9+r10*1], 0x1 movsxd r10, dword ptr [rdx+rsi*4+0x10] shl r10, 0x4 vunpcklpd ymm4, ymm2, ymm3 vunpckhpd ymm5, ymm2, ymm3 vmovupd ymmword ptr [rcx], ymm4 vmovupd xmm6, xmmword ptr [r9+r10*1] vmovupd ymmword ptr [rax], ymm5 movsxd r10, dword ptr [rdx+rsi*4+0x18] shl r10, 0x4

```
ptr[r8+xmm12*8],ymm7
vgatherdpd ymm10,ymmword
ptr[r8+xmm14*8],ymm9
vmovupd ymmword ptr [rax+rdx*8], ymm10
vmovupd ymmword ptr [rax+rdx*8+0x20],
ymm11
add rdx, 0x8
cmp rdx, r11
jb loop
```

```
vinsertf128 ymm8, ymm6, xmmword ptr
[r9+r10*1], 0x1
movsxd r10, dword ptr [rdx+rsi*4+0x14]
shl r10, 0x4
vmovupd xmm7, xmmword ptr [r9+r10*1]
movsxd r10, dword ptr [rdx+rsi*4+0x1c]
add rsi, 0x8
shl r10, 0x4
vinsertf128 ymm9, ymm7, xmmword ptr
[r9+r10*1], 0x1
vunpcklpd ymm10, ymm8, ymm9
vunpckhpd ymm11, ymm8, ymm9
vmovupd ymmword ptr [rcx+0x20], ymm10
add rcx, 0x40
vmovupd ymmword ptr [rax+0x20], ymm11
add rax, 0x40
cmp rsi, r8
jl loop
```

AOS から SOA へのストライドなしの通常のアクセスパターンで、VINSERTF128 と複数の要素をインターリーブしてパックするインテル® AVX のコードシーケンスは、より適切であると考えられます。

表 15-11 インデックスによる AOS から SOA への転置の比較

マイクロアーキテクチャー	VPGATHERPD	インテル® AVX の VINSRTF128/VUNPCK*
Broadwell†	1X	1.4X
Skylake†	1.3X	1.7X

15.16.5 インテル® MMX® 命令のスループットの制限によるインテル® AVX2 への変換方法

Skylake† マイクロアーキテクチャー・ベースのプロセッサーでは、インテル® MMX® 命令セットの機能性はこれまでの世代から変更されていません。しかし、多くのインテル® MMX® 命令は、これまでのマイクロアーキテクチャーと比べると、1 つのポートでのみ実行されるため、命令のスループットが半分に抑制されます。スループットが抑制されるインテル® MMX® 命令には次のものがあります。

- PADDS[B/W]、PADDUS[B/W]、PSUBS[B/W]、PSUBUS[B/W]
- PCMPGT[B/W/D]、PCMPEQ[B/W/D]
- PMAX[UB/SW]、PMIN[UB/SW]
- PAVG[B/W]、PABS[B/W/D]、PSIGN[B/W/D]

インテル® MMX® 命令のスループット減少への対策として、アセンブリーや組込み関数のコードをインテル® AVX2 命令を使用するように変換することで劇的なパフォーマンスの向上がもたらされます。例 15-48 に、インテル® AVX2 と等価なインテル® MMX® 命令を使用したアセンブリーのシーケンスを示します。Skylake† マイクロアーキテクチャーでは、例 15-48 のインテル® MMX® コードは Broadwell† マイクロアーキテクチャーに対しておよそ半分のスピードで実行されます。

これは、PMAXSW/PMINSW のスループットが単一ポートによる実行制限のため、半分に減少したことによります。同じタスクが等価なインテル® AVX2 シーケンスで実装されると、Skylake† マイクロアーキテクチャー上のインテル® AVX2 コードのパフォーマンスは、Broadwell† マイクロアーキテクチャーで実行されるインテル® MMX® コードの ~3.9 倍となります。

例 15-48 スループットが減少したインテル® MMX® シーケンスをインテル® AVX2 に変換する

インテル® MMX® コード	インテル® AVX2 コード

mov rax, pIn
mov rbx, pOut
mov r8, len
xor rcx, rcx
mov rcx, 8
movq mm0, [rax]
movq mm1, [rax + 8]
movq mm2, mm0
movq mm3, mm1
cmp rcx, r8
jge end
loop:
movq mm4, [rax + 2*rcx]
movq mm5, [rax + 2*rcx + 8]
pmaxsw mm0, mm4
pmaxsw mm1, mm5
pminsw mm2, mm4
pminsw mm3, mm5
add rcx, 8
cmp rcx, r8
jl loop
end:
//リダクション
pmaxsw mm0, mm1
pshufw mm1, mm0, 0xE
pmaxsw mm0, mm1
pshufw mm1, mm0, 1
pmaxsw mm0, mm1
pminsw mm2, mm3
pshufw mm3, mm2, 0xE
pminsw mm2, mm3
pshufw mm3, mm2, 1
pminsw mm2, mm3
movd eax, mm0
mov wordptr [rbx], ax
movd eax, mm2
mov WORD PTR [rbx + 2], ax
emms
	mov rax, pIn
	mov rbx, pOut
	mov r8, len
	xor rcx, rcx
	mov rcx, 32
	vmovdqu ymm0, ymmword ptr [rax]
	vmovdqu ymm1, ymmword ptr [rax + 32]
	vmovdqu ymm2, ymm0
	vmovdqu ymm3, ymm1
	cmp rcx, r8
	jge end
	loop:
	vmovdqu ymm4, ymmword ptr [rax + 2*rcx]
	vmovdqu ymm5, ymmword ptr [rax + 2*rcx + 32]
	vpmaxsw ymm0, ymm0, ymm4
	vpmaxsw ymm1, ymm1, ymm5
	vpminsw ymm2, ymm2, ymm4
	vpminsw ymm3, ymm3, ymm5
	add rcx, 32
	cmp rcx, r8
	jl loop
	end:
	//リダクション
	vpmaxsw ymm0, ymm0, ymm1
	vextracti128 xmm1, ymm0, 1
	vpmaxsw xmm0, xmm0, xmm1
	vpshufd xmm1, xmm0, 0xe
	vpmaxsw xmm0, xmm0, xmm1
	vpshuflw xmm1, xmm0, 0xe
	vpmaxsw xmm0, xmm0, xmm1
	vpshuflw xmm1, xmm0, 1
	vpmaxsw xmm0, xmm0, xmm1
	vmovd eax, xmm0
	mov wordptr [rbx], ax
	vpminsw ymm2, ymm2, ymm3
	vextracti128 xmm1, ymm2, 1
	vpminsw xmm2, xmm2, xmm1
	vpshufd xmm1, xmm2, 0xe
	vpminsw xmm2, xmm2, xmm1
	vpshuflw xmm1, xmm2, 0xe
	vpminsw xmm2, xmm2, xmm1
	vpshuflw xmm1, xmm2, 1
	vpminsw xmm2, xmm2, xmm1
	vmovd eax, xmm2
	mov wordptr [rbx + 2], ax

第 16 章
インテル® TSX の推奨事項

16.1 はじめに

インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) は、ロックベースのプログラミング・モデルを維持する一方、ロックで保護されたクリティカル・セクションのパフォーマンスを向上することを目的としています。

インテル® TSX を利用すると、プロセッサーは、スレッドをロックで保護されたクリティカル・セクションによりシリアル化する必要があるかどうかを動的に判断して、必要な場合にのみシリアル化を行います。これにより、ハードウェアは、Lock Elision (ロックの省略) として知られている手法を用いて、不要な動的同期によって損なわれているアプリケーションの並行性 (コンカレンシー) を生かすことができます。

ロックの省略により、ハードウェアは、開発者が指定したクリティカル・セクション (**トランザクション領域**とも呼ばれる) をトランザクション実行します。実行の際、ロックされた変数はトランザクション領域内で読み取られるだけで書き込まれません (したがって取得されない)。つまり、ロックされた変数はトランザクション領域で変更されないため、並行性を生かすことができます。

トランザクション実行に成功すると、ハードウェアはトランザクション領域内で行われたすべてのメモリー操作が、ほかの論理プロセッサーからは瞬時に起こったように見えるようにします。プロセッサーは、コミットに成功した場合のみ、ほかの論理プロセッサーに見える領域内でアーキテクチャーの更新を行います。このプロセスは**アトミックコミット**とも呼ばれます。トランザクション領域内で行われた変更は、アトミックコミットが行われることでほかの論理プロセッサーに見えるようになります。

成功したトランザクション実行ではアトミックコミットが保証されるため、プロセッサーは明示的な同期を行うことなくプログラマーが指定したコード領域を安全だと推定して実行します。特定の実行で同期が不要だった場合、交差するスレッド間のシリアル化を行うことなく実行をコミットできます。

トランザクション実行が成功しなかった場合、プロセッサーは更新をアトミックにコミットできません。安全だと仮定した実行に失敗すると、プロセッサーは実行をロールバックし、プロセスはトランザクション・アボートと見なされます。トランザクションがアボートすると、プロセッサーは領域で実行された更新をすべて破棄し、安全だと推定した実行が行われなかったように見えるようにアーキテクチャー上の状態を復元し、非トランザクションに実行を再開します。有効なポリシーに応じて、ロックの省略が再試行されるか、処理を進めるため明示的にロックが取得されます。

インテル® TSX では 2 つのプログラム・インターフェイスが用意されています。

- **Hardware Lock Elision** (HLE) は、従来のプロセッサーと互換性のある命令セット拡張 (XACQUIRE および XRELEASE プリフィクス) です。
- **Restricted Transactional Memory** (RTM): 新しい命令セット・インターフェイス (XBEGIN および XEND 命令を包括) です。

従来のハードウェアでインテル® TSX 対応のソフトウェアを実行する場合は、HLE インターフェイスを使用してロックの省略を実装します。一方、従来のハードウェアに対応する必要がなく、より複雑なロック・プリミティブを扱う場合は、インテル® TSX の RTM インターフェイスによりロックの省略を実装します。新しい命令を使用する後者のケースでは、トランザクション実行が行われない場合に備えて、開発者はトランザクション・アボートが生じた後に実行する (省略されたロックを取得するコードを含む) 非トランザクション・パスを常に提供する必要があります。

さらに、インテル® TSX には、論理プロセッサーがトランザクション実行しているかどうかをテストする XTEST 命令と、トランザクション領域をアボートする XABORT 命令も用意されています。

プロセッサーは、さまざまな理由によりトランザクション実行をアボートします。最も多い原因は、トランザクション実行している論理プロセッサーと別のプロセッサー間のデータアクセス競合によるものです。このようなアクセス競合はトランザクション実行の成功の妨げとなります。トランザクション領域内から読み取られたメモリーアドレスによりトランザクション領域の**読み取りセット**が構成され、トランザクション領域内に書き込まれたアドレスによりトランザクション領域の**書き込みセット**が構成されます。インテル® TSX は、キャッシュラインの粒度で読み取りセットと書き込みセットを維持します。RTM を使用するロックの省略では、明示的にロックを取得する別のスレッドがある場合でもトランザクション実行するスレッドが正しく動作するように保証するため、省略されるロックのアドレスを読み取りセットに追加する必要があります。

別の論理プロセッサーがトランザクション領域の書き込みセットの一部の場所で読み取りを行うか、トランザクション領域の読み取りセットまたは書き込みセットの一部の場所で書き込みを行うと、データアクセス競合が発生します。これは**データ競合**と呼ばれます。インテル® TSX は、キャッシュライン単位でデータ競合を検出するため、同じキャッシュラインに配置された無関係なデータの場所も競合として検出されます。トランザクション・アボートは、トランザクション・リソースの制限により発生することもあります。例えば、領域でアクセスしたデータ量が実装固有の処理能力を超えた場合です。CPUID や I/O などの命令は、実装により常にトランザクション実行をアボートすることがあります。

インテル® TSX のインターフェイスに関する詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 16 章で確認できます。

以降のセクションでは、インテル® TSX 命令を使用するソフトウェア開発者向けのガイドラインを説明します。ガイドラインでは、プリフィクス・ヒント (HLE) や新しい命令 (RTM) を通じて、ロックで保護されたクリティカル・セクションの並行性を生かせるようにロックを省略するインテル® TSX 命令の使用法に注目します。インテル® TSX 命令はロックの省略以外にも利用できますが、それらの使用法はここでは説明しません。

以後**ロックの省略**という用語は、ロックを省略する **HLE** ベースまたは **RTM** ベースの実装のいずれかを指します。

16.1.1 最適化の概略

ここからは、インテル® TSX 命令を使用してロックを省略する際に、マルチスレッド・アプリケーションを最適化およびチューニングするための推奨事項を示します。インテル® TSX は、ロックを取得した後のアプリケーションの動作を見逃しがちなマイクロカーネルの代わりに、アプリケーションのパフォーマンスを向上させます (16.2 節「アプリケーション・レベルのチューニングと最適化」を参照)。この後のセクションでは、インテル® TSX を使用してロックの省略を行う同期ライブラリーを作成する方法 (16.3 節「インテル® TSX 対応の同期ライブラリーの開発」を参照)、インテル® TSX のパフォーマンス監視機能を効果的に使用する方法 (16.4 節「インテル® TSX のパフォーマンス監視サポートを利用する」を参照)、そして初期実装のパフォーマンス・ガイドライン (16.5 節「パフォーマンスのガイドライン」を参照) についても述べていきます。

最初に、すべてのクリティカル・セクションのロックを有効化してから、問題のあるクリティカル・セクションを特定することを推奨します。この「ボトムアップ」アプローチによってアプリケーションの評価とチューニングが単純化され、開発者は適切なクリティカル・セクションに注目できます。インテル® TSX のチューニングに関連するリソースは、http://www.intel.com/software/tsx (英語) で入手できます。

16.2 アプリケーション・レベルのチューニングと最適化

アプリケーションは通常、**同期ライブラリー**を利用してクリティカル・セクションに関連するロック取得とロック解放機能を実装しています。これらのアプリケーションでインテル® TSX ベースのロックの省略を活用する最も簡単な方法は、インテル® TSX 対応の同期ライブラリーを利用することです。既存のライブラリーがインテル® TSX 命令 (16.2.1 節を参照) の利点を活用できるかもしれません。既製品で、インテル® TSX 対応のライブラリーが利用できない場合、16.3 節「インテル® TSX 対応の同期ライブラリーの開発」で、インテル® TSX 未対応のロック・ライブラリーをインテル® TSX 命令を使うように拡張する方法を示しています。インテル® TSX 対応の同期ライブラリーと従来の同期ライブラリーは、互換性を持って使用できます。

これらのライブラリーを使用するアプリケーションは、アプリケーションを変更することなくインテル® TSX を使用できますが、トランザクション実行のコミット率を増加させ、トランザクション・アボートによる無駄な実行サイクルを抑える基本的なチューニングとプロファイリングを行うことでパフォーマンスを向上できます。チューニングでは、最初にプロファイリング・ツール (16.4 節「インテル® TSX のパフォーマンス監視サポートを利用する」を参照) を用いてアプリケーションのトランザクション動作の特性を把握することを推奨します。プロファイリング・ツールは、ハードウェアに実装されているパフォーマンス監視機能とサンプリング機能を利用して、アプリケーションのトランザクション動作に関する詳細な情報を提供します。ツールは、パフォーマンス・モニタリング・カウンターやプリサイス・イベント・ベース・サンプリング (PEBS) メカニズムなどのプロセッサー機能が使用します。詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 18 章を参照してください。

インテル® TSX 対応の同期ライブラリーを使用するアプリケーションは、従来の同期ライブラリーを使用する場合と同じ動作になる必要がありますが、インテル® TSX によりレイテンシーが短縮されてスレッド間の同期が以前よりも速くなるため、コードの潜在的なバグが表面化する可能性があります。

16.2.1 既存のインテル® TSX 対応ロック・ライブラリー

このセクションでは、ロックの省略のためすでにインテル® TSX に対応している既製品のライブラリーについてまとめています。これらは完全なものではなく、2015 年前半時点のスナップショットを示しています。ここで紹介するすべてのライブラリーが、完全にチューニングされているわけではありません。

16.2.1.1 プログラムの変更なしにロックの省略を可能にするライブラリー

- Linux* では、GNU* glibc 2.18 で PTHREAD_MUTEX_DEFAULT タイプの pthread ミューテックスのロック省略のサポートが追加されました。glibc 2.19 では読み取り/書き込みミューテックスのロック省略のサポートが加えられました。ロックの省略が有効であるかどうかは、ライブラリーのコンパイル時に --enable-lock-elision=yes パラメーターが設定されているかどうかに依存します。
- Java* JDK 8u20 以降では、-XX:+UseRTMLocking オプションが有効である場合に同期セクションで適応型の省略がサポートされます。
- インテル® Composer XE 2013 SP1 以降では、OpenMP* の omp_lock_t 向けのロックの省略がサポートされています。ロックの省略を有効にするには、"export KMP_LOCK_KIND=adaptive" を設定します。

16.2.1.2 プログラムの修正を必要とするライブラリー

- インテル® スレッディング・ビルディング・ブロック (インテル® TBB) では、speculative_spin_rw_mutex でロックの省略をサポートしています。この新しいロックタイプを使用するようにプログラムを変更する必要があります。
- gcc 4.8 以降では、インテル® TSX によるソフトウェア・トランザクション・メモリーの実装の加速をサポートします。
- Concurrency Kit は、ck_elide ラッパーでスピンロックのロック省略をサポートします。
- DPDK ライブラリーは、スピンロックと read-write ロックのロック省略をサポートします ("_tm" サフィックス付きの lock/unlock 呼び出しを介して)。

16.2.2 初期のチェック

いくつかの簡単なチェックを行うことで、チューニングの労力を軽減できます。特に、優れたライブラリー実装の利用とクリティカル・セクション内部の統計収集の扱いについては重要です。

- インテル® TSX 対応の優れた同期ライブラリーを使用します。アプリケーションは、インテル® TSX 対応の同期ライブラリーを直接使用する必要があります。インテル® TSX 対応ライブラリーの上に独自のカスタム・ライブラリーを実装すると、トランザクション領域を識別できないことがあります。インテル® TSX 向けに同期ライブラリーを作成する方法は、16.3 節を参照してください。
- クリティカル・セクション内部の統計を収集しないようにします。クリティカル・セクション (および同期ライブラリー自身) は共有のグローバル統計カウンターを使用することがありますが、これらのカウンターはデータ競合

やトランザクション・アボートを引き起こします。アプリケーションには通常、これらの統計収集を無効にするフラグが用意されているので、初期のチューニング段階でこれを無効にしておくと、本来のデータ競合に注目しやすくなります。

16.2.3 アプリケーションの実行とプロファイル

マルチスレッド・アプリケーションにおけるスレッドの相互作用を視覚化することは困難です。最初のステップでは、インテル® TSX 対応の同期ライブラリーを使ってアプリケーションを実行し、パフォーマンスを測定します。次に、プロファイリング・ツールで結果を解析します。つまり、プロファイリング・ツールによりトランザクション実行されたアプリケーション・サイクルを測定し、アプリケーションのトランザクション実行の割合を把握すべきです (16.4 節を参照)。

トランザクション実行サイクルの比率が低くなる要因はいくつかあります。

アプリケーションがクリティカル・セクションで同期をほとんど使用していない。この場合、ロックの省略の利点は得られません。

- アプリケーションの同期ライブラリーがすべてのプリミティブにインテル® TSX を使用していない。アプリケーションが、クリティカル・セクションのロックの一部に内部カスタム関数やカスタム・ライブラリーを使用している場合、この問題が発生します。これらのロックの実装をトランザクション実行に変更する必要があります (16.4.2 節を参照)。
- アプリケーションが、ロックの省略に対応した同期ライブラリーで提供されるものとは異なる、より高いレベルのロック構造 (本書ではメタロックと呼んでいます) を使用している。このような場合、構造をロックの省略に対応させる必要があります (16.3.7 節を参照)。
- プログラムが LOCK プリフィクス命令をクリティカル・セクション以外で使用している。この場合、アルゴリズムがトランザクション実行に対応していない限り、インテル® TSX を利用する利点はありません。このようなロック目的以外での使用法は本書では触れていません。

インテル® TSX パフォーマンス・チューニングの「ボトムアップ」アプローチでは、チューニング手法を次のようにモジュール化できます。

- ロックをすべて特定します。
- ロックをすべて省略するインテル® TSX 同期ライブラリーを使用してプログラムを (変更せずに) 実行します。
- プロファイリング・ツールでトランザクション実行を測定します。
- 必要に応じて、トランザクション・アボートの原因を調べて対処します。

16.2.4 トランザクション・アボートを最小限に抑える

データ競合はキャッシュ・コヒーレンス・プロトコルを介して検出されます。データ競合はトランザクションのアボートにつながります。初期の実装では、データ競合を検出するスレッドはトランザクションをアボートします。

HLE ベースのトランザクション実行がトランザクションのアボートに遭遇すると、現在の実装ではハードウェアは HLE 実行を開始した XACQUIRE プリフィクス付きの命令を再開しますが、XACQUIRE プリフィクスは無視されます。その結果、ロックの省略なしで再実行し、明示的にロックが取得されます。RTM ベースのトランザクション実行がトランザクションのアボートに遭遇すると、現在の実装ではハードウェアは XBEGIN 命令で提供される命令アドレスから再開します。

初期のインテル® TSX 実装は制限付きの入れ子をサポートします。RTM は 7 階層の入れ子レベルをサポートします。HLE は 1 階層の入れ子レベルをサポートします。これは実装依存であり、同じ世代のプロセッサー・ファミリーの今後の実装では変更される可能性があります。

『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 16 章では、トランザクションのアボートを引き起こすさまざまな原因が詳しく説明されています。インテル® TSX 命令とプリフィ

クスに関する詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2B』で確認できます。

プロファイリング・ツールのパフォーマンス監視機能を使用して、アボートしたトランザクション実行に費やされたサイクル数を計算できます。すべてのトランザクション・アボートがパフォーマンスを低下させるとは限りません。別のスレッドが取得したロックを待機するために実行がストールしている場合や、トランザクション実行にデータ・プリフェッチ効果がある場合もあります。

プロファイリング・ツールは、PEBS を用いてアボートしたトランザクション領域を認識し、相対的なコストに関する情報を提供します (16.4 節を参照)。次に、トランザクション・アボートの一般的な原因と対策を示します。

チューニングの推奨事項 4: プロファイリング・ツールを用いて、パフォーマンス低下の多くの原因であるトランザクション・アボートを識別します。

トランザクション・アボートの原因には次のようなものがあります。

- データアクセス競合によるアボート。
- ロック変数の競合によるアボート。
- リソースバッファーの超過によるアボート。
- HLE インターフェイス固有の制限によるアボート。
- 『Intel® Architecture Instruction Set Extensions Programming Reference』の第 8 章で説明されているその他のアボート。

16.2.4.1 データ競合によるトランザクション・アボート

別の論理プロセッサーがトランザクション領域の書き込みセットの一部の場所で読み取りを行うか、トランザクション領域の読み取りセットまたは書き込みセットの一部の場所で書き込みを行うと、データ競合が発生します。初期の実装では、データ競合は、キャッシュライン単位で処理されるキャッシュ・コヒーレンス・プロトコルにより検出されます。

ここでは、トランザクション・アボートの原因となるさまざまなデータ競合について説明します。アプリケーションにもともと内在する競合は回避できるものもあります。

フォルス・シェアリングによる競合

異なる変数を同じキャッシュライン (64 バイト) へ配置して複数のスレッドで個別に書き込みを行うと、フォルス・シェアリングが発生します。ハードウェアはキャッシュライン単位でデータ競合をチェックするため、アドレスがオーバーラップしていなくてもこれらの変数は同じアドレスを持つように見え、不要なトランザクション・アボートを引き起こします。

このフォルス・シェアリングの問題はインテル® TSX 固有の問題ではありません。キャッシュ・コヒーレンス・プロトコルはシステム内でキャッシュラインを移動しますが、これには大きなオーバーヘッドが伴います。少なくとも変数の 1 つが異なるスレッドによって頻繁に書き込まれる場合、異なる変数を同じキャッシュラインに配置するべきではありません。

チューニングの推奨事項 5: 競合する変数はパディングを追加して別々のキャッシュラインに配置します。

チューニングの推奨事項 6: 可能な場合、フォルス・シェアリングが最小限になるようにデータ構造を再構成します。

真の共有による競合

このトランザクション・アボートはフォルス・シェアリングによるものではなく、競合データが実際に共有されている場合に発生しますが、ソフトウェアを変更することで軽減できることがあります。次のセクションで、このような競合への対処方法を説明します。

統計管理による競合

一部のソフトウェアは、複数のスレッド間で共有されるグローバル統計カウンターを使用しています。例えば、クリティカル・セクションのロックを取得したか、または取得された回数をカウントする同期ライブラリーなどがあります。また、複数のスレッドによってアクセスされるグローバル変数やオブジェクトの数をカウントするような場合も考えられます。これらの統計はトランザクション・アボートの原因となります。このようなトランザクション・アボートに対処するには、統計の使用目的を理解する必要があります。

プログラムのロジックに影響しない場合は、統計を無効にしたり、条件付きでスキップすることができます。例えば、クリティカル・セクションのシリアル実行の回数をカウントするケースについて考えてみます。ロックが省略されない場合、実行はすでにシリアル化されているため、クリティカル・セクション内で統計が更新されます。ロックが省略されている場合、省略されているロックをカウントしても意味がありません。この統計が役立つのは、ロックが省略されなかった場合だけです。その場合、ソフトウェアは統計を利用してシリアル化のレベルを追跡できます。ロックを省略しない実行の場合 (つまり、シリアル化されている場合) のみ、XTEST 命令で統計を更新できます。これらの統計はプログラムの開発中にのみ有用であり、製品版では無効にできます。

しかし、場合によっては、統計を無効にしたり、スキップできないこともあります。これらの統計を論理スレッドごとに維持することで、(フォルス・シェアリングを回避しつつ) 不要なトランザクション・アボートを回避できます。このアプローチでは、統計の読み取り時にすべてのスレッドの結果をまとめる必要があります。これはまた、スレッド間の通信を最小限に抑えることで、インテル® TSX 命令を利用しなくてもアプリケーションのパフォーマンスを向上できます。

このほかにも、クリティカル・セクションの外に統計を移動したり、アトミック操作で統計を更新する方法があります。これらのアプローチではトランザクション・アボートは軽減されますが、アトミック操作によるオーバーヘッドや、通信のオーバーヘッドは軽減されません。

チューニングの推奨事項 7: グローバル統計は操作ごとに更新する代わりにサンプリングすることもできます。

チューニングの推奨事項 8: クリティカル・セクションでは不要な統計は避けます。

チューニングの推奨事項 9: クリティカル・セクションでは統計をスレッド単位で維持することを検討します。

プログラマーは、共有グローバル統計によるトランザクション・アボートを軽減するため最適なアプローチを選択する必要があります。初期テストでは、すべてのグローバル統計を無効にすると、グローバル統計が問題かどうかを判断しやすくなります。

データ構造内の資源管理による競合

データ構造内の資源管理による操作も、データ競合を引き起こす原因の 1 つです。例えば、データ構造は、構造内のエントリー数を追跡するために変数を持つことがあります。これは、統計カウンターと同様の影響があり、不要なトランザクション・アボートを引き起こします。

状況によっては (例えば、ヒープの再構成を引き起こすエントリーの数など)、アトミック更新を使用することでクリティカル・セクション外に更新を移動できます。

また別のケースでは、データ競合が発生するタイミングを減らすアプローチを採用できることもあります (16.2.4.1 節「データ競合によるトランザクション・アボート」を参照)。

メモリー割り当ての競合

クリティカル・セクションでメモリー割り当てが行われることがあります。その場合、スレッドのローカル領域にフリーリストを保持し、割り当て済みメモリーのフォルス・シェアリングを回避するスレッド対応のメモリー・アロケーション・ライブラリーを使用することを推奨します。

条件付き書き込みリダクションによる競合

一般的なソフトウェア・パターンには、値がほとんど変わらない共有変数やフラグの更新が含まれます。そのような操作は、(値が同じ場合でも) キャッシュラインを更新するためキャッシュラインへの書き込み許可が必要になります。そのような操作では、共有変数へアクセスするほかのスレッドでトランザクション・アボートが発生します。このようなデータ競合は、値が同じ場合はストアを実行せずに必要な場合のみ更新を行うことで回避できます (例 16-1 を参照)。

例 16-1 条件付き更新とデータ競合の軽減

```
state = true; // 毎回更新
var |= flag;
```

```
if (state != true) state = true;
if (!(var & flag)) var |= flag;
```

データ競合範囲の軽減

ここで示す手法では、頻発する実際のデータ競合によるトランザクション・アボートを回避できないケースがあります。その場合、データ競合が発生する範囲を短くすることを目標にすべきです。例えば、実際に競合するメモリーアクセスをクリティカル・セクションの最後に移動してタイミングを短縮します。

16.2.4.2 トランザクション・リソースの制限によるトランザクション・アボート

インテル® TSX 実装は、共通のトランザクション領域を実行する十分なリソースを提供しますが、トランザクション領域の実装の制限やデータ使用量によりトランザクションがアボートすることがあります。アーキテクチャーがトランザクション実行に必要なリソースを保証したり、トランザクション実行の成功を保証することはありません。

プロセッサーは、L1D (L1 データ) キャッシュ内の**読み取りセット**アドレスと**書き込みセット**アドレスの両方を追跡します。

読み取りセットアドレスの追い出し (evisction) は、第 2 レベルの構造でそのラインが追跡される可能性があるため、即座にトランザクション・アボートにつながるとは限りません。現在の実装では、第 2 レベルの構造は追い出された読み取りセットアドレスを確立的に追尾します。その結果、他のスレッドからのアクセスよって偽りの一致が生じ、必要がないトランザクションのアボートが発生する可能性があります。このような偽りの競合の割合は、異なるスレッドからのアドレスストリームと正確なハードウェア実装によります。Broadwell† マイクロアーキテクチャーの実装では第 2 レベルの構造が改善されています。偽りの競合の割合は、将来の実装では減少すると考えられます。

アーキテクチャーはバッファリングに関し何も保証しないため、ソフトウェアはどのような保証も期待してはいけません。

Haswell†、Broadwell† および Skylake† マイクロアーキテクチャーの L1 データキャッシュの連想性は 8 です。これはこの実装では、同じキャッシュセットに割り当てられる 9 番目の位置への書き込みを実行するトランザクションがアボートすることを意味します。しかし、このマイクロアーキテクチャーの実装が、同じセットへのわずかなアクセスではアボートしないことが保証されることを意味するものではありません。

さらに、インテル® ハイパースレッディング・テクノロジーが有効である構成では、L1 キャッシュが同じコアの 2 つの論理プロセッサー間で共有されるため、同じコアの一方の論理プロセッサーの操作が追い出しを引き起こし、有効な読み取りと書き込みセットのサイズを大幅に減少させます。

プロファイラーを利用して、処理能力の制限により頻繁にアボートするトランザクション領域を特定できます (16.4.4 節を参照)。ソフトウェアは、そのようなトランザクション領域内でデータの過剰なアクセスを避けるべきです。一般に、大量のデータアクセスには時間がかかるため、そのようなアボートは多くの実行サイクルを無駄にします。

クリティカル・セクションでのデータ使用量はアルゴリズムの変更により軽減できることもあります。例えば、ソートされた配列には、リニア検索ではなくバイナリー検索を利用することで、クリティカル・セクション内でアクセスするアドレスの数を減らすことができます。

トランザクション領域の特定のコードパスで大量のデータにアクセスすることが想定される場合、アルゴリズムで (XABORT 命令により) 早期にトランザクション・アボートを強制したり、無効化されたロックを取得してアボートせずに非トランザクション実行に遷移できます (16.2.6 節を参照)。

トランザクション領域内の動作が原因で、処理能力によるアボートが発生することがあります。例えば、アプリケーションが初めてダイナミック・ライブラリー関数を呼び出す場合、ソフトウェア・システムはダイナミック・リンカーを呼び出してシンボルを解決する必要があります。初めての呼び出しがトランザクション領域内で行われる場合、大量のデータアクセスが生じて通常アボートが発生します。このアボートは、ダイナミック・ライブラリー関数が初めて呼び出されるときにのみ発生します。これが頻繁に発生する場合は、非トランザクション実行中にトランザクション実行パスが使用されていないことが原因である可能性が高いと考えられます。

16.2.4.3 ロック省略固有のトランザクション・アボート

データ競合に加えて、ロック自体の競合によってトランザクション・アボートが生じることもあります。その場合、クリティカル・セクションのトランザクション実行と非トランザクション実行がオーバーラップするタイミングを検出する必要があります。インテル® TSX を利用してロックを省略する場合、ロックは読み取りセットに追加されます。これは、HLE では自動的に行われますが、RTM ではソフトウェア・ライブラリーで明示的に行う必要があります。これにより、明示的にロックを取得するほかのスレッドとの競合を確認できます。これは、アボートし、再開して、最終的にロックを取得するというトランザクション実行の自然な流れの一部です。

HLE と RTM を利用するロックの省略では、このようにロック変数に対する二次的な競合が原因でアボートが発生することがよくあります。トランザクション・スレッドのアボートは通常の非トランザクション実行に遷移し、その過程で明示的にロックを取得します。このロックの取得により、ほかのトランザクション実行スレッドをシリアル化しなければならなくなりアボートします。

RTM の場合、ロックが解放されるのを待って取得を試みることで、フォールバック・ハンドラーはこれらの二次的なアボートを減らせる可能性があります (16.3.5 節を参照)。

16.2.4.4 HLE 固有のトランザクション・アボート

一部のトランザクション・アボートは、HLE ベースのロックの省略でのみ発生します。これらについては、次のセクションで説明します。

サポートされていないロックの省略パターン

トランザクション実行を正常にコミットするには、ロックがある特性を持ち、ロックへのアクセスが特定のガイドラインに従っていなければなりません。XRELEASE プリフィクス命令では、ロックの省略の値を、対応する XACQUIRE プリフィクス命令 (ロックの取得) が実行される前の値に復元する必要があります。

これにより、ハードウェアは書き込みセットに追加することなく、安全にロックを省略できます。XRELEASE プリフィクス命令 (ロックの解放) と XACQUIRE プリフィクス命令 (ロックの取得) のデータサイズとデータアドレスは、どちらも一致していなければなりません。また、ロックはキャッシュ境界をまたぐことはできません。例えば、アドレス A への XACQUIRE プリフィクス命令 (ロックの取得) の後に別のアドレス B への XRELEASE プリフィクス命令 (ロックの解放) がある場合、アドレス A とアドレス B は一致しないためアボートします。

サポートされていない HLE 領域内のロック変数へのアクセス

通常、ロック変数は HLE 領域からアボートせずに読み取ることができます。しかし、ある種の一般的でないアクセスはトランザクション・アボートを引き起こすことがあります。例えば、省略されたロック変数へのアライメントされていないアクセスや一部がオーバーラップするアクセスは、トランザクション・アボートとなります。その場合、省略されたロック変数へ適切にアライメントされたアクセスが行われるように、ソフトウェアを変更すべきです。

HLE ベースのトランザクション領域内で省略されたロックへ書き込みを行う場合は、必ず XRELEASE プリフィクス命令を使用します。そうしないと、トランザクション・アボートが発生します。

16.2.4.5 その他のトランザクション・アボート

プログラマーは、トランザクション領域内ですべての命令を安全に使用でき、すべての特権レベルでトランザクション領域を使用できます。しかし、一部の命令は常にトランザクション実行をアボートさせ、実行を非トランザクション・パスに安全かつシームレスに遷移させます。そのようなトランザクション・アボートは、プロファイリング・ツールによって収集される PEBS レコードのトランザクション・アボート・ステータスには命令アボート (Instruction Aborts) として表示されます (16.4 節を参照)。

この命令の一覧は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』に記載されています。一般的な例として、x87 およびインテル® MMX® 命令のアーキテクチャー・ステートに対する操作、セグメント、制御、デバッグレジスターを更新する操作、I/O 命令、SYSENTER、SYSCALL、SYSEXIT、SYSRET などのリング遷移を引き起こす命令が挙げられます。

プログラマーは、トランザクション領域内では、x87/インテル® MMX® 命令の代わりに、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)/インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) 命令を使用するべきです。ただし、トランザクション領域内でインテル® SSE 命令とインテル® AVX 命令を混在させる場合は注意が必要です。XMM レジスターにアクセスするインテル® SSE 命令と YMM レジスターにアクセスするインテル® AVX 命令の混在使用は、トランザクション・アボートの原因となります。VZEROUPPER 命令もアボートの原因になることがあるため、この命令はクリティカル・セクションの前に移動すべきです。

特定の 32 ビット呼び出し規約は、引数と戻り値の受け渡しに x87 ステートを使用しています。プログラマーは別の呼び出し規約を利用するか、あるいは関数をインライン展開することを検討すべきです。long double 型などの一部のデータ型は x87 命令を使用するため避けるべきです。

命令ベースによるアボートに加えて、各種ランタイムイベントによりトランザクション実行がアボートされるケースもあります。

トランザクション実行中に非同期イベント (NMI、SMI、INTR、IPI、PMI、その他) が発生すると、トランザクション実行はアボートされ、非トランザクション実行に移行します。そのようなアボートの発生率は、オペレーティング・システムのバックグラウンド・ステートに依存します。例えば、オペレーティング・システムのタイマーを用いて割り込みをかけると、トランザクション・アボートの原因になることがあります。

トランザクション実行中に同期例外イベント (#BR, #PF, #DB, #BP/INT3 など) が発生すると、トランザクション実行はコミットされず、非トランザクション実行が必要になります。これらのイベントは、発生しなかったように処理されます。

ページフォルト (#PF) は通常、プログラムの起動時に発生することが最も多いイベントです。ページが初めて割り当てられるため、この間にトランザクション領域がアボートする確率が高くなります。このようなアボートは、プログラムが定常状態になると発生しなくなります。ただし、実行時間の非常に短いプログラムではこのアボートが頻発することがあります。同様の現象は、大きなメモリー領域が割り当てられた際にも起こります。

トランザクション領域内のメモリーアクセスを行うには、プロセッサーが参照するページ・テーブル・エントリーの Accessed フラグと Dirty フラグをセットする必要があります。この処理は、ページへの初回アクセス時と書き込み時

に行われます。現在の実装では、これらの処理はトランザクション・アボートを引き起こします。非トランザクション・モードで再実行するとこれらのビットが適切に更新され、通常、後続のトランザクション実行ではこれが原因のトランザクション・アボートが発生しません。このトランザクション・アボートは、PEBS レコードのトランザクション・アボート・ステータスには命令アボート (Instruction Aborts) として表示されますが、頻発しない限り特に注意する必要はありません。

さらに、実装固有の条件やバックグラウンド・システム・アクティビティーがトランザクション・アボートの原因になることもあります。例えば、システムのキャッシュ階層によるアボート、プロセッサーのマイクロアーキテクチャー実装との微妙な相互作用、システムタイマーによる割り込みなどが挙げられます。このようなアボートは、インテル® TSX を使用するロックの省略ではごくまれです。

チューニングの推奨事項 10: プログラム起動時のトランザクション領域では、定常状態よりも高い確率でアボートが発生します。

チューニングの推奨事項 11: バックグラウンド・アクティビティーにより、まれにオペレーティング・システムのサービスがトランザクション・アボートの原因になることがあります。

16.2.5 トランザクション実行専用のコードパスの使用

インテル® TSX を使用するプログラマーは、非トランザクション・フォールバック・パスとは異なる、トランザクション領域でのみ実行されるコードを記述できます。これは RTM (フォールバック・ハンドラーを利用) や HLE と XTEST 命令を使用することで可能になります。

ただし、トランザクション実行時に実行されるコードが、非トランザクション実行時に実行されるコードと大きく異なる場合は注意が必要です。(命令およびデータの) ページフォルトなどのイベントや、アクセスビットとダーティービットを変更するページ操作は、トランザクション実行を繰り返しアボートする可能性があります。そのため、非トランザクション・フォールバック・パスでもこれらの操作が実行されるようにしなければなりません。そうしないと、そのトランザクション領域はいつまでも成功できません。ロックの省略により、アプリケーションのトランザクション・パスと非トランザクション・パスは同じになり、唯一の違いは同期ライブラリーだけなので、これは一般的な問題ではありません。

XTEST 命令は、トランザクション実行時にアボートの原因になる可能性の高い不要なコードシーケンスをスキップするのに利用できます。また、巻き戻された (unwind) コードや実際にロックを取得したときのみ必要なエラー処理コード (デッドロック検出など) をスキップして最適化を実装するのにも利用できます。

チューニングの推奨事項 12: トランザクション実行専用のコードパスは単純にしてインライン展開します。

チューニングの推奨事項 13: トランザクション実行専用のコードパスは最小限に抑えます。

16.2.6 アボートが頻発するトランザクション領域またはトランザクション・パスへの対応

一部のトランザクション領域は高い確率でアボートし、これまでに示した手法ではうまくアボートを減らすことができないことがあります。その場合は、以降のセクションに記載する手法を検討してみると良いでしょう。

16.2.6.1 アボートしないで非省略実行へ遷移する

システムコールや I/O 操作など、トランザクション・アボートが避けられないこともあります。

トランザクション実行パスでアボートが避けられない場合、ロックの省略に RTM を利用しているソフトウェアでは、非トランザクション実行に遷移しロックの取得を試みて、成功した場合はトランザクション実行をコミットすることができます。例 16-2 に簡単な例を示します。実際のコードでは入れ子処理などの追加が必要になります。

例 16-2 アボートしないで非省略実行へ遷移する

```
/* RTM トランザクションでトランザクション実行がアボートした場合は */
/* 非トランザクション実行でロックを取得する */
<オリジナルのロック取得コード>
_xend(); /* コミット */
/* アボート処理を行う */
```

16.2.6.2 早期のアボート強制

トランザクション領域内のアボートを引き起こすパスで、早い段階に PAUSE 命令または XABORT 命令を挿入し、強制的にトランザクション・アボートの発生を試みることができます。これにより、破棄が必要な作業を最小限に抑えることが可能です。

16.2.6.3 選択したロックを省略しない

ロックの省略によりアプリケーションのパフォーマンスが低下し、どの手法でもトランザクション・アボートを軽減できない場合、トランザクション・アボートの発生率が高く、コストの高い特定のロックの省略をソフトウェアで無効にできます。その場合、アボートの発生率が高くてもパフォーマンスが向上することがあるため、アプリケーション・レベルのパフォーマンス・メトリックを用いて検証すべきです。

16.3 インテル® TSX 対応の同期ライブラリーの開発

このセクションでは、インテル® TSX 命令を使用して、既存の同期ライブラリーをロックの省略に対応させる方法を示します。

16.3.1 HLE プリフィクスの追加

プログラマーは、クリティカル・セクションを保護するロックの取得に使用する命令の前に XACQUIRE プリフィクスを使用します。プログラマーは、クリティカル・セクションを保護するロックの解放に使用する命令の前に XRELEASE プリフィクスを使用します。この命令にはロックに対する書き込みも含まれます。

この命令により、同じロックの XACQUIRE プリフィクスが付加されたロック取得操作の前の値にロックの値が戻された場合、プロセッサーはロックの解放に関連付けられている外部書き込み要求を省略し、書き込みセットにロックのアドレスを追加しません。

16.3.2 省略に適したクリティカル・セクションのロック

ライブラリー自体がデータ競合の原因にならないようにする必要があります。ライブラリーのデータ競合の一般的な例を次に示します。

- ロックの所有権フィールドの競合
- ロック関連の統計の競合

ロックの省略に HLE を利用する場合、プログラマーは既存のコードパスにトランザクション実行を追加する必要があります (HLE では、ロックが省略される場合も、されない場合も実行されるコードパスは同じであるため)。また、共有された場所への書き込み操作は、そのロック変数に対するロックの取得/解放命令によってのみ行われることを確認すべきです。共有された場所へのほかの書き込み操作は、通常、ロック省略ライブラリーを利用して共通のロックを省略する 2 つのスレッド間でデータ競合を引き起こします。これは、複数のスレッドを使って共有ロックで保護された空のクリティカル・セクションを繰り返し実行するテストによって簡単に特定できます。

16.3.3 ロックの省略における HLE または RTM の使用

プロセッサーが HLE 拡張と RTM 拡張をサポートしているかどうかは、CPUID 情報によって判断できます。ただし、HLE プリフィクス (XACQUIRE と XRELEASE) は、プロセッサーが HLE をサポートしているかどうかを確認しなくても使用できます。HLE をサポートしていないプロセッサーはこれらのプリフィクスを無視し、トランザクション実行に入らずにコードを実行します。一方、RTM 命令 (XBEGIN、XEND、XABORT) を使用するアプリケーションは、プロセッサーが RTM 命令をサポートしているかどうかを確認する必要があります。RTM 命令をサポートしていないプロセッサーで実行すると、#UD (未定義オペコード) 例外が発生します。XTEST 命令も、CPUID 情報をチェックして HLE か RTM のいずれかがサポートされていることを確認する必要があります。どちらもサポートされていない場合、この命令も #UD 例外を引き起こします。繰り返し確認しなくても済むように、CPUID 情報をある変数に格納しておくと良いでしょう。

HLE では、トランザクション実行中のプロセッサーがクリティカル・セクションでロックの値を読み取ると、プロセッサーがロックを取得したように見えます (トランザクション実行のものではない値が返されます)。そのため、HLE 実行と HLE プリフィクスなしの実行が機能的に同じになります。

RTM インターフェイスを利用して、プログラマーはより複雑な同期アルゴリズムを記述したり、トランザクション・アボート後の再試行ポリシーを制御することができます。RTM ベースのロック実装では、複数のコードパスを持つラッパーとして実装することが推奨されます。つまり、1 つのパスで RTM ベースのロックを実行し、別のパスで RTM ベースではないロックを実行します (16.3.4 節を参照)。通常、RTM ベースではないロックのコードを変更する必要はありません。一度だけ再試行するプリミティブを使うことでパフォーマンスが向上する可能性があります。このプリミティブにより、スレッドはロックが解放された後にロックの省略を再試行できます。

RTM 命令は明示的にロックに関連付けられていないため、ロックの省略に RTM 命令を使用するソフトウェアは、トランザクション領域内でロックの状態を確認し、ロックがフリーの場合のみトランザクション実行を継続すべきです。さらに、ロックがフリーでないときの再試行ポリシーも定義する必要があります。

HLE との違いは、RTM ベースのクリティカル・セクション内でロックを読み取ると、ロックはフリーのままであり、取得されていないかのように見えることです。そのため、ロックの値を返すライブラリー関数は、トランザクション実行をアボートして非トランザクション実行の値を返す必要があります (16.3.9 節を参照)。HLE 命令では、明示的にロックアドレスが関連付けられており、正しい値が返されることがハードウェアによって保証されているため、このような状況は発生しません。

ユーザー/ソース・コーディング規則 36: ロックの省略に RTM を利用する場合は、常にトランザクション領域内でロックをテストします。

チューニングの推奨事項 14: ラッパーでロック変数を読み取れない場合、RTM ラッパーは使用しないようにします。

16.3.4 ロックの省略に RTM を使用するラッパーの例

このセクションでは、RTM 命令を使用してロックの省略を実装するラッパーを記述する方法を示します。既存のロック実装 (省略なし) をラッパーで囲み、ラッパー内に新しいパスを追加してロックの省略を実装します。ラッパーでロックを省略する場合と、しない場合のそれぞれのコードパスを記述します。ロックを省略しないで取得するパスは、ロックを省略するパスが成功しなかったときにのみ実行されます。このアプローチでは、ロックを省略しない場合のパスは変更されません。これは、チケットロックや reader-writer (読み取り/書き込み) ロックなどのさまざまなロックに適用できます。

例 16-3 にコードシーケンスの例を示します (使用されている組込み関数の説明は、16.7 節を参照してください)。

例 16-3 ロックの取得/解放プリミティブに RTM を使用するラッパーの例

```
void rtm_wrapped_lock(lock) {
   if (_xbegin() == _XBEGIN_STARTED) {
      if (lock is free)
         /* 読み取りセットにロックを追加する */
         return; /* トランザクション実行 */
      _xabort(0xff);
      /* 0xff はロックがフリーでないことを示す */
   }
   /* トランザクション・アボートの後に実行されるコード */
   original_locking_code(lock);
}
```

```
void rtm_wrapped_unlock(lock) {
   /* ロックがフリーの場合は、ロックが省略されたと仮定する */
   if (lock is free)
      _xend(); /* コミット */
   else
      original_unlocking_code(lock);
}
```

例 16-3 の _xabort() はロックがフリーでない場合、トランザクション実行を終了します。代わりに、_xend() を使用しても同様の動作になります。ただし、プロファイリング・ツールは _xabort() 操作とアボートコード 0xff (ソフトウェア規約) を簡単に認識し、ロックがフリーでなかったケースであると判断できます。_xend() が使用されると、プロファイリング・ツールは、このケースとロックの省略に成功したケースを区別することができません。

上記の例は、1 回だけ再試行し、トランザクション・アボートのさまざまな原因は区別しないという基本ポリシーを示すために単純化されています。より高度な実装では、トランザクション・アボートの原因に関する情報に基づいてロックごとに省略を試すかどうかを決定するヒューリスティックを追加します。また、ロックがフリーでない場合、ブロックした後にロックの省略を再試行するコードを追加します。これには、同期ライブラリーへのわずかな変更が必要になることがあります。

プログラミング・エラーが原因で、スレッドが解放済みのロックを解放しようとすることがあります。このエラーはすぐに明らかにならないことがあります。しかし、前述のラッパーでロックの解放関数を前述の RTM 対応のライブラリーに置換すると、XEND 命令がトランザクション領域外で実行されます。この場合、ハードウェアは #GP 例外を発生します。一般に、このエラーはオリジナルのアプリケーションで修正したほうが良いでしょう。代替手段として、ソフトウェアでエラーになったコードパスを保持しようとする場合は XTEST で XEND を保護できます。

16.3.5 RTM フォールバック・ハンドラーのガイドライン

RTM 用のフォールバック・ハンドラーでは、RTM ベースのトランザクション実行が成功しなかったときに実行されるコードパスを記述します。インテル® TSX はトランザクション実行の成功を保証していないため、RTM フォールバック・ハンドラーは単にトランザクション実行の再試行を繰り返すのではなく、処理を進めることを保証しなければなりません。

チューニングの推奨事項 15: ロックの省略に RTM を利用する場合、ロックを取得することで処理が進むことを簡単に保証できます。

フォールバック・ハンドラーが明示的にロックを取得すると、そのロックを省略する他のすべてのトランザクション実行スレッドがアボートし、実行がシリアル化されます。これは、ロックがトランザクション領域の読み取りセットにあることを保証することで達成されます。

ソフトウェアは、EAX レジスターのアボート情報を参照して、トランザクション実行を再試行する場合とフォールバックして明示的にロックを取得する場合のヒューリスティックを作成できます。例えば、_XABORT_RETRY ビットが

設定されていないと、トランザクション実行の再試行は他のアボートを引き起こす可能性が高くなります。フォールバック・ハンドラーは、このような場合とロックがフリーでない場合 (例えば、_XABORT_EXPLICIT ビットが設定されているが、_XABORT_CODE()[23] が「ロックがビジー」状態であることを示す 0xff を返す場合) を区別する必要があります。ロックがフリーでない場合は待機後に再試行しなければなりません。

アボートの原因がデータ競合 (_XABORT_CONFLICT) であれば、ほとんどのデータ競合は一時的であるため、時間をおいて再試行することでパフォーマンスが向上する可能性があります。ただし、このような再試行の回数は制限し、無制限に繰り返すべきではありません。

ハイパースレッディングが有効である構成では、容量アボート (_XABORT_CAPACITY) に対する少数の再試行は有効である場合があります。L1 キャッシュは HT スレッド間で共有されるリソースであり、一方のスレッドが他のスレッドのデータを押し出す可能性があります。再試行には合理的な成功の見込みがあります。この場合、ステータスコードの _XABORT_RETRY ビットを無視する必要があります。それ以外ではいかなる理由があっても _XABORT_RETRY ビットを無視するべきではありません。

一般にコア数が多く複数のソケットを持つシステムでは再試行の回数が増加します。

一般に、ロックがフリーでない場合、フォールバック・ハンドラーはロックがフリーになるまでトランザクション実行の再試行を待機すべきです。そうすることでロックの省略なしで非トランザクション実行にとどまるのを防ぐことができます。このような状況は、フォールバック・ハンドラーが、ロックがフリーでもトランザクション実行を試みることができないために生じます (16.3.8 節を参照してください)。

ユーザー/ソース・コーディング規則 37: RTM アボートハンドラーは有効なテスト済みの非トランザクション・フォールバック・パスを提供する必要があります。

チューニングの推奨事項 16: ロックがビジー状態の場合は、ロックがフリーになるまで再試行を待機します。

16.3.6 インテル® TSX による省略に適したロックの実装

このセクションでは、一般的なロック・アルゴリズムにおいて、インテル® TSX 命令を使用するロックの省略に適したバージョンを実装するアプローチについて述べています。このセクションで触れないアルゴリズムにも同様のアプローチを適用できます。

16.3.6.1 HLE を使用する単純なスピンロックの実装

スピンロックは、単純でよく使用されているロック・アルゴリズムです。このアルゴリズムでは、スレッドはロックがフリーかどうかを確認してから、LOCK プリフィクス命令でロックを取得します。ロックがフリーでない場合、スレッドはロックがフリーになるまで (通常は、ロックの値を保持するローカル・データ・キャッシュの読み取り操作により) スピンして待機します。

例えば、値がゼロのときはロックがフリーで、そうでない場合はほかのスレッドによって取得されていると仮定します。ロックは通常のストア命令によって解放されます。

23 _XABORT_CODE は RTM アボートコードを示す xabort ステータスにアクセスします。

例 16-4 は、C11 規格に似た gcc 4.8 以降の**アトミック組込み関数**を用いています。ここでは、推奨されるアプローチに従って gcc 4.8+ 組込み関数を使用したスピンロックを実装しています。このスピンロックで HLE を有効にするのに必要な変更は、__ATOMIC_HLE_ACQUIRE フラグと __ATOMIC_HLE_RELEASE フラグの追加だけです。残りのコードは HLE を利用しない場合と同じです。

例 16-4 GCC 4.8 以降で HLE を使用するスピンロックの例

```
#include <immintrin.h> /* _mm_pause() に必要 */
/* ロックを 0 に初期化 */
void hle_spin_lock(int *lock)
{
   while (__atomic_exchange_n(lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE) != 0)
   { int val;
   /* 再試行する前にロックがフリーになるのを待機する*/
   do {
      _mm_pause(); /* アボートのスペキュレーション */
      __atomic_load_n(lock, &val, __ATOMIC_CONSUME);
   } while (val == 1);
   }
}
```

```
void hle_spin_unlock(int *lock)
{
   __atomic_clear(lock, __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
}
```

以下に、同じスピンロックを Windows* 用の C/C++ コンパイラー (Microsoft* Visual Studio* 2012 とインテル® C++ コンパイラー 17.0) の組込み関数を用いて実装する例を示します。

例 16-5 インテル® コンパイラーと Microsoft* コンパイラーの組込み関数で HLE を使用するスピンロックの例

```
#include <intrin.h> /* _mm_pause() に必要 */
#include <imminitrin.h> /* HLE 組込み関数に必要 */
/* ロックを 0 に初期化 */
void hle_spin_lock(int *lock)
{
   while (_InterlockedCompareExchange_HLEAcquire(&lock, 1, 0) != 0){
   /* 再試行する前にロックがフリーになるのを待機する */
      do {
         _mm_pause(); /* アボートのスペキュレーション */
         /* コンパイラーによる命令の並べ替えと待機ループのスキップを防ぎ
         IA 上で追加のフェンス命令が生成されないようにする */
         _ReadWriteBarrier();
      } while (lock == 1);
   }
}
```

```
void hle_spin_unlock(int *lock)
{
   _Store_HLERelease (lock, 0);
}
```

HLE スピンロックのアセンブラー実装については、16.7 節を参照してください。

16.3.6.2 インテル® TSX を使用する reader-writer (読み取り/書き込み) ロックの実装

reader-writer (読み取り/書き込み) ロックは、クリティカル・セクションの大部分が読み取り専用のときによく使用されます。このロックはクリティカル・セクションの読み取りアクセスがシリアル化されるのを防止しますが、共有された場所に対してはアトミック操作 (LOCK プリフィクスが付加された XADD または CMPXCHG のことが多い) を要求し、複数の読み取り間で通信が必要になります。ロックの省略は、読み取りと競合しない書き込みが通信することなく同時に処理できることを除いて、基本的にすべてのロックが reader-writer (読み取り/書き込み) ロックと同様の動作となります。

前述のラッパーアプローチにより、RTM を使って reader-writer (読み取り/書き込み) ロックを省略できます。ロックの省略との唯一の違いは、reader-writer (読み取り/書き込み) ロックのアルゴリズムでは通常、読み取りと書き込みの両方の状態を確認してロックがフリーかどうかを判断していることです。そのため、読み取り/書き込みのそれぞれの状態を別々のキャッシュラインに配置できれば、読み取りのトランザクション実行と非トランザクション実行を並列に行えます。読み取りは、書き込みの状態がフリーであることを確認するだけで済みます。

HLE を利用する場合、ロックが省略されるときと、されないときのコードパスは同じになります。一部の reader-writer (読み取り/書き込み) ロックの実装では実際のクリティカル・セクションではなく、読み取り/書き込み状態の保護にロックを使用しています。この場合、まずロックを 1 つのアトミック操作から成る高速なパスに変更する必要があります。このパスでは、ロック変数が配置されているキャッシュラインは変更すべきではありません。具体的には、読み取りの数と書き込みの数を 1 つのフィールドにまとめ、ロックの取得/解放関数に LOCK プリフィクスを付加した XADD 命令や CMPXCHG 命令を使ってこのフィールドをアトミックに確認/更新します。HLE プリフィクス (XACQUIRE と XRELEASE) を、LOCK プリフィクスが付加されたこれらの命令に追加します。興味深いことに、このアプローチは、インテル® TSX を使用しなくても reader-writer (読み取り/書き込み) ロックのパフォーマンスを向上させます。また、RTM ラッパーを使用することで、トランザクション実行用と非トランザクション実行用に異なるロックの取得パスが用意されるため、ロック構造の変更を回避できます。

チューニングの推奨事項 17: 読み取り/書き込みロックでは基本ブロックのロックではなく、ロック操作全体を省略します。

16.3.6.3 インテル® TSX を使用するチケットロックの実装

チケットロックも一般的なアルゴリズムです。チケットロックはスピンロックのバリエーションで、共有する場所でスピンしてロックがフリーになったら取得する代わりに、チケットによってどのスレッドがクリティカル・セクションに入ることができるかを決定します。

前述のラッパーアプローチにより、RTM を使ってチケットロックを省略することができます (16.3.4 節を参照)。

一部のチケットロックの実装は、チケットの値が増えることを想定します。そのようなロックは、取得前と取得後のロックの値が同じでなければならないとする HLE 要件を満たしません。

チューニングの推奨事項 18: RTM を使用してチケットロックを省略します。

16.3.6.4 インテル® TSX を使用するキューベースのロックの実装

一般に、ロックの省略は複数のスレッドが共通のクリティカル・セクションへ同時に入り、コミットを試みることを前提としています。公正なロックでは、スレッドが先着順にクリティカル・セクションに入り、解放しなければなりません。この 2 つの考え方は、場合によっては対立するように見えますが、一般的な目的はこれよりも柔軟性があります。

キューベースのロックはスレッドがロック要求のキューを作成する公正なロック方式です。これはチケットロックの亜種とも言えます。

一部の実装では、初期の LOCK プリフィクス命令によりキューが作成されます。その際、その命令に HLE XACQUIRE プリフィクスを追加してロックを省略できます。トランザクション・アボートが発生しなかった場合、ロックの解放後にキューは空になります。しかし、トランザクション・アボートが発生し、アボートしたスレッドがロックを明示的に取得している場合 (つまり、キューが生成されている)、後続のスレッドはキューに追加され、ロックが解放されると先頭のスレッドのみがロックの省略を試みます。さらに、別のスレッドがクリティカル・セクションに到達してキューに追加されると、トランザクション実行中のスレッドはアボートし、キューが空になるまで非トランザクション実行になる可能性があります。

これは、キューを使用してロックの省略を試みる実装でのみ発生します。スピンフェーズを省略し最初のスピンが失敗した後にのみキューで待機する適応スピン・スリープ・ロックのような、最初のアトミック操作の後にのみキューを作成する実装には当てはまりません。この問題は、ラッパーを使用する実装 (RTM を使用するものなど) でも存在しません。これらの実装では、キューの処理でスレッドはロックの省略を試みません。

チューニングの推奨事項 19: 最初のアトミック操作でキューを実装するロックに RTM ラッパーを使用します。

16.3.7 インテル® TSX を使用するアプリケーション固有のメタロックの省略

一部のアプリケーションは、同期ライブラリーを利用してメタロックと呼ばれる独自のロックを構築します。このアプローチでは、アプリケーションは同期ライブラリーのロックによりメタロックのデータを保護します。データを更新したら、ロックを解放します。これは、16.3.6.2 節の reader-writer (読み取り/書き込み) ロックの実装アプローチに似ています。

アプリケーションは、メタロックを保持しているときにクリティカル・セクションを実行し、メタロックを解放しているときは同期ライブラリーのロックを使ってメタロックデータを保護します。このシーケンスでは同期ライブラリーのロックを省略しても意味がありません。同期ライブラリー内のコードではなく、メタロック自体を省略してアプリケーションのコードをトランザクション実行すべきです。プロファイリング・ツールによってクリティカル・セクションを特定し、(16.3.4 節の例に似た) RTM ラッパーによりロックの省略中のメタロックを回避できます。

次のメタロックの実装例について考えてみます。

例 16-6 メタロックの例

```
void meta_lock(Metalock *metalock) {
   __lock(metalock->lock);
   /* ロックのためメタロックの状態を変更 */
   unlock(metalock->lock);
}
```

```
void meta_unlock(Metalock *metalock) {
   lock(metalock->lock);
   /* メタロック状態を解放する */
   unlock(metalock->lock);
}
```

```
meta_lock(metalock);
   /* クリティカル・セクション */
meta_unlock(metalock);
```

上記のコードは次のコードに置き換えることができます。

例 16-7 RTM を使用するメタロックの例

```
void rtm_meta_lock(Metalock *metalock) {
   if (_xbegin() == _XBEGIN_STARTED){
      if (meta_state_is_all_free(metalock))
         return;
      _xabort(0xff);
   }
   meta_lock(metalock);
}
void rtm_meta_unlock(Metalock *metalock) {
   if (meta_state_is_all_free(metalock))
      _xend();
   else
      meta_unlock(metalock);
}
```

```
rtm_meta_lock(metalock);
   /* クリティカル・セクション */
rtm_meta_unlock(metalock);
```

チューニングの推奨事項 20: メタロックでは基本ブロックのロックではなく、外側のロックをすべて省略します。

16.3.8 永続的な非省略実行を回避する

トランザクション・アボートが起こると、最終的にロックを省略しない非トランザクション実行に遷移します。これは、処理が続行することを保証します。ただし、特定の状況下と一部のロック取得アルゴリズムでは、スレッドがロックの省略を試みずに非トランザクション実行に留まることがあります。これはパフォーマンスの妨げとなります。

この状況を理解するため、HLE を使用する単純なスピンロックの実装例について考えてみます (同様のシナリオは RTM でも試せます)。ロックは値が 0 の場合はフリーで、値が 1 の場合は別のスレッドによって取得されていることを意味します。

HLE を利用するロックの取得シーケンスは、例 16-8 のように記述できます。

例 16-8 HLE を利用するロックの取得/解放シーケンス

```
   mov eax,$1
Retry:
   XACQUIRE; xchg LockWord,eax
   cmp eax,$0 # 値が 0 ならロックの取得に成功
   jz Locked
   SpinWait:
   cmp LockWord, $1
   jz SpinWait# 値がまだ 1 のまま
   jmp Retry# ロックがフリーなので取得を試みる
Locked:
   XRELEASE; mov LockWord,$0
```

スレッドがロックを省略できないときは、省略なしでロックを取得します。ほかのスレッドが同じロックの取得を試みる場合、"XACQUIRE; xchg lockWord, eax" 命令を実行して、ロック操作を省略しトランザクション実行に入ります。しかし、この時点でロックはほかのスレッドによってすでに取得されているため、このスレッドはトランザクション実行中に SpinWait ループに入ります。

これは、トランザクション実行中にハードウェアがクリティカル・セクション・ロックだと認識できず、ロック変数に対するアトミック操作と見なすためです。ハードウェアはロックがフリーでない、ということの意味を理解できません。

ロックを取得しているスレッドがロックを解放すると、そのロックへの書き込み操作によって、現在その場所をスピンしているスレッドのトランザクションがアボートします (ロックの解放操作とトランザクション実行中のスレッドによるロックの読み取りループの間で競合が発生するため)。一度アボートすると、スレッドはロックを省略しないで実行を再開します。これは、すべてのスレッドで起こる可能性があります。スレッドはトランザクション実行中にスピンしますが、ロックが解放されるとロックの省略なしで非トランザクション実行します。これは、ほかにロックの取得を試みるスレッドがなくなるまで繰り返されます。つまり、スレッドは非トランザクション実行に留まることになります。

この問題への簡単な対処法は、SpinWait ループで PAUSE 命令 (アボートを引き起こす) を使用することです。これは、インテル® TSX を使用しない場合でもロックの解放の待機に推奨されるアプローチです。PAUSE 命令は SpinWait ループの非トランザクションへの移行を強制し、ロックが解放されたらスレッドがロックを省略できるようにします。

例 16-9 HLE を使用したスピンウェイトの例

```
   mov eax,$1
Retry:
   XACQUIRE; xchg LockWord,eax
   cmp eax,$0# 値が 0 ならロックの取得に成功
   jz Locked
SpinWait:
   pause
   cmp LockWord, $1
   jz SpinWait# 値がまだ 1 のまま
   jmp Retry# ロックがフリーなので取得を試みる
Locked:
```

チューニングの推奨事項 21: HLE スピンロックの待機ループには常に PAUSE 命令を使用します。

16.3.9 RTM ベースのライブラリーで省略されたロックの値を読み取る

一部の同期ライブラリーは、ロックの値を読み取るインターフェイスを提供しています。RTM を使用してロックを省略するライブラリーは、ロックが読み取られるだけでライブラリー内へ書き込まれないため、省略を行うスレッドがロック変数を取得したかどうかを正確に判断できないことがあります。

場合によっては、ライブラリーのインターフェイスはロックが取得されているかどうかをチェックするだけの単純なテストで、ソフトウェアに正当性チェックを提供します。RTM ベースのライブラリーを使用して関数に正しい値を確実に渡すには、トランザクション実行をアボートして明示的にロックを取得する必要があります。具体的には、XABORT 命令 (_xabort(0xfe) を使用して) でアボートを強制します。フォールバック・ハンドラーは 0xfe コードでこの状況を特定し、読み取りを排除する最適化を行います。また、_xtest() 組込み関数で不要なトランザクション・アボートを回避できます。

assert(is_locked(my_lock)) => assert(_xtest() || is_locked(my_lock))

省略された同期ライブラリー向けの効率良いプリミティブは、取得されたロックや進行中のロックの省略を結合します。次に例を示します。

bool is_atomic(lock) { return _xtest() || is_locked(lock); }

また、動作を想定できる場合、ロック変数は関数の一部として読み取ることができます。例えば、ロックを取得する **try-lock** インターフェイスは、スレッドがロックの取得を 1 回だけ試みてロックがフリーかどうかを示す値を返します。これは、ロックを取得するためスピンを繰り返すスピンロックとは対照的です。一般に try-lock は問題になりませんが、入れ子の try-lock により返される値に対しソフトウェアが暗黙の仮定を行っていることがあります。RTM ベースの実装ではロックが省略されるため、ロックがフリーであることを示す値が返されます。ソフトウェアでこの値に対して暗黙の仮定を行っている場合、同期ライブラリーは XABORT 命令で (_xabort(0xfd) を使用して) トランザク

ション・アボートを強制できます。ただし、これは一部のプログラムで不要なアボートを引き起こします。プログラムでこのような暗黙の仮定を行うことは推奨されません。また、このような暗黙の仮定が行われることはまれであるため、try-lock の同期ライブラリーではアボートしないことを推奨します。

16.3.10 HLE と RTM を混在させる

HLE と RTM は、一般的なトランザクション実行機能に対する 2 つのソフトウェア・インターフェイスの選択肢を提供します。RTM 内で HLE を使用したり、HLE 内で RTM を使用する場合の動作は実装固有です。第 4 世代インテル® Core™ プロセッサーの初期実装では、HLE と RTM を混在させるとトランザクション・アボートにつながります。以降のプロセッサーでは動作が変わる可能性がありますが、トランザクション・コミットのセマンティクスは維持されます。

一般に HLE と RTM はロックを省略するという目的は同じですが、ソフトウェア・インターフェイスが異なるため、アプリケーションでは混在しないようにすべきです。しかし、ロックの省略を実装するライブラリー関数は、呼び出し関数があるかどうか、また呼び出し関数が RTM あるいは HLE でロックを省略するライブラリー関数を呼び出しているかどうかを把握していない可能性があります。

これらの状態は、_xtest() 関数によりソフトウェアで対応できます。例えば、ライブラリーがトランザクション領域内で呼び出されたかどうか、そしてロックがフリーかどうかを確認できます。トランザクション領域内で呼び出された場合は、新しいトランザクション領域を開始しないようにできます。ロックがフリーでない場合、ライブラリーは _xabort(0xff) 関数で結果を返すことができます。この場合、ロックの解放時に呼び出される関数は、取得操作がスキップされたことを認識できなければいけません。

例 16-10 にこの概念を示します。

例 16-10 HLE と RTM を混在させる概念上の例

```
// ロックの取得シーケンス
// 関数またはスレッドのローカルを使用する
bool lock_in_transactional_region = false;
if (_xtest() && my lock is free) { /* すでにトランザクション領域内である */
   lock_in_transactional_region = true;
} else {
// ロックがフリーの場合は取得し、そうでない場合はアボートする
}
// ロックの解放シーケンス
if (!lock_in_transactional_region) {
   // ロックを解放する
}
```

16.4 インテル® TSX のパフォーマンス監視サポートを利用する

インテル® TSX を使用するアプリケーションの解析は、パフォーマンス・カウンターベースのプロファイルにより、トランザクション実行の動作とトランザクション・アボートの原因を明らかにします。インテル® TSX で優れたパフォーマンスを得るには、プロファイリング・ツールで収集したデータを基にアボートを最小限に抑えるチューニングが必要です。通常、アプリケーションのインストルメンテーションよりも、パフォーマンス・カウンターを使用するほうが簡単でコード変更が少ないため推奨されます。『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 18 章と第 19 章に、現在の実装でサポートされる各種パフォーマンス監視に関する情報が記載されています。

一般に、プロファイリング・ツールは周期的に割り込みをかけて情報を収集しますが、この割り込みによってトランザクション・アボートするため、プロファイリングはトランザクション実行に影響します。そのため、プロファイリングではこの影響を最小限に抑えるべきです。ただし、トランザクション・アボートのみをプロファイリングする場合は問題になりません。

プログラムの起動時に一度しか実行されないイベントも多数あります。複雑なプログラムのプロファイリングでは起動時のプロファイリングをスキップすることで、これらのイベントによる不要なデータの収集を大幅に減らすことができます。

Linux* perf、インテル® Performance Counter Monitor、およびインテル® VTune™ プロファイラーを含むプロファイル・ツールは、インテル® TSX をサポートしています。http://www.intel.com/software/tsx (英語) を参照してください。

16.4.1 トランザクション成功を測定する

最初のステップは、アプリケーションのトランザクション成功を測定することです。これは、次の 3 つのカウンターと Unhalted_Core_Cycles イベントを設定することで測定できます。

1. 固定サイクルカウンター (IA32_FIXED_CTR0) を使用して FixedCyclesCounter を測定します。
2. IA32_PERFEVTSEL2 に IN_TX フィルターと IN_TXCP フィルターを設定して、IA32_PMC2 の CyclesInTxCP を測定します。
3. MSR IA32_PERFEVTSELx (x= 0, 1, 3) に IN_TX フィルターを設定して、対応するカウンターの CyclesInTXOnly を測定します。

サンプリングはトランザクション・アボートを引き起こす可能性があるため、これらのサイクルの測定にはサンプリングではなくカウンターを使用します。この 3 つの値から、合計サイクル数、トランザクション実行で費やされたサイクル数、アボートされたトランザクション領域で費やされたサイクル数を計算できます。

```
CyclesTotal = FixedCycleCounter
%CyclesTransactionalAborted = ((CyclesInTxOnly - CyclesInTxCP) / CyclesTotal) * 100.0
%CyclesTransactional = (CyclesInTx / CyclesTotal) * 100.0
%CyclesNonTransactional = 100.0 - %CyclesTransactional
```

CyclesTransactional がほぼゼロの場合、アプリケーションはロックベースの同期を使用していないか、インテル® TSX の Hardware Lock Elision (HLE) が有効な同期ライブラリーを使用していません。後者の場合、インテル® TSX 対応の同期ライブラリーを使用すべきです (16.3 節を参照してください)。

CyclesTransactionalAborted が CyclesTransactional と比較して少ない場合、トランザクションの成功率は高く、追加のチューニングは必要ありません。

CyclesTransactionalAborted が CyclesTransactional とほぼ同じ (で少なくない) 場合、ほとんどのトランザクション領域がアボートしていて、HLE は役立ちません。次のステップでは、トランザクション・アボートの原因を特定して、トランザクション・アボート数を減らします (16.2.4 節を参照してください)。

16.4.2 省略するロックを特定してすべてのロックが省略されることを確認する

このステップは、トランザクション実行で費やされたサイクルが少ない場合に有効です。これは、省略されるロックが少ないことが原因の可能性があります。MEM_UOP_RETIRED.LOCK_LOADS イベントをカウントし、RTM_RETIRED.START イベントまたは HLE_RETIRED.START イベントと比較すべきです。ロックのロード数が、開始されたトランザクションの数よりも大幅に多い場合、すべてのロックが省略として認識されていない可能性があります。MEM_uop_RETIRED.LOCK_LOADS の PEBS バージョンでサンプリングを行い、失われたロックを特定できます。ただし、この手法は省略の対象になっていないメタロックを迅速に検出するのには効果的ではありません (16.3.7 節を参照してください)。また、MEM_uop_RETIRED.LOCK_LOADS イベントのコールグラフをプロファイリングすることで、アプリケーション・レベルのクリティカル・セクションをトランザクション実行するためインテル® TSX を使用すべき高レベルの同期ライブラリーを特定できます。

16.4.3 トランザクション・アボートのサンプリング

ハードウェア実装は、トランザクション・アボートをサンプリングするため PEBS プリサイスイベントを定義しています (HLE の場合は HLE_RETIRED.ABORTED、RTM の場合は RTM_RETIRED.ABORTED)。これは実行中のすべてのトランザクション・アボートを正確にプロファイルすることを可能にします。PEBS を有効にしてサンプリングし、トランザクション・アボートが発生するコードの位置を特定します。PEBS ハンドラー (プロファイリング・ツールの一部) は、PEBS レコードの EventingIP フィールドを用いてトランザクション・アボートの正確なコード位置を報告します。

次のステップでは最も一般的なトランザクション・アボートについて検証し対処します。トランザクション・アボートをサンプリングすることで追加のアボートが発生することはありません。

16.4.4 プロファイリング・ツールを利用してアボートを分類する

トランザクション・アボートのプロファイリングにより生成される PEBS レコードには、トランザクション・アボートの原因に関する追加情報を示す TX Abort Information フィールドがあります。TX Abort Information の下位 32 ビットは Cycles_Last_TX と呼ばれ、アボート前の最後のトランザクション領域で費やされたサイクル数を示します。このデータからトランザクション・アボートのおよそのコストが分かります。

RelativeCostOfAbortForIP = SUM(Cycles_Last_TX_For_IP)

トランザクション・アボートにはパフォーマンスを低下させないものもあれば、パフォーマンスに大きく影響するものもあります。プログラマーは、この情報を基にどのトランザクション・アボートに注目すべきかを判断できます。

PEBS レコードの詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の 18.10.5.1 節を参照してください。

アボートを分類できるように、プロファイリング・ツールはアボートコストを表示できなければいけません。

チューニングの推奨事項 22: 最もコストの高いアボートを最初に検証します。

チューニングの推奨事項 23: TX Abort Information にはトランザクション・アボートに関する追加情報が含まれています。

PEBS レコードの **Instruction_Abort** ビット (ビット 34) が設定されている場合、トランザクション・アボートの原因を命令に直接関連付けることができます。それらのアボートに対して、PEBS レコードはトランザクション・アボートの原因となった命令アドレスを記録します。ページフォルト (通常プログラムを終了させるものやプログラム起動時のワーキングセットでフォルトになるものを含む) などの例外もこのカテゴリーに含まれます。

PEBS レコードの **Non_Instruction_Abort** ビット (ビット 35) が設定されている場合、アボートの原因は PEBS レコードで報告された命令アドレスの命令ではない可能性があります。例えば、ほかのスレッドとの間でデータ競合が発生した場合が考えられます。この場合、**Data_Conflict** ビット (ビット 37) も設定されます。別の例として、トランザクション実行する読み取りセット/書き込みセットの容量の制限によるトランザクション・アボートが挙げられます。これは、Capacity_Write (ビット 38) フィールドと Capacity_Read (ビット 39) フィールドに記録されます。

データ競合によるアボートは、トランザクション領域内のどの命令でも発生する可能性があります。そのため、クリティカル・セクション全体にわたって競合の原因を調査したほうが良いでしょう。PEBS によって報告される **EventingIP** ではなく、リターン IP (アボートコードの IP) とコールグラフに注目すべきです。通常リターン IP はロックがインライン展開されていない限り、同期ライブラリーを指しているため、呼び出し元からクリティカル・セクションを特定できます。

容量が原因の場合、クリティカル・セクション全体にわたってメモリー使用量を減らすように変更する必要があるので、クリティカル・セクション全体 (ReturnIP のプロファイリング) を調査すると良いでしょう。

チューニングの推奨事項 24: 命令のアボートは早期に分析すべきですが、プログラム起動後に発生するコストの高いもののみ分析します。

チューニングの推奨事項 25: データ競合や容量の制限によるアボートは、アボート時に報告される命令アドレスだけでなく、クリティカル・セクション全体を調査します。

チューニングの推奨事項 26: プロファイラーは、命令が原因でないアボートイベントでは ReturnIP とコールグラフの表示を、命令が原因のアボートイベントでは EventingRIP の表示をサポートしなければなりません。

チューニングの推奨事項 27: プロファイリング・ツールはすべての PEBS TX Abort 情報ビットを表示できなければなりません。

16.4.5 RTM フォールバック・ハンドラー向けの XABORT 引数

RTM ベースのトランザクション領域のアボートに XABORT 命令が使用されると、EAX レジスターを介してフォールバック・ハンドラーに命令オペランドが渡されます。この情報は、RTM 用の PEBS ベースのプロファイリング・ツールでも提供されます。プロファイリング・ツールはこの情報を使用してさまざまな XABORT ベースのトランザクション・アボートを分類できます。アボートステータスを定義することは、優れたフォールバック・ハンドラーを記述する上でも役立ちます。

次の表に、ここで使用するアボートステータスの定義を示します。

表 16-1 RTM アボートステータスの定義

アボートコード	説明
0xff	テスト時にロックがフリーでなかったことが原因の XABORT ベースのアボート (「15.3.4 ロック省略に RTM を使用するラッパーの例」を参照)
0xfe	省略されたロックの値がテストされたことが原因の XABORT ベースのアボート (「15.3.9 RTM ベースのライブラリーで省略されたロックの値を読み取る」を参照)
0xfd	入れ子の try-lock 内で発生した XABORT ベースのアボート (15.3.9 を参照)
0xfc: 0xf0	予約済み

チューニングの推奨事項 28: プロファイリング・ツールは RTM アボートコードを表示できなければなりません。

16.4.6 トランザクション・アボートのコールグラフ

プロファイリング・ツールは、パフォーマンス監視情報を収集する際に割り込みをかけます。この割り込みはトランザクション・アボートの原因になります。つまり、プロファイリング・ツールはトランザクション・アボートが発生した後にのみ情報を収集することが可能であり、トランザクション領域内で発生したスタック上の関数呼び出しは把握できず、トランザクション実行の開始時のコールグラフのみ見ることができるということです。PEBS でトランザクション・アボートをサンプリングする場合、RIP フィールドにはアボート後の命令ポインターが、EventingIP フィールドにはアボート時のトランザクション領域内の命令ポインターが含まれます。すべてのサンプリングがトランザクション・アボートの原因となるため、非アボートイベントのサンプリングでも同じことが言えます。

アボートの種類に応じて、ReturnIP または EventingIP のいずれかをプロファイリングすると良いでしょう。プロファイリング・ツールによって収集されるスタック・コールグラフは常に ReturnIP と関連付けられています。この情報と EventingIP を組み合わせると、トランザクション領域内に関数呼び出しが含まれず、連続していないように見えることがあります (EventingIP は最下位レベルの呼び出し元と関連付けられていない可能性があります)。アボートの原因を理解するためトランザクション領域内の関数呼び出しに関する情報が必要な場合は、LBR (最後の分岐レコードの略、16.4.7 節を参照) または SDE ソフトウェア・エミュレーション (16.4.8 節を参照) を使用できます。

チューニングの推奨事項 29: プロファイラーは ReturnIP と EventingIP を表示できなければなりません。

チューニングの推奨事項 30: スタック・コールグラフは常に ReturnIP に関連付けられており、EventingIP と一緒に見た場合、連続していないように見えることがあります。

チューニングの推奨事項 31: トランザクション領域内の関数呼び出しを確認するには LBR またはインテル® SDE を利用します。

16.4.7 LBR とトランザクション・アボート

LBR (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の 17.4 節を参照) は、トランザクション実行とアボートに関する情報を提供します。一般に LBR はインテル® TSX と互換性があります。通常のコールグラフが利用できない場合、LBR を使用することでトランザクション内の情報が得られます。lcall フィルターをコールグラフの代わりに使用できます。ただし、LBR コールグラフ・スタック (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の 17.8 節) はインテル® TSX と互換性がなく、完全な情報が得られないことがあります。

チューニングの推奨事項 32: プロファイリング・ハンドラーはアボート時に LBR をサンプリングし、その結果を報告できなければなりません。

16.4.8 インテル® SDE によるインテル® TSX ソフトウェアのプロファイリングとテスト

インテル® Software Development Emulator (インテル® SDE) ツール (http://software.intel.com/en-us/articles/intel-software-development-emulator (英語)) は、ハードウェアに実装される前に新しい命令セット拡張をソフトウェア開発に利用できます。このツールは、新しい命令を利用するソフトウェアの広範なテスト、デバッグ、解析にも役立ちます。

インテル® SDE の各種機能によって、インテル® TSX 命令を使用するプログラムの機能テスト、プロファイル、デバッグが行えます。このツールは一般的なトランザクション・アボートの詳細な情報と、ハードウェアで直接利用できない追加のプロファイリング機能をもたらします。エミュレーションによる非常に大きなオーバーヘッドが発生するため、このツールでランタイムおよび絶対的なパフォーマンス特性を得るべきではありません。

16.4.4 節で述べたとおり、アボートの原因がデータ競合やリソースの制限によるものでない限り、ハードウェアはアボートの原因となった命令の正確なアドレスを報告します。インテル® SDE はそのような命令の正確な命令アドレスと命令に関する追加情報を提供します。また、アプリケーションのソースコード位置、ソースファイル名、行番号、コールスタック、命令が操作したデータアドレスの情報も提供します。さらに、犠牲となったトランザクション (競合でアボートされた) 向けに、競合するメモリーアクセスが行われたソースコードの位置も出力できます。

次のオプションを指定することで、これらの情報を得られます。

-tsx -hle_enabled 1 -rtm-mode full -tsx_stats 1 -tsx_stats_call_stack 1

フォールバック・ハンドラーは EAX レジスターからアボートの原因を判断します。インテル® SDE ツールにエミュレーター・パラメーターとして特定の EAX レジスター値を渡すと、トランザクション・アボートを強制できます。開発者はさまざまな EAX 値でフォールバック・ハンドラーをテストできます。このモードでは、すべての RTM ベースのトランザクション実行は、パラメーターとして渡された EAX レジスター値で直ちにアボートします。これは、未解決のページフォルトや同様の操作が原因でトランザクション実行がアボートするケース (EAX = 0) の機能テストに有効です。

次のオプションを指定することで、これらの情報を得られます。

-tsx -rtm-mode abort -rtm_abort_reason EAX

インテル® SDE は、容量アボートのデバッグに有効な命令とメモリーアクセスのログ取得機能を備えています。インテル® SDE のログデータから、容量オーバーフローを引き起こしている不均等 (non-uniform) キャッシュセットがあるか調査するため、キャッシュセットの過密度を診断できます。この洗練されたログデータは、アボートの原因を診断する際に利用できます。ログ取得機能は、以下のオプションを指定して有効にできます。

-tsx_debug_log 3 -tsx_log_inst 1 -tsx_log_file 1

さらに、インテル® SDE は、トランザクション内部の機能的なデバッグを行うため標準デバッガー (gdb や Microsoft* Visual Studio*) とともに使用できます。

16.4.9 HLE 固有のパフォーマンス監視イベント

インテル® TSX のパフォーマンス・イベントには HLE 固有のトランザクション・アボート条件が含まれています。これらのイベントは、16.2.4.4 節に示す原因のアボートを追跡します。多くの場合、これらのアボートは同期ライブラリーの実装の問題により発生します。インテル® TSX 対応の同期ライブラリーでは、これらのイベントを測定してその値が無視できるくらいになるまでライブラリーを改善すると良いでしょう。

TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK は、XRELEASE プリフィクスを持たないストア操作が、省略バッファーで省略されたロックの操作を行ったために発生したトランザクション・アボートの数をカウントします。これは多くの場合、ロックの解放命令に XRELEASE プリフィクスがないことが原因です。

TX_MEM.ABORT_ELISION_BUFFER_NOT_EMPTY は、トランザクション実行をコミットする XRELEASE プリフィクスが付加されたロックの解放命令が、省略されたロックを持つ省略バッファーを見つけたために発生したトランザクション・アボートの数をカウントします。これは多くの場合、省略されなかった (つまり、省略バッファーにない) ロックに対して XRELEASE を実行するコードシーケンスで発生します。

TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH は、XRELEASE ロックが省略バッファーのアドレスと値の条件を満たさないために発生したトランザクション・アボートの数をカウントします。これは、例えば XRELEASE 操作によって書き込まれる値が、同じロックに対する XACQUIRE 操作で読み取られた値と異なる場合に発生します。

TX_MEM.ABORT_HLE_ELISION_UNSUPPORTED_ALIGNMENT は、トランザクション領域の読み取りが省略バッファーのロックにアクセスしたが、読み取れなかったために発生したトランザクション・アボートの数をカウントします。これは通常、アクセスが適切にアライメントされていないか、アクセスが部分的にオーバーラップしているか、あるいは読み取り操作のリニアアドレスが省略されたロックと異なるが物理アドレスは同じ場合に発生します。これらのイベントの発生は非常にまれです。

16.4.10 インテル® TSX の有用なメトリックを計算する

ここでは、パフォーマンス・イベントを使って有用な指標を計算する式を示します。イベントのカウントをそのまま利用できることもありますが、場合によってはカウンターのデータを基に計算が必要になります。

次の式は、HLE または RTM トランザクション実行が開始された回数を計算します。ここでは、すべての入れ子の領域を 1 つの領域にまとめています。

#HLE Regions Started: HLE_RETIRED.COMMIT + HLE_RETIRED.ABORTED
#RTM Regions Started: RTM_RETIRED.COMMIT + RTM_RETIRED.ABORTED

次の式は、アボートした HLE または RTM トランザクション実行の比率を計算します。

%AbortedHLE = 100.0 * (HLE_RETIRED.ABORTED/HLE_RETIRED.START)
%AbortedRTM = 100.0 * (RTM_RETIRED.ABORTED/RTM_RETIRED.START)

次の式は、トランザクション領域で費やされたサイクル数の平均を計算します (CyclesInTX の計算については 16.4.1 節を参照)。

AvgCyclesInHLE = CyclesInTX/HLE_RETIRED_START
AvgCyclesInRTM = CyclesInTX/RTM_RETIRED.START
AvgCyclesInTX=CyclesInTX/(HLE_RETIRED.START + RTM_RETIRED.START)

次の式は、データ競合によりアボートした HLE または RTM トランザクション実行の比率を計算します。

%AbortedHLEDataConflict = TX_MEM.ABORT_CONFLICT/HLE_RETIRED.START;
%AbortedRTMDataConflict = TX_MEM.ABORT_CONFLICT / RTM_RETIRED.START;
%AbortedTXDataConlict= TX_MEM.ABORT_CONFLICT / (HLE_RETIRED.START+RTM_RETIRED.START);

次の式は、トランザクション・ストアのリソースの制限によりアボートした HLE または RTM トランザクション実行の数を計算します。

%AbortedTXStoreResource = TX_MEM.ABORT_CAPACITY_WRITE

Broadwell † と Skylake † マイクロアーキテクチャーをベースとするプロセッサーでは、"TX_MEM.ABORT_CAPACITY_WRITE" イベントは、読み取りまたは書き込みによるアボートをカウントする TX_MEM.ABORT_CAPACITY で置き換えられます。

次の式は、リソースの制限によりアボートした HLE または RTM トランザクション実行の合計数を計算します。L1 データキャッシュから追い出されたトランザクション読み取りは、直ちにアボートにならない可能性があるため区別されます。

%AbortedHLEResource = HLE_RETIRED.ABORTED_MISC1 - TX_MEM.ABORT_CONFLICT
%AbortedRTMResource = RTM_RETIRED.ABORTED_MISC1- TX_MEM.ABORT_CONFLICT
%AbortedTXResource = (HLE_RETIRED.ABORTED_MISC1+RTM_RETIRED.ABORTED_MISC5) - TX_EM.ABORT_CONFLICT

HLE_RETIRED.ABORTED_MISC1 は、16.4.9 節で示したいくつかのイベントの影響を受けることがあります。正確な結果を得るためには、まずこれらを最小限に抑えるようにロック・ライブラリーをチューニングする必要があります。

HLE_RETIRED.ABORTED_MISC1 は、HLE_RETIRED.ABORTED_MIEM としても知られています。同様に、RTM_RETIRED.ABORTED_MISC1 は RTM_RETIRED.ABORTED_MEM とも呼ばれます。

16.5 パフォーマンスのガイドライン

第 4 世代インテル® Core™ プロセッサーはインテル® TSX をサポートする最初のプロセッサーです。トランザクション実行には実装固有のオーバーヘッドが伴います。パフォーマンスは、将来のマイクロアーキテクチャーで改善される可能性があります。インテル® TSX の初期実装はアプリケーションのクリティカル・セクションにおける一般的な用途を想定しています。そのため、このようなオーバーヘッドは相殺され、通常アプリケーション・レベルのパフォーマンスには影響しません。

しかし、考慮すべきいくつかのガイドラインがあります。

チューニングの推奨事項 33: インテル® TSX はクリティカル・セクション向けに設計されているため、XBEGIN/XEND 命令と XACQUIRE/XRELEASE プリフィクスのレイテンシーは、LOCK プリフィクス命令のレイテンシーと一致するように意図されています。これらの命令のレイテンシーは通常のロード操作とは異なることに注意してください。

トランザクション領域の実行には実装固有のオーバーヘッドがあります。そのほとんどは固定コストで、残りはさまざまな動的コンポーネントによるものです。このオーバーヘッドはクリティカル・セクションのサイズやメモリー使用量とはほとんど関係なく、通常マイクロアーキテクチャーのアウトオブオーダー実行によって相殺されます。しかし、第 4 世代インテル® Core™ プロセッサー実装では、特定のシーケンスでこのオーバーヘッドが大きくなる可能性があります。特にクリティカル・セクションが非常に小さく、タイトなループ内にある場合 (例えば、マイクロベンチマークで行われる処理など)、オーバーヘッドが大きくなります。実際のアプリケーションでは、通常このような動作は見られません。

このオーバーヘッドは大きなクリティカル・セクションでは相殺されますが、非常に小さなクリティカル・セクションでは相殺されません。オーバーヘッドを減らす簡単なアプローチの 1 つは、クリティカル・セクションの早期にトランザクション・キャッシュラインへアクセスすることです。コミットのオーバーヘッドは、Broadwell† マイクロアーキテクチャー・ベースのプロセッサーでは軽減されています。

16.6 デバッグのガイドライン

インテル® TSX を使用するロック省略の実装はアプリケーションのセマンティクスを変更しません。つまり、アボートされたトランザクション実行中に更新されたすべてのアーキテクチャー・ステートは、ハードウェアによって自動的に破棄されます。アプリケーションにトランザクション実行でのみ実行される新しいコードパスを追加する際は注意が必要です (16.2.5 節を参照)。

ただしロックの省略では、データ競合が起こった場合にのみスレッド間の通信が発生するため、スレッド間の実行タイミングが変わります。そのためロックが通常よりも速く行われるように見えることがあります。このタイミングの違いはアプリケーションに潜在的な問題をもたらす可能性があります。この潜在的な問題はインテル® TSX 固有のものではなく、新しい世代のすべてのハードウェアで見られます。

コードのインストルメンテーションは、マルチスレッド・ソフトウェアのデバッグでよく使用される手法です。タイミング関連の問題をデバッグする場合と同様に、コードのインストルメンテーションを行う際は、タイミングを大きく変えたり、不要なアボートを引き起こさないように注意する必要があります。スレッドごとにバッファーを利用して、実行をトレースし特定のイベントを記録できます。タイムスタンプの取得には RDTSC 命令を使用できます。バッファーの出力はクリティカル・セクション外で行うべきです。

トランザクション・アボートは、トランザクション領域内で更新されたメモリー状態をすべて破棄します。この情報はインストルメンテーションでなければトレースできません。トランザクション領域内の問題は、プロファイリング・ツールではトランザクション・アボートとして検出され、LBR 情報から制御フローを再構成できます。インテル® プロセッサー・トレースをサポートするプロセッサーでは、トランザクション内部の制御フローを完全に追跡して記録することを可能にするトレースログを使用できます。このトレースには、トランザクションの開始、コミット、およびアボートを示すマーカーが含まれます。

通常の assert() 関数はトランザクション・アボートになり、その出力情報はトランザクション領域外では利用できません。RTM 命令を使用することで assert 関数は、トランザクション実行を終了し、その影響を可視化して、assert 関数でプログラムを終了することができます。次に例を示します。

```
assert(x) => if (!(x)) { while (_xtest()) _xend(); assert(0); }
```

16.7 インテル® TSX 用の一般的な組込み関数

新しいアセンブラー (GNU* binutils 2.23、Microsoft* Visual Studio* 2012) はインテル® TSX 命令をサポートしています。以前のツールチェーンではインテル® TSX 命令をバイト値として表現します。

16.7.1 RTM C 組込み関数

新しい C/C++ コンパイラー (gcc 4.8、Microsoft* Visual Studio* 2012、インテル® C++ コンパイラー 17.0) は、immintrin.h ヘッダーファイルで RTM 組込み関数を定義しています。RTM は新しい命令セットであり、CPUID 命

令で RTM 機能フラグを確認してから使用すべきです (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の第 3 章を参照)。

_xbegin()

_xbegin() はトランザクション領域を開始して、トランザクション領域に入ると _XBEGIN_STARTED を返し、そうでない場合はアボートコードを返します。_xbegin() の戻り値が _XBEGIN_STARTED (0 でない値) であることを確認することが重要でです。ゼロはアボートコードです。値が _XBEGIN_STARTED でない場合、リターンコードには、_xabort() によって渡される各種ステータスビットとオプションの 8 ビット定数が含まれます。

以下に、有効なステータスビットを示します。

- _XABORT_EXPLICIT: _xabort() によって発生したアボート。_XABORT_CODE(status) には、_xabort() へ渡された値が含まれます。
- _XABORT_RETRY: このビットが設定されている場合は、再試行によりトランザクション領域をコミットできる可能性があります。設定されていなければ、再試行しても成功する確率が低いままです。
- _XABORT_CAPACITY: 容量のオーバーフローによるアボートです。
- _XABORT_DEBUG: デバッグトラップによるアボートです。
- _XABORT_NESTED: 入れ子のトランザクションで発生したアボートです。

_xend()

_xend() はトランザクションをコミットします。

_xtest()

_xtest() は、コードが現在トランザクション実行中の場合は真を返します。HLE でも利用できます。

_xabort()

_xabort(constant) は現在のトランザクションをアボートします。constant は 8 ビットの定数でなければなりません。この定数は _xbegin() によって返されるステータスコードに含まれており、_XABORT_EXPLICIT フラグが設定されている場合、_XABORT_CODE() でアクセスできます。推奨される利用法については 16.4.5 節を参照してください。

gcc 4.8 以降では、-mrtm コンパイラー・オプションを指定してこれらの組込み関数を有効にする必要があります。

16.7.1.1 古い gcc* 互換コンパイラーによる RTM 組込み関数のエミュレート

immintrin.h で RTM 組込み関数をサポートしていない古い gcc 互換コンパイラーでは、例 16-11 に示す等価なインライン・アセンブラーを利用できます。

例 16-11 古い gcc 互換コンパイラーによる RTM 組込み関数のエミュレート

```
/* immintrin.h でこのインターフェイスをサポートしている新しいツールでは不要 */
#define _XBEGIN_STARTED (~0u)
#define _XABORT_EXPLICIT (1 << 0)
#define _XABORT_RETRY (1 << 1)
#define _XABORT_CONFLICT (1 << 2)
#define _XABORT_CAPACITY (1 << 3)
#define _XABORT_DEBUG (1 << 4)
#define _XABORT_NESTED (1 << 5)
#define _XABORT_CODE(x) (((x) >> 24) & 0xff)
#define __force_inline __attribute__((__always_inline__)) inline
```

```
static __force_inline int _xbegin(void)
{
   int ret = _XBEGIN_STARTED;
   asm volatile(".byte 0xc7,0xf8 ; .long 0" : "+a" (ret) :: "memory");
   return ret;
}
```

```
static __force_inline void _xend(void)
{
   asm volatile(".byte 0x0f,0x01,0xd5" ::: "memory");
}
```

```
static __force_inline void _xabort(const unsigned int status)
{
   asm volatile(".byte 0xc6,0xf8,%P0" :: "i" (status) : "memory");
}
static __force_inline int _xtest(void)
{
   unsigned char out;
   asm volatile(".byte 0x0f,0x01,0xd6 ; setnz %0" : "=r" (out) :: "memory");
   return out;
}
```

16.7.2 gcc およびその他の Linux* 互換コンパイラーの HLE 組込み関数

Linux* および互換システムでは、HLE は gcc 4.8 および古い形式の C11 のアトミック・プリミティブ拡張として実装されています。HLE XACQUIRE を使用するにはメモリーモデル引数に __ATOMIC_HLE_ACQUIRE フラグを設定し、HLE XRELEASE を使用するには __ATOMIC_HLE_RELEASE フラグを設定します。

メモリーモデルは、__ATOMIC_HLE_ACQUIRE では __ATOMIC_ACQUIRE 以上、__ATOMIC_HLE_RELEASE では __ATOMIC_RELEASE 以上でなければなりません。失敗メモリーモデルと成功メモリーモデルを含む操作 (__atomic_compare_exchange_n など) では、HLE フラグは成功メモリーモデルでのみサポートされます。

HLE は、IA アトミック命令に直接変換可能なアトミック操作でのみサポートされます。次の場合はサポートされません。

- 32 ビットのターゲット上の 8 バイト値
- 16 バイト値
- 加算/減算を除く、結果にアクセスするフェッチ命令または命令フェッチ
- __atomic_store と __atomic_clear は、__ATOMIC_HLE_RELEASE のみサポート

16.7.2.1 gcc 4.8 による HLE 組込み関数の生成

gcc 4.8 のいくつかのバージョンでは、コンパイラーの不具合により、アトミック組込み関数を用いて HLE ヒントを生成するには最適化レベル -O2 以上を指定しなければなりません。

16.7.2.2 C++11 atomic のサポート

gcc 4.8 は C++11 の <atomic> ヘッダーをサポートしています。このヘッダーで定義されているメモリーモデルは、C アトミック・インターフェイスに似た HLE フラグで拡張されています。2 つの新しいフラグ __memory_order_hle_acquire と __memory_order_hle_release が定義されています。C アトミック組込み関数の制限が適用されます。

例 16-12 に C++ を使用した HLE 組込み関数の例を示します。

例 16-12 HLE 組込み関数の C++ の例

```
#include <atomic>
#include <immintrin.h>
using namespace std;
atomic_flag lock;
for (;;) {
   if (!lock.test_and_test(memory_order_acquire|__memory_order_hle_acquire) {
   // HLE によるロックの省略を使用するクリティカル・セクション
      lock.clear(memory_order_release|__memory_order_hle_release);
      break;
   } else {
      // ロックを取得できなかったため待機して再試行する
      while (lock.load())
         _mm_pause(); // ロックがビジーなためトランザクション領域をアボートする
   }
}
```

16.7.2.3 古い gcc 互換コンパイラーによる HLE 組込み関数のエミュレート

これらの組込み関数をサポートしていない古いコンパイラーではインライン・アセンブリーを利用できます。例 16-13 に、__atomic_exchange_n(&lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE) をエミュレートする例を示します。

表 16-13 古い GCC コンパイラーでの HLE 組込み関数のエミュレーション

```
#define XACQUIRE ".byte 0xf2; " /* XACQUIRE をサポートしない古いアセンブラー向け*/
#define XRELEASE ".byte 0xf3; "
static inline int hle_acquire_xchg(int *lock, int val)
{
   asm volatile(XACQUIRE "xchg %0,%1" : "+r" (val), "+m" (*lock) :: "memory");
   return val;
}
static void hle_release_store(int *lock, int val)
{
   asm volatile(XRELEASE "mov %0,%1" : "r" (val), "+m" (*lock) :: "memory");
}
```

16.7.3 Windows* C/C++ コンパイラーの HLE 組込み関数

Windows* C/C++ コンパイラー (Microsoft Visual Studio 2012 およびインテル® C++ コンパイラー 17.0) は、HLE プレフィクスを持つ固有の atomic 組込み関数を提供しています。例 16-14 を参照してください。

例 16-14 インテルと Microsoft コンパイラーによる HLE 組込み関数のサポート

アトミックな比較-交換操作:

```
long _InterlockedCompareExchange_HLEAcquire(long volatile *Destination, long
Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLEAcquire(__int64 volatile *Destination,
__int64 Exchange, __int64 Comparand);
void * _InterlockedCompareExchangePointer_HLEAcquire(void * volatile *Destination,
void * Exchange, void *Comparand);
long _InterlockedCompareExchange_HLERelease(long volatile *Destination, long
Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLERelease(__int64 volatile *Destination,
__int64 Exchange, __int64 Comparand);
void * _InterlockedCompareExchangePointer_HLERelease(void * volatile
*Destination, void * Exchange, void *Comparand);
```

アトミックな加算:

```
long _InterlockedExchangeAdd_HLEAcquire(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLEAcquire(__int64 volatile *Addend, __int64
Value);
long _InterlockedExchangeAdd_HLERelease(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLERelease(__int64 volatile *Addend, __int64
Value);
```

HLE プリフィクス付きのストア組込み関数:

```
void _Store_HLERelease(long volatile *Destination, long Value);
void _Store64_HLERelease(__int64 volatile *Destination, __int64 Value);
void _StorePointer_HLERelease(void * volatile *Destination, void * Value);
```

組込み関数の詳細については、コンパイラーのドキュメントを参照してください。

第 17 章
モバイル利用における電力の最適化

17.1 概要

モバイル・コンピューティングでは、時間や場所にかかわらずコンピューターを使用するため、バッテリー持続時間が重要な要素となっています。モバイル・アプリケーションでは、パフォーマンスと消費電力を考慮したソフトウェアの最適化が求められます。この章では、モバイル・プロセッサーの省電力手法[24] に関する説明を行い、バッテリー持続時間の延長のために開発者が利用できる推奨事項を紹介します。

マイクロプロセッサーは、アクティブに命令を実行して目的とする処理を行っている間、電力を消費します。また、非アクティブな状態 (停止状態) でも電力を消費します。プロセッサーがアクティブな状態で消費する電力は、アクティブ電力と呼ばれます。そして、プロセッサーが停止した状態で消費する電力は、スタティック電力と呼ばれます。

ACPI 3.0 (ACPI は Advanced Configuration and Power Interface の略) は、高度な電力の管理および消費を達成するための規格です。必要に応じてデバイスの電源を投入したり、アプリケーションの要求に対してプロセッサー速度の制御を管理することを可能にします。この規格では、アクティブ消費電力の管理を容易にする P ステートと、スタティック消費電力の管理を容易にする C ステートタイプ[25] がそれぞれ定められています。

インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、インテル® Core™ Duo プロセッサー、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーには、アクティブ消費電力とスタティック消費電力を削減することを目的として設計された以下の機能が実装されています。

- 拡張版 Intel SpeedStep® テクノロジー: オペレーティング・システム (OS) を、ワークロード実行中にプロセッサーが周波数/電圧レベルを低い状態に遷移できるようにプログラムできます。
- スリープ状態や ACPI C ステートなど各種の活動状態をサポートします。プロセッサー内のサブシステムへの電力を切断することによって、スタティック消費電力を削減します。

拡張版 Intel SpeedStep® テクノロジーでは、P ステートに対応した動作ポイント間で低レイテンシーの遷移が可能です。一般に、数値の大きな P ステートは、低周波数で動作しアクティブ消費電力を削減できます。数値の大きな C ステートタイプは、スタティック電力をより積極的に削減します。ただし、数値の大きな C ステートほど、他のステートに遷移するときのレイテンシーが増加します。

17.2 モバイル環境での利用シナリオ

モバイル環境での利用モデルでは、バッテリー電力での運用時に大きな負荷が突発的に発生することがあります。オフィス、Web、ストリーミング・アプリケーションのワークロードのほとんどは、それほどパフォーマンスを必要としません。拡張版 Intel SpeedStep® テクノロジーを利用すると、パフォーマンスの履歴を追跡し、プロセッサーの周波数と電圧を調節するためのポリシーを OS に実装できます。直前の 300ms[26] で要求が変化した場合、OS は要求に対応できる最低周波数を選択して、ターゲットの P ステートを最適化できます。

例えば、プロセッサー使用率が 100% から低下した後、100% に戻るアプリケーションについて考えてみます。図 17-1 では、OS がプロセッサーの周波数を変更することによって要求を満たし、消費電力を調節する仕組みを示しています。次に、OS のパワー・マネジメント・ポリシーとパフォーマンス履歴との相互作用について説明します。

---

[24] インテル® Centrino® モバイル・テクノロジーとインテル® Centrino® Duo モバイル・テクノロジーについて、本書ではプロセッサーに関連するテクノロジーのみをカバーします。

[25] ACPI 3.0 規格では、4 つの C ステート (C0、C1、C2、C3) を定義しています。ACPI 規格を実装するマイクロプロセッサーは、それぞれの ACPI C ステートタイプごとにプロセッサー固有のステートを割り当てます。

[26] この章では、大きさの順序や相対的な大きさを示す例として、電力管理における時定数 (300ms、100ms など) を使用します。実際の値は実装によって異なり、同じベンダーから供給されても製品ごとに異なる可能性があります。

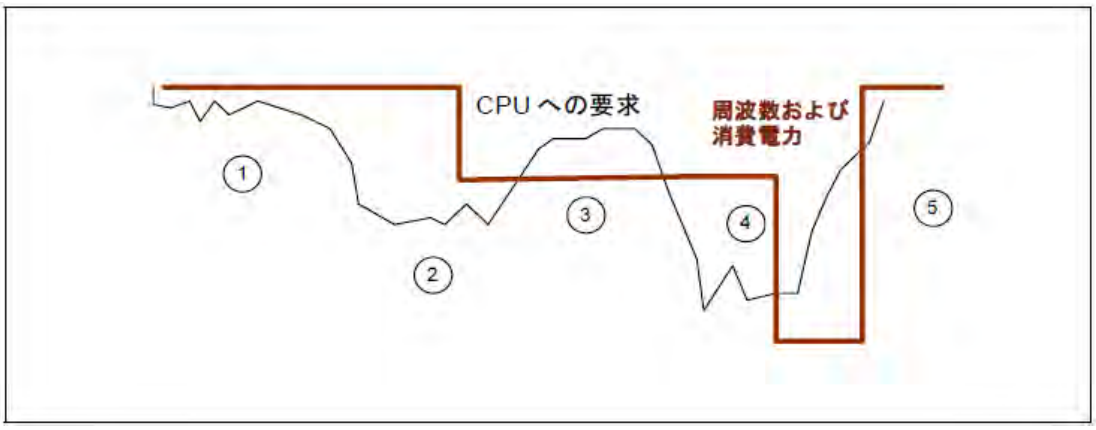


図 17-1 パフォーマンス履歴と状態遷移

1. 要求が多い場合、プロセッサーは最高周波数 (P0) で動作します。
2. 要求が減少すると、少し遅れて OS が認識し、プロセッサーを低周波数 (P1) に移行します。
3. プロセッサーの周波数が減少し、プロセッサー使用率が最も効率的なレベル (最高周波数の 80 ~ 90%) になります。その結果、同じ量のワークがより低い周波数で実行されます。
4. 要求が減少すると、OS はプロセッサーを最低周波数に移行します。この状態は、低周波数モード (LFM) とも呼ばれます。
5. 要求が増加すると、OS はプロセッサーを最高周波数に戻します。

17.2.1 電力効率に優れたソフトウェア

電力テクノロジーの近年の進歩とエンドユーザーが要求するさまざまなコンピューティング・シナリオにより、消費電力とパフォーマンスの適正なバランスはますます重要となっています。最新世代のアーキテクチャーで提供されるハードウェアの省電力機能を検討する上で、電力効率に優れたソフトウェアは重要な役割を果たします。コードが適切に作成されていないと、システムは新しいハードウェアの機能を利用したり、エンドユーザーの動的な要求に応じることができません。

モバイル・プラットフォームは、CPU、LCD、HDD、DVD、チップセットなど、さまざまなコンポーネントから構成され、それぞれがノートブックの消費電力に関係します。プラットフォームの主要コンポーネントの消費電力を理解することで、プラットフォーム全体の消費電力を適切に把握し、消費電力の最適化に対する手引きを得ることができ、ソフトウェアが構成コンポーネント間で消費電力量を動的に調整するのに役立ちます。

以下に、電力効率に優れたソフトウェアに関する全般的なガイドラインのいくつかを示します。

- アプリケーションは、プロセッサー周波数を自身で設定するのではなく、最新の OS 機能を利用して、適切な動作周波数を選択するべきです。プロセッサー周波数を独自に設定すると、消費電力とパフォーマンスの両方に悪影響を及ぼす可能性があります。
- アプリケーションがユーザー入力や何らかのイベントの待機状態になった場合、すぐにアイドルモードに切り替わるように最適化されたサービスを使用できるようにします。アイドル動作は、消費電力に大きな影響を及ぼす可能性があります。アプリケーションのほとんどがアイドル・コンテキストで動作することが判明した場合、プロセッサーをウェイクアップするアプリケーション・イベントの頻度を減らし、定期的なポーリングを控えて、メモリー内のアクティブなサービスの数を減らします。
- アプリケーションにコンテキストを認識する機能を組込み、長時間のバッテリー持続性と最適なユーザー・エクスペリエンスを実現します。
- 状況に応じた利用および最適なエンドユーザー・エクスペリエンスのため、消費電力の把握や動的なパワーポリシーをアプリケーションの設計に取り入れます。詳細は OS によって異なる可能性があります。Microsoft* Windows* については、http://msdn.microsoft.com/ja-jp/windows/hardware/gg463243.aspx# を参照してください。
- アプリケーションの消費電力の特性を評価します。プラットフォームの消費電力を測定する多くの手法が提供されています。

- Fluke NetDAQ* などのハードウェア・インストルメンテーションを使用します。これにより、CPU、HDD、メモリーなど、各コンポーネントの消費電力を測定できます。
- C ステート存在カウンターを使用します。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 4』の第 2 章「Model-Specific Registers (MSRs)」を参照してください。
- CPU の利用状況、カーネル時間、時間割り込み率などのパラメーターを検討して、ソフトウェアの動作を調査します。ハードウェア・インストルメンテーションを使用できない場合に、プラットフォームの消費電力に関連付けることができます。

17.5 節に、消費電力とパフォーマンスを関連付ける方法、およびソフトウェアを最適化する手法に関するいくつか例を示します。

17.3 ACPI C ステート

処理要求が 100% 未満の場合、プロセッサーは、一部の時間は有効な処理を行い、残りの時間はアイドル状態となります。例えば、プロセッサーは、Sleep() 関数によってセットされたアプリケーションのタイムアウト、Web サーバーの応答、ユーザーによるマウスクリックを待機することがあります。

図 17-2 に、アクティブ時間とアイドル時間との関係を示します。アプリケーションが待機状態になると、OS が HLT 命令を発行して、プロセッサーは停止状態に移行し、次の割り込みまで待機します。割り込みは、定期的なタイマー割り込みや、イベントを通知する割り込みの場合があります。

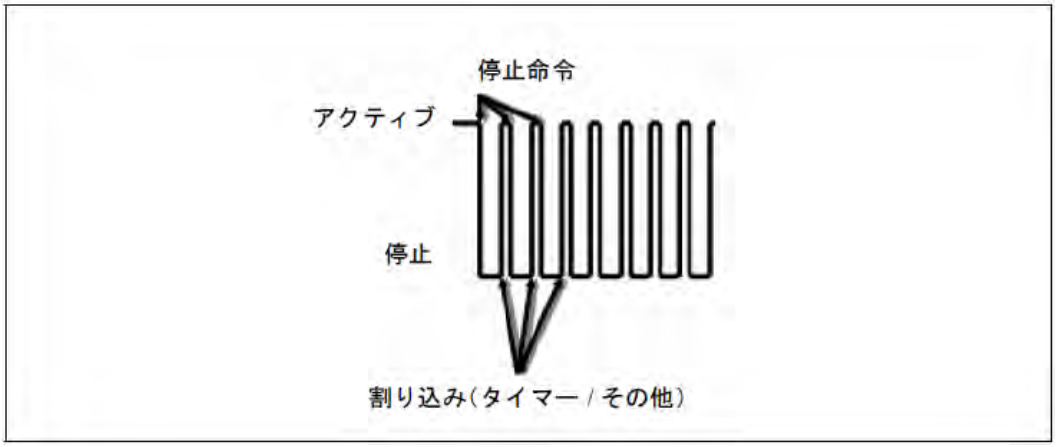


図 17-2 プロセッサーのアクティブ時間と停止時間

図 17-2 に示すように、プロセッサーはアクティブ状態またはアイドル (停止) 状態のいずれかです。ACPI では、4 つの C ステート (C0、C1、C2、C3) を定義しています。ACPI 規格のメカニズムを利用すると、プロセッサー固有の C ステートを ACPI C ステートタイプに割り当てることができます。C ステートタイプは、アクティブとアイドルの 2 つのカテゴリーに分けられ、アクティブ (C0) では、プロセッサーが電力をフルに消費します。そして、アイドル (C1 ~ 3) では、プロセッサーはアイドル状態であり、消費電力が大幅に減少します。

C ステートタイプのインデックスは、スリープ状態の深さを表しています。値が大きいほど、スリープ状態が深く、消費電力は少なくなります。ただし、ウェイクアップに要する時間が増加します (終了レイテンシーが長い)。

各 C ステートタイプは、以下のように定義されます。

- C0 — プロセッサーがアクティブで、処理を行うため命令を実行しています。
- C1 — レイテンシーが最も低いアイドル状態であり、終了レイテンシーも極めて低くなります。C1 の電力状態では、プロセッサーはシステムキャッシュの内容を維持します。
- C2 — このレベルは、C1 ステートよりも省電力が強化されています。主な強化点は、プラットフォーム・レベルで提供されます。
- C3 — このレベルは、C1 や C2 よりも省電力が強化されています。C3 では、プロセッサーがクロック発生とスヌープ活動を停止します。また、システムメモリーはセルフ・リフレッシュ・モードに移行できます。

OS のパワー・マネジメント・ポリシーを実装してスタティック消費電力を削減する基本的な手法としては、プロセッサーのアイドル時間を評価した上で、値の大きな C ステートタイプへの遷移を開始します。これは、プロセッサー利用率を評価した上で P ステートへの遷移を開始することにより、アクティブ消費電力を削減する手法と似ています。OS は、図 17-3 に示すように、特定の時間枠内の履歴を調べてから、次の時間枠でターゲットになる C ステートタイプを設定します。

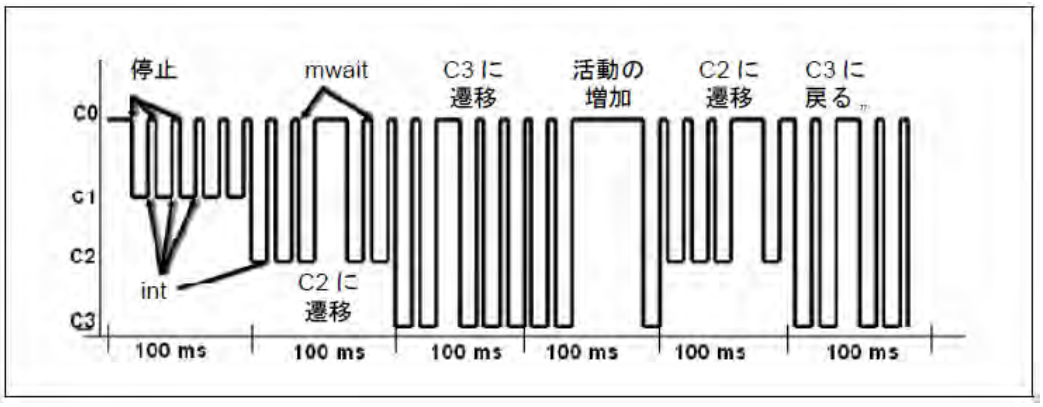


図 17-3 アイドル時間に対する C ステートの適用

プロセッサーが最低周波数 (LFM: 低周波数モード) で動作し、利用率が低い場合について考えてみます。最初のタイムスライス (図 17-3 の例では、C ステートの遷移の決定に 100ms のタイムスライスを使用) ではプロセッサー利用率が低く、OS は次のタイムスライスで C2 に遷移することを決定します。2 番目のタイムスライスの後も、プロセッサー利用率が依然として低いため、OS は C3 への遷移を決定します。

17.3.1 プロセッサー固有の C4 ステートとディープ C4 ステート

インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、インテル® Core™ Duo プロセッサー、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーには、プロセッサー固有の C ステート (および関連するサブ C ステート) が追加されており、ACPI C3 ステートタイプに割り当てることができます。[27] プロセッサー固有の C ステートとサブ C ステートに対しては、MWAIT 拡張命令を使用したアクセスや、CPUID 命令を使用した検出が可能です。スタティック消費電力の削減に適用されるプロセッサー固有の状態の 1 つは、C4 ステートと呼ばれます。C4 では、次の方法で節電が行われます。

- L2 キャッシュが状態を維持できる最低限のレベルまで、プロセッサーの電圧が削減されます。
- インテル® Core™ Solo プロセッサー、インテル® Core™ Duo プロセッサー、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーでは、C4 の状態が長時間続くと、スタティック電力をさらに節約するためディープ C4 ステートに移行する場合があります。

[27] インテル® Pentium® M プロセッサーは、ファミリー 6、モデル 9 または 13 の CPUID シグネチャーによって検出できます。インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーの CPUID シグネチャーは、ファミリー 6、モデル 14 です。インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーの CPUID シグニチャーは、ファミリー 6、モデル 15 です。

プロセッサーは、プロセッサーのコンテキストを維持するのに必要な最低レベルまで電圧を落とします。ディープ C4 ステートを終了するには、キャッシュのウォームが必要な場合があります。ただし、パフォーマンス上のペナルティーは小さく、バッテリー持続時間を延長できるメリットの方が、ディープ C4 ステートのレイテンシーを上回ります。

17.3.2 プロセッサー固有のディープ C ステートとインテル® ターボ・ブースト・テクノロジー

Nehalem† マイクロアーキテクチャー・ベースのプロセッサーでは、プロセッサー固有の C ステートが複数実装されています。

表 17-1 Nehalem† マイクロアーキテクチャー・ベースのモバイル・プロセッサーでの ACPI C ステートタイプとプロセッサー固有の C ステートのマッピング

ACPI C ステートタイプ	プロセッサー固有の C ステート
C0	C0
C1	C1
C2	C2
C3	C3

プロセッサー固有のディープ C ステートは実装依存です。通常、低電力の C ステート (値の大きい C ステート) ほど、終了レイテンシーが長くなります。例えば、コアがすでに C7 状態にあると、ラスト・レベル・キャッシュ (L3) がフラッシュされます。プロセッサーはディープ C ステート (C3/C7) に対する OS 要求のオートデモーションをサポートし、柔軟なパワーパフォーマンス設定をサポートするため、C1/C3 ステートへ降格 (デモート) します。

低電力のディープ C ステートだけでなく、インテル® ターボ・ブースト・テクノロジーは、P1 ステートをプロセッサーの適切な高周波数モード動作にマッピングすることで、通常ステート (C0) のパフォーマンスを状況に応じて向上できます。システムの TDP のヘッドルームを、P1 ステートのターゲットよりもさらに高い周波数へ転向できます。オペレーティング・システムが P0 ステートを要求すると、プロセッサーは P1 の範囲と P0 の範囲の間にコア周波数を設定します。1 つのコアだけがビジー状態の P0 ステートでは、インテル® ターボ・ブースト・テクノロジーの最大周波数が設定されるのに対し、プロセッサーが 2～ 4 のコアを使用している場合、周波数はプロセッサーによって制限されます。通常の状況下では、例えすべてのコアが実行中であっても、周波数が P1 を下回ることはありません。

17.3.3 Sandy Bridge† マイクロアーキテクチャーのプロセッサー固有ディープ C ステート

Sandy Bridge† マイクロアーキテクチャー・ベースのプロセッサーでは、プロセッサー固有の C ステートが複数実装されています。

表 17-2 Sandy Bridge† マイクロアーキテクチャー・ベースの ACPI C ステートタイプとプロセッサー固有の C ステートのマッピング

ACPI C ステートタイプ	プロセッサー固有の C ステート
C0	C0
C1	C1
C2	C2
C3	C6+C7

プロセッサー固有のディープ C ステートのマイクロアーキテクチャーの動作は、実装依存です。主な節電特性およびインテリジェントな応答特性の一部を以下に示します。

- モバイル・プラットフォームでは、コアがすでに C7 状態にあると、ラスト・レベル・キャッシュ (L3) がフラッシュされます。
- オートデモーション: 以下の場合、プロセッサーはターゲットの C ステート (コア C6/C7 ステートまたは C3 ステート) に対する OS 要求を、それよりも数値が小さい C ステート (コア C3 ステートまたは C1 ステート) に降格 (デモート) できます。
  - これまでの履歴により、C6/C7 ステートまたは C3 ステートが C3 ステートまたは C1 ステートよりも電力効率が低いことが判明している場合。
  - これまでの履歴により、ディーパー・スリープ・ステートがパフォーマンスに影響を及ぼすことが判明している場合。
  - ディーパー C ステートの遷移のオーバーヘッドが頻繁に発生することにより、電力効率が低下したり、パフォーマンスが低下することがあります。Sandy Bridge† マイクロアーキテクチャーには、この機能による電力利得を向上させる高度なアルゴリズムが装備されています。

- アンデモーション: ディーパー C ステートに対する OS 要求をオートデモーションにより降格 (デモート) し、C1 ステートまたは C3 ステートに移行できます。降格されたステートに長時間留まると、ハードウェアは制御を OS に戻します。そして、OS はディーパー C ステート要求を繰り返し、ハードウェアのアンデモーションにより、OS で要求されたディーパー C ステートに移行します。

17.3.4 インテル® ターボ・ブースト・テクノロジー 2.0

インテル® ターボ・ブースト・テクノロジー 2.0 は、インテル® ターボ・ブースト・テクノロジーの第 2 世代として機能が強化されています。インテル® ターボ・ブースト・テクノロジーは、TDP ヘッドルームに応じて、定格の周波数よりも高い周波数にプロセッサー・コアの周波数を上げることができます。

Sandy Bridge† マイクロアーキテクチャー・ベースのインテル® Core™ プロセッサーの TDP には、プロセッサー・コアおよびプロセッサー・グラフィック・サブシステムが含まれます。インテル® ターボ・ブースト・テクノロジー2.0 では、サーマルバジェットやパワーバジェットのヘッドルームを転化して、プロセッサー・コア周波数やプロセッサー・グラフィック・サブシステムの動作周波数を上げる可能性をさらに増やします。

プロセッサー・コアやプロセッサー・グラフィック・ユニットの消費電力は、一連の MSR インターフェイス[28] を介して測定できます。インテル® ターボ・ブースト・テクノロジーをサポートしたり、ヒントを基にターボモード動作でパフォーマンスと電力の偏りを最適化するのに必要なオペレーティング・システム要件については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A』の第 14 章「Power and Thermal Management」を参照してください。

17.4 バッテリー持続時間の延長に関するガイドライン

バッテリーを節約し、モバイル・コンピューティングの利用状況に応じて最適化を行うには、以下のガイドラインに従います。

- 最高のパフォーマンスではなく、必要な機能や体験を実現するのに十分なだけのパフォーマンスを得られるパワー・マネジメント方式を採用します。
- スピンループの使用を避けます。
- バッテリーでの動作中にアプリケーションが実行する処理量を減らします。
- ACPI C3 ステートタイプによるハードウェアの省電力機能を活用し、同じ物理プロセッサー上のプロセッサー・コア間の調整を行います。
- システムのスリープ状態 (S1 ~ S4) との遷移を適切に実装します。

---

[28] 一般に、こうした MSR インターフェイスに基づいた電力測定およびパワー・マネジメントの決定は、同じプロセッサー・ファミリー/モデル内で行い、異なるファミリー/モデルやサポートされていない環境条件で推定しないようにすべきです。

- プロセッサー・パフォーマンスに対する要求が少ない場合は、プロセッサーが値の大きな P ステート (周波数は低いが、消費電力あたりのパフォーマンスでは効率が高い) で動作できるようにします。
- プロセッサーの動作に対するユーザー要求の頻度が低い場合は、プロセッサーが値の大きな ACPI C ステートタイプ (より深い低電力状態) に移行できるようにします。

17.4.1 機能品質を満たすためのパフォーマンスの調整

システムがバッテリーで動作している場合、アプリケーションは、パフォーマンスや機能品質の制限、バックグラウンド処理の停止、またはその両方によってバッテリー持続時間を延長できます。このようなオプションをアプリケーションに実装することで、プロセッサーのアイドル時間が増加します。アイドル時のプロセッサーの消費電力は、アクティブ時よりも大幅に低いため、バッテリー持続時間の延長が可能です。

以下に、試みるべき手法の例を示します。

- ビデオやオーディオの再生時の品質/色分解能/解像度を制限します。
- 自動スペルチェックや文章校正を無効にします。
- ログ活動を無効にするか、頻度を減らします。
- ディスク操作を集中させ、ハードドライブの不要な回転をなくします。
- ビジュアルなアニメーションの量を減らすか、品質を制限します。
- ファイルスキャンやインデックス付けを無効にするか、可能な限り減らします。
- AC 電源が得られるまで待てる処理は延期します。

パフォーマンス、品質、バッテリー持続時間のバランスは、単一のセッション中でも変化することがあるので、実装は複雑です。ユーザーがそれぞれのニーズに合わせて設定を最適化できるようにするには、アプリケーションによるサポートが必要になることもあります (図 16-4 を参照)。

アプリケーションがバッテリー電力を認識できるようにするには、適切な OS API を使用します。Windows* の場合、以下の API が使用できます。

- GetSystemPowerStatus — システムの電力情報を取得します。このステータスでは、システムが AC 電源と DC 電源 (バッテリー) のいずれで動作しているか、現在バッテリーが充電中であるかどうか、またバッテリー寿命がどの程度残っているか示されます。
- GetActivePwrScheme — アクティブな電力方式 (現在のシステム電力方式) のインデックスを取得します。アプリケーションではこの API を利用して、システムが最適な電力方式で動作していることを保証できます。スピンループの使用は避けます。

スピンループは、短時間の待機や同期に使用されます。スピンループの主な利点は、応答時間が短いことです。Windows* API の PeekMessage() を使用しても、即座の応答という点で同様の利点が得られます (ただし、現在のマルチタスク OS ではほとんど必要とされていません)。

ただし、スピンループや、メッセージループにおける PeekMessage() を使用した場合、プロセッサーは常に動作していなければならないので、低電力状態に移行できません。可能な場合は適切な API に置き換える必要があります。次に例を示します。

- アプリケーションが数ミリ秒以上待機しなければならない場合、スピンループの使用は避け、WaitForSingleObject() など Windows* の同期 API を使用します。
- 即時応答が不要な場合、PeekMessage() の使用は避けます。メッセージがキューに格納されるまでスレッドをサスペンドするには、WaitMessage() を使用します。

インテル® モバイル・プラットフォーム SDK では、モバイル・プロセッサーや、プラットフォームのそのほかのコンポーネントの消費電力を管理および最適化するためモバイル・ソフトウェア向けに、豊富な API を提供しています。

17.4.2 処理量の削減

プロセッサーが C0 ステートにある場合、プロセッサーがバッテリーを消費する電力は、プロセッサーがアクティブなワークロードを実行する時間に比例します。最も明白な省電力手法は、ワークロードの完了に要するサイクル数を減らすことです (通常は、プロセッサーが実行に要する命令数の削減、またはアプリケーション・パフォーマンスを最適化することに相当します)。

アプリケーションの最適化は、まず効率良いアルゴリズムの採用から始めます。次に、インテル® VTune™ プロファイラー、インテル® コンパイラー、インテル® パフォーマンス・ライブラリーなどのインテル® ソフトウェア開発ツールを利用してそのアルゴリズムを改善します。

アプリケーション・ワークロードの完了時間を短縮するパフォーマンス最適化の詳細については、第 3 章から第 9 章を参照してください。

17.4.3 プラットフォーム・レベルの最適化

アプリケーションでは、デバイスを適切に使用し、ワークロードを再分配することによって、プラットフォーム・レベルの電力を節約できます。以下の手法により、パフォーマンスに影響を与えることなくさらに節電が可能となります。

- あらかじめ CD/DVD からデータを読み出し、メモリーやハードディスクにキャッシュしておくと、DVD ドライブの回転を停止できます。
- 使用していないデバイスの電源を切ります。
- ネットワークへの負荷が高いアプリケーションを開発する際は、節電可能な選択肢を活かせるようにします。例えば、WLAN と LAN が両方とも接続されている場合、WLAN から LAN に切り替えます。
- WLAN ではデータを大きなチャンクにして送信すると、連続するパケットとパケットの間に WiFi カードが低電力モードに移行できます。この節電方法は、省電力モードによっては、迅速なウェイクアップを可能にするため個々の送受信処理の後も WiFi カードが高電力モードを最大で数秒間維持することを考慮しています。
- 頻繁なディスクアクセスを回避します。ディスクをアクセスするごとに、ドライブが回転し、最後のアクセスからしばらくの間は高電力モードが維持されます。小規模なディスク読み出しと書き込みは RAM にバッファリングし、まとめてディスク操作を行います。また、GetDevicePowerState() Windows* API を使用してディスクの状態を調べ、回転していない場合はディスクアクセスを延期します。

17.4.4 スリープ状態への遷移

一部のケースでは、スリープ状態に遷移すると、アプリケーションが悪影響を受けます。例えば、アプリケーションがネットワーク上のファイルを使用中に、システムがサスペンドモードに移る場合などが考えられます。レジュームした時点でネットワーク接続を利用できないと、情報が失われる可能性があります。

スリープ状態への遷移を認識できるようにすることで、そのような状況でのアプリケーションの動作を改善できます。これは、WM_POWERBROADCAST メッセージの利用によって可能です。このメッセージには、アプリケーションが適切に反応するために必要なすべての情報が含まれています。

スリープモードへの遷移に対するアプリケーションの反応の例をいくつか紹介します。

- スリープ状態への遷移の前に状態やデータを保存し、ウェイクアップ後に復元する場合。
- ファイルや I/O デバイスなど、使用中のシステム・リソース・ハンドルをすべてクローズする場合 (複製されたハンドルも含みます)。
- スリープ状態への遷移の前に通信リンクをすべて切断し、ウェイクアップ時に再確立する場合 (ウェイクアップ時にリモート活動をすべて同期します)。
- リモートファイルやリモート・データベースへのライトバックを行う場合。
- スリープ状態への遷移の前に、ストリーミング・ビデオの再生やファイルのダウンロードなど実行中のユーザー活動をすべて中止し、ウェイクアップ後に再開する場合。

推奨事項: サスペンドイベントを適切に処理できれば、より高度なパフォーマンスを発揮する堅牢なアプリケーションを実現できます。

17.4.5 拡張版 Intel SpeedStep® テクノロジー

拡張版 Intel SpeedStep® テクノロジーを利用すると、低周波数で動作して電力を節約するようにプロセッサーを調整できます。基本的な概念としては、処理を小さく分割し、OS のパワー・マネジメント・ポリシーを適用することで、高い P ステートへの遷移を実現します。

OS は通常、およそ数 10 ミリ秒から数 100 ミリ秒の時定数[29] を利用して、プロセッサー・ワークロードでの要求を検出します。例えば、必要なサービス品質 (QOS) を達成するのにプロセッサー・リソースの 50% しか要さないアプリケーションについて考えてみます。タスクのスケジューリングは、プロセッサーが P0 ステート (最高のパフォーマンスを得るための最高周波数) を 0.5 秒間維持してから、スリープ状態に 0.5 秒間遷移する方法で行われます。この要求パターンが繰り返されます。

プロセッサーへの要求は 0.5 秒ごとに 0% と 100% の間で切り替わり、プロセッサー・リソースの使用率は平均で 50% になります。これに伴い、周波数は最高周波数と最低周波数の間で切り替わります。消費電力も同様に切り替わるため、平均消費電力は Paverage = (Pmax+Pmin)/2 の式で表せます。

図 17-4 は、広い間隔 (> 300ms) でのタスク・スケジューリングと、動作周波数と消費電力に対するその影響を示した時系列グラフです。

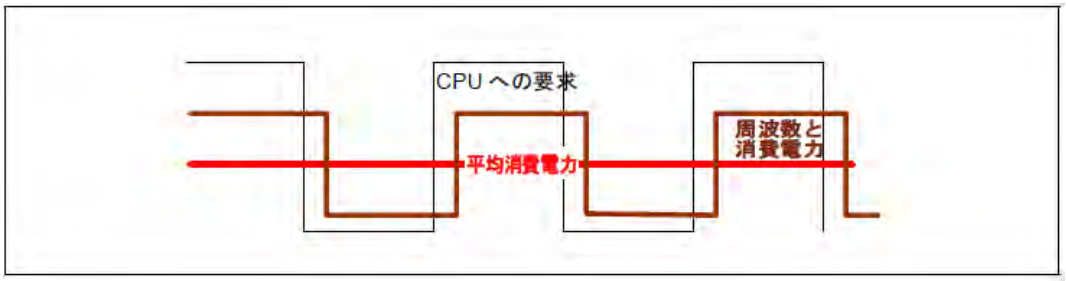


図 17-4 広い間隔でのタスク・スケジューリングおよび消費電力のグラフ

このアプリケーションは、処理単位を小さく分割することも可能で、各処理単位と Sleep() がもっと頻繁な間隔 (100ms など) で発生するようにスケジューリングすれば、同等の QOS を確保できます (50% の時間はフル・パフォーマンスで動作)。この場合、ワークロードは各サンプリングで 300ms の間フル・パフォーマンスを必要としているわけではないことを OS が認識します。すると、パワー・マネジメント・ポリシーは、QOS のレベルを維持しながら、プロセッサーの周波数と電圧を下げ始めます。

アクティブ消費電力、周波数、電圧の関係は、次の式で表せすことができます。

$$Power = \alpha * C * V^2 * F$$

この式では、「V」はコア電圧、「F」は動作周波数、「α」は作動係数を表わします。通常、50% のデューティー・サイクルにおける 100% のパフォーマンスの QOS は、100% のデューティー・サイクルにおける 50% のパフォーマンスによって達成できます。ほとんどのワークロードでの周波数スケーリング効率の傾きは 1 未満なので、コア周波数を 50% に削減しても、元のパフォーマンスの 50% 以上になります。また、コア周波数を 50% に削減すると、コア電圧も大幅に減少します。

高い P ステート (低電力状態) で命令を実行する場合、1 命令あたりの消費電力は P0 ステートよりも少なくなります。そのため、P0 ステートにおける半分のデューティー・サイクル (Pmax/2) に比例した節電では、非アクティブ

[29] 実際の値は、OS や OS のリリースによって異なります。

な消費電力に比例して半分のデューティー・サイクル (Pmin /2) が増加しても、それを補う効果が得られます。周波数に対する消費電力と電圧との非リニアな関係は、タスク単位を小さく変更すると、電力を大幅に節約できることを示しています。このような最適化は、メディア・ストリーミングの利用、DVD の再生、リソースをあまり必要としないアプリケーション (ワード・プロセッサー、電子メール、Web 閲覧) の実行時など、プロセッサーへの要求が少ない場合に可能です。

連続的に低周波数で動作させることには、ほかにもプラスの効果があります。消費電力 (この場合は低から高) やバッテリー電流が頻繁に変更される場合、バッテリーに悪影響を及ぼし、最終的にバッテリーの劣化が進みます。

最低限の動作ポイント (最高の P ステート) に達した場合、処理を分割する必要はありません。代わりに、アイドル状態を長時間維持することで、プロセッサーがより深い低電力モードに移れるようにします。

17.4.6 拡張版インテル® ディーパー・スリープを有効にする

一般的なモバイル・コンピューティングでは、プロセッサーはほとんどの時間アイドル状態です。バッテリーを節約するには、スタティック消費電力の削減に取り組まなければなりません。

一般の OS のパワー・マネジメント・ポリシーでは、定期的に低電力の C ステートに移行することによってスタティック消費電力を削減できる可能性を評価します。OS のパワー・マネジメント・ポリシーでは、通常アイドル状態が長くなるほど、より深い低電力の C ステートに移行するようプロセッサーに指示します。

アプリケーションは最低限の P ステートに達した場合、処理を大きなチャンクにまとめることによって、プロセッサーが処理の間により深い C ステートに移行できるようにします。この手法では、周波数変更の決定はディープ・スリープ・ステートへの遷移の決定よりも長い時間枠に基づいて行われる事実を利用しています。プロセッサー固有の C4 ステートに移行して積極的なスタティック電力削減機能を利用する場合、以下に基づいて判断します。

- プロセッサーが長時間にわたり終了レイテンシーの高い低電力状態にあっても、QOS を維持できるか。
- プロセッサーが C4 の状態である時間が、この低電力 C ステートの高い終了レイテンシーを吸収できるだけ長いか。

この時間が十分に長ければ、プロセッサーはディーパー・スリープ・ステートに移行して、大幅に電力を節約できます。アプリケーションで拡張版インテル® ディーパー・スリープを利用する際は、以下のガイドラインが役立ちます。

- 高い割り込み率の設定を回避します。割り込み間の時間が短いと、OS が低電力状態に移れなくなる可能性があります。ディープ C ステートとの遷移では、レイテンシー・ペナルティーが生じるのに加えて、電力が消費されます。一部のケースでは、オーバーヘッドが節電量を上回ることがあります。
- ハードウェアのポーリングを回避します。ACPI C3 タイプステートでは、プロセッサーがスヌープを停止する場合があります。また、各バス・アクティビティー (DMA やバス・マスタリングを含む) では、プロセッサーは値の小さな C ステートタイプへ移行することを要求します。値の小さな C ステートタイプは通常 C2 を指しますが、C0 のこともあります。インテル® Core™ Solo プロセッサーでは、従来世代のインテル® Pentium® M プロセッサーに比べて、状況が大幅に改善されています。ただし、ポーリングを行うと、値の最も大きなプロセッサー固有の C ステートに移行できなくなる可能性があります。

17.4.7 マルチコアに関する考慮事項

マルチコア・プロセッサーでは、節電の計画に際して特別な考慮が必要です。インテル® Core™ Duo プロセッサーとインテル® Core™ マイクロアーキテクチャー・ベースのモバイル・プロセッサーのデュアルコア・アーキテクチャーでは、マルチスレッド・アプリケーションのさらなる節電の可能性を提供しています。

17.4.7.1 拡張版 Intel SpeedStep® テクノロジー

ドメイン構成によりシングルスレッド・アプリケーションを再構成することで、マルチコア・プロセッサーの活用が可能となります。2 つのドメインスレッドに再構成すると、各スレッドは元の命令数の約半分を実行します。デュアルコア・アーキテクチャーでは、2 つのスレッドを同時に実行でき、各スレッドはプロセッサー・コアに専用リソースを持っています。モバイル環境での利用を目的としたアプリケーションでは、各スレッドの命令数を削減することで、物理プロセッサーはシングルスレッド・バージョンに比べて低い周波数で動作できます。その結果、プロセッサーが低電圧で動作できるようになり、バッテリーの節約につながります。

OS は、物理プロセッサー上の各論理プロセッサーやコアを個別のエンティティーとして認識しており、論理プロセッサーやコアごとに独立して CPU 使用率を計算します。要求があれば、OS は物理パッケージ上で利用可能な最高の周波数で動作することを選択します。その結果、2 つのコアを搭載した物理プロセッサーは、多くの場合、目標の QOS を満たすのに求められる周波数よりも高い周波数で動作します。

例えば、一方のスレッドがシングルスレッド実行サイクルの 60% を必要とし、他方のスレッドが 40% を必要とする場合、OS のパワー・マネジメント・ポリシーでは、最大周波数の 60% で動作するように物理プロセッサーを設定することがあります。

ただし、各スレッドで必要な実行サイクルが 50% になるように、スレッド間で均等に処理を分割することもできます。その結果、いずれのコアも最大周波数の 60% ではなく 50% で動作可能になります。これにより、物理プロセッサーが低電圧で動作できるようになり、バッテリーを節約できます。

したがって、アプリケーションのプランニングとチューニングでは、最低限の周波数と電圧ポイントで動作できるように、スレッドをできる限り対称的にする必要があります。

17.4.7.2 スレッドの移行に関する考慮事項

OS スケジューリングとマルチコアに対応しないパワー・マネジメント・ポリシーの相互作用により、マルチスレッド・アプリケーションでパフォーマンス異常が生じる可能性があります。この問題は、スレッドが頻繁にコア間を移行するマルチスレッド・アプリケーションで発生します。

最高速で動作しているスレッドが一方のコアから、長時間アイドル状態となっている他方のコアに移行する場合、マルチコアを認識可能な P ステート調整ポリシーを備えていない OS では、各コアがプロセッサー・リソースの 50% しか要求しない (アイドル履歴に基づいて誤った判断を下す) 場合があります。このような場合、マルチコアをサポートしない P ステート調整ポリシーによってプロセッサーの周波数が落とされ、パフォーマンスに影響する可能性があります。図 17-5 を参照してください。

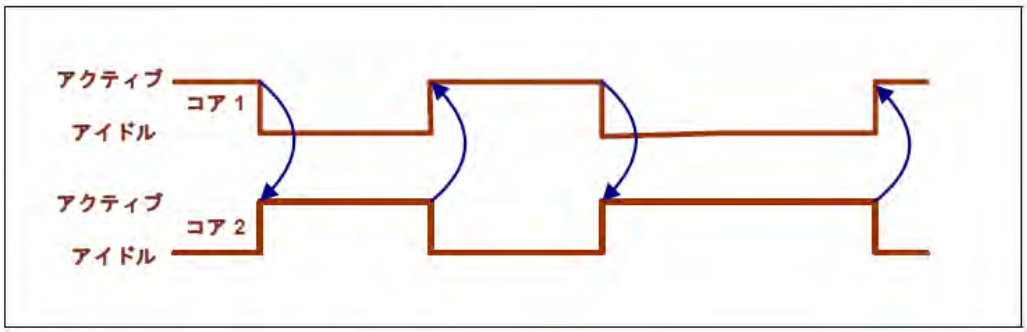


図 17-5 マルチコア・プロセッサーでのスレッドの移行

ソフトウェア・アプリケーションでは、この問題の発生を防ぐいくつかの手法があります。

- スレッド・アフィニティーの管理 — マルチスレッド・アプリケーションは、プロセッサー・トポロジーを検出して、プロセッサー・アフィニティーをアプリケーション・スレッドに割り当てることにより、スレッドの移行を防止できます。この方法では、マルチコアを認識可能な P ステート調整ポリシーを OS が備えていない場合の問題を回避できます。
- マルチコアをサポートする P ステート調整ポリシーを備えた OS へのアップグレード — 多くの新しい OS リリースでは、マルチコアをサポートする P ステート調整ポリシーを備えています。詳細については、各 OS ベンダーにお問い合わせください。

17.4.7.3 C ステートでのマルチコアに関する考慮事項

マルチコア・プロセッサーで C ステートに影響を与えるのは、以下の 2 点です。

マルチコアを認識不可能な C ステート調整では、十分な節電を達成できません。

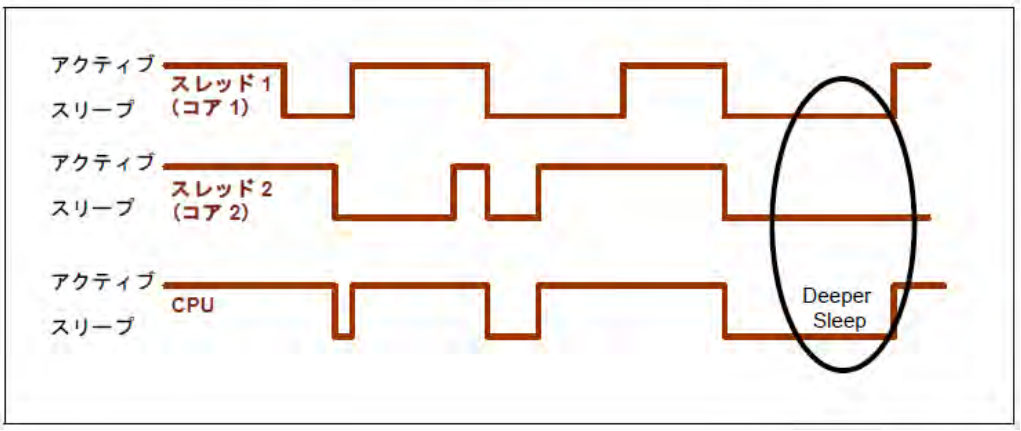


図 17-6 ディーパー・スリープへの遷移

マルチコア・プロセッサーの各コアが異なる C ステートタイプへ移行するのに必要な条件を満たしている場合、マルチコアをサポートしない (認識不可能な) ハードウェアでは、物理プロセッサーが最低限の C ステートタイプに遷移します (値の小さな C ステートは節電効果が低くなります)。例えば、コア 1 が ACPI C1 の条件を満たし、コア 2 が ACPI C3 の条件を満たしている場合、マルチコアをサポートしない OS 調整では、物理プロセッサーが ACPI C1 に遷移します。図 17-6 を参照してください。

両方のコアで拡張版インテル® ディーパー・スリープの利点を活用します。

マルチスレッド・アプリケーションでは、プロセッサー固有の C ステート (拡張されたディーパースリープなど) を最大限に利用してバッテリーを節約するには、スレッドを同期する必要があります。つまり OS 同期プリミティブを用いて、スレッドを同時に処理し、同時にスリープ状態に移すべきです。長時間フルアイドル状態にしておくと (ACPIC3 の条件を満たすと)、物理プロセッサーは、プロセッサー固有のディープ C4 ステートを透過的に利用できます (利用可能な場合)。

マルチスレッド・アプリケーションは、スレッド実行の負荷インバランスを認識および修正してから、スレッド同期を実装すべきです。スレッドのインバランスの認識には、パフォーマンス監視イベントを利用できます。インテル® Core™ Duo プロセッサーでは、この用途のイベントを用意しています。このイベント (Serial_Execution_Cycle) は、以下の条件でカウントアップされます。

- コアが C0 ステートでコードをアクティブに実行している。
- 物理プロセッサー上の第 2 のコアがアイドル状態 (C1 ~ C4) である。

ソフトウェア開発者は、Serial_Execution_Cycle と Unhalted_Ref_Cycles を比較することによって、シリアル実行されているコードを検出できます。シリアル化されたコード領域を 2 つのスレッドで並列化し、スレッド同期を調整することで、節電効果を高められます。

Serial_Execution_Cycle は、インテル® Core™ Duo プロセッサー上でのみ利用できます。ただし、負荷がインバランスなアプリケーション・スレッドは通常、基盤となるマイクロアーキテクチャーの違いにかかわらず、対称的なアプリケーション・スレッドや、対称的に構成されたマルチコア・プロセッサーでも同じように振る舞います。

このような理由から、負荷インバランスを識別する手法は、インテル® Core™ Duo プロセッサー固有のものではなく、マルチスレッド・アプリケーション全般に適用できます。

17.5 インテリジェントな電力消費を実現するソフトウェアの調整

この節では、消費電力とパフォーマンス両方のバランスをとるようにソフトウェアを調整するいくつかの手法について説明します。電力を最適化する手法のほとんどは汎用的なものです。最後の項 (17.5.8 節) で、Sandy Bridge† マイクロアーキテクチャーに固有の機能について説明します。これらの機能を検討して、パフォーマンスとそれに対する消費電力の利点が得られるようにソフトウェアを最適化します。

17.5.1 アクティブサイクルの削減

アクティブサイクルを減らしてタスクの完了を早め、システム・アイドル・ループに制御を移行することで、オペレーティング・システムの電力最適化機能を生かすことができます。

アクティブサイクルの削減は、第 3 章で取り上げたパフォーマンス重視のコーディング手法の適用から、SIMD 命令を使用したベクトル化、マルチスレッディングまで、いくつかの方法で達成できます。

17.5.1.1 マルチスレッド化によるアクティブサイクルの削減

スレッドレベルの並列処理が、ある一定量の計算処理からなるタスクである場合、データ分割によるマルチスレッド化を適用できます。アクティブサイクルの削減量は並列処理の程度に応じて異なります。同じような原則が機能分割した場合にも適用できます。

バランスのとれたマルチスレッド処理を実装できれば、結果としてさらに賢明で効率良いパフォーマンスと消費電力の利点を実現できます。適切な同期プリミティブを選択することも、消費電力とパフォーマンスの両方に影響します。

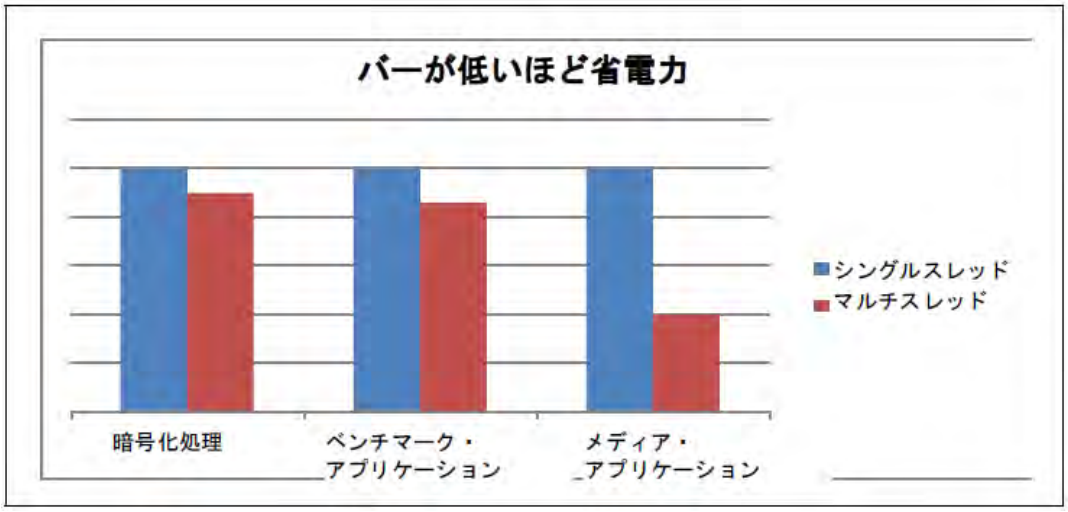


図 17-7 パフォーマンスの最適化による電力の削減

図 17-7 は、異なるアプリケーション・ドメインの 3 つのアプリケーションを使用して、シングルスレッド・ワークロードのプロセッサーの消費電力と、パフォーマンスを最適化した消費電力を比較して調査結果を示したものです。この調査によると、アプリケーション 1 (暗号化処理) は、最適化によりパフォーマンスが 2 倍向上しています。同時に、消費電力はおよそ 12% 削減されています。アプリケーション 3 (メディア・アプリケーション) では、マルチスレッド化やそのほかのパフォーマンス最適化によって、パフォーマンスが 10 倍向上しました。消費電力はおよそ 60% 削減されました。

17.5.1.2 ベクトル化

SIMD 命令を使用することで、特定の計算処理を完了するためのパスを短くでき、アクティブサイクルを削減できます。複数の独立したデータ要素に対し同じ操作を行うコードは、ベクトル化の候補となります。ベクトル化は、通常、単一の命令で処理できる要素のループを伴うアプリケーションに適用されます。一般的に、SIMD 命令を使用することで、単位時間当たりの消費電力が多少増えても、アクティブサイクルは大幅に削減されます。そして、最終的に消費電力の削減につながります。

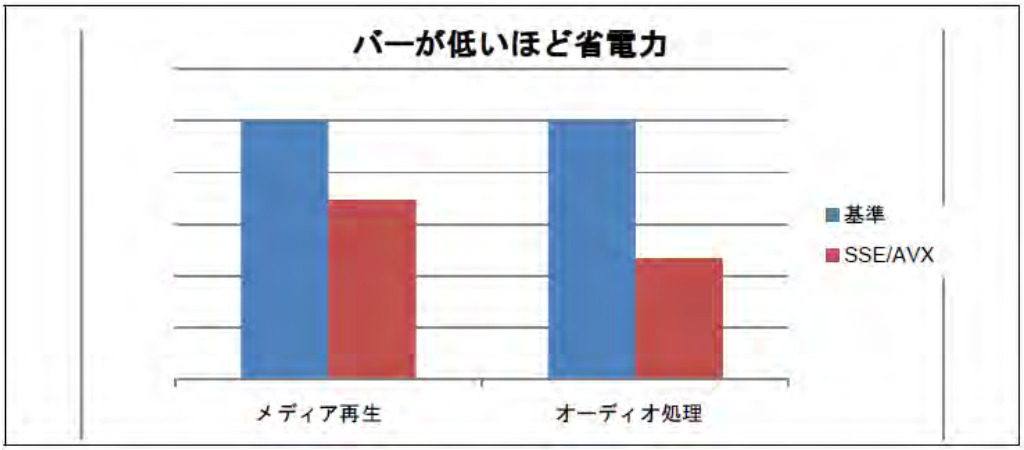


図 17-8 ベクトル化による電力の削減

図 17-8 に、ベクトル化による電力の削減効果に関する調査結果を示します。メディア再生のワークロードでは、インテル® SSE2/インテル® SSE4 命令セットを使用することにより、2.15 倍の高速化を達成しています。オーディオ処理ワークロードでは、インテル® AVX 命令セットを使用することにより、パフォーマンスが最大 5 倍向上しました。同時に、オーディオ処理ワークロードでは電力も削減されました。

17.5.2 PAUSE および Sleep(0) ループの最適化

マルチスレッド構成では、スレッドの同期は一般的であり、スケジューリングにより別のスレッドに移行するためタスクの実行を待機する構造を持つことがあります。これは、ループ内で Sleep(0) を発行することで実装されます。

これは、「スリープループ」と呼ばれます (例 17-1 を参照)。また、SwitchToThread 呼び出しも使用できます。「スリープループ」は、ロック・アルゴリズムやスレッドが処理を待機しているスレッドプールでよく見られます。

例 17-1 最適化されていないスリープループ

```
while(!acquire_lock())
{ Sleep( 0 ); }
do_work();
release_lock();
```

密なループにとどまり、パラメーター 0 で Sleep() サービスを呼び出すこの構造は、実際にはポーリングループであり、以下のような悪影響を招きます。

- Sleep() 呼び出しは、コンテキスト・スイッチングのコストが高く、10000 サイクル以上になることもあります。
- また、リング 3 からリング 0 への遷移もコストがかかり、1000 サイクル以上になる可能性があります。
- 制御の取得を待機しているスレッドがない場合、このスリープループは OS にとって CPU リソースを消費する非常にアクティブなタスクとなり、OS が CPU を低電力状態にする妨げとなります。

例 17-2 PAUSE を使用して省電力化されたスリープループ

```
if (!acquire_lock())
{ /* Sleep 関数に入る前に、max_spin_count 回まで pause 命令を繰り返す */
    for(int j = 0; j < max_spin_count; ++j)
    { /* PAUSE 命令の組込み関数 */
        _mm_pause();
        if (read_volatile_lock())
        {
           if (acquire_lock()) goto PROTECTED_CODE;
        }
    }
    /* PAUSE ループが動作しない場合に、Sleep 関数を実行 */
    Sleep(0);
    goto ATTEMPT_AGAIN;
}
PROTECTED_CODE:
do_work();
release_lock();
```

例 17-2 に、PAUSE 命令を使用して、スリープループを省電力化する手法を示します。

PAUSE 命令で「スピンウェイト」の速度を落とすことで、マルチスレッド・ソフトウェアでは以下のような利点が得られます。

- 待機状態のタスクがビジーウェイトからより簡単にリソースを獲得できるようにすることで、パフォーマンスが向上します。
- スピン時に使用するパイプラインのパスを減らします。
- Sleep(0) 呼び出しのオーバーヘッドを排除することで、消費電力を削減します。

あるケースでは、この手法によってパフォーマンスが 4.3 倍向上しました。これはプロセッサー・レベルでは 21% の消費電力の削減、プラットフォーム・レベルでは 13% の消費電力の削減となります。

17.5.3 スピンウェイト・ループ

すべてのスピンウェイト・ループで PAUSE 命令を使用します。PAUSE 命令によって、スピンウェイト・ループのパイプライン処理がなくなり、実行リソースの過度な消費や、不必要な電力の消費を防止できます。

スピンウェイト・ループを実行する場合、プロセッサーはメモリーアクセス順序違反の可能性を検出し、コア・プロセッサーのパイプラインをフラッシュするため、ループ終了時にかなりパフォーマンスが低下する可能性があります。

PAUSE 命令は、プロセッサーに対して、そのコードシーケンスがスピンウェイト・ループであるというヒントを与えます。プロセッサーはこのヒントにより、メモリーアクセス順序違反を回避し、パイプラインのフラッシュを防止できます。ただし、PAUSE によるスピンウェイト・ループは短くする必要があります。

17.5.4 コードでのポーリングに代わりイベントドリブンのサービスを使用

デバイスや状態の変化を絶えずポーリングすると、プラットフォームがウェイクアップし、消費電力が増える可能性があります。できる限りポーリングを最小限に抑えて、可能であればイベントドリブンのフレームワークを使用します。AC からバッテリーへの移行など各種デバイスの状態変化に対する通知サービスを OS が提供している場合、デバイスの状態の変化をポーリングする代わりにこれらのサービスを使用します。この新しいイベント通知フレームワークを使用すると、ステータスが変化した場合、コードは非同期に通知を取得できるため、電源ステータスをコードがポーリングするオーバーヘッドを削減できます。

17.5.5 割り込み率の低減

割り込み率が高いと、次の 2 つの影響により、プロセッサーの消費電力とパフォーマンスに支障をきたします。

- プロセッサーのパッケージおよびそのコアがディーパー・スリープ・ステート (C ステート) に入るのを阻止され、ハードウェアが省電力機能を利用できなくなります。
- インテル® ターボ・ブースト・テクノロジー 2.0 で上昇可能な周波数が制限されるため、プロセッサーで実行中のほかのアプリケーションのパフォーマンスが低下します。

ユーザーセッションやアプリケーションの割り込み率が 1 秒あたり数 1000 件に及ぶ場合、プロセッサーはパフォーマンスと省電力の適正なバランスをとることができなくなります。

この状況を回避するには、散発的なウェイクアップを最小限に抑えます。アプリケーションやドライバーの散発的なすべてのアクティビティーを 1 つのウェイクアップ期間内にスケジュールして、割り込み率を必要最小限に減らします。

メディア・アプリケーションの多くは、非常に短いタイマー周期 (1ms) を設定します。可能な限り、オペレーティング・システムのデフォルトのタイマー周期を使用します。どうしても短い周期が必要な場合は、タスクの終了時にソフトウェアがタイマー周期をリセットするようにします。

17.5.6 特権時間の削減

アプリケーションが特権モードで多くの時間を費やすと、さまざまな理由により、過度に電力が消費されます。例えば、システム呼び出し率が高くなったり、I/O ボトルネックが生じます。Windows* のパフォーマンス・モニター (リソースモニター) を使用すると、特権モードの時間を推測できます。

1 秒あたりのシステム呼び出し数によって測定されるシステム呼び出し率が高い場合、ソフトウェアはカーネルモードへの遷移を頻繁に行っていることを示します。つまり、アプリケーションはリング 3 (ユーザーモード) からリング 0 (カーネルモード) へ頻繁に移行します。典型的な例として、Win32 API の WaitForSingleObject() などのコストが高い同期呼び出しが頻繁に利用されることで発生します。これは、特にプロセス間通信で利用される非常に重要な同期 API です。ロックが行われているかどうかに関係なく、カーネルモードに移行します。ロックの競合が短時間または全くないマルチスレッド・コードでは、スピンカウント付きの EnterCriticalSection を使用すべきでしょう。WaitForSingleObject() と比較した場合、この API の利点はロックの競合がない限りカーネルモードに遷移しないことです。そのため、競合がない状態では、スピンカウント付きの EnterCriticalSection を使用した方がはるかにコストが低く、特権モードで費やす時間を削減できます。

Sandy Bridge† マイクロアーキテクチャー・ベースのシステムで、4 つのアクティブなスレッドを実行する小さなテスト・アプリケーションを使用して調査を実施しました。テストケースのロックの実装には、WaitForSingleObject と EnterCriticalSection を使用しています。ロックの競合がなかったため、各スレッドは 1 回のみロックを行っています。

下のグラフに示すように、競合がない状況で WaitForSingleObject() を使用すると、EnterCriticalSection() を使用したときと比べ、消費電力とパフォーマンスの両方に悪影響を及ぼします。

以下のグラフに示すように、競合しないロックで WaitForSingleObject() を使用した場合、より多くの電力が消費されます。EnterCriticalSection() を使用した方が、パフォーマンスは 10 倍向上し、消費電力が 60% 削減されます。

詳細は以下を参照してください。https://software.intel.com/en-us/articles/implementing-scalable-atomiclocks-for-multi-core-intel-em64t-and-ia32-architectures (英語)

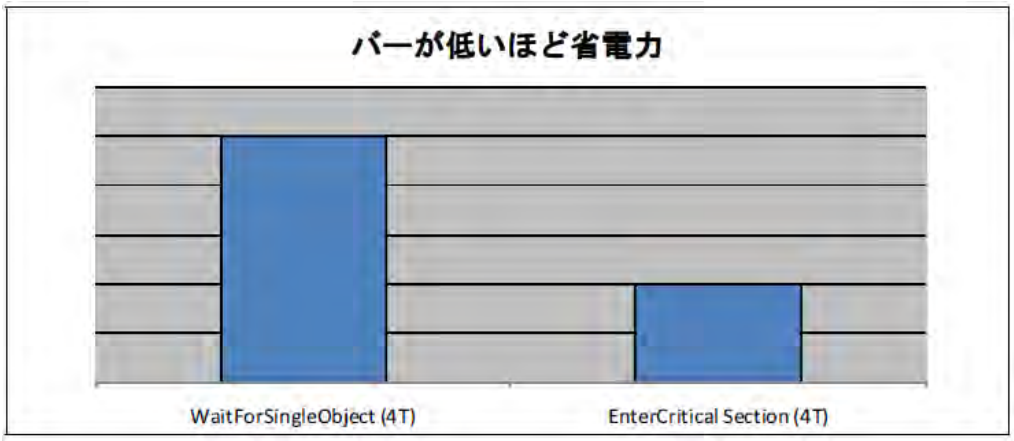


図 17-9 同期プリミティブの消費電力の削減の比較

17.5.7 コードでのコンテキスト認識の設定

コンテキストの認識は、電力リソースが限られている場合にソフトウェアが電力を節約する手段となります。電力を節約するために実装可能なコンテキスト認識の例をいくつか示します。

- バッテリー電力を使用してラップトップでゲームをプレイする場合、フレームレートを無制限にせず 60FPS に変更します。
- バッテリーでの動作中、および使用していないときは、ディスプレイを暗くします。
- ネットワークに接続していないときは、ワイヤレスなどのデバイスをオフにする簡単なオプションをエンドユーザーに提供します。

アプリケーションは、バッテリー電力でゲームをプレイしている時にこうした変更を透過的に行ったり、バッテリー持続時間を延長するヒントをユーザーに提供できます。いずれの場合も、アプリケーションはコンテキスト認識により、AC 電源ではなく、バッテリー電力が使用されていることを把握する必要があります。

異なるフレームレートで 2 つのゲームをプレイし、調査を行いました。左のバーは、これらのゲームの基準となるデフォルトのフレームレート (無制限) を示します。真ん中のバーは 60FPS でゲームをプレイした場合、右のバーは 30FPS でゲームをプレイした場合です。この調査から、フレームレートを制限すると、消費電力を削減できることが分かります。

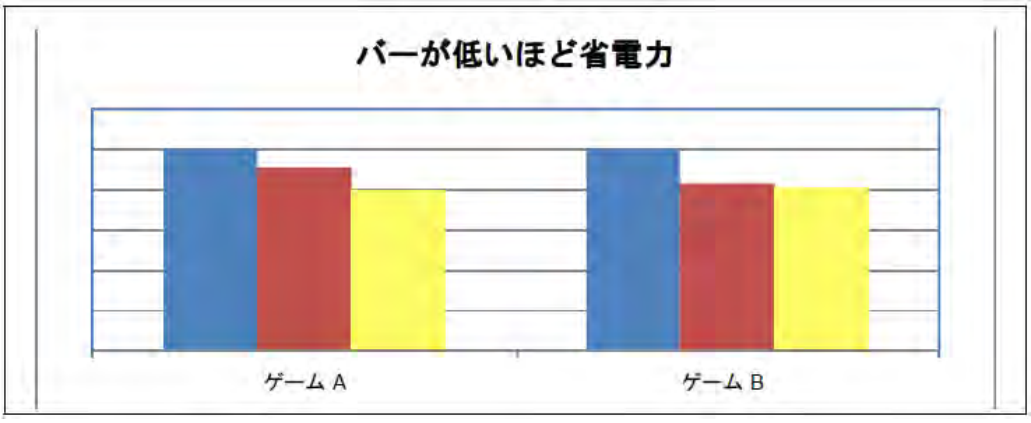


図 17-10 電源に対応したフレームレート設定による消費電力の削減の比較

17.5.8 パフォーマンスの最適化による消費電力の削減

一般的には、パフォーマンスの最適化によって、CPU がコードを実行しているサイクル数を減らすことにより消費電力も削減できます。消費電力の削減をもたらすマイクロアーキテクチャー機能について、それぞれパフォーマンスの最適化例を示します。

3.6.1 節に示すように、Sandy Bridge† マイクロアーキテクチャーで追加されたロードポート機能により、サイクル数を削減できます。これにより、ロードポート機能は消費電力の軽減に貢献できます。例えば、3.6.1.3 節のカーネルの実証では、エンジニアリング・システムを使用して、バンクの競合がある状態とない状態のサンプル・アプリケーションを実行しました。バンクの競合がないと、第 2 のロードポートを使用できるため、パフォーマンスが向上するとともに、1 時間あたり 25mW の電力が削減されました。

もう 1 つの機能であるデコード済み命令キャッシュと LSD によって、マイクロオペレーション (uop) がキャッシュに格納され、デコーダーによる消費電力が削減されます。例えば、switch 文に対してアライメントが合った 32 バイト・チャンク (3.4.2.5 節を参照) 内で 3 つ以下の無条件分岐を並べ替えるコード・アライメント手法を使用した場合、処理速度が 1.23 倍向上し、同じアライメントが合った 32 バイト・チャンク内であっても無条件分岐の密度が高く、デコード済み命令キャッシュに収まらない場合と比べ、消費電力が 1.12 倍削減されされます。

インテル® AVX によるベクトル化により、複数の要素を 1 つのサイクル内で処理することができるため、全体的な処理サイクルが減るとともに、消費電力が削減されます。図 17-8 に消費電力の削減例を示します。

17.6 システム・ソフトウェアのプロセッサー固有のパワー・マネジメントの最適化

この節では、プロセッサー固有のパワー・マネジメント情報を取り上げます。具体的には、Sandy Bridge† マイクロアーキテクチャー・ベースの第 2 世代インテル® Core™ プロセッサーに適用されます。これらのプロセッサーの CPUID DisplayFamily_DisplayModel シグネチャーは 06_2AH です。このようなプロセッサー固有の能力により、システム・ソフトウェアはプロセッサーの応答性と消費電力を最適化し、バランスをとることができます。

17.6.1 プロセッサー固有の非アクティブ状態構成のパワー・マネジメントの推奨事項

各種の非アクティブ状態において、適切な終了レイテンシー値で ACPI の _CST オブジェクトをプログラミングすることで OS パワー・マネジメントによる最適な消費電力削減を実現できます。インテルの推奨値はモデル固有です。

表 17-3 および表 17-4 に、CPUID DisplayFamily_DisplayModel シグネチャーが 06_2AH のプロセッサー、および電圧レギュレーターのスルーレート機能の 2 つの構成について、パッケージ C ステートの開始/終了レイテンシーを示します。表 17-3 は低速 VR 構成で、表 17-4 は高速 VR 構成です。各構成について、VR デバイスは MSR_POWER_CTL.[bit 4] の設定に応じ、高速割り込み中断モード、または低速割り込み中断モードいずれかで動

作できます。これらの C ステート開始/終了レイテンシーはプロセッサーの仕様ではなく、経験則から導き出した予測です。状況によっては、コアからの終了レイテンシーが表 17-3 および表 17-4 に示されている値よりも高くなることもあります。

表 17-3 低速 VR の場合のクライアント・システムの C ステートのプロセッサーの合計の終了レイテンシー (コア + パッケージの終了レイテンシー)

C ステート [1]	典型的な終了レイテンシー[2]	最悪なケースの終了レイテンシー
	MSR_POWER_CTL MSR.[4] =0	MSR_POWER_CTL MSR.[4] =1
C1	1 µS	1 µS
C3	156 µS	80 µS
C6	181 µS	104 µS
C7	199 µS	109 µS

注意:

1. この C ステートの開始/終了レイテンシーは、インテルの予測に過ぎず、プロセッサーの仕様ではありません。
2. コアの 1 つがアクティブになったときにパッケージが C0 であるものと仮定します。
3. MSR_POWER_CTL.[4] = 1 の場合に、高速割り込み中断モードが有効になります。
4. PCH と接続しているデバイスのレイテンシーは、低速割り込み中断モードのレイテンシーに相当します。

表 17-4 高速 VR の場合のクライアント・システムの C ステートのプロセッサーの合計の終了レイテンシー (コア + パッケージの終了レイテンシー)

C ステート [1]	典型的な最悪なケースの終了レイテンシー時間 (すべての製品)[2]	
	MSR_POWER_CTL MSR.[4] =0	MSR_POWER_CTL MSR.[4] =1
C1	1 µS	1 µS
C3	156 µS	80 µS
C6	181 µS	104 µS
C7	199 µS	109 µS

注意:

1. この C ステートの開始/終了レイテンシーは、インテルの予測に過ぎず、プロセッサーの仕様ではありません。
2. コアの 1 つがアクティブになったときにパッケージが C0 であるものと仮定します。
3. パッケージがディーパー C ステートの場合、ローカル APIC タイマーのウェイクアップの終了レイテンシーは、典型的なコアレベルの終了レイテンシーに依存します。パッケージが C0 の場合は、それぞれのコアレベルの終了レイテンシーの代表値と最悪のケース値の間で変動する可能性があります。

表 17-5 に、CPUID DisplayFamily_DisplayModel シグネチャーが 06_2AH のプロセッサー、および電圧レギュレーターのスルーレート機能の 2 つの構成について、コアのみの C ステートの開始/終了レイテンシーを示します。コアのみの終了レイテンシーは、MSR_POWER_CTL.[4] の影響を受けることはありません。

表 17-5 低速 VR の場合のクライアント・システムの C ステートのコアのみの終了レイテンシー

C ステート [1]	典型的な最悪なケースの終了レイテンシー (すべての製品)[2]	
C1	1 µS	1 µS
C3	21 µS	240 µS
C6	46 µS	250 µS
C7	46 µS	250 µS

注意:

1. この C ステートの開始/終了レイテンシーは、インテルの予測に過ぎず、プロセッサーの仕様ではありません。

2. 低速 VR デバイスとは、ランプ時間が高速モードでは 10mv/µs、低速モードでは 2.5mv/µs のデバイスを指します。

17.6.1.1 非アクティブ状態からアクティブ状態へ遷移する場合のパワー・マネジメントと応答性のバランス

MSR_PKGC3_IRTL、MSR_PKGC6_IRTL、および MSR_PKGC7_IRTL は、システム・ソフトウェアが消費電力とシステムの応答性のバランスをとるためのプロセッサー固有のインターフェイスです。システム・ソフトウェアは、実行時にシステム固有の要件に合わせて、設定値をパッケージ非アクティブ状態から C0 に変更することがあります。例えば、バッテリー動作時と AC 電源動作時を対比して、より積極的なタイミングを設定する場合でです。

終了レイテンシーは VR スイングレートの影響を大きく受けます。表 17-4 に、「高速」終了レート (内部の HPET タイマーと CPU タイマー以外のすべてのイベントに対する PCH および CPU のデフォルトの推奨構成) の状態あたりの合計割り込み応答時間 (コアの要素を含む) を示します。

PCIe* からの各種イベントに対し BIOS で「低速」レートを選択する (POWER_CTL MSR ビット 4 で高速中断イベントメソッドを無効にする) には、それぞれ上記の設定値を拡張する必要があります。さもないと、はるかに低速な VR スイングレートで要求を満たすため、CPU がより浅い PKG_C ステートを選択する可能性があります。

各種の PCH 接続デバイスに対して「低速」終了レートを選択すると (PCH BIOS 設定)、上記のレイテンシー計算メカニズムでは認識されないため、結果として CPU は必要なレイテンシー目標を満たせません。

表 17-6 インテル® プロセッサー (Sandy Bridge† マイクロアーキテクチャ世代) の POWER_CTL MSR

レジスターアドレス		レジスター名	スコープ	ビットの説明
16 進表記	**10 進表記**			
1FCH	508	MSR_POWER_CTL	コア	電力制御レジスター
		3:0		予約済み。
		4		FAST_Brk_Int_En. 1 に設定した場合、PCIe* 割り込みウェイクアップ・イベントの高速スロープに対して電圧レギュレーターが有効になります。
		63:5		予約済み。

第 18 章
インテル® AVX-512 命令向けのソフトウェア最適化

Skylake† Server マイクロアーキテクチャーと Knights Landing† マイクロアーキテクチャー・ベースのインテル® Xeon Phi™ プロセッサーからサポートされるインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) は、次のような 512 ビット命令セットの拡張を含んでいます。

- インテル® AVX-512 基本 (F)
  - 512 ビット・ベクトル幅
  - 32 個の 512 ビット長ベクトルレジスター
  - データのエクスパンドとコンプレス命令
  - 三項論理命令
  - 8 個の 64 ビット長マスクレジスター
  - 2 つのソースのレーン間のパーミュート命令
  - スキャッター命令
  - 組込みブロードキャスト/丸め
  - 超越関数のサポート

- インテル® AVX-512 競合検出命令 (CD)
- インテル® AVX-512 指数および逆数命令 (ER)
- インテル® AVX-512 プリフェッチ命令 (PF)
- インテル® AVX-512 バイトおよびワード命令 (BW)
- インテル® AVX-512 ダブルワードおよびクワッドワード命令 (DQ)
  - 新しい QWORD の計算と変換命令

- インテル® AVX-512 ベクトル長の拡張 (VL)

図 18-1 は、2 つのプロセッサー・ファミリーの異なる拡張を示しています。この章のパフォーマンス・レポートは、データ・キャッシュ・ユニット (DCU) に存在するデータの測定を基にしています。Skylake† Server サーバーシステムでは、インテル® ターボ・ブースト・テクノロジーと Intel SpeedStep® テクノロジーは無効化され、コアとアンコアの周波数は 1.80GHz に設定されています。この固定周波数の構成は、他の要因によるコード変更に影響を分かりやすくするために導入されています。インテル® AVX-512 を使用した際の電力と周波数への影響については、2.4.3 節「Skylake† Server の電力管理」を参照してください。

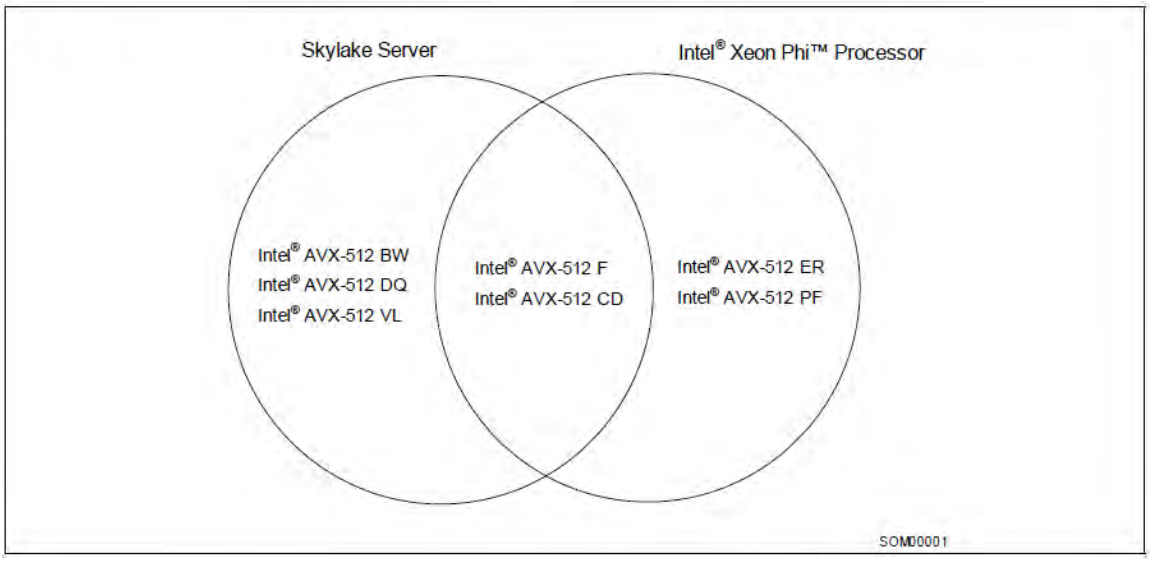


図 18-1 Skylake† Server と Knights Landing† マイクロアーキテクチャーでサポートされるインテル® AVX-512 拡張

18.1 インテル® AVX-512 とインテル® AVX2 コーディングの基礎

ほとんどの場合、インテル® AVX-512 のパフォーマンス向上は 512 ビット幅のレジスターによってもたらされます。この節では、インテル® AVX2 とインテル® AVX-512 コード間の相違点を示し、インテル® AVX2 からインテル® AVX-512 へ簡単に移行する方法を説明します。最初の項では組込み関数、次の項ではアセンブリー・コードの変換について説明します。以下の節では、コード変換を行う際に考慮して扱うべき高度な視点をハイライトします。

次の項の例は、デカルト座標系回転を実装しています。デカルト座標系回転のポイントは、(x, y) のペアによって表されます。次の図は、角度 θ による (x', y') のデカルト回転 (x, y) を示しています。

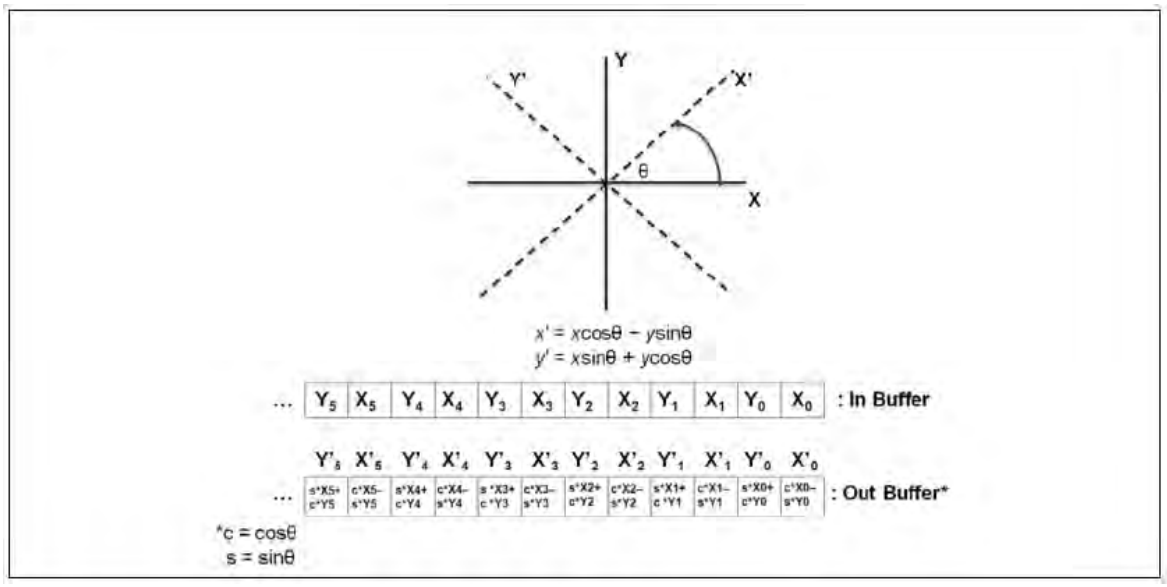


図 18-2 デカルト回転

18.1.1 組込み関数によるコーディング

次のインテル® AVX2 とインテル® AVX-512 の比較では、インテル® AVX2 の組込み関数で記述されたコードシーケンスをインテル® AVX-512 へ変換する方法を示しています。この例は、インテル® AVX 命令形式、64 バイト ZMM レジスター、64 バイト・データ境界での静的および動的メモリーの割り当て、および ZMM レジスター内の 16 個の浮動小数点を表現する C のデータ型を使用しています。この変換を行うガイドラインを以下に示します。

- 静的および動的に割り当てられているバッファーを 64 バイトにアライメントします。
- 定数向けの補助バッファーサイズを倍にします。
- "__mm256_組込み関数名" のプリフィクスを "__mm512_" に変更します。
- 変数のデータ型名を __m256 から __m512 へ変更します。
- ループの反復回数を半分にします (倍のストライド長に)。

例 18-1 組込み関数を使用したデカルト座標変換

インテル® AVX2 組込み関数のコード	インテル® AVX-512 組込み関数のコード

#include <immintrin.h>
int main()
{
int len = 3200;
// 32 バイト境界で動的にメモリーを割り当て
//
float* pInVector = (float *)
_mm_malloc(len*sizeof(float),32);
float* pOutVector = (float *)
_mm_malloc(len*sizeof(float),32);
 	
// データの初期化
for (int i=0; i<len; i++)
pInVector[i] = 1;
 	
float cos_teta = 0.8660254037;
float sin_teta = 0.5;
 	
// 8 つの float を 32 バイトのアライメントで静的にメモリー割り当て
__declspec(align(32)) float
cos_sin_teta_vec[8] =
{cos_teta, sin_teta, cos_teta, sin_teta,
cos_teta, sin_teta,
cos_teta, sin_teta};
 	
__declspec(align(32)) float
sin_cos_teta_vec[8] =
{sin_teta, cos_teta, sin_teta, cos_teta,
sin_teta, cos_teta,
sin_teta, cos_teta};
 	
// __m256 データ型は、Ymm レジスターの
// 8 つの float 要素を表します
__m256 Ymm_cos_sin =
_mm256_load_ps(cos_sin_teta_vec);
 	
// インテル® AVX2 の 256 ビットの
// パックド単精度ロード
__m256 Ymm_sin_cos =
_mm256_load_ps(sin_cos_teta_vec);
__m256 Ymm0, Ymm1, Ymm2, Ymm3;
	#include <immintrin.h>
	int main()
	{
	int len = 3200;
	// 64 バイト境界で動的にメモリーを割り当て
	//
	float* pInVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	float* pOutVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	
	// データの初期化
	for (int i=0; i<len; i++)
	pInVector[i] = 1;
	
	float cos_teta = 0.8660254037;
	float sin_teta = 0.5;
	
	// 16 個の float を 64 バイトのアライメントで静的にメモリー割り当て
	__declspec(align(64)) float
	cos_sin_teta_vec[16] =
	{cos_teta, sin_teta, cos_teta, sin_teta,
	cos_teta, sin_teta,
	cos_teta, sin_teta, cos_teta, sin_teta,
	cos_teta, sin_teta,
	cos_teta, sin_teta, cos_teta, sin_teta};
	
	__declspec(align(64)) float
	sin_cos_teta_vec[16] =
	{sin_teta, cos_teta, sin_teta, cos_teta,
	sin_teta, cos_teta,
	sin_teta, cos_teta, sin_teta, cos_teta,
	sin_teta, cos_teta,
	sin_teta, cos_teta, sin_teta, cos_teta};
	
	// __m512 データ型は、Zmm レジスターの
	// 16 個の float 要素を表します
	__m512 Zmm_cos_sin =
	_mm512_load_ps(cos_sin_teta_vec);
	
	// インテル® AVX-512 の 512 ビットの
	// パックド単精度ロード
	__m512 Zmm_sin_cos =
	_mm512_load_ps(sin_cos_teta_vec);
	__m512 Zmm0, Zmm1, Zmm2, Zmm3;

// 2 回分アンロールされたループで 16 個の要素を処理
//
for(int i=0; i<len; i+=16)
{
Ymm0 = _mm256_load_ps(pInVector+i);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector + i,Ymm3);
 	
Ymm0 = _mm256_load_ps(pInVector+i+8);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2, Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector+i+8,Ymm3);
}
_mm_free(pInVector);
_mm_free(pOutVector);
return 0;
}
	// 2 回分アンロールされたループで 32 個の要素を処理
	//
	for(int i=0; i<len; i+=32)
	{
	Zmm0 = _mm512_load_ps(pInVector+i);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2,Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector + i,Zmm3);
	
	Zmm0 = _mm512_load_ps(pInVector+i+16);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2, Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector+i+16,Zmm3);
	}
	_mm_free(pInVector);
	_mm_free(pOutVector);
	return 0;
	}

ベースライン	**スピードアップ: 1.95x**

18.1.2 アセンブリーによるコーディング

組込み関数の移植のガイドラインと同様に、アセンブリーで記述されたコードを移植するガイドラインを以下に示します。

- 静的および動的に割り当てられているバッファーを 64 バイトにアライメントします。
- 必要であれば補足のバッファーサイズを 2 倍にします。
- 命令ニーモニックの前に “v” プリフィクスを追加します。
- レジスター名を ymm から zmm へ変更します。
- ループの反復回数を半分にします (またはストライド長を倍にします)。

例 18-2 アセンブリーを使用したデカルト座標系の回転

インテル® AVX2 アセンブリー・コード	インテル® AVX-512 アセンブリー・コード

#include <immintrin.h>
int main()
{
int len = 3200;
// 32 バイト境界で動的にメモリーを割り当て
float* pInVector = (float *)
_mm_malloc(len*sizeof(float),32);
float* pOutVector = (float *)
_mm_malloc(len*sizeof(float),32);
 	
// データの初期化
for (int i=0; i<len; i++)
pInVector[i] = 1;
 	
float cos_teta = 0.8660254037;
float sin_teta = 0.5;
 	
// 8 つの float を 32 バイトのアライメントで静的にメモリー割り当て
__declspec(align(32)) float
cos_sin_teta_vec[8] =
{cos_teta, sin_teta, cos_teta, sin_teta,
cos_teta, sin_teta, cos_teta, sin_teta};
 	
__declspec(align(32)) float
sin_cos_teta_vec[8] =
{sin_teta, cos_teta, sin_teta, cos_teta,
sin_teta, cos_teta,
sin_teta, cos_teta};
 	
__asm
{
mov rax,pInVector
mov r8,pOutVector
// ymm レジスターに 32 バイトをロード
vmovups ymm3,ymmword ptr[cos_sin_teta_vec]
vmovups ymm4,ymmword ptr[sin_cos_teta_vec]
 	
mov edx, len
shl edx, 2
xor ecx, ecx
loop1:
vmovsldup ymm0, [rax+rcx]
vmovshdup ymm1, [rax+rcx]
vmulps ymm1, ymm1, ymm4
vfmaddsub213ps ymm0, ymm3, ymm1
	#include <immintrin.h>
	int main()
	{
	int len = 3200;
	// 64 バイト境界で動的にメモリーを割り当て
	float* pInVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	float* pOutVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	
	// データの初期化
	for (int i=0; i<len; i++)
	pInVector[i] = 1;
	
	float cos_teta = 0.8660254037;
	float sin_teta = 0.5;
	
	// 16 個の float を 64 バイトのアライメントで静的にメモリー割り当て
	__declspec(align(64)) float
	cos_sin_teta_vec[16] =
	{cos_teta, sin_teta, cos_teta, sin_teta,
	cos_teta, sin_teta, cos_teta, sin_teta,
	cos_teta, sin_teta, cos_teta, sin_teta,
	cos_teta, sin_teta, cos_teta, sin_teta};
	
	__declspec(align(64)) float
	sin_cos_teta_vec[16] =
	{sin_teta, cos_teta, sin_teta, cos_teta,
	sin_teta, cos_teta,
	sin_teta, cos_teta, sin_teta, cos_teta,
	sin_teta, cos_teta,
	sin_teta, cos_teta, sin_teta, cos_teta};
	__asm
	{
	mov rax,pInVector
	mov r8,pOutVector
	// zmm レジスターに 64 バイトをロード
	vmovups zmm3, zmmword ptr[cos_sin_teta_vec]
	vmovups zmm4, zmmword ptr[sin_cos_teta_vec]
	
	mov edx, len
	shl edx, 2
	xor ecx, ecx
	loop1:
	vmovsldup zmm0, [rax+rcx]
	vmovshdup zmm1, [rax+rcx]
	vmulps zmm1, zmm1, zmm4
	vfmaddsub213ps zmm0, zmm3, zmm1

// ymm レジスターから 32 バイトのストア
vmovaps [r8+rcx], ymm0
vmovsldup ymm0, [rax+rcx+32]
vmovshdup ymm1, [rax+rcx+32]
vmulps ymm1, ymm1, ymm4
vfmaddsub213ps ymm0, ymm3, ymm1
// 前のストアからの 32 バイトのオフセット
vmovaps [r8+rcx+32], ymm0
 	
// このループで処理された 64 バイト
// (コードは 2 回アンロール)
add ecx, 64
cmp ecx, edx
jl loop1
}
 	
_mm_free(pInVector);
_mm_free(pOutVector);
return 0;
}
	// zmm レジスターから 64 バイトのストア
	vmovaps [r8+rcx], zmm0
	vmovsldup zmm0, [rax+rcx+64]
	vmovshdup zmm1, [rax+rcx+64]
	vmulps zmm1, zmm1, zmm4
	vfmaddsub213ps zmm0, zmm3, zmm1
	// 前のストアからの 64 バイトのオフセット
	vmovaps [r8+rcx+64], zmm0
	
	// このループで処理された 128 バイト
	// (コードは 2 回アンロール)
	add ecx, 128
	cmp ecx, edx
	jl loop1
	}
	
	_mm_free(pInVector);
	_mm_free(pOutVector);
	return 0;
	}

ベースライン	**スピードアップ: 1.95x**

18.2 マスク処理

拡張 VEX (EVEX) コード構成を使用するインテル® AVX-512 命令は、要素ごと計算操作を条件付きで制御し、結果をデスティネーション・オペランドに更新するためプレディケート・オペランドをエンコードします。プレディケート・オペランドは opmask (オペレーションマスク) レジスターとも呼ばれます。opmask はそれぞれ 64 ビット長の 8 つのアーキテクチャー・レジスターです。この 8 つのアーキテクチャー・レジスターは、プレディケート・オペランドとして k1 から k7 を介してアクセスできます。k0 は通常のソースまたはデスティネーションとして使用できますが、プレディケート・オペランドとしてはエンコードできません。

プレディケート・オペランドは、メモリー・ソース・オペランドを持つ命令でメモリーフォルトを抑制するために使用できます。

プレディケート・オペランドとして opmask レジスターは、ベクトルレジスターの各要素を操作または更新するのを管理する 1 ビットを含んでいます。Skylake† マイクロアーキテクチャーのマスク命令は、すべてのデータサイズをサポートします: バイト (int8)、ワード (int16)、単精度浮動小数点 (float32)、整数ダブルワード (int32)、倍精度浮動小数点 (float64)、整数クワッドワード (int64)。ベクトルレジスターはデータサイズに応じて 8、16、32 または 64 個の要素を格納しています。そのため、ベクトル・マスク・レジスターは 64 ビット長となっています。

Skylake† マイクロアーキテクチャーでは、128 ビット、256 ビット、および 512 ビット長ベクトルのマスクが可能です。データ型とベクトル長に応じて、各命令は最下位マスクビットを要素の数だけアクセスします。例えば、インテル® AVX-512 命令が 512 ビット・ベクトルの 64 ビット・データ要素を操作する場合、opmask レジスターの下位 8 ビット (512/64) のみを使用します。opmask レジスターは、要素単位でインテル® AVX-512 命令に影響します。そのため、各データ要素のすべての数値または非数値操作、およびデスティネーション・オペランドへの要素ごとの中間結果の更新は、opmask レジスターの対応するビットで決定されます。

インテル® AVX-512 のプレディケート・オペランドとして機能する opmask は次のような特性を持っています。

- 命令操作は、対応する opmask ビットが設定されている要素に対してのみ行われます。これは、例外や違反もマスクがセットされていない要素の操作では引き起こされないことを意味します。その結果、マスクが設定されていない操作の結果として MXCSR 例外フラグが更新されることもありません。

- 対応する書込みマスクビットが設定されていない場合、デスティネーション要素には操作の結果は反映されません。その場合、デスティネーション要素の値は、保護されるか (マージマスク)、ゼロに設定されます (ゼロ化マスク)。
- メモリー操作を伴う命令のいくつかでは、マスクビット 0 の要素に対するメモリーフォルトが抑制されます。

マスクは、インテル® AVX-512 ベクトルレジスターのデスティネーションに対するマージ動作を提供するため、制御フローのプレディケーションを実装する強力な構造をもたらします。代替手段として、マージの代わりにゼロ化マスクを利用できます。これは、以前の値を保持する代わりにゼロで要素を更新します。古い値を必要としない場合、ゼロ化により暗黙的に依存関係が排除されます。

マスクを使用するほとんどの命令では、両方のマスク形式を利用できます。0 ではない (gather と scatter) EVEX.aaa ビットを持つ命令と、メモリーへの書き込みを行う命令は、マージマスクのみを受け入れます。

また、デスティネーション・オペランドがメモリー・ロケーションである場合、要素ごとにデスティネーション更新規則が適用されます。ベクトルは、プレディケート・オペランドとして使用される opmask レジスターに基づいて要素単位で書き込まれます。

opmask レジスターの値は次のように設定されます。

- ベクトル命令 (CMP、FPCLASS など) の結果として生成される。
- メモリーからロードされる。
- GPR レジスターからロードされる。
- マスクからマスクの操作で変更される。

18.2.1 マスクの例

マスク付き命令は、各データ要素に関連付けられたマスクビットに応じて、条件付きパックドデータ要素の操作を行います。各データ要素のマスクビットは、マスクレジスター内の対応するビットです。

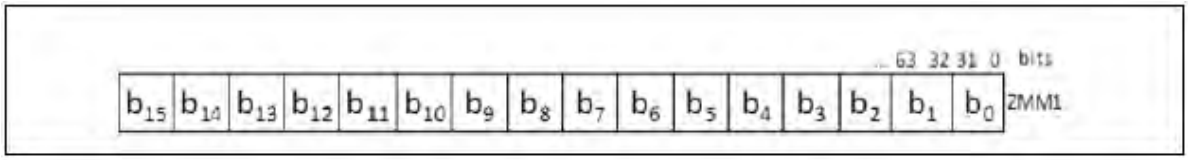
マスク付き命令を実行する場合、マスク値 0 に対応する要素には 0 が返されます。デスティネーション・レジスターの対応する値は、ゼロ化フラグに依存します。

- フラグが設定されていれば、メモリー・ロケーションにはゼロが書き込まれます。
- 設定されていない場合、メモリー・ロケーションの値は保護されます。

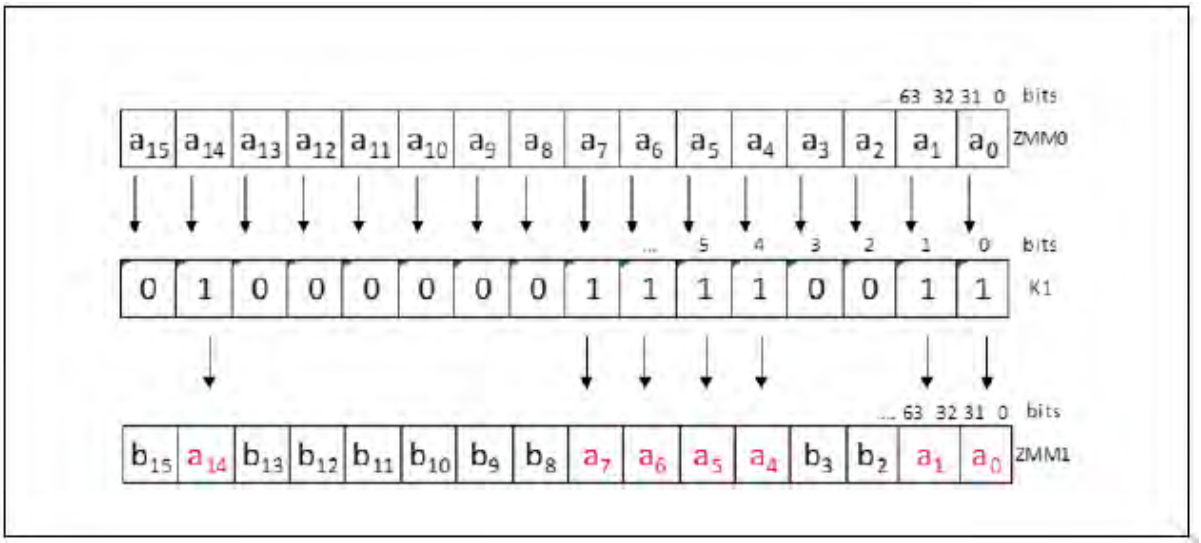
次の図では、マージマスクを使用したレジスターから別のレジスターへのマスク移動の例を示しています。

```
vmovaps zmm1 {k1}, zmm0
```

命令が実行される前のデスティネーション・レジスターの内容は、次のようになっています。

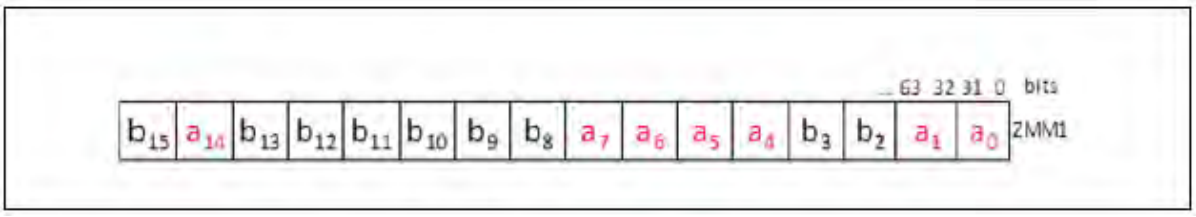


次のように操作されます。



ゼロ化マスクの実行結果には、次の命令を使用します (命令中の {z} に注目)。

```
vmovaps zmm1 {k1}{z}, zmm0
```



マージマスク操作はデスティネーションとの依存関係がありますが、ゼロ化マスクにはそのような依存関係がないことに注意してください。

次の例では、インテル® AVX2 に対してインテル® AVX-512 でどのようにマスク操作が行われるかを示しています。

C のコードに示します。

```
const int N = miBufferWidth;
const double* restrict a = A;
const double* restrict b = B;
double* restrict c = Cref;

for (int i = 0; i < N; i++){
  double res = b[i];
  if(a[i] > 1.0){
    res = res * a[i];
  }
  c[i] = res;
}
```

例 18-3 組込み関数とマスク

インテル® AVX2 組込み関数のコード	インテル® AVX-512 組込み関数のコード

for (int i = 0; i < N; i+=32){
__m256d aa, bb, mask;
#pragma unroll(8)
for (int j = 0; j < 8; j++){
aa = _mm256_loadu_pd(a+i+j*4);
bb = _mm256_loadu_pd(b+i+j*4);
mask =
_mm256_cmp_pd(_mm256_set1_pd(1.0), aa,
1);
aa = _mm256_and_pd(aa, mask); // 偽の値を
ゼロにする
false values
aa = _mm256_mul_pd(aa, bb);
bb = _mm256_blendv_pd(bb, aa, mask);
_mm256_storeu_pd(c+4*j, bb);
}
c += 32;
}
	for (int i = 0; i < N; i+=32){
	__m512d aa, bb;
	__mmask8 mask;
	#pragma unroll(4)
	for (int j = 0; j < 4; j++){
	aa = _mm512_loadu_pd(a+i+j*8);
	bb = _mm512_loadu_pd(b+i+j*8);
	mask =
	_mm512_cmp_pd_mask(_mm512_set1_pd(1.0),
	aa, 1);
	bb = _mm512_mask_mul_pd(bb, mask, aa,
	bb);
	_mm512_storeu_pd(c+8*j, bb);
	}
	c += 32;
	}

ベースライン	**スピードアップ: 2.9x**

例 18-4 アセンブリーとマスク

インテル® AVX2 アセンブリー・コード	インテル® AVX-512 アセンブリー・コード

mov rax, a
mov r11, b
mov r8, N
shr r8, 5
mov rsi, c
xor rcx, rcx
xor r9, r9
 	
loop:
vmovupd ymm1, ymmword ptr [rax+rcx*8]
inc r9d
vmovupd ymm6, ymmword ptr [rax+rcx*8+0x20]
vmovupd ymm2, ymmword ptr [r11+rcx*8]
vmovupd ymm7, ymmword ptr [r11+rcx*8+0x20]
vmovupd ymm11, ymmword ptr [rax+rcx*8+0x40]
vmovupd ymm12, ymmword ptr [r11+rcx*8+0x40]
vcmppd ymm4, ymm0, ymm1, 0x1
vcmppd ymm9, ymm0, ymm6, 0x1
vcmppd ymm14, ymm0, ymm11, 0x1
vandpd ymm16, ymm1, ymm4
vandpd ymm17, ymm6, ymm9
vmulpd ymm3, ymm16, ymm2
vmulpd ymm8, ymm17, ymm7
vmovupd ymm1, ymmword ptr [rax+rcx*8+0x60]
vmovupd ymm6, ymmword ptr [rax+rcx*8+0x80]
vblendvpd ymm5, ymm2, ymm3, ymm4
	mov rax, a
	mov r11, b
	mov r8, N
	shr r8, 5
	mov rsi, c
	xor rcx, rcx
	xor r9, r9
	mov rdi, 1
	cvtsi2sd xmm8, rdi
	vbroadcastsd zmm8, xmm8
	loop:
	vmovups zmm0, zmmword ptr [rax+rcx*8]
	inc r9d
	vmovups zmm2, zmmword ptr [rax+rcx*8+0x40]
	vmovups zmm4, zmmword ptr [rax+rcx*8+0x80]
	vmovups zmm6, zmmword ptr [rax+rcx*8+0xc0]
	vmovups zmm1, zmmword ptr [r11+rcx*8]
	vmovups zmm3, zmmword ptr [r11+rcx*8+0x40]
	vmovups zmm5, zmmword ptr [r11+rcx*8+0x80]
	vmovups zmm7, zmmword ptr [r11+rcx*8+0xc0]
	vcmppd k1, zmm8, zmm0, 0x1
	vcmppd k2, zmm8, zmm2, 0x1
	vcmppd k3, zmm8, zmm4, 0x1
	vcmppd k4, zmm8, zmm6, 0x1
	vmulpd zmm1{k1}, zmm0, zmm1
	vmulpd zmm3{k2}, zmm2, zmm3
	vmulpd zmm5{k3}, zmm4, zmm5
	vmulpd zmm7{k4}, zmm6, zmm7

vblendvpd ymm10, ymm7, ymm8, ymm9
vmovupd ymm2, ymmword ptr [r11+rcx*8+0x60]
vmovupd ymm7, ymmword ptr [r11+rcx*8+0x80]
vmovupd ymmword ptr [rsi], ymm5
vmovupd ymmword ptr [rsi+0x20], ymm10
vcmppd ymm4, ymm0, ymm1, 0x1
vcmppd ymm9, ymm0, ymm6, 0x1
vandpd ymm18, ymm11, ymm14
vandpd ymm19, ymm1, ymm4
vandpd ymm20, ymm6, ymm9
vmulpd ymm13, ymm18, ymm12
vmulpd ymm3, ymm19, ymm2
vmulpd ymm8, ymm20, ymm7
vmovupd ymm11, ymmword ptr [rax+rcx*8+0xa0]
vmovupd ymm1, ymmword ptr [rax+rcx*8+0xc0]
vmovupd ymm6, ymmword ptr [rax+rcx*8+0xe0]
vblendvpd ymm15, ymm12, ymm13, ymm14
vblendvpd ymm5, ymm2, ymm3, ymm4
vblendvpd ymm10, ymm7, ymm8, ymm9
vmovupd ymm12, ymmword ptr [r11+rcx*8+0xa0]
vmovupd ymm2, ymmword ptr [r11+rcx*8+0xc0]
vmovupd ymm7, ymmword ptr [r11+rcx*8+0xe0]
vmovupd ymmword ptr [rsi+0x40], ymm15
vmovupd ymmword ptr [rsi+0x60], ymm5
vmovupd ymmword ptr [rsi+0x80], ymm10
vcmppd ymm14, ymm0, ymm11, 0x1
vcmppd ymm4, ymm0, ymm1, 0x1
vcmppd ymm9, ymm0, ymm6, 0x1
vandpd ymm21, ymm11, ymm14
add rcx, 0x20
vandpd ymm22, ymm1, ymm4
vandpd ymm23, ymm6, ymm9
vmulpd ymm13, ymm21, ymm12
vmulpd ymm3, ymm22, ymm2
vmulpd ymm8, ymm23, ymm7
vblendvpd ymm15, ymm12, ymm13, ymm14
vblendvpd ymm5, ymm2, ymm3, ymm4
vblendvpd ymm10, ymm7, ymm8, ymm9
vmovupd ymmword ptr [rsi+0xa0], ymm15
vmovupd ymmword ptr [rsi+0xc0], ymm5
vmovupd ymmword ptr [rsi+0xe0], ymm10
add rsi, 0x100
cmp r9d, r8d
jb loop
	vmovups zmmword ptr [rsi], zmm1
	vmovups zmmword ptr [rsi+0x40], zmm3
	vmovups zmmword ptr [rsi+0x80], zmm5
	vmovups zmmword ptr [rsi+0xc0], zmm7
	add rcx, 0x20
	add rsi, 0x100
	cmp r9d, r8d
	jb loop

ベースライン	**スピードアップ: 2.9x**

18.2.2 マスクのコスト

マスクを使用すると、マスクなしのコードに比べパフォーマンスが低下します。これは、次のいずれかが原因で発生します。

- それぞれのロードで追加のブレンド操作が行われるため。
- マージマスクによりデスティネーションで依存関係が存在するため。この依存関係はセロ化マスクでは発生しません。
- より制限されたマスクのフォワード規則 (詳細については、フォワードとメモリーマスクを参照してください)。

次の例では、マージマスクを使用する際にどのようにデスティネーション・レジスターと依存関係が生じるか示しています。

例 18-5 マスク処理の例

マスクなし	マージマスク	ゼロマスク
mov rbx, iter loop: vmulps zmm0, zmm9, zmm8 vmulps zmm1, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}, zmm9, zmm8 vmulps zmm1{k1}, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}{z}, zmm9, zmm8 vmulps zmm1{k1}{z}, zmm9, zmm8 dec rbx jnle loop
ベースライン	**スローダウン: 4x**	**スローダウン: ベースラインと同じ**

マスクなしでは、プロセッサーは 2 つの FMA ポートでサイクルごとに 2 つの乗算を実行できます。

マスクを伴うと、反復 N の乗算が反復 N - 1 の乗算の出力と依存性があるため、プロセッサーは 4 サイクルごとに 2 つの除算を実行します。

ゼロ化マスクではデスティネーション・レジスターと依存関係を持たないため、2 つの FMA ポートでサイクルごとに 2 つの乗算を実行できます。

推奨事項: マスクにはコストが伴うため必要な場合にのみ使用します。可能であれば、マージマスクではなくゼロ化マスクを使用します。

18.2.3 マスクとブレンド

この節では、条件付きコードにおいてブレンドとマスクを使用する利点と欠点について説明します。

次のコードについて考えてみます。

```
for ( i=0; i<SIZE; i++ )
{
  if ( a[i] > 0 )
  {
    b[i] *= 2;
  }
  else
  {
    b[i] /= 2;
  }
}
```

次の例では、2 つのコンパイル可能なコードを示しています。

- 代替 1 は、マスクされたコードと簡単なデータの数値処理を使用します。
- 代替 2 では、コードを連続して処理される 2 つの独立したマスクなしのフローに分割し、次にメモリーへストアする前にマスクされた移動 (ブレンド) を行います。

例 18-6 マスクとブレンドの例 1

代替 1	**代替 2**

`mov rax, pImage`
`mov rbx, pImage1`
`mov rcx, pOutImage`
`mov rdx, len`
`vpxord zmm0, zmm0, zmm0`
`mainloop:`
`vmovdqa32 zmm2, [rax+rdx*4-0x40]`
`vmovdqa32 zmm1, [rbx+rdx*4-0x40]`
`vpcmpgtd k1, zmm1, zmm0`
`knotw k2, k1`
`(1) vpslld zmm2 {k1}, zmm2, 1`
`(2) vpsrld zmm2 {k2}, zmm2, 1`
`(3) vmovdqa32 [rcx+rdx*4-0x40], zmm2`
`sub rdx, 16`
`jne mainloop`
	`mov rax, pImage`
	`mov rbx, pImage1`
	`mov rcx, pOutImage`
	`mov rdx, len`
	`vpxord zmm0, zmm0, zmm0`
	`mainloop:`
	`vmovdqa32 zmm2, [rax+rdx*4-0x40]`
	`vmovdqa32 zmm1, [rbx+rdx*4-0x40]`
	`vpcmpgtd k1, zmm1, zmm0`
	`vmovdqa32 zmm3, zmm2`
	`vpslld zmm2, zmm2, 1`
	`vpsrld zmm3, zmm3, 1`
	`(1) vmovdqa32 zmm3 {k1}, zmm2`
	`(2) vmovdqa32 [rcx+rdx*4-0x40], zmm3`
	`sub rdx, 16`
	`jne mainloop`

ベースライン·サイクル 1x **ベースライン命令 1x**	**スピードアップ: 1.23x** **命令: 1.11x**

代替 1 には、命令 (1) と (2)、および (2) と (3) の間に依存関係があります。これは、命令 (2) は実行を開始する前に命令 (1) のブレンドの結果を待機する必要があり、また命令 (3) は命令 (2) を待機することを意味します。

代替 2 では、それぞれの条件コードの分岐はすべてのデータに対し並列に実行され、マスクはメモリーにデータを書き戻す前に 1 つのレジスターへのブレンドに使用されるため、依存関係は 1 つしかありません。

ブレンドは高速ですが、マスクなしデータに対して発生する可能性がある例外をマスクしません。

代替 2 は 11% 多くの命令を実行しますが、全体の実行で 23% のスピードアップをもたらします。代替 2 ではレジスターを 1 つ多く使用します (zmm3)。この追加レジスターの使用は、レジスター·プレッシャー (レジスターをメモリーに退避し、その後ロードする) が高まると追加のレイテンシーを生じさせます。

次のコードはマスクとブレンドのもう一つの例です。

```
for (int i=0;i<len;i++){
  if (a[i] > b[i]){
    a[i] += b[i]
  }
}
```

例 18-7 マスクとブレンドの例 2

代替 1	代替 2

mov rax,a
mov rbx,b
mov rdx,size2
 	
loop1:
vmovdqa32 zmm1,[rax +rdx*4 -0x40]
vmovdqa32 zmm2,[rbx +rdx*4 -0x40]
(1) vpcmpgtd k1,zmm1,zmm2
(2) vmovdqa32 zmm3{k1}{z},zmm2
(3) vpaddd zmm1,zmm1,zmm3
vmovdqa32 [rax +rdx*4 -0x40],zmm1
sub rdx,16
jne loop1
	mov rax,a
	mov rbx,b
	mov rdx,size2
	
	loop1:
	vmovdqa32 zmm1,[rax +rdx*4 -0x40]
	vmovdqa32 zmm2,[rbx +rdx*4 -0x40]
	(1)vpcmpgtd k1,zmm1,zmm2
	(2)vpaddd zmm1{k1},zmm1,zmm2
	vmovdqa32 [rax +rdx*4 -0x40],zmm1
	sub rdx,16
	jne loop1

ベースライン・サイクル 1x **ベースライン命令 1x**	**スピードアップ: 1.05x** **命令: 0.87x**

代替 1 には、命令 (1) と (2)、および (2) と (3) の間に依存関係があります。

代替 2 では、2 つの命令 (1) と (2) にのみ依存関係のチェーンがあります。

18.2.4 入れ子になった条件/マスク集合

インテル® AVX-512 には、入れ子と (または) 複数の条件操作の実装を容易にするため、マスクレジスターですべてのビット単位の論理演算子の実行を可能にする、一連のマスク操作命令が含まれます。次の例では、論理和 (&&) は *kandw* 命令を使用して実行されます。

```
for(int iX = 0; iX < iBufferWidth; iX++)
{
  if ((*pInImage)>0 && ((*pInImage)&3)==3)
  {
    *pRefImage = (*pInImage)+5;
  }
  else
  {
    *pRefImage = (*pInImage);
  }
  pRefImage++;
  pInImage++;
}
```

例 18-8 複数条件式の実行

スカラー	インテル® AVX2	インテル® AVX-512

mov rsi, pImage
mov rdi, pOutImage
mov rbx, len
xor rax, rax
mainloop:
mov r8d, dword ptr
[rsi+rax*4]
mov r9d, r8d
cmp r8d, 0
jle label1
and r9d, 0x3
cmp r9d, 3
jne label1
add r8d, 5
label1:
mov dword ptr [rdi+rax*4],
r8d
add rax, 1
cmp rax, rbx
jne mainloop
	mov rsi, pImage
	mov rdi, pOutImage
	mov rbx, [len]
	xor rax, rax
	vpbroadcastd ymm1, [five]
	vpbroadcastd ymm7, [three]
	vpxor ymm3, ymm3, ymm3
	mainloop:
	vmovdqa ymm0, [rsi+rax*4]
	vmovaps ymm6, ymm0
	vpcmpgtd ymm5, ymm0, ymm3
	vpand ymm6, ymm6, ymm7
	vpcmpeqd ymm6, ymm6, ymm7
	vpand ymm5, ymm5, ymm6
	vpaddd ymm4, ymm0, ymm1
	vblendvps ymm4, ymm0, ymm4,
	ymm5
	vmovdqa [rdi+rax*4], ymm4
	add rax, 8
	cmp rax, rbx
	jne mainloop
		mov rsi, pImage
		mov rdi, pOutImage
		mov rbx, [len]
		xor rax, rax
		vpbroadcastd zmm1, [five]
		vpbroadcastd zmm5, [three]
		vpxord zmm3, zmm3, zmm3
		mainloop:
		vmovdqa32 zmm0, [rsi+rax*4]
		vpcmpgtd k1, zmm0, zmm3
		vpandd zmm6, zmm5, zmm0
		vpcmpeqd k2, zmm6, zmm5
		kandw k1, k2, k1
		vpaddd zmm0 {k1}, zmm0, zmm1
		vmovdqa32 [rdi+rax*4], zmm0
		add rax, 16
		cmp rax, rbx
		jne mainloop

ベースライン 1x	**スピードアップ: 5x**	**スピードアップ: 11x**

18.2.5 メモリーマスクのマイクロアーキテクチャーを改善

次の表に、Broadwell† マイクロアーキテクチャー以降で改善されたマスク操作の一覧を示します。

表 18-1 Skylake† Server と Broadwell† マイクロアーキテクチャーのキャッシュの比較

項目	Broadwell† マイクロアーキテクチャー	Skylake† Server マイクロアーキテクチャー
1	VMASKMOV ストアのアドレスは、マスクが判明した後にのみ解決されると考えられるべきです。マスク付きストアに続くロードは、マスク値が判明するまでブロックされる可能性があります (メモリー・ディスアンビゲーションによって解決されない限り)。	この問題は解決されています。vmaskmov ストアのアドレスは、マスクが認識される前に解決されます。
2	マスクがすべて 1 または 0 でないロードはマスク付きストアに依存し、ストアデータがキャッシュに書き込まれるまで待機します。マスクがすべて 1 のデータは、マスク付きストアから依存関係のあるロードへフォワードされます。マスクがすべて 0 のロードはマスク付きストアと依存関係はありません。	マスクがすべて 1 または 0 でないロードはマスク付きストアに依存し、ストアデータがキャッシュに書き込まれるまで待機します。マスクがすべて 1 のデータは、マスク付きストアから依存関係のあるロードへフォワードされます。マスクがすべて 0 のロードはマスク付きストアと依存関係はありません。
3	マスク付きロード (vmaskmov 命令を使用した) が不正なメモリーアドレス範囲を含んでいる場合、プロセッサーはどの不正な範囲にマスク値が「1」に設定されているかどうかを確認するため、複数サイクルのアシストコードを実行する可能性があります。 このアシストは、マスクが “ゼロ” であってもプログラマーにとってロードを実行すべきでないことが明白な場合に行われる可能性があります。	インテル® AVX-512 のマスク処理では、マスク値がすべてゼロであればメモリーフォルトは無視されアシストは発行されません。

18.2.6 ピーリングとリマインダーのマスク

アライメントされたデータのキャッシュラインへのアクセスは、アライメントされていないデータへのアクセスより高いパフォーマンスをもたらします。多くの場合、アドレスはコンパイル時には不明であるか、判明していてもアライメントされていません。このような場合、最初の要素からアライメントされているアドレスまでをマスク付きモードで処理し、その後ループボディーをマスクなしで処理するピーリング・アルゴリズムを導入することを推奨します。必要に応じてループの終端をリマインダー・ループでマスク付きで処理します。この方式はコードサイズを増加させますが、データ処理全体のパフォーマンスを改善します。

次のコードはピーリングとリマインダーのマスクの例です。

```
for (size_t i = 0; i < len; i++)
    pOutImage[i] = (pInImage[i] * alfa) + add_value;
```

次の例は、2 つのコードバージョン (両者ともアライメントされていない出力データ配列を使用) の実装と実行スピードの違いを示しています。

例 18-9 ピーリングとリマインダーのマスク

ピーリングなし、マスクなしのボディー、マスクされたリマインダー	ピーリングあり、マスクなしのボディー、マスクされたリマインダー

mov rbx, pOutImage // 出力
mov rax, pImage // 入力
mov rcx, len
mov edx, addValue
vpbroadcastd zmm0, edx
mov edx, alfa
vpbroadcastd zmm3, edx
mov rdx, rcx
sar rdx, 4 // 反復あたり 16 要素、RDX - ループ総数
 	
jz remainder // すべてを反復しない
xor r8, r8
vmovups zmm10, [indices]
mainloop:
vmovups zmm1, [rax + r8]
vfmadd213ps zmm1, zmm3, zmm0
vmovups [rbx + r8], zmm1
add r8, 0x40
sub rdx, 1
jne mainloop
 	
remainder:
// リマインダーのマスクを作成
and rcx, 0xF // リマインダーでの要素数
jz end // リマインダーの要素なし
vpbroadcastd zmm2, ecx
vpcmpd k2, zmm10, zmm2, 1 // 低位を比較
 	
vmovups zmm1 {k2}{z}, [rax + r8]
vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
vmovups [rbx + r8] {k2}, zmm1
end:
	mov rax, pImage // 入力
	mov rbx, pOutImage // 出力
	mov rcx, len
	movss xmm0, addValue
	vpbroadcastd zmm0, xmm1
	movss xmm1, alfa
	vpbroadcastd zmm3, xmm0
	xor r8, r8
	xor r9, r9
	vmovups zmm10, [indices]
	vmovups zmm11, [itersize]
	vpbroadcastd zmm12, ecx
	
	peeling:
	mov rdx, rbx
	and rdx, 0x3F
	jz endofpeeling // ピールの必要なし
	neg rdx
	add rdx, 64 // 64 - X
	// rdx にアライメントされた位置までのバイト数が格納されている
	mov r9, rdx
	sar r9, 2 // r9 にはピーリングされた要素数が含まれる
	vpbroadcastd zmm12, r9d
	vpcmpd k2, zmm10, zmm12, 1 // ピーリングのマスクを生成するため低位を比較
	
	vmovups zmm1 {k2}{z}, [rax]
	vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
	vmovups [rbx] {k2}, zmm1 // アライメントされていないストア
	
	endofpeeling:

	sub rcx, r9
	mov r8, rcx
	sar r8, 4 // ループの総数
	jz remainder // すべてを反復しない
	
	mainloop:
	vmovups zmm1, [rax + rdx]
	vfmadd213ps zmm1, zmm3, zmm0
	vmovaps [rbx + rdx], zmm1 // アライメントされたストア
	add rdx, 0x40
	sub r8, 1
	jne mainloop
	
	remainder:
	// リマインダーのマスクを作成
	and rcx, 0xF // リマインダーでの要素数
	jz end // リマインダーの要素なし
	vpbroadcastd zmm2, ecx
	vpcmpd k2, zmm10, zmm2, 1 // 低位を比較
	vmovups zmm1 {k2}{z}, [rax + rdx]
	vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
	vmovaps [rbx + rdx] {k2}, zmm1 //アライメントされている
	end:

ベースライン 1x	**スピードアップ: 1.04x**

18.3 フォワーディングとマスク付き操作

マスクなしのストア命令を使用しその後にロード命令が続く場合、データのフォワーディングはロードのタイプ、サイズそしてストアアドレスからのアドレス·オフセットに依存しますが、ストアアドレスそのものには関係しません (ストアアドレスはキャッシュラインにアライメントおよびフィットする必要がなく、フォワーディングはアライメントされずライン分割されたストアで発生します)。

次の図は、データのフォワーディングが起こる可能性があるすべてのケースを示しています。

図 18-3 データがフォワードされる場合

データ・フォワーディングを使用する際に考慮すべき 2 つの重要なことがあります。

1. GPR へのデータフォワードは、ストア命令の下位 256 ビットのみが可能です。直前に書き込まれたデータを GPR にロードする際は注意が必要です。
2. フォワーディングは特定のマスクでのみ行われるため、マスクを使用してはいけません。

18.4 フォワーディングとメモリーマスク

マスク付きロードやストアを使用する場合、次のことを考慮してください。

- マスク値がすべてゼロまたはすべて非ゼロではない場合、マスク付きストアに続くロード操作が同じアドレスを持っていると、データがキャッシュに書き込まれるまでブロックされます。
- GPR のフォワーディングとは異なり、ロードとストアアドレスが完全に同じではない場合、ベクトルロードのマスクありなしにかかわりなくフォワードは行われません。
  - st_mask = 10101010、ld_mask = 01010101、フォワードなし、ブロックされる。
  - st_mask = 00001111、ld_mask = 00000011、フォワードなし、ブロックされる。

- マスク値がすべて 1 である場合、データはロード操作にフォワードされるためブロックされません。
  - st_mask = 11111111、ld_mask = 影響なし、フォワードあり、ブロックされない。

- また、マスク値がすべてゼロである場合、フォワードされませんがブロックされることもありません。
  - st_mask = 00000000、ld_mask = 影響なし、フォワードなし、ブロックされない。

マスク付きのストアは慎重に利用されるべきです。例えば、リマインダーのサイズがコンパイル時に 1 であることが判明していれば、その後に同じキャッシュラインからロードする操作があれば (または、アドレス + ベクトル長でオーバーラップする)、マスク付きのリマインダー・ブロックよりもスカラーのリマインダーを使用した方がいいでしょう。

18.5 データコンプレス

データコンプレス操作は、入力バッファーからマスクレジスター 1 ビットで指定されたインデックスの要素を読み込みます。読み込まれた要素はデスティネーション・バッファーへ書き込まれます。要素数がデスティネーション・レジスターのサイズ未満であれば、余りの空間はゼロで埋められます。

次の図はデータコンプレス操作を示します。

```
if (k[i] == 1)
{
  dest[a] = src[i];
  a++;
}
```

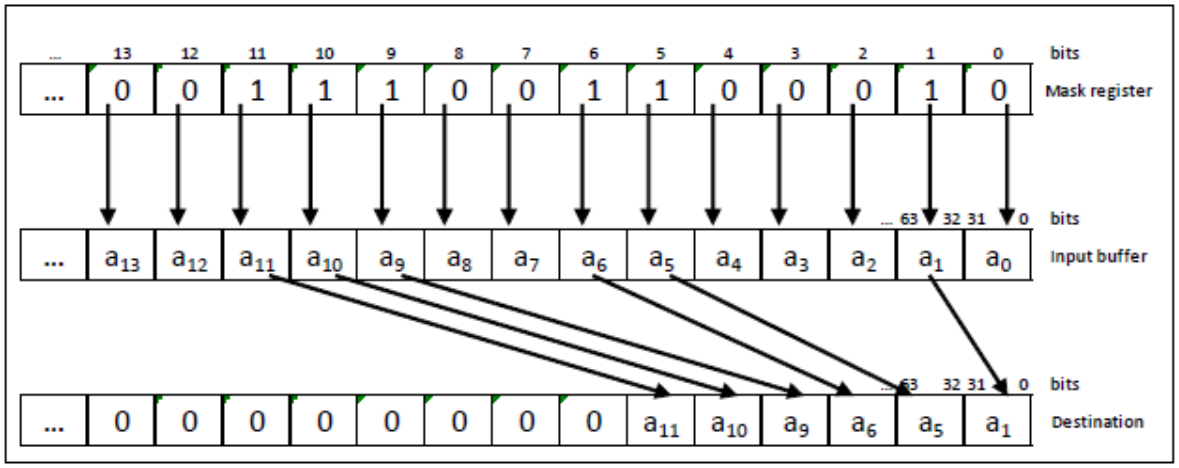


図 18-4 データコンプレス操作

18.5.1 データコンプレスの例

次のコードは、配列 a から正の値を持つデータを配列 b に移動 (収集) しています。

```
for (int i=0; i<SIZE; i++)
{
  if ( a[i] > 0 )
    b[j++] = a[i];
}
```

次の 4 つの実装は dword 要素の配列からのコンプレス操作を行っています。

- 代替 1 では、スカラーのデータアクセスを行い、各要素を個別にチェックしています。0 より大きな値はデスティネーション配列に書き込まれます。
- 代替 2 は、事前割り当てによりシャッフルキーで初期化されたテーブルと、インテル® AVX の shuffle 命令を使用しています。比較命令はシャッフルテーブルのエントリーポイント数を提供し、次にキーがロードされ、キーに従って元の配列がシャッフルされます。反復ごとに 4 つの要素が処理されます。
- 代替 3 は、代替 2 と同じアルゴリズムを使用しますが、インテル® AVX2 の 256 ビット・レジスターと、バイト・シャッフルに変わって dword 命令で並べ替えを使用します。反復ごとに 8 つの要素が処理されます。
- 代替 4 は、コンプレスキーとしてマスクレジスターと *vpcompress* 命令を使用するインテル® AVX-512 のアルゴリズムです。反復ごとに 16 の要素が処理されます。

例 18-10 インテル® AVX-512 のデータコンプレスと他の代替実装を比較

代替 1 スカラー

```
   mov rsi, source
   mov rdi, dest
   mov r9, len

   xor r8, r8
   xor r10, r10
mainloop:
   mov r11d, dword ptr [rsi+r8*4]
   test r11d, r11d
   jle m1
   mov dword ptr [rdi+r10*4], r11d
   inc r10
m1:
   inc r8
   cmp r8, r9
   jne mainloop
```

ベースライン 1x

代替 2 インテル® AVX

```
   mov rsi, source
   mov rdi, dest
   mov r14, shuffle_LUT
   mov r15, write_mask
   mov r9, len

   xor r8, r8
   xor r11, r11
   vpxor xmm0, xmm0, xmm0
mainloop:
   vmovdqa xmm1, [rsi+r8*4]
   vpcmpgtd xmm2, xmm1, xmm0
   mov r10, 4
   vmovmskps r13, xmm2
   shl r13, 4
   vmovdqu xmm3, [r14+r13]
   vpshufb xmm2, xmm1, xmm3
   popcnt r13, r13
   sub r10, r13
   vmovdqu xmm3, [r15+r10*4]
   vmaskmovps [rdi+r11*4], xmm3, xmm2
   add r11, r13
   add r8, 4
   cmp r8, r9
   jne mainloop

shuffle_LUT:
   .int 0x80808080, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x80808080, 0x80808080, 0x80808080
   .int 0x07060504, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x80808080, 0x80808080
   .int 0x0b0A0908, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x0b0A0908, 0x80808080, 0x80808080
```

```
   .int 0x07060504, 0x0b0A0908, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x0b0A0908, 0x80808080
   .int 0x0F0E0D0C, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x07060504, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x0F0E0D0C, 0x80808080
   .int 0x0b0A0908, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x03020100, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
   .int 0x07060504, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
   .int 0x03020100, 0x07060504, 0x0b0A0908, 0x0F0E0D0C

write_mask:
   .int 0x80000000, 0x80000000, 0x80000000, 0x80000000
   .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

スピードアップ: 2.87x

代替 3 インテル® AVX

```
   mov rsi, source
   mov rdi, dest
   mov r14, shuffle_LUT
   mov r15, write_mask
   mov r9, len

   xor r8, r8
   xor r11, r11
   vpxor ymm0, ymm0, ymm0
mainloop:
   vmovdqa ymm1, [rsi+r8*4]
   vpcmpgtd ymm2, ymm1, ymm0
   mov r10, 8
   vmovmskps r13, ymm2
   shl r13, 5
   vmovdqu ymm3, [r14+r13]
   vpermd ymm2, ymm3, ymm1
   popcnt r13, r13
   sub r10, r13
   vmovdqu ymm3, [r15+r10*4]
   vmaskmovps [rdi+r11*4], ymm3, ymm2
   add r11, r13
   add r8, 8
   cmp r8, r9
   jne mainloop

// The lookup table is too large to reproduce in the document.
// It consists of 256 rows of 8 32 bit integers.
//The first 8 and the last 8 rows are shown below.

shuffle_LUT:
   .int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
   .int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
   .int 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
   .int 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
   .int 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
   .int 0x0, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
```

```
   .int 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
   .int 0x0, 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0

// Skipping 240 lines
   .int 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0, 0x0
   .int 0x0, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
   .int 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
   .int 0x0, 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
   .int 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
   .int 0x0, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
   .int 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
   .int 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7

write_mask:
   .int 0x80000000, 0x80000000, 0x80000000, 0x80000000
   .int 0x80000000, 0x80000000, 0x80000000, 0x80000000
   .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
   .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

スピードアップ: 5.27x

代替 4 インテル® AVX-512

```
   mov rsi, source
   mov rdi, dest
   mov r9, len

   xor r8, r8
   xor r10, r10
   vpxord zmm0, zmm0, zmm0
mainloop:
   vmovdqa32 zmm1, [rsi+r8*4]
   vpcmpgtd k1, zmm1, zmm0
   vpcompressd zmm2 {k1}, zmm1
   vmovdqu32 [rdi+r10*4], zmm2
   kmovd r11d, k1
   popcnt r12, r11
   add r8, 16
   add r10, r12
   cmp r8, r9
jne mainloop
```

スピードアップ: 11.9x

18.6 データ・エクスパンド

データ・エクスパンド操作は、ソース配列 (レジスター) から要素を読み込み、マスクレジスターのビットで有効化された位置の要素をデスティネーション・レジスターに書き込みます。有効化されたビット数がデスティネーション・レジスターのサイズ未満である場合、余分な値は無視されます。

```
if (k[i] == 1)
{
  dest[i] = src[a];
  a++;
}
```

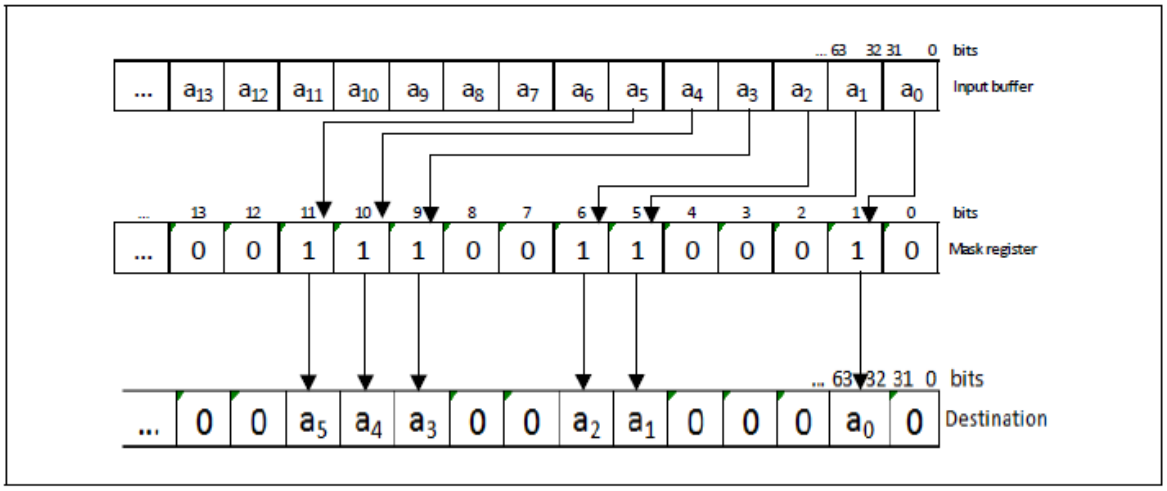


図 18-5 データ・エクスパンド操作

18.6.1 データ・エクスパンドの例

次のコードはエクスパンド操作を使用した例を示しています。配列 a のすべての正数に対し、コードは一連の番号を設定します。

```
for (int i=0; i<SIZE; i++){
  if (a[i] > 0)
  dest[i] = a[count++];
  else
  dest[i] = 0;
}
```

次の 3 つの実装は、16 個の dword 要素の配列からのエクスパンド操作を行っています。

- 代替 1 では、スカラーのデータアクセスを行い、各要素を個別にチェックしています。値が 0 より大きければデスティネーション配列の対応する要素は、インデックス・カウンターで示されるソースの値で上書きされ、カウンターがインクリメントされます。
- 代替 2 は、事前割り当てによりシャッフルキーで初期化されたテーブルと、インテル® AVX2 の shuffle 命令を使用しています。比較命令はシャッフルテーブルのエントリーポイント数を提供し、次にキーがロードされ、キーに従って元の配列がシャッフルされます。反復ごとに 4 つの要素が処理されます。
- 代替 3 は、エクスパンド・キーとしてマスクレジスターと *vpexpandd* 命令を使用するインテル® AVX-512 のアルゴリズムです。反復ごとに 16 の要素が処理されます。

例 18-11 インテル® AVX-512 のデータ・エクスパンドと他の代替実装を比較

代替 1: スカラー	代替 2: インテル® AVX2 コード	代替 3: インテル® AVX-512 コード

mov rsi, input
mov rdi, output
mov r9, len
xor r8, r8
xor r10, r10
 	
mainloop:
mov r11d, dword ptr [rsi+r8*4]
test r11d, r11d
jle m1
mov r11d, dword ptr [rsi+r10*4]
mov dword ptr [rdi+r8*4], r11d
inc r10
m1:
inc r8
cmp r8, r9
jne mainloop
	mov rsi, input
	mov rdi, output
	mov r9, len
	xor r8, r8
	xor r10, r10
	vpxord ymm0, ymm0, ymm0
	mov r14, shuf2
	xor r15,r15
	mainloop:
	vmovdqa ymm1,[rsi+r8*4]
	vpxor ymm4, ymm4, ymm4
	vpbroadcastd ymm4,r15d
	vpcmpgtd ymm2,ymm1,ymm0
	vmovdqu ymm1,[rsi+r10*4]
	vmovmskps r13,ymm2
	shl r13, 5
	vmovdqa ymm3,[r14+r13]
	vpermd ymm4,ymm3,ymm1
	popcnt r13,r13
	add r10, r13
	vmaskmovps [rdi+r8*4],ymm2, ymm4
	add r8, 8
	cmp r8, r9
	jne mainloop
	// The lookup table is too large
	// to reproduce in the document.
	// It consists of 256 rows of 8
	// 32-bit integers. The first 8
	// and the last 8 rows are shown
	// below. The table needs to be
	// 32-byte aligned.
	
	shuf2:
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 1, 0, 0, 0, 0, 0, 0
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 0, 1, 0, 0, 0, 0, 0
	.int 0, 0, 1, 0, 0, 0, 0, 0
	.int 0, 1, 2, 0, 0, 0, 0, 0
	// Skipping 240 lines
	.int 0, 0, 0, 0, 1, 2, 3, 4
	.int 0, 0, 0, 1, 2, 3, 4, 5
	.int 0, 0, 0, 1, 2, 3, 4, 5
	.int 0, 1, 0, 2, 3, 4, 5, 6
	.int 0, 0, 0, 1, 2, 3, 4, 5
	.int 0, 0, 1, 2, 3, 4, 5, 6
	.int 0, 0, 1, 2, 3, 4, 5, 6
	.int 0, 1, 2, 3, 4, 5, 6, 7
		vpxord zmm0, zmm0, zmm0
		
		mainloop:
		vmovdqa32 zmm1,[rsi+r8*4]
		vpcmpgtd k1, zmm1, zmm0
		vmovdqu32 zmm1,[rsi+r10*4]
		vpexpandd zmm2 {k1}{z}, zmm1
		vmovdqu32 [rdi+r8*4],zmm2
		add r8, 16
		kmovd r11d, k1
		popcnt r12, r11
		add r10, r12
		cmp r8, r9
		jne mainloop

ベースライン 1x	**スピードアップ: 4.23x**	**スピードアップ: 8.58x**

18.7 三項論理

三項論理 *vpternlog* 命令は、指定される 3 つのオペランド間のどのような論理関数も実行します。命令は 3 つのオペランドと即値 (この論理式の真理値表) を必要とします。最初のオペランドはデスティネーションでもあるため、実行後元の値は上書きされます。

18.7.1 三項論理の例 1

次の例は 3 つの値のビット単位の論理関数を示しています。この例の関数は次の真理値表で定義されます。

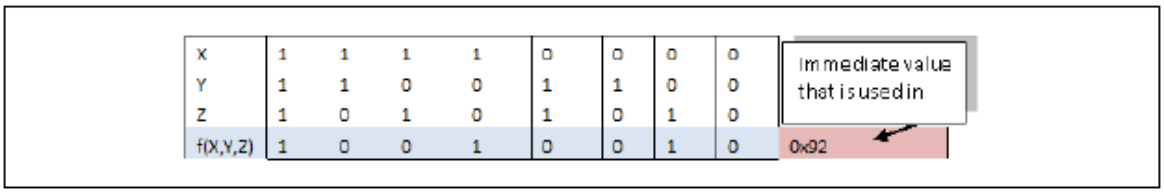


図 18-6 三項論理の例 1 - 真理値表

この真理値表のカルノー図を使用して、次のように関数を定義できます。

f(X,Y,Z) = $\bar{x}\,\bar{y}\,z \vee xyz \vee x\,\bar{y}\,\bar{z}$

または、短い表記で、より少ないブール演算を使用します。

f(X,Y,Z) = $\bar{y}(z \oplus x) \vee xyz$

上記の関数の C コードは次のようになります。

```
for (int i=0; i<SIZE; i++)
{
  Dst[i] = ((~Src2[i]) & (Src1[i] ^ Src3[i])) | (Src1[i] & Src2[i] & Src3[i]);
}
```

X、Y および Z のそれぞれの組み合わせによる関数の値は、命令で使用される即値を与えます。

以下に X、Y および Z のすべての値に、この論理関数を適用する 3 つの実装を示します。

- 代替 1 は、インテル® AVX2 で利用可能なビット単位の論理関数を使用した、256 ビット・ベクトル計算です。
- 代替 2 は、インテル® AVX-512 で利用可能なビット単位の論理関数を使用した 512 ビット・ベクトル計算です (*vpternlog* 命令なし)。
- 代替 2 は、*vpternlog* 命令を使用したインテル® AVX-512 による 512 ビット・ベクトル計算です。すべての例で、2 回のアンロールが行われています。

例 18-12 三項論理と代替手法の比較

代替 1: インテル® AVX2

```
    mov rax, src1
    mov rbx, src2
    mov rcx, src3
    mov r11, dst
    mov r8, len
    xor r10, r10
mainloop:
    vmovdqu ymm1, ymmword ptr [rax+r10*4]
    vmovdqu ymm3, ymmword ptr [rdx+r10*4]
    vmovdqu ymm2, ymmword ptr [rcx+r10*4]
    vmovdqu ymm10, ymmword ptr [rcx+r10*4+0x20]
    vpand ymm0, ymm1, ymm3
    vpxor ymm4, ymm1, ymm2
    vpand ymm5, ymm0, ymm2
    vpandn ymm6, ymm3, ymm4
    vpor ymm7, ymm5, ymm6
    vmovdqu ymmword ptr [r11+r10*4], ymm7
    vmovdqu ymm9, ymmword ptr [rax+r10*4+0x20]
    vmovdqu ymm11, ymmword ptr [rdx+r10*4+0x20]
    vpxor ymm12, ymm9, ymm10
    vpand ymm8, ymm9, ymm11
    vpandn ymm14, ymm11, ymm12
    vpand ymm13, ymm8, ymm10
    vpor ymm15, ymm13, ymm14
    vmovdqu ymmword ptr [r11+r10*4+0x20], ymm15
    add r10, 0x10
    cmp r10, r8
    jb mainloop
```

ベースライン 1x

例 18-12 三項論理と代替手法の比較 (続き)

代替 2: インテル® AVX-512 論理命令	代替 3: インテル® AVX-512 の vpternlog 命令を使用

mov rdi, src1
mov rsi, src2
mov rdx, src3
mov r10, dst
mov r11, dst
mov r8, len
xor r10, r10
mainloop:
vmovups zmm2,k0,zmmword ptr [rdi+r10*4]
vmovups zmm4,k0,zmmword ptr [rdi+r10*4+0x40]
vmovups zmm6,k0,zmmword ptr [rsi+r10*4]
vmovups zmm8,k0,zmmword ptr [rsi+r10*4+0x40]
vmovups zmm3,k0,zmmword ptr [rdx+r10*4]
vmovups zmm5,k0,zmmword ptr [rdx+r10*4+0x40]
vpandd zmm0,k0, zmm2, zmm6
vpandd zmm1,k0, zmm4, zmm8
vpxord zmm7,k0, zmm2, zmm3
vpxord zmm9,k0, zmm4, zmm5
vpandd zmm10,k0, zmm0, zmm3
vpandd zmm12,k0, zmm1, zmm5
vpandnd zmm11,k0, zmm6, zmm7
vpandnd zmm13,k0, zmm8, zmm9
vpord zmm14,k0, zmm10, zmm11
vpord zmm15,k0, zmm12, zmm13
vmovups zmmword ptr [r10+r10*4],k0,zmm14
vmovups zmmword ptr [r10+r10*4+0x40],k0,zmm15
add r10, 0x20
cmp r10, r9
jb mainloop
	mov r9, src1
	mov r8, src2
	mov r10, src3
	mov r11, dst
	mov rsi, len
	xor rax rax
	
	mainloop:
	vmovaps zmm1,[r8+rax*4]
	vmovaps zmm0,[r9+rax*4]
	vpternlogd zmm0,zmm1,[r10], 0x92
	vmovaps [r11],zmm0
	vmovaps zmm1,[r8+rax*4+0x40]
	vmovaps zmm0,[r9+rax*4+0x40]
	vpternlogd zmm0,zmm1,[r10+0x40],0x92
	vmovaps [r11+0x40],zmm0
	add rax,32
	add r10,0x80
	add r11, 0x80
	cmp rax, rsi
	cmp eax,esi
	jne mainloop

スピードアップ: 1.94x	**スピードアップ: 2.36x** **(インテル® AVX-512 論理命令に対し 1.22x)**

18.7.2 三項論理の例 2

次の例は、Fortran で良く使用される符号変換操作です。浮動小数点数を持つ 2 つの配列を使用するコードを考えてみましょう。

```
for (int i=0; i<SIZE; i++)
{
  b[i] = a[i] > 0 ? b[i] : -b[i];
}
```

これは、次のコードと等価です。

```
for (int i=0; i<SIZE; i++)
{
  b[i] = ( a[i] & 0x80000000 ) ^ b[i];
}
```

言い換えると、

$x = (y \wedge z) \oplus x$

は、次の真理値表を定義します。

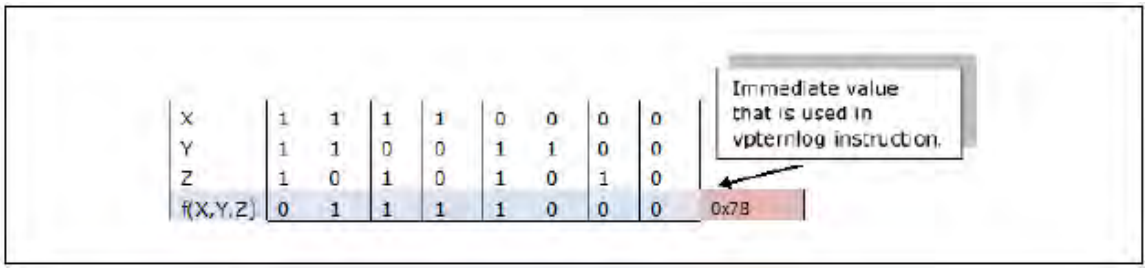


図 18-7 三項論理の例 2 - 真理値表

つまり 2 つの論理命令 (*vpand* と *vpxor*) は、*vpternlog* 命令に置き換えることができます。

```
vpternlog x,y,z,0x78
```

18.8 新しいシャッフル命令

インテル® AVX-512 では 3 つの新しい shuffle 操作が追加されています。

- vpermw: いずれの word も並べ替え可能な新しい単一ソース命令
- permt2[w/d/q//ps/pd]: 2 つのソースのいずれも並べ替え可能な新しい命令 (ソースレジスターを上書き)
- permi2[w/d/q/ps/pd]: 2 つのソースのいずれも並べ替え可能な新しい命令 (制御レジスターを上書き)

次の図に *vpermi2ps* 命令の仕組みを示します。この例では zmm0 はシャッフルの制御に使用されますが、出力レジスター (制御レジスターは上書きされます) でもあることに注意してください。

```
vpermi2ps zmm0, zmm1, zmm2
```

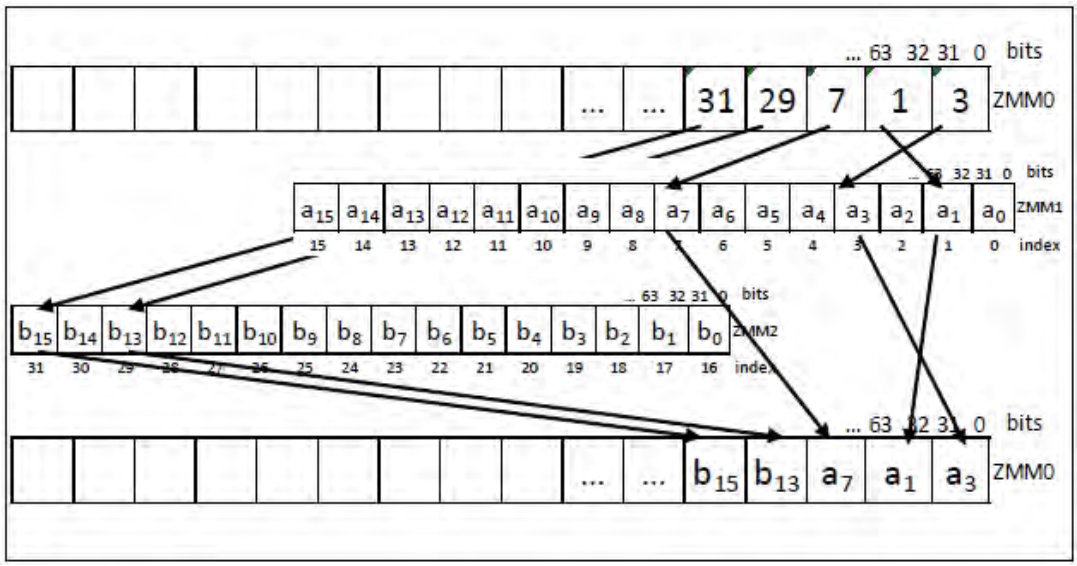
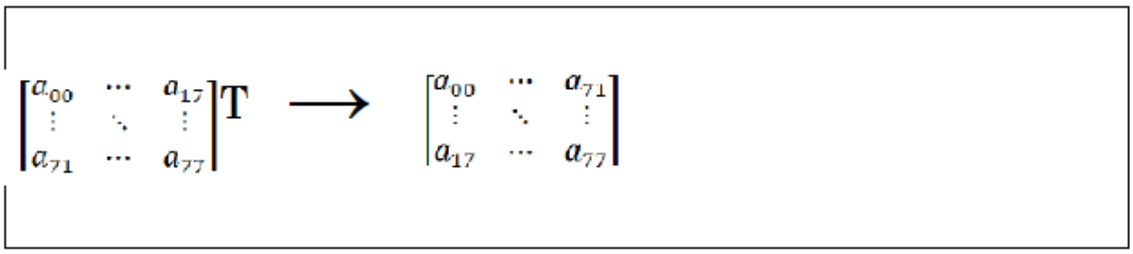


図 18-8 vpermi2ps 命令の操作

インデックス·レジスターの値は、命令とソースレジスターで同じ解像度でなければならないことに注意してください (word を扱うときは word、dword の時は dword など)。

18.8.1 2 つのソースのパーミュートの例

この例では、行列の転置操作における 2 つのソースを持つパーミュート命令の使い方を示します。転置する行列は word 要素の 8x8 正方形行列です。



C コードで表すと次のようになります (各行列は 8*8*2 = 128 バイトの連続したブロックであると仮定しています)。

```
for(int iY = 0; iY < 8; iY++)
{
  for(int iX = 0; iX < 8; iX++)
  {
    trasposedMatrix[iY*8+iX] = originalMatrix[iX*8+iY];
  }
}
```

この行列転置の 3 つの実装を用意しました。

- 代替 1 は、スカラーコードで、ソース行列のそれぞれの要素をアクセスして、デスティネーション行列の対応する位置にそれを配置します。このコードは、1 つの行列ごとに 64 (8x8) 回反復します。
- 代替 2 は、インテル® AVX2 コードで、インテル® AVX2 のパーミュートとシャッフル (アンパック) 命令を使用します。8x8 行列ごとに必要な反復回数は 1 回のみです。
- 代替 3 は、インテル® AVX-512 コードで、2 ソースのパーミュート命令を使用します。このコードは最初にパーミュート・マスクをロードして、次に行列データをロードしていることに注目してください。パーミュート操作に使用されるマスクは、次の配列に保存されています。

```
short permMaskBuffer [8*8] = { 0, 8, 16, 24, 32, 40, 48, 56,
                               1, 9, 17, 25, 33, 41, 49, 57,
                               2, 10, 18, 26, 34, 42, 50, 58,
                               3, 11, 19, 27, 35, 43, 51, 59,
                               4, 12, 20, 28, 36, 44, 52, 60,
                               5, 13, 21, 29, 37, 45, 53, 61,
                               6, 14, 22, 30, 38, 46, 54, 62,
                               7, 15, 23, 31, 39, 47, 55, 63 };
```

各代替コードは、50 個の行列 (それぞれ 2 バイト要素の 8x8) を転置します。

例 18-13 行列転置の代替手法

代替 1: スカラーコード	代替 2: インテル® AVX2 コード	代替 3: インテル® AVX-512 コード

mov rsi, pImage
mov rdi, pOutImage
xor rdx, rdx
 	
matrix_loop:
xor rax, rax
outerloop:
xor rbx, rbx
innerloop:
mov rcx, rax
shl rcx, 3
add rcx, rbx
mov r8w, word ptr
[rsi+rcx*2]
mov rcx, rbx
shl rcx, 3
add rcx, rax
mov word ptr[rdi+rcx*2],r8w
add rbx, 1
cmp rbx, 8
jne innerloop
add rax, 1
cmp rax, 8
jne outerloop
add rdx, 1
add rsi, 64*2
add rdi, 64*2
cmp rdx, 50
jne matrix_loop
	mov rsi, pImage
	mov rdi, pOutImage
	xor rdx, rdx
	
	matrix_loop:
	vmovdqa xmm0, [rsi]
	vmovdqa xmm1, [rsi+0x10]
	vmovdqa xmm2, [rsi+0x20]
	vmovdqa xmm3, [rsi+0x30]
	vinserti128 ymm0, ymm0,
	[rsi+0x40], 0x1
	vinserti128 ymm1, ymm1,
	[rsi+0x50], 0x1
	vinserti128 ymm2, ymm2,
	[rsi+0x60], 0x1
	vinserti128 ymm3, ymm3,
	[rsi+0x70], 0x1
	vpunpcklwd ymm4, ymm0, ymm1
	vpunpckhwd ymm5, ymm0, ymm1
	vpunpcklwd ymm6, ymm2, ymm3
	vpunpckhwd ymm7, ymm2, ymm3
	vpunpckldq ymm0, ymm4, ymm6
	vpunpckhdq ymm1, ymm4, ymm6
	vpunpckldq ymm2, ymm5, ymm7
	vpunpckhdq ymm3, ymm5, ymm7
	vpermq ymm0, ymm0, 0xD8
	vpermq ymm1, ymm1, 0xD8
	vpermq ymm2, ymm2, 0xD8
	vpermq ymm3, ymm3, 0xD8
	vmovdqa [rdi], ymm0
	vmovdqa [rdi+0x20], ymm1
	vmovdqa [rdi+0x40], ymm2
	vmovdqa [rdi+0x60], ymm3
	add rdx, 1
	add rsi, 64*2
	add rdi, 64*2
	cmp rdx, 50
	jne matrix_loop
		mov rax, permMaskBuffer
		vmovdqa32 zmm10, [rax]
		vmovdqa32 zmm11, [rax+0x40]
		mov rsi, pImage
		mov rdi, pOutImage
		xor rdx, rdx
		matrix_loop:
		vmovdqa32 zmm2, [rsi]
		vmovdqa32 zmm3, [rsi+0x40]
		vmovdqa32 zmm0, zmm10
		vmovdqa32 zmm1, zmm11
		vpermi2w zmm0, zmm2, zmm3
		vpermi2w zmm1, zmm2, zmm3
		vmovdqa32 [rdi], zmm0
		vmovdqa32 [rdi+0x40], zmm1
		add rdx, 1
		add rsi, 64*2
		add rdi, 64*2
		cmp rdx, 50
		jne matrix_loop

ベースライン 1x	**スピードアップ: 13.7x**	**スピードアップ: 37.3x** **(インテル® AVX2 コードと比べて 2.7x)**

18.9 ブロードキャスト処理

18.9.1 組込みブロードキャスト

インテル® AVX-512 では、ブロードキャストなしの命令シンタックス内でも暗黙的にブロードキャスト操作を適用できる、組込みブロードキャスト操作が実装されています。メモリーからのソースをブロードキャストできます。つまり、有効なソースオペランドのすべての要素に渡って、追加のソースレジスターを使用することなく、32 ビット・データ要素では最大 16 回 (64 ビット・データでは最大 8 回) 繰り返されます。これは、ベクトル命令のすべての操作で同じスカラーオペランドを繰り返して使用する際に便利です。

組込みブロードキャストは、要素サイズが 32 または 64 ビットの命令で利用できます。バイトとワード要素のブロードキャストは、組込みブロードキャストではサポートされません。そのためバイトとワードのブロードキャストには、通常のブロードキャストを使用してください。

組込みブロードキャストは使用するレジスター数が少ないため、レジスター・プレッシャーが高いコードでは特に有効です。

さらに、組込みブロードキャストを使用すると、ロード uop は操作を行う uop と同じ命令にあるため、マイクロフュージョンの恩恵を受けることができます。

例えば、次のコードの置き換えについて考えてみます。

```
vbroadcastss zmm3, [rax]
vmulps zmm1, zmm2, zmm3
```

置換後:

```
vmulps zmm1, zmm2, [rax] {1to16}
```

{1to16} プリミティブでは、次のことが行われます。
1. メモリーから float32 (単精度) 要素を 1 つロードします。
2. 16 個の 32 ビット浮動小数点要素を形成するため、16 回繰り返します。

インテル® AVX-512 命令のストア・セマンティクスと純粋なロード命令は、ブロードキャスト・プリミティブをサポートしていません。

18.9.2 ロードポートで実行されるブロードキャスト

Skylake† Server マイクロアーキテクチャーでは、32 ビット以上のメモリーオペランドを持つブロードキャスト命令は、ロードポートで実行されます。シャッフルのようにポート 5 では実行されません。次の代替 2 の例では、ロードポートでブロードキャストを実行すると、どのようにポート 5 のワークロードが減少して、パフォーマンスが向上するか示しています。代替 3 は、組込みブロードキャストが、ロードポートとマイクロフュージョンでブロードキャストを実行する利点を示します。

例 18-14 ロードポートでブロードキャストを代替実行

代替 1: 32 ビットのロード操作とレジスター・ブロードキャスト	**代替 2: 32 ビット・メモリー・オペランドとブロードキャスト**	**代替 3: 32 ビット組込みブロードキャスト**

loop:
vmovd xmm0, [rax]
vpbroadcastd zmm0, xmm0
vpaddd zmm2, zmm1, zmm0
vpermd zmm2, zmm3, zmm2
add rax, 0x4
sub rdx, 0x1
jnz loop
	Loop:
	vpbroadcastd zmm0, [rax]
	vpaddd zmm2, zmm1, zmm0
	vpermd zmm2, zmm3, zmm2
	add rax, 0x4
	sub rdx, 0x1
	jnz loop
		loop:
		vpaddd zmm2, zmm1, [rax]{1to16}
		vpermd zmm2, zmm3, zmm2
		add rax, 0x4
		
		sub rdx, 0x1
		jnz loop

ベースライン 1x	**スピードアップ: 1.57x**	**スピードアップ: 1.9x**

次の例は、Skylake† Server マイクロアーキテクチャーでは、16 ビット・ブロードキャストがポート 5 で実行されるため、メモリー・オペランド・ブロードキャストからは利点を得られないことを示しています。

例 18-15 ポート 5 での 16 ビット・ブロードキャストの実行

代替 1: 16 ビット・ロードとレジスター・ブロードキャスト	代替 2: 16 ビット・メモリー・オペランドとブロードキャスト

loop:
vmovd xmm0, [rax]
vpbroadcastw zmm0, xmm0
vpaddw zmm2, zmm1, zmm0
vpermw zmm2, zmm3, zmm2
add rax, 0x4
sub rdx, 0x1
jnz loop
	loop:
	vpbroadcastw zmm0, [rax]
	vpaddw zmm2, zmm1, zmm0
	vpermw zmm2, zmm3, zmm2
	add rax, 0x2
	sub rdx, 0x1
	jnz loop

ベースライン 1x	**スピードアップ: ベースラインと同じ**

組込みブロードキャストは、16 ビット・メモリー・オペランドではサポートされないことに注意してください。

18.10 組込み丸め操作

デフォルトでは、丸めモードは MXCSR レジスターの 13:14 ビットで設定されます。

インテル® AVX-512 では、(命令ごとの) 静的丸めモード (RM) や丸めモードの上書きと呼ばれる新しい命令属性が導入されています。この属性は、MXCSR の RM ビットの値にかかわりなく、特定の演算丸めモードを適用することを可能にします。丸めモードと組み合わせて、インテル® AVX-512 はまた SAE (“suppress-all-exceptions” - すべての例外を抑制) 属性を持っています。これにより、MXCSR のすべての浮動小数点例外フラグを無効化できます。丸めモードが有効であると SAE は常に適用されます。

静的丸めモードと SAE の制御は、レジスター-レジスターベクトル命令の EVEX.b ビットを 1 に設定することで、命令のエンコードで有効にできます。この場合、ベクトル長は最大ベクトル長 (インテル® AVX-512 では 512 ビット) であると想定されます。次の表にインテル® AVX-512 で設定可能な静的丸めモードの一覧を示します。いくつかの命令では、静的に指定される即値ビットで、すでに丸めモードが設定されていることに注意してください。この場合、即値ビットは MXCSR.RM のビットに優先されるように、組込み丸めモードよりも優先されます。

18.10.1 静的丸めモード

静的丸めモードの機能と説明を以下に示します。

表 18-2 静的丸めモードの機能

機能	説明
{rn-sae}	最も近い偶数に丸める + SAE
{rd-sae}	切り下げ (負の無限大へ) + SAE
{ru-sae}	切り上げ (正の無限大へ) + SAE
{rz-sae}	ゼロへ丸め (切り捨て) + SAE

次に利用例を示します。

例 18-16 組込みと非組込み丸め

組込み丸めを使用	組込み丸めを使用しない

vaddps zmm7 {k6}, zmm2, zmm4, {ru-sae}	; rax & rcx は、MXCSR 値のロードとセーブ (復元用) に使用されるメモリー内の一次的な dword 値を指します
	
	vstmxcsr [rax] ; mxcsr 値をメモリーへロード
	mov ebx, [rax] ; レジスターへ移動
	and ebx, 0xFFFF9FFF ; ゼロ RM ビット
	または ebx, 0x5F80 ;{ru} を RM ビットに設定してすべての例外を抑制
	mov [rcx], ebx ; 新しい値をメモリーへ移動
	vldmxcsr [rcx] ; MXCSR へ保存
	
	vaddps zmm7 {k6}, zmm2, zmm4 ; 自身を操作
	
	vldmxcsr [rax] ; 以前の MXCSR 値をリストア

このコードは、ベクトル zmm2 と zmm4 の単精度浮動小数数を正の無限大への切り上げで加算しており、結果は k6 の条件付き書き込みマスクによってベクトル zmm7 に生成されます。MXCSR.RM ビットはこの命令では無視され、結果に影響しないことに注意してください。

以下は静的丸めモードが許されない命令の例です。

```
; 丸めモードは命令即値で設定済み
vrndscaleps zmm7 {k6}, zmm2 {rd}, 0x00

; メモリーオペランド付き命令
vmulps zmm7 {k6}, zmm2, [rax] {rd}

; 最大ベクトル長 (512 ビット) とは異なるベクトル長を持つ命令
vaddps ymm7 {k6}, ymm2, ymm4 {rd}

; 浮動小数点命令なし
vpaddd zmm7 {k6}, zmm2, zmm4 {rd}
```

18.11 スキャッター命令

この命令は連続しないデータのストア (分散) を実行します。命令は、与えられたベースアドレス、符号付きのオフセット、およびデータ項目により、データレジスターの各要素を、ベースアドレスとオフセットから計算されるメモリー・ロケーションに書き込みます。命令は、ベースアドレスとインデックス・ベクトルで示されるメモリー・ロケーションに、ダブルワード・ベクトルでは最大 16 要素 (qword インデックスでは 8 要素)、またはクワッドワード・ベクトルでは最大 8 要素をストアします。各要素は対応するマスクビットが 1 の場合にのみストアされます。図 18-9 は次の操作を示しています。

vscatterdpd [rax + zmm0]{k1} , zmm1

この例では、rax にはベースアドレス、zmm0 にはオフセット、そして zmm1 には書き込むデータが含まれます。

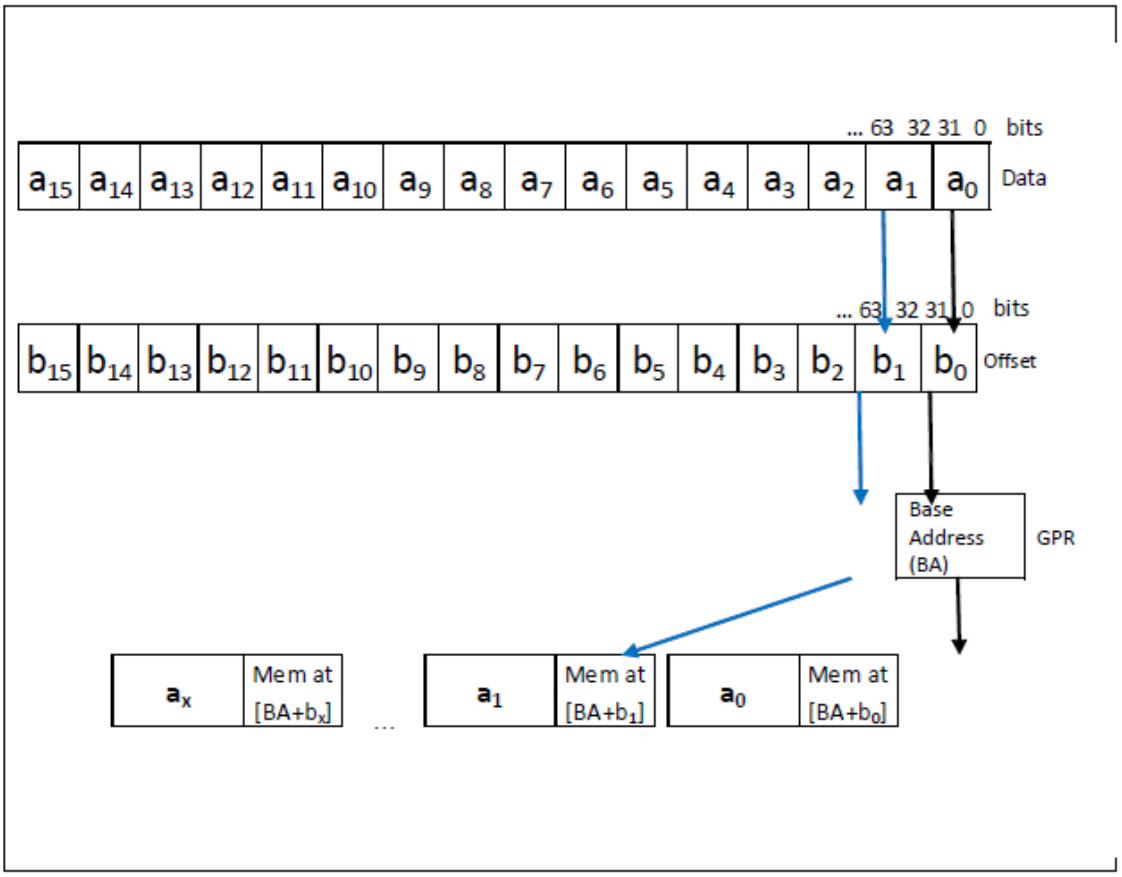


図 18-9 vscatterdpd 命令の操作

18.11.1 データ・スキャッターの例

与えられた 0 から N の範囲のユニークなインデックスの配列で、long long 整数 (64 ビット) から浮動小数点数 (32 ビット) へ値を変換しながら、対応するインデックスに従って N 個の配列をソートします。

```
for ( int i=0; i < N; i++ )
{
dst[ ind [i] ] = (float)src[i];
}
```

上記のコードの 3 つの実装例を示します。

- 代替 1 はスカラーコードです。
- 代替 2 はスキャッターのソフトウェア・シーケンスです。
- 代替 3 は、ハードウェア・スキャッターです。

注意

ハードウェア・スキャッター操作では、ベクトルの要素数と同じ数のストア操作が発行されます。連続した要素のストアにはスキャッターを使用せず、vmov 命令を使用してストアを行います。

例 18-17 スキャッター

スカラー

```
mov rax, pImage //入力
mov rcx, pOutImage //出力
mov rbx, pIndex //インデックス
mov rdx, len //レングス
xor r9, r9
mainloop:
mov r9d, [rbx+rdx-0x4]
vcvtsi2ss xmm0, xmm0, qword ptr [rax+rdx*2-0x8]
vmovss [rcx+r9*4], xmm0
sub rdx, 4
jnz mainloop
```

ベースライン 1x

ソフトウェア・シーケンス

```
shufMaskP:
 .quad·0x0000000200000001
 .quad·0x0000000400000003
 .quad·0x0000000600000005
 .quad·0x0000000800000007
  mov rax, pImage //入力
  mov rcx, pOutImage //出力
  mov rbx, pIndex //インデックス
  mov rdx, len //レングス
  mov r9, shufMaskP
  vmovaps ymm2, [r9]
mainloop:
  vmovaps zmm1, [rax + rdx*2 - 0x80] //データロード
  vcvtuqq2ps ymm0, zmm1 //float へ変換
  movsxd r9, [rbx + rdx - 0x40] //8 番目のインデックスをロード
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
  movsxd r9, [rbx + rdx - 0x3c] //7 番目のインデックスをロード
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
  movsxd r9, [rbx + rdx - 0x38] //6 番目のインデックスをロード
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
  movsxd r9, [rbx + rdx - 0x34] //5 番目のインデックスをロード
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
  movsxd r9, [rbx + rdx - 0x30] //4 番目のインデックスをロード
```

ハードウェア・スキャッター

```
  mov rax, pImage //入力
  mov rcx, pOutImage //出力
  mov rbx, pIndex //インデックス
  mov rdx, len //レングス
mainloop:
  vmovdqa32 zmm0, [rbx+rdx-0x40]
  vmovdqa32 zmm1, [rax+rdx*2-0x80]
  vcvtuqq2ps ymm1, zmm1
  vmovdqa32 zmm2, [rax+rdx*2-0x40]
  vcvtuqq2ps ymm2, zmm2
  vshuff32x4 zmm1, zmm1, zmm2, 0x44
  kxnorw k1,k1,k1
  vscatterdps [rcx+4*zmm0] {k1}, zmm1
  sub rdx, 0x40
  jnz mainloop
```

vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x2c] //3 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x28] //2 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x24] //最初のインデックスをロード vmovss [rcx + 4*r9], xmm0 vmovaps zmm1, [rax + rdx*2 - 0x40] //データロード vcvtuqq2ps ymm0, zmm1 //float へ変換 movsxd r9, [rbx + rdx - 0x20] //8 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x1c] //7 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x18] //6 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x14] //5 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x10] //4 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0xc] //3 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x8] //2 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x4] //最初のインデックスをロード vmovss [rcx + 4*r9], xmm0 sub rdx, 0x40 jnz mainloop	
スピードアップ: 1.48x	**スピードアップ: 1.53x**

18.12 静的丸めモード、すべての例外を抑制 (SAE)

インテル® AVX-512 の浮動小数点命令には、すべての例外を抑制する機能 (SAE) が導入されています。この機能は、スプリアスフラグの設定が望ましくない場合に有用です。現在のベクトル数学関数の実装では、通常スプリアスフラグの設定を許容するため、例外が有効化されたアプリケーションの実行で問題が生じることがあります。標準化に準拠するコードはスプリアスフラグの設定を許可しません。

静的丸めモードには、規格化された用途 (IEEE、OpenCL) に加え、デフォルトの丸めモード (動的に設定可能) で動作する数学ライブラリーのアプリケーションがあります。

18.13 QWORD 命令のサポート

インテル® AVX-512 では、インテル® AVX とインテル® AVX2 で導入された多くの命令の QWORD 拡張をサポートします。QWORD は次の命令でサポートされます。

18.13.1 算術命令での QUADWORD サポート

インテル® AVX-512 では、vpmaxsq、vpmaxuq、vpminsq、vpminuq、および vpmullq 命令に新たな quadword のサポートが追加されました。次の例は、2 つの 64 ビット数の加算と乗算の最大値を配列 C に格納しています。

```
const int N = miBufferWidth;
const __int64* restrict a = A;
const __int64* restrict b = B;
__int64* restrict c = Cref;

for (int i = 0; i < N; i++){
  int64 sum = a[i] + b[i];
  int64 mul = a[i] * b[i];
  c[i] = mul > sum ? mul : sum;
}
```

次のコードは、新たな機能がどのように命令数をインテル® AVX2 の 118 からインテル® AVX-512 の 30 に減らし、3.1 倍のスピードアップをもたらすかを示します。

例 18-18 インテル® AVX2 とインテル® AVX-512 での QWORD の例

インテル® AVX2 組込み関数	インテル® AVX-512 組込み関数

for (int i = 0; i < N; i+= 32){
  __m256i aa, bb, aah, bbh, mul, sum;
  #pragma unroll(8)
  for (int j = 0; j < 8; j++){
  aa = _mm256_loadu_si256((const
  __m256i*)(a+i+4*j));
  bb = _mm256_loadu_si256((const
  __m256i*)(b+i+4*j));
  sum = _mm256_add_epi64(aa, bb);
  mul = _mm256_mul_epu32(aa, bb);
  aah = _mm256_srli_epi64(aa, 32);
  bbh = _mm256_srli_epi64(bb, 32);
  aah = _mm256_mul_epu32(aah, bb);
  bbh = _mm256_mul_epu32(bbh, aa);
  aah = _mm256_add_epi32(aah, bbh);
  aah = _mm256_slli_epi64(aah, 32);
  mul = _mm256_add_epi64(mul, aah);
  aah = _mm256_cmpgt_epi64(mul, sum);
  aa=_mm256_castpd_si256 (
_mm256_blendv_pd(_mm256_castsi256_pd(sum),
_mm256_castsi256_pd(mul),_mm256_castsi256_pd
(
  aah)));
  _mm256_storeu_si256((__m256i*)(c+4*j),aa)
  ;
  }
  c += 32;
}
	for (int i = 0; i < N; i+= 32){
	  __m512i aa, bb, mul, sum;
	  #pragma unroll(4)
	  for (int j = 0; j < 4; j++){
	  aa = _mm512_loadu_si512((const
	  __m512i*)(a+i+8*j));
	  bb = _mm512_loadu_si512((const
	  __m512i*)(b+i+8*j));
	  sum = _mm512_add_epi64(aa, bb);
	  mul = _mm512_mullo_epi64(aa, bb);
	  aa = _mm512_max_epi64(sum, mul);
	  _mm512_storeu_si512((__m512i*)(c+8*j),aa
	  );
	  }
	  c += 32;
	}

ベースライン 1x	**スピードアップ: 3.1x**
インテル® AVX2 アセンブリー	**インテル® AVX-512 アセンブリー**
loop: vmovdqu32 ymm28,ymmword ptr[rax+rcx*8+0x20] inc r9d vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0x20] vmovdqu32 ymm17, ymmword ptr [r11+rcx*8] vmovdqu32 ymm19, ymmword ptr [rax+rcx*8] vmovdqu ymm13, ymmword ptr [rax+rcx*8+0x40] vmovdqu ymm11, ymmword ptr [r11+rcx*8+0x40] vpsrlq ymm25, ymm28, 0x20 vpsrlq ymm27, ymm26, 0x20 vpsrlq ymm16, ymm19, 0x20 vpsrlq ymm18, ymm17, 0x20 vpaddq ymm6, ymm28, ymm26 vpsrlq ymm10, ymm13, 0x20 vpsrlq ymm12, ymm11, 0x20 vpaddq ymm0, ymm19, ymm17 vpmuludq ymm29, ymm25, ymm26 vpmuludq ymm30, ymm27, ymm28 vpaddd ymm31, ymm29, ymm30	loop: vmovups zmm0, zmmword ptr [rax+rcx*8] inc r9d vmovups zmm5, zmmword ptr [rax+rcx*8+0x40] vmovups zmm10, zmmword ptr [rax+rcx*8+0x80] vmovups zmm15, zmmword ptr [rax+rcx*8+0xc0] vmovups zmm1, zmmword ptr [r11+rcx*8] vmovups zmm6, zmmword ptr [r11+rcx*8+0x40] vmovups zmm11, zmmword ptr [r11+rcx*8+0x80] vmovups zmm16, zmmword ptr [r11+rcx*8+0xc0] vpaddq zmm2, zmm0, zmm1 vpmullq zmm3, zmm0, zmm1 vpaddq zmm7, zmm5, zmm6 vpmullq zmm8, zmm5, zmm6 vpaddq zmm12, zmm10, zmm11 vpmullq zmm13, zmm10, zmm11 vpaddq zmm17, zmm15, zmm16 vpmullq zmm18, zmm15, zmm16 vpmaxsq zmm4, zmm2, zmm3 vpmaxsq zmm9, zmm7, zmm8

```
vmovdqu32 ymm29,ymmword ptr [r11+rcx*8+0x80]
vpsllq ymm5, ymm31, 0x20
vmovdqu32 ymm31,ymmword ptr [rax+rcx*8+0x80]
vpsrlq ymm30, ymm29, 0x20
vpmuludq ymm20, ymm16, ymm17
```

```
vpmaxsq zmm14, zmm12, zmm13
vpmaxsq zmm19, zmm17, zmm18
vmovups zmmword ptr [rsi], zmm4
vmovups zmmword ptr [rsi+0x40], zmm9
```

```
vpmuludq ymm21, ymm18, ymm19
vpmuludq ymm4, ymm28, ymm26
vpaddd ymm22, ymm20, ymm21
vpaddq ymm7, ymm4, ymm5
vpsrlq ymm28, ymm31, 0x20
vmovdqu32 ymm20, ymmword ptr
                [r11+rcx*8+0x60]
vpsllq ymm24, ymm22, 0x20
vmovdqu32 ymm22, ymmword ptr
                [rax+rcx*8+0x60]
vpsrlq ymm21, ymm20, 0x20
vpaddq ymm4, ymm22, ymm20
vpcmpgtq ymm8, ymm7, ymm6
vblendvpd ymm9, ymm6, ymm7, ymm8
vmovups ymmword ptr [rsi+0x20], ymm9
vpmuludq ymm14, ymm10, ymm11
vpmuludq ymm15, ymm12, ymm13
vpmuludq ymm8, ymm28, ymm29
vpmuludq ymm9, ymm30, ymm31
vpmuludq ymm23, ymm19, ymm17
vpaddd ymm16, ymm14, ymm15
vpsrlq ymm19, ymm22, 0x20
vpaddd ymm10, ymm8, ymm9
vpaddq ymm1, ymm23, ymm24
vpsllq ymm18, ymm16, 0x20
vmovdqu32 ymm28, ymmword ptr
             [rax+rcx*8+0xc0]
vpsllq ymm12, ymm10, 0x20
vpmuludq ymm23, ymm19, ymm20
vpmuludq ymm24, ymm21, ymm22
vpaddd ymm25, ymm23, ymm24
vmovdqu32 ymm19, ymmword ptr
             [rax+rcx*8+0xa0]
vpsllq ymm27, ymm25, 0x20
vpsrlq ymm25, ymm28, 0x20
vpsrlq ymm16, ymm19, 0x20
vpcmpgtq ymm2, ymm1, ymm0
vblendvpd ymm3, ymm0, ymm1, ymm2
vpaddq ymm0, ymm13, ymm11
vmovups ymmword ptr [rsi], ymm3
vpmuludq ymm17, ymm13, ymm11
vpmuludq ymm11, ymm31, ymm29
vpaddq ymm1, ymm17, ymm18
vpaddq ymm13, ymm31, ymm29
vpaddq ymm14, ymm11, ymm12
vmovdqu32 ymm17, ymmword ptr
              [r11+rcx*8+0xa0]
vmovdqu ymm12, ymmword ptr [r11+rcx*8+0xe0]
vpsrlq ymm18, ymm17, 0x20
```

```
vmovups zmmword ptr [rsi+0x80], zmm14
vmovups zmmword ptr [rsi+0xc0], zmm19
add rcx, 0x20
add rsi, 0x100
cmp r9d, r8d
jb loop
```

vpcmpgtq ymm2, ymm1, ymm0
vpmuludq ymm26, ymm22, ymm20
vpcmpgtq ymm15, ymm14, ymm13
vblendvpd ymm3, ymm0, ymm1, ymm2
vblendvpd ymm0, ymm13, ymm14, ymm15
vmovdqu ymm14, ymmword ptr [rax+rcx*8+0xe0]
vmovups ymmword ptr [rsi+0x40], ymm3
vmovups ymmword ptr [rsi+0x80], ymm0
vpaddq ymm5, ymm26, ymm27
vpsrlq ymm11, ymm14, 0x20
vpsrlq ymm13, ymm12, 0x20
vpaddq ymm1, ymm19, ymm17
vpaddq ymm0, ymm14, ymm12
vmovdqu32 ymm26, ymmword ptr
[r11+rcx*8+0xc0]
vpmuludq ymm20, ymm16, ymm17
add rcx, 0x20
vpmuludq ymm21, ymm18, ymm19
vpaddd ymm22, ymm20, ymm21
vpsrlq ymm27, ymm26, 0x20
vpsllq ymm24, ymm22, 0x20
vpmuludq ymm29, ymm25, ymm26
vpmuludq ymm30, ymm27, ymm28
vpmuludq ymm15, ymm11, ymm12
vpmuludq ymm16, ymm13, ymm14
vpmuludq ymm23, ymm19, ymm17
vpaddd ymm31, ymm29, ymm30
vpaddd ymm17, ymm15, ymm16
vpaddq ymm2, ymm23, ymm24
vpsllq ymm19, ymm17, 0x20
vpcmpgtq ymm6, ymm5, ymm4
vblendvpd ymm7, ymm4, ymm5, ymm6
vpsllq ymm6, ymm31, 0x20
vmovups ymmword ptr [rsi+0x60], ymm7
vpaddq ymm7, ymm28, ymm26
vpcmpgtq ymm3, ymm2, ymm1
vpmuludq ymm5, ymm28, ymm26
vpmuludq ymm18, ymm14, ymm12
vblendvpd ymm4, ymm1, ymm2, ymm3
vpaddq ymm8, ymm5, ymm6
vpaddq ymm1, ymm18, ymm19
vmovups ymmword ptr [rsi+0xa0], ymm4
vpcmpgtq ymm9, ymm8, ymm7
vpcmpgtq ymm2, ymm1, ymm0
vblendvpd ymm10, ymm7, ymm8, ymm9
vblendvpd ymm3, ymm0, ymm1, ymm2
vmovups ymmword ptr [rsi+0xc0], ymm10
vmovups ymmword ptr [rsi+0xe0], ymm3
add rsi, 0x100
cmp r9d, r8d
jb loop	

ベースライン 1x	**スピードアップ: 3.1x**

18.13.2 変換命令での QUADWORD サポート

次の表は、変換命令のクワッドワード拡張を示しています。

表 18-3 ベクトル Quadword 拡張

From / To	ベクトル SP	ベクトル DP	ベクトル int64	ベクトル uint64
ベクトル SP	-		vcvtps2qq	vcvtps2uqq
ベクトル DP		-	vcvtpd2qq	vcvtpd2qq
ベクトル int64	vcvtqq2ps	vcvtqq2pd	-	
ベクトル uint64	vcvtqq2ps	vcvtuqq2pd		-

表 18-4 スカラー Quadword 拡張

From / To	スカラー SP	スカラー DP	スカラー int64	スカラー uint64
スカラー SP	-		vcvtss2si	vcvtss2usi
スカラー DP		-	vcvtsd2si	vcvtsd2usi
スカラー int64	vcvtsi2sd	vcvtsi2sd	-	
スカラー uint64	vcvtusi2sd	vcvtusi2sd		-

18.13.3 切り捨て変換命令での QUADWORD サポート

次の表は、切り捨てモードの変換命令のクワッドワード拡張を示しています。

表 18-5 ベクトル Quadword 拡張

From / To	ベクトル int64	ベクトル uint64
ベクトル SP	vcvttps2qq	vcvttps2uqq
ベクトル DP	vcvttpd2qq	vcvttpd2qq

表 18-6 スカラー Quadword 拡張

From / To	スカラー int64	スカラー uint64
スカラー SP	vcvttss2si	vcvttss2usi
スカラー DP	vcvttsd2si	vcvttsd2usi

18.14 ベクトル長の直交性

ベクトル長拡張 (VL) をサポートするプロセッサーでは、すべてのインテル® AVX-512 命令が 128 ビット、256 ビット、そして 512 ビットのベクトルを操作できます。組込み丸めを伴う命令を除き、これら 3 つのベクトル長がインテル® AVX-512 命令でサポートされます。命令のエンコードにおいて、ベクトル長と組込み丸め制御は同じ 2 つのビットを使用してコード化されているため、組込み丸めが使用される場合ベクトル長は自動的に 512 ビット (インテル® AVX-512 における最大ベクトル長) と仮定されます。

“組込み丸め” については 18.10 節を参照してください。

18.15 超越計算サポート向けのインテル® AVX-512 命令

この節では、インテル® AVX-512 で導入された新しい超越計算向けの命令について説明します。

18.15.1 VRCP14、VRSQRT14 - 1/x、x/y、sqrt(x) 向けのソフトウェア・シーケンス

シンタックス:

```
VRCP14PD/PS dest, src
VRSQRT14PD/PS dest, src
```

18.15.1.1 アプリケーションの例

逆数、除算、平方根、逆平方根向けのソフトウェア・シーケンス。

1/x、x/y、sqrt(x) 向けのソフトウェア・シーケンスは、スループットに利点があります (レイテンシーはそれほどでもなく、精度がかなり低いわけでもありません)。通常これらは、ニュートンラフソン近似や多項式近似によって実装されます。

VRCP14 と VRSQRT14 の利点の 1 つは、これまでの RCPPS や RSQRTPS に比べ精度が改善されていることです。これは、特に倍精度では計算を短縮するのに役立ちます (倍精度では 50 - 52 ビットの近似のため 3 つのニュートンラフソン反復が必要です)。

これらの命令のもう 1 つの利点は、倍精度バージョンがサポートされることです (RCP/RSQRT 命令ではサポートされていませんでした)。これらの機能により倍精度のパフォーマンスが向上します。Skylake† Server マイクロアーキテクチャー上では、倍精度逆数と平方根ソフトウェア・シーケンスは、512 ビット・ベクトル・モードの倍精度超越引数リダクション (log、cbrt など) における VDIV や VSQRT よりもかなり高いスループットをもたらします。

log() や cbrt() (立方根) などの関数では、高価な逆数テーブル索引に代わって丸められた VRCP14PD の結果を使用できます。同様の手法を RCPPS にも使用できますが、倍精度ではそれほど効果はありません。

log() 引数リダクションの例は、18.15.3 節「VRNDSCALE - ベクトル丸めスケール」を参照してください。

18.15.2 VGETMANT、VGETEXP - ベクトル仮数とベクトル指数の取得

シンタックス:

```
VGETMANTPD/PS dest_mant, src, imm
VGETEXPPD/PS dest_exp, src
```

18.15.2.1 アプリケーションの例

対数関数。

```
log2(x) = VGETEXP(x) + log2(VGETMANT(x,8))
log(x) = VGETEXP(x)*log(2.0) + log(VGETMANT(x,8))
```

上記に見られるように、VGETMANT(x,8) がすべての有効な関数入力で [1,2) であり、不正な入力 (x < 0) が NaN であることが保証されると、log(VGETMANT(x,8)) の計算を軽減できます。

これは仮数の対数を計算するさまざまなアルゴリズムに適用できます。特定のアルゴリズムの選択は、必要とする精度、最適化の目標 (レイテンシーやスループットの最適化)、またはマイクロアーキテクチャーに依存します。いくつかのアルゴリズムでは、仮数の正規化にほかの手法を使用する可能性があります。[0.5, 1) または [0.75, 1.5); しかし、計算の基礎となる基本的な同一性は上に示されています。

X^{alpha} (alpha 定数) と除算の詳細については、18.15.5 節「VSCALEF - ベクトルスケール」を参照してください。

18.15.3 VRNDSCALE - ベクトル丸めスケール

シンタックス:
`VRNDSCALEPD/PS dest, src, imm`

18.15.3.1 アプリケーションの例

テーブル索引は超越関数の実装でよく利用されます。テーブル・インデックスは、多くの場合入力の先頭数ビットに基づいています。テーブル・インデックスに対応する浮動小数点入力値を生成するため、引数リダクション処理の一部として VRNDSCALE 命令を使用できます。以下は、1 ≤ x < 2 における log(x) の引数リダクションの実装例です。

```
y = RCP14(x);                 // y is in (0.5, 1]
y0=RNDSCALE(y, k*16);         // y0 には k 仮数ビットがあります
                              // (先頭 1 ビットを含みます)
R = x?y0 - 1;                 // |R|<2-14+2-k.
```

そして、log(x) = -log(y0) + log(1+R) となります。

log(1+R) は多項式で計算でき、log(y0) は 2k - 1 +1 要素または 2k - 1 要素のテーブル索引から求められます。追加のチェックは犠牲になります。

18.15.4 VREDUCE - ベクトルレデュース

シンタックス:
`VREDUCEPD/PS dest, src, imm`

18.15.4.1 アプリケーションの例

VREDUCE の最も有益な点は、exp2 や pow (exp2 も含まれます) などの一般的な超越操作においてレイテンシーを軽減できることです。また、atan() などほかの超越関数でも使用できます。

18.15.5 節「VSCALEF - ベクトルスケール」も参照してください。

18.15.5 VSCALEF - ベクトルスケール

シンタックス:
`VSCALEFPD/PS dest, src1, src2`

18.15.5.1 アプリケーションの例

```
exp2 (2x)
exp2(x) = VSCALEF( 2VREDUCE(x, RD_mode), x)
```

R(x) = VREDUCE(x, RD_mode) = x - floor(x) は、[0, 1) の範囲にあります。2R(x) は、多項式近似や多項式近似とテーブル索引によって求められます。VSCALEF はオーバーフローとアンダーフローを適切に処理します。また、exp() の特殊ケース (入力は無限大など) も処理できるように定義されているため、ベクトル実装において特殊な処理パスを必要としません。VSCALEF を使用しないと、入力が非常に大きな場合に異なる処理パスが必要になります。

もはや明示的に指数を操作する必要はなく、VSCALEF はスループットを改善します。

```
Exp(x)
Exp(x) = VSCALEF( 2R(x), x*(1/log(2.0)),
```

ここで、

```
R(x) = x - log(2.0)*floor(x*(1/log(2.0));
```

log(2.0) 近似 (ネイティブの浮動小数点形式より長い) を使用することで、R(x) を正確に計算できます。

exp2() のように、VSCALEF を使用する利点は、スループットを改善し二次的な分岐を排除できることです。

x^{alpha} (alpha 定数)

例えば、alpha=1/3 (立方根 cbrt) の場合、この計算の基本的なリダクションは次のようになります。

```
x^alpha = VSCALEF( (VGETMANT(x, imm))alpha?2VREDUCE (VGETEXP(x)*alpha, RD_mode),
VGETEXP(x)*alpha)
```

即値 (imm) の選択は alpha 定数の値を基にしています

除算:

```
a/b = VSCALEF(VGETMANT(a,0)/VGETMANT(b,0), VGETEXP(a)-VGETEXP(b))
```

このリダクションは分岐なしの除算実装を可能にし、オーバーフロー、アンダーフロー、および特殊入力 (ゼロ、無限大、デノーマル) を処理できます。

すべての非 NaN 入力では、|VGETMANT(x,0)| は [1,2) の範囲にあります。

VGETMANT(a,0)/VGETMANT(b,0) は、必要な精度で計算できます。

インテル® AVX-512 で利用可能な「すべての例外を抑制 (SAE)」する機能は、スプリアスフラグが設定されないことを確実にします。フラグは、計算の結果として正しく設定できます。ただし、ゼロ除算は追加の処理が必要となるため除外されます。

高い精度や IEEE の順守が必要である場合、ハードウェア命令が一般により高いパフォーマンス (特にレイテンシーに関して) をもたらします。

18.15.6 VFPCLASS - ベクトル浮動小数点クラス

シンタックス:

```
VFPCLASSPD/PS dest_mask, src, imm
```

18.15.6.1 アプリケーションの例

VFPCLASS は、特殊なケースを検出して直接処理を行うパスに導いたり、代替としてメインパスでマスク操作に使用されます。以下に、逆数シーケンスと平方根シーケンスの 2 つの例を示します。

1/x 計算の引数リダクションは、e=1-x*RCP14(x) です。x が ±0 または ±Inf である場合、この式は NaN と評価され、RCP14 は特殊ケースに正しい結果を返します。VFPCLASS は、x が ±0 または ±Inf では mask=1 を設

定し、それ以外の x には mask=0 を設定することを可能にします。そして、このマスクを使用して RCP14 の出力 (特殊ケースの結果) や、RCP14 から始まる相反精密計算の結果 (一般的な入力) を選択できます。

同様に、RSQRT14 を基にする平方根計算では、=±0 または x=+Inf のマスクを作成するため VFPCLASS を使用できます。

Pow(x,y) 関数:

pow(x,y)=2y*log2(x) のメインパスは、x?0、x=Inf/NaN、または y=Inf/NaN を処理しません。最初の VFPCLASS 操作は、x?0 や x=Inf/NaN のケースに special_x_mask=1 を設定するために使用できます。2 番目の VFPCLASS 操作は、x?0 や x=Inf/NaN のケースに special_y_mask=1 を設定するために使用します。どちらかのマスクが設定されると、2 番目のパスへ分岐します。

18.15.7 VPERM, VPERMI2, VPERMT2 - 小規模テーブル索引の実装

18.15.7.1 アプリケーションの例

数学ライブラリー関数は、頻繁にテーブル索引を使用して実装されます。ベクトルモードでは、大きなテーブル索引はベクトルギャザーが使用されます。小さなテーブル索引は、VPERM* 命令を使用することで劇的に高速化できます。

VPERM* 命令によるテーブル索引を使用することで劇的な高速化を達成できた超越関数には、単精度と倍精度の exp()、log()、pow() などがあります。

18.16 競合検出

インテル® AVX-512 競合検出命令は、インテル® AVX-512 基本命令とともに、ベクトルの依存関係の可能性があるループを効率良くベクトル化することを可能にします。VPCONFLICT は、単一ベクトルレジスター内で要素の水平比較を行います。VPCONFLICT は、ベクトルレジスターの各要素と直前の要素をすべて比較して、その比較結果を出力します。水平比較はその他の用途にも利用できます。

ほかの競合検出命令は、比較結果の効率良い操作を考慮しています。VPLZCNT 命令により、ベクトル要素と一致する値を結合する際に使用されるレジスター内パーミュート操作の制御を生成できます。

18.16.1 競合検出とベクトル化

インテル® AVX-512CD 命令は、配列ポインター (*ptr[i] += val[i]) や A[B[i]] += val[i] などの間接アドレス配列など) を使用した読み込みと書き込みを伴うループを効率良くベクトル化することを可能にします。

次のようなヒストグラムの計算を考えてみます。

```
for(int i = 0; i < num_inputs; i++)
{
  histogram[input[i] & (num_bins - 1)]++;
}
```

input[0] = input[1] = 3 である場合、SIMD 命令を使用して histogram[input[0]] と histogram[input[1]] をレジスターへ (ギャザーを使用して) 読み込み、インクリメントを行って書き戻す (スキャッターを使用して) と、不正な結果がもたらされます。この操作を行うと histgram[3] の値は 1 になります。本来は 2 でなければなりません。

この問題はインデックスが複製されることで生じます。反復 0 の histogram への書き込みと反復 1 の histogram からの読み込みに依存関係がありますが、読み込みは以前の書き込みの値を取得する必要があります。

この問題を特定するため、VPCONFLICT 命令を使用してインデックス (またはポインター値) の複製を検出します。この命令はレジスター内のすべての以前の要素とベクトルレジスターの各要素を比較します。

例:

```
vpconflictd zmm0, zmm1
```

以下の図は、VPCONFLICTD 命令の実行例を示しています。入力 ZMM1 は 16 個の整数要素を含みます (青色のボックス)。図の上部にある ZMM1 は、左に示されるように転移しています。白いボックスは、ハードウェアが ZMM1 の異なる要素を比較した結果を示しています (0 = 等しくない、1 = 等しい)。命令からのそれぞれの比較出力は単一のビットです。実行されなかった比較 (灰色のボックス) は、単一ビット '0' を生成します。最終的に結果は ZMM0 に生成されます (図の下にある黄色のボックス)。各要素は上記のビットに相当する 10 進数で表されています。

VPCONFLICT は異なる方法でループをベクトル化するのに役立ちます。

最も簡単なものは、特定の SIMD レジスターのインデックスの複製をチェックすることです。複製されていなければ、すべての要素を安全に SIMD 命令で計算できます。競合が検出された場合、そのグループの要素をスカラーループで実行します。

インデックスの複製がそれほど多くない場合、ループのスカラーバージョンへの分岐はうまく動作します。しかし、ベクトル化されたループ反復が複製できるほど十分に大きければ、可能な限り並列性を高めるためできるだけ多くの SIMD 命令を使用する方がいいでしょう。

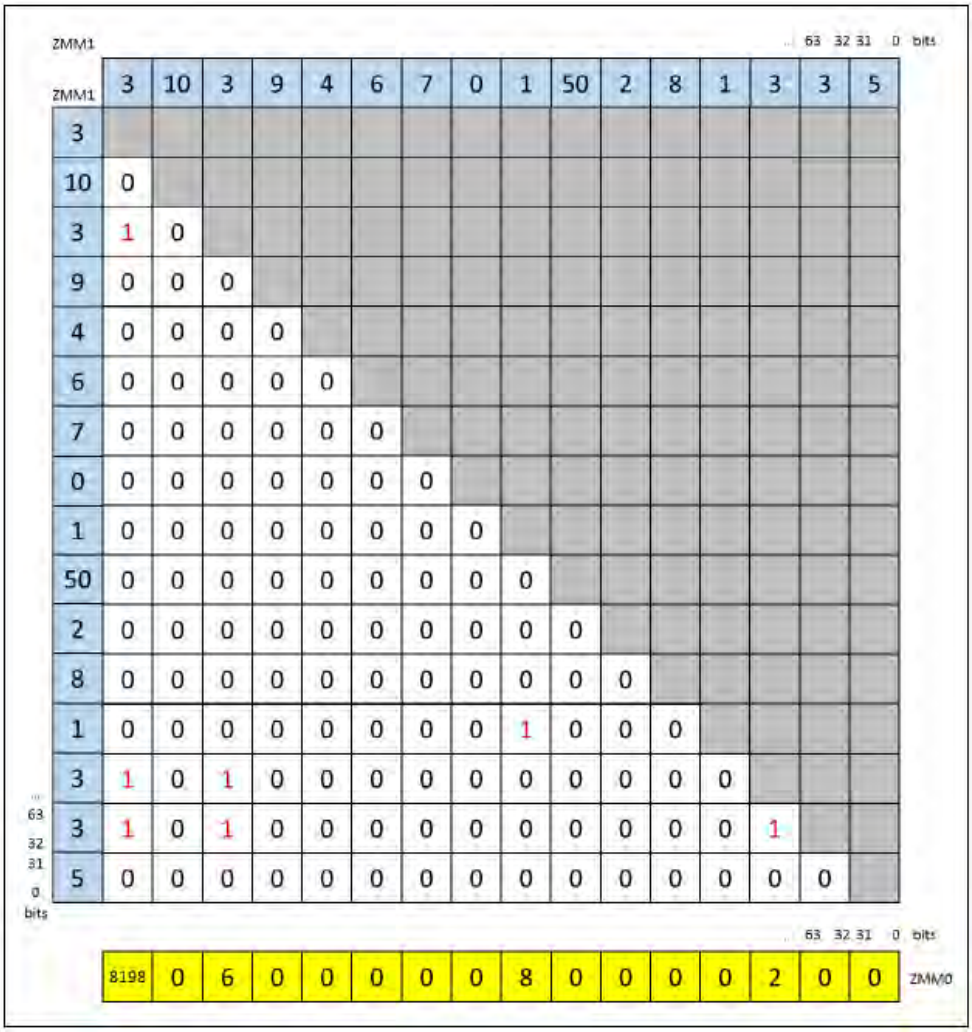


図 18-10 VPCONFLICTD 命令の実行

ヒストグラムの例のようにメモリー・ロケーションを更新するループでは、データがレジスター内に存在する間に個別のインデックスを使用して更新をマージし、それぞれのメモリー・ロケーションに一度だけ書き込みを行うことで、ストア・ロード・フォワードを最小限にできます。さらに、マージは並列に実行できます。

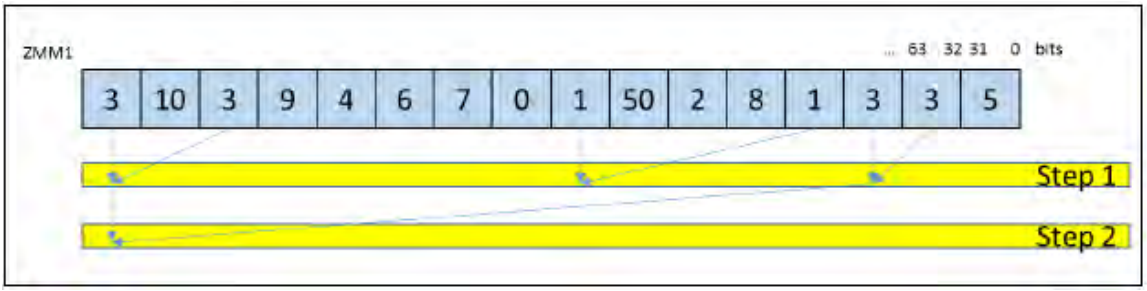


図 18-11 VPCONFLICTD マージ処理

図 18-11 は、インデックスのセットの例をマージする様子を示しています。図ではインデックスのみが示されていますが、実際には値をマージします。ほとんどのインデックスはユニークであり、マージする必要はありません。ステップ 1 では 3 組のインデックスを結合します: 2 つの '3' のペアと 1 つの '1' のペア。ステップ 2 は、ステップ 1 の '3' 向けの中間結果を結合します。これにより、それぞれのインデックスは 1 つの値のみを指します。ステップ 2 のみで、3 つのインデックス値を持つ 4 つの要素がマージされていることに注目してください。これは、ツリーのリダクションを行ったことで、結果のペアまたは各ステップの中間結果をマージしたことによるものです。

上記に示すマージ処理 (結合やリダクション) は、一連のパーミュート操作によって行われます。

最初のパーミュート制御は、VPLZCNT + VPSUB シーケンスで生成されます。VPLZCNT は、それぞれのベクトル要素に先行するゼロ数を供給します (例えば、最上位ビットに隣接する連続したゼロなど)。各ベクトル要素のビット数から VPLZCNT の結果を引くことで、VPCONFLICT 命令の結果の再上位 1 ビットの位置を示し、または -1 であれば要素間に競合なないことを示します。上記の例によるシーケンスの結果、次のパーミュート制御が得られます。

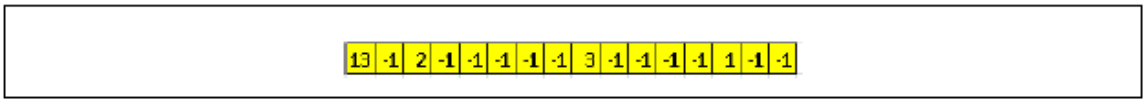


図 18-12 VPCONFLICTD パーミュート制御

一致するインデックスをマージするパーミュート・ループと、次のパーミュート・インデックスのセットを生成する処理は、パーミュート制御の値がすべて '-1' と等しくなるまで繰り返されます。

以下のアセンブリー・コードは、スカラーバージョンのヒストグラム・ループとベクトルバージョンのツリー・リダクションを示しています。ループはわずかな計算しか含んでいないためスピードアップは控え目です。SIMD の利点は論理積和操作とインクリメントをベクトル化することで得られています。ループがさらにベクトル化可能な計算を含んでいれば、SIMD のスピードアップはより高くなります。

例 18-19 スキャッター実装の代替

スカラーコード (2 回アンロール)	インテル® AVX-512 コード

mov r9d, bins_minus_1
mov ebx, num_inputs
mov r10, pInput
mov r15, pHistogram
xor rax, rax
histogram_loop:
lea ecx, [rax + rax]
inc eax
movsxd rcx, ecx
mov esi, [r10+rcx*4]
and esi, r9d
mov r8d, [r10+rcx*4+4]
movsxd rsi, esi
and r8d, r9d
movsxd r8, r8d
inc dword ptr [r15+rsi*4]
inc dword ptr [r15+r8*4]
cmp eax, ebx
jb histogram_loop
	vmovaps zmm4, all_1 // {1, 1, …, 1}
	vmovaps zmm5, all_negative_1
	vmovaps zmm6, all_31
	vmovaps zmm7, all_bins_minus_1
	mov ebx, num_inputs
	mov r10, pInput
	mov r15, pHistogram
	xor rcx, rcx
	histogram_loop:
	vpandd zmm3, zmm7, [r10+rcx*4]
	vpconflictd zmm0, zmm3
	kxnorw k1, k1, k1
	vmovaps zmm2, zmm4
	vpxord zmm1, zmm1, zmm1
	vpgatherdd zmm1{k1}, [r15+zmm3*4]
	vptestmd k1, zmm0, zmm0
	kortestw k1, k1
	je update
	
	vplzcntd zmm0, zmm0
	vpsubd zmm0, zmm6, zmm0
	
	conflict_loop:
	vpermd zmm8{k1}{z}, zmm0, zmm2
	vpermd zmm0{k1}, zmm0, zmm0
	vpaddd zmm2{k1}, zmm2, zmm8
	vpcmpned k1, zmm5, zmm0
	kortestw k1, k1
	jne conflict_loop
	
	update:
	vpaddd zmm0, zmm2, zmm1
	kxnorw k1, k1, k1
	add rcx, 16
	vpscatterdd [r15+zmm3*4]{k1}, zmm0
	cmp ecx, ebx
	jb histogram_loop

スカラーのベースライン 1x	**スピードアップ: 1.11x (ランダム入力); 1.34x (同じ入力値)**

競合するループの結果には (すべてのマージが完了した結果のベクトル。上記のシーケンスでは ZMM2)、部分和のすべてが含まれることに注意してください。つまり、各要素の結果には、同じインデックス値を持つ以前の要素がすべてマージされているものが含まれます。例 18-19 を実行した結果、ZMM2 には次の値が格納されます。

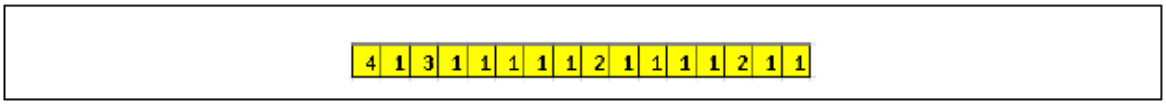


図 18-13 VPCONFLICTD ZMM2 の結果

18.16.2 VPCONFLICT による疎ドット積

疎なベクトルが 1 組の配列として格納されることを想定します。1 つには非ゼロの値が含まれ、もう一方はこれらの値の元の場所のベクトルを含んでいます。インデックスは増分でソートされていることに注意してください。

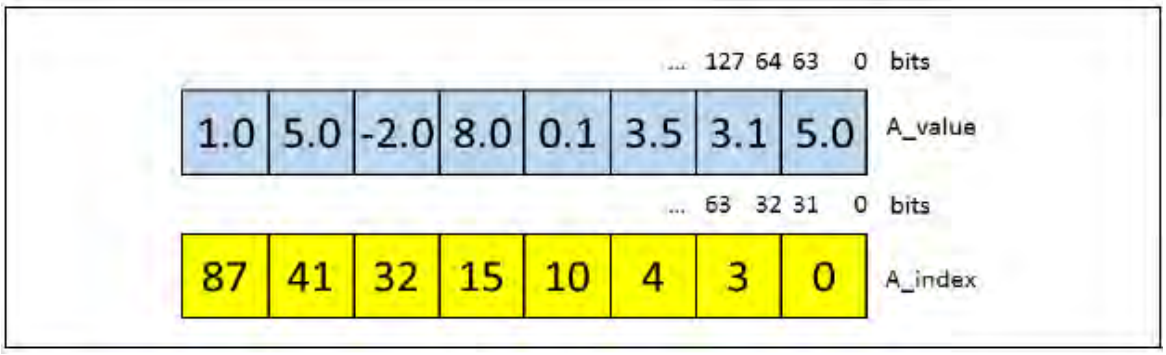


図 18-14 疎ベクトルの例

2 つの疎ベクトルのドット積を効率良く実行するには、インデックスが一致する要素を検出する必要があります。そして、それらに対し乗算と累加を行います。これを行うスカラー手法では、2 つのインデックス配列の先頭からそれらのインデックスを比較して、一致すれば乗算と累加を行い、次にインデックスを進めます。一致しなければ少ない方のインデックスを進めます。

```
A_offset = 0; B_offset = 0; sum = 0;
while ((A_offset < A_length) && (B_offset < B_length))
{
  if (A_index[A_offset] == B_index[B_offset]) // 一致
  {
    sum += A_value[A_offset] * B_value[B_offset];
    A_offset++;
    B_offset++;
  }
  else if (A_index[A_offset] < B_index[B_offset])
  {
    A_offset++;
  }
  else
  {
    B_offset++;
  }
}
```

インテル® AVX-512CD 命令は、このループを効率良くベクトル化します。各ベクトルの 1 つのインデックスを比較する代わりに、8 つを同時に比較します。最初に、各ベクトル向けの 8 つのインデックスを単一のベクトルレジスターに結合します。そして、VPCONFLICT 命令でインデックスを比較します。その出力をベクトル A の一致する要素のマスクを生成するため使用し、また B の値を対応する場所へ移動するパーミュート制御を作成します。そして、ベクトル FMA 命令を実行します。

次のアセンブリー・コードでは、スカラーとベクトルバージョンの単一比較と FMA を行っています。簡潔にするため、オフセットの更新とループは省略されています。

例 18-20 インテル® AVX-512CD を使用してスカラーとベクトルを更新

スカラーコード	インテル® AVX-512 コード

mov rdx, A_index
mov rcx, A_offset
mov rax, A_value
mov r12, B_index
mov r13, B_offset
mov rbx, B_value
mov r10d, [rdx+rcx*4]
mov r11d, [r12+r13*4]
cmp r10d, r11d
jne skip_fma
// 一致したら fma を実行
movsd xmm5, [rbx+r13*8]
mulsd xmm5, [rax+rcx*8]
addsd xmm4, xmm5
skip_fma:
	mov rdx, A_index
	mov rcx, A_offset
	mov rax, A_value
	mov r12, B_index
	mov r13, B_offset
	mov rbx, B_value
	mov r14, all_31s // {31, 31, …} の配列
	vmovaps zmm2, [r14]
	mov r15, upconvert_control // {0, 7, 0, 6, 0, 5,0, 4, 0, 3, 0, 2, 0, 1, 0, 0} の配列
	vmovaps zmm1, [r15]
	vpternlogd zmm0, zmm0, zmm0, 255
	movl esi, 21845
	kmovw k1, esi // 基数ビットを設定
	
	// A のインデックスを 8 つリード
	vmovdqu ymm5, [rdx+rcx*4]
	// B のインデックスを 8 つリードして、
	// zmm6 の上位へ配置
	vinserti64x4 zmm6, zmm5, [r12+r13*4], zmm5
	vpconflictd zmm7, zmm6
	// A と B の比較を抽出
	vextracti64x4 ymm8, zmm7, 1
	// 比較結果を permute 制御へ変換
	vplzcntd zmm9, zmm8
	vptestmd k2, zmm8, zmm0
	vpsubd zmm10, zmm2, zmm9
	// データが 64 ビットであるため、パーミュート制御を
	// 32 ビットから 64 ビットへアップコンバート
	vpermd zmm11{k1}, zmm1, zmm10
	// A の値を対応する B の値に移動し、
	// FMA を実行
	vpermpd zmm12{k2}{z}, zmm11, [rax+rcx*8]
	vfmadd231pd zmm4, zmm12, [rbx+r13*8]

ベースライン 1x	**スピードアップ: 4.4x**

18.17 インテル® AVX-512 ベクトルバイト操作命令 (VBMI)

インテル® AVX-512 VBMI 命令は、ビット操作を高速化するため設計された 512 ビット命令セットです。この節では新しい命令について説明し、簡単な用法を示します。命令の完全な定義については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』を参照してください。VBMI1 および VBMI2 をサポートするプロセッサーは、CPUID 機能フラグ CPUID:(EAX=07H, ECX=0):ECX[bit 01] = 1 と CPUID:(EAX=07H, ECX=0):ECX[bit 06] = 1 で示されます。

18.17.1 レーン間のパケットバイト要素の置換 (VPERMB)

VPERMB 命令は、単一ソースの任意の位置から任意の位置へのバイト置換命令です。次の図は、VPERMB 命令の操作例を示しています。

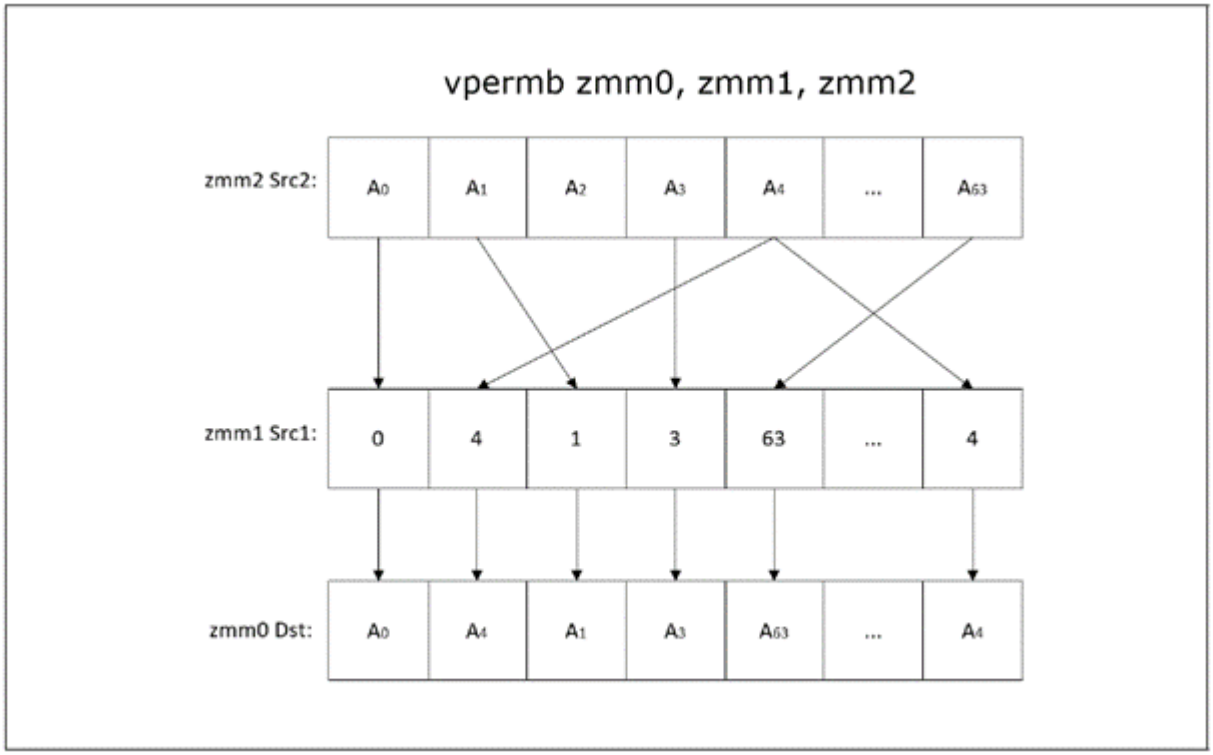


図 18-15 VPERMB 命令の操作

VPERMB の操作:

```
// vpermb zmm Dst {k1}, zmm Src1, zmm Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;
for(int i=0;i<64;i++){
    if(k1[i]){
        Dst[i]= Src2[Src1[i]];
    }else{
        Dst[i]= zero_masking? 0 : Dst[i];
    }
}
```

次の例は、64 バイトのルックアップ・テーブルの実装例を示しています。

スカラーコード:

```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char*
dictionary_bytes, int numOfElements){
   for(int i = 0; i < numOfElements; i++) {
       out_bytes[i] = dictionary_bytes[in_bytes[i] & 63];
   }
}
```

例 18-21 VPERMB 実装での改善

代替 1: VBMI なしのベクトル実装	代替 2: VPERMB 実装

mov rsi, dictionary_bytes
mov r11, in_bytes
mov rax, out_bytes
mov r9d, numOfElements
xor r8, r8
vpmovzxbw zmm3, [rsi]
vpmovzxbw zmm4, [rsi+32]
 	
loop:
vpmovzxbw zmm1, [r11+r8*1]
vpmovzxbw zmm2, [r11+r8*1+32]
vpermi2w zmm1, zmm3, zmm4
vpermi2w zmm2, zmm3, zmm4
vpmovwb [rax+r8*1], zmm1
vpmovwb [rax+r8*1+32], zmm2
add r8, 64
cmp r8, r9
jl loop
	mov rsi, dictionary_bytes
	mov r11, in_bytes
	mov rax, out_bytes
	mov r9d, numOfElements
	xor r8, r8
	vmovdqu32 zmm2, [rsi]
	
	loop:
	vmovdqu32 zmm1, [r11+r8*1]
	vpermb zmm1, zmm1, zmm2
	vmovdqu32 [rax+r8*1], zmm1
	add r8, 64
	cmp r8, r9
	jl loop

ベースライン: 1x	**スピードアップ: 6.5x**

18.17.2 レーン間の 2 つのソースバイトの置換 (VPERMI2B、VPERMT2B)

VPERMI2B と VPERMT2B 命令は、2 ソースバイトの置換命令です。デスティネーションは操作ソースでもあります。VPERMI2B ではデスティネーションは操作インデックスであり、VPERMT2B ではデスティネーションはデータソースの 1 つです。

次の図は VPERMI2B 命令の操作例を示します。

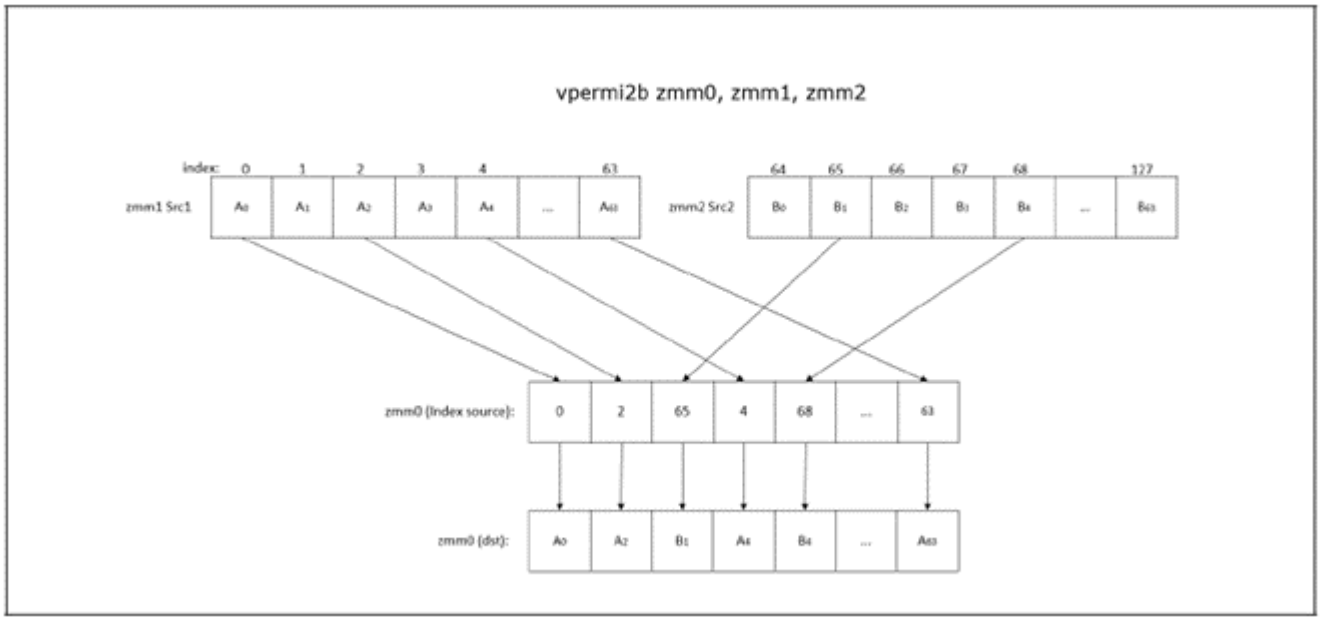


図 18-16 VPERMI2B 命令の操作

VPERMI2B 操作:

```
/// vpermi2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;
for(int i=0;i<64;i++){
   if(k1[i]){
      Dst[i]= Dst [i]>63 ? Src1[Dst [i] & 63] : Src2[Dst [i] & 63] ;
   }else{
      Dst[i]= zero_masking? 0 : Dst[i];
   }
}
```

次の図は VPERMT2B 命令の操作例を示します。

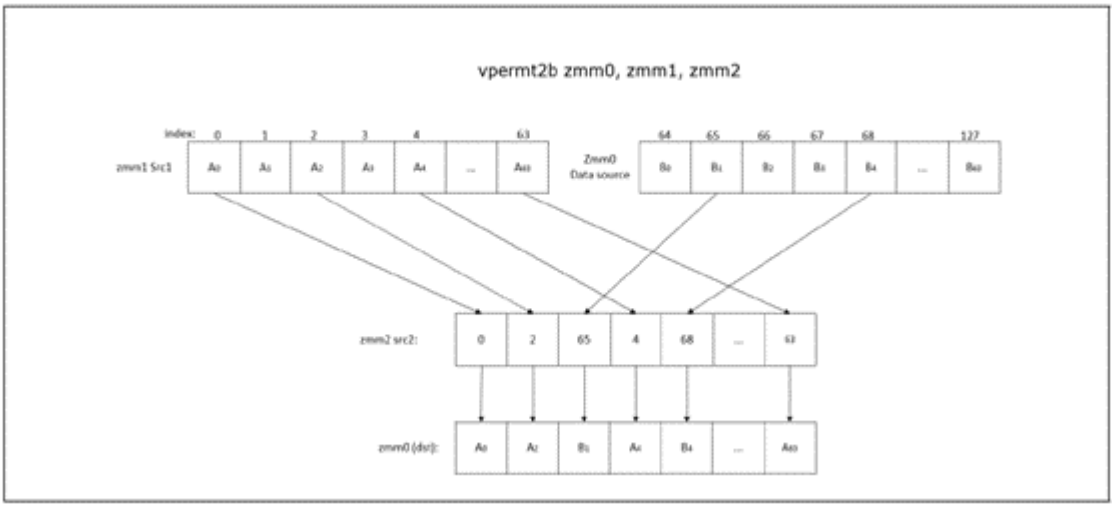


図 18-17 VPERMT2B 命令の操作

VPERMT2B の操作:

```
// vpermt2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, * Src2;
data2= copy(Dst);
for(int i=0;i<64;i++){
   if(k1[i]){
      Dst[i]= Src2[i]>63 ? Src1[Src2 [i] & 63] : Dst[Src2[i] & 63] ;
   }else{
      Dst[i]= zero_masking? 0 : Dst[i];
   }
}
```

次の例は 128 バイトのルックアップ・テーブルの実装を示します。

C コード:

```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char*
dictionary_bytes, int numOfElements){
   for(int i = 0; i < numOfElements; i++) {
      out_bytes[i] = dictionary_bytes[in_bytes[i] & 127];
   }
}
```

例 18-22 VPERMT2B 実装による改善

代替 1: VBMI なしのベクトル実装	**代替 2: VPERMI2B の実装**

// 関数に送信されたデータを取得
mov rsi, dictionary_bytes
mov r11, in_bytes
mov rax, out_bytes
mov r9d, numOfElements
xor r8, r8
// 辞書を再構成
vpmovzxbw zmm10, [rsi]
vpmovzxbw zmm15, [rsi+64]
vpsllw zmm15, zmm15, 8
vpord zmm10, zmm15, zmm10
vpmovzxbw zmm11, [rsi+32]
vpmovzxbw zmm15, [rsi+96]
vpsllw zmm15, zmm15, 8
vpord zmm11, zmm15, zmm11
// 定数を初期化
mov r10, 0x00400040
vpbroadcastw zmm12, r10d
mov r10, 0
vpbroadcastd zmm13, r10d
mov r10, 0x00ff00ff
vpbroadcastd zmm14, r10d
// 反復を開始
loop:
vpmovzxbw zmm1, [r11+r8*1]
vpandd zmm2, zmm1, zmm12
vpcmpw k1, zmm2, zmm13, 4
vpermi2w zmm1, zmm10, zmm11
vpsrlw zmm1{k1}, zmm1, 8
vpandd zmm1, zmm1, zmm14
vpmovwb [rax+r8*1], zmm1
add r8, 32
cmp r8, r9
jl loop
	mov rsi, dictionary_bytes
	mov r11, in_bytes
	mov rax, out_bytes
	mov r9d, numOfElements
	xor r8, r8
	vmovdqu32 zmm2, [rsi]
	vmovdqu32 zmm3, [rsi+64]
	loop:
	vmovdqu32 zmm1, [r11+r8*1]
	vpermi2b zmm1, zmm2, zmm3
	vmovdqu32 [rax+r8*1], zmm1
	add r8, 64
	cmp r8, r9
	jl loop

ベースライン: 1x	**スピードアップ: 5.3x**

18.17.3 クワッドワードのソースからパックされたアライメントされていないバイトを選択 (VPMULTISHIFTQB)

VPMULTISHIFTQB 命令は、2 番目のソースオペランドの各入力 qword 要素から 8 つのアライメントされていないバイトを選択し、デスティネーション・オペランドの各 qword 要素に対して 8 つの組み立てられたバイトを書き込みます。

次の図は VPMULTISHIFTQB 命令操作の例を示します。

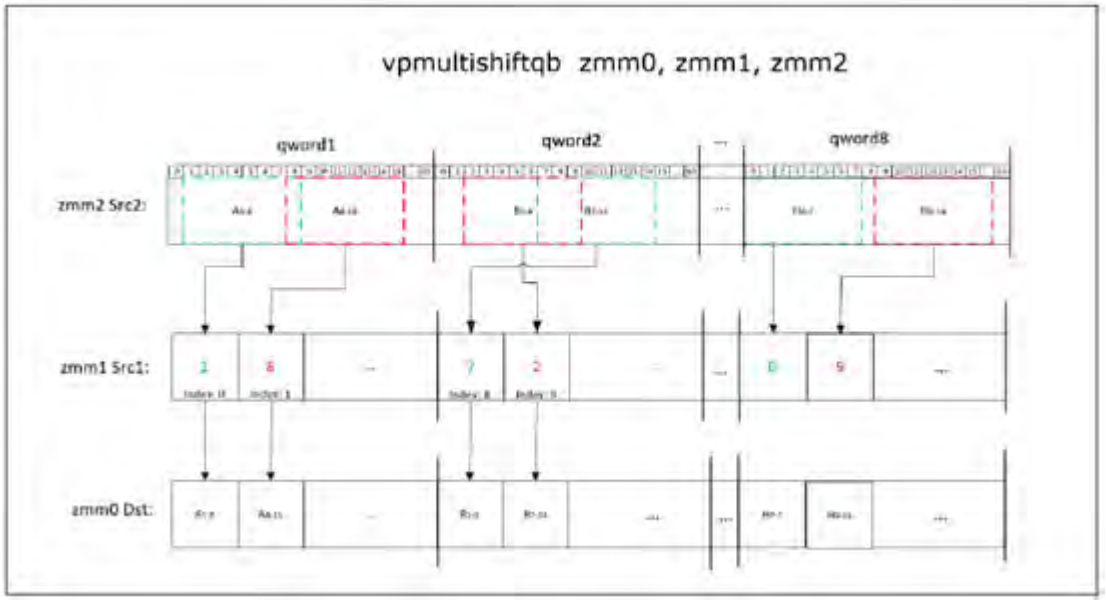


図 18-18 VPMULTISHIFTQB 命令の操作

VPMULTISHIFTQB 操作:

```
// vpmultishiftqb Dst{k1},Src1,Src2
bool zero_masking=false;
unsigned char *Dst, * Src1;
unsigned __int64 *Src2;
bit * k1;
for(int i=0;i<8;i++){
   for(int j=0;j<8;j++){
      if(k1[i*8 +j]){
         Dst[i*8 +j]= (src2[i]>> Src1[i*8 +j]) &0xFF ;
      }else{
         Dst[i*8 +j]= zero_masking? 0 : Dst[i*8 +j];
      }
   }
}
```

次の例では、5 ビットの符号なし整数配列を 1 バイトの符号なし整数配列に変換します。

C コード:

```
void decompress (unsigned char* compressedData, unsigned char* decompressedData,
int numOfElements){
   for(int i = 0; i < numOfElements; i += 8){
      unsigned __int64 * data = (unsigned __int64 * )compressedData;
      decompressedData[i+0] = * data & 0x1f;
      decompressedData[i+1] = (*data >> 5 ) & 0x1f;
      decompressedData[i+2] = (*data >> 10 ) & 0x1f;
      decompressedData[i+3] = (*data >> 15 ) & 0x1f;
      decompressedData[i+4] = (*data >> 20 ) & 0x1f;
      decompressedData[i+5] = (*data >> 25 ) & 0x1f;
      decompressedData[i+6] = (*data >> 30 ) & 0x1f;
      decompressedData[i+7] = (*data >> 35 ) & 0x1f;
      compressedData += 5;
   }
}
```

例 18-23 VPMULTISHIFTQB 実装による改善

代替 1:; VBMI なしのベクトル実装	**代替 2: VPMULTISHIFTQB の実装**

mov rdx, compressedData
mov r9, decompressedData
mov eax, numOfElements
shr eax,3
xor rsi, rsi
loop:
mov rcx, qword ptr [rdx]
mov r10, rcx
and r10, 0x1f
mov r11, rcx
mov byte ptr [r9+rsi*8], r10b
mov r10, rcx
shr r10, 0xa
add rdx, 0x5
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x2], r10b
mov r10, rcx
shr r10, 0xf
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x3], r10b
mov r10, rcx
shr r10, 0x14
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x4], r10b
mov r10, rcx
shr r10, 0x19
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x5], r10b
mov r10, rcx
shr r11, 0x5
shr r10, 0x1e
and r11, 0x1f
	// 定数:
	__declspec (align(64)) const unsigned
	__int8
	permute_ctrl[64] = {
	0, 1, 2, 3, 4, 0, 0, 0
	5, 6, 7, 8, 9, 0, 0, 0
	10, 11, 12, 13, 14, 0, 0, 0
	15, 16, 17, 18, 19, 0, 0, 0
	20, 21, 22, 23, 24, 0, 0, 0
	25, 26, 27, 28, 29, 0, 0, 0
	30, 31, 32, 33, 34, 0, 0, 0
	35, 36, 37, 38, 39, 0, 0, 0
	};
	__declspec (align(64)) const unsigned
	__int8
	multishift_ctrl[64] = {
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	};
	// asm:
	mov rsi, compressedData
	mov rdi, decompressedData
	mov r8d, numOfElements
	lea r8, [rdi+r8]
	mov r9, 0x1F1F1F1F
	vpbroadcastd zmm12, r9d

shr rcx, 0x23
and r10, 0x1f
and rcx, 0x1f
mov byte ptr [r9+rsi*8+0x1], r11b
mov byte ptr [r9+rsi*8+0x6], r10b
mov byte ptr [r9+rsi*8+0x7], cl
inc rsi
cmp rsi, rax
jb loop
	vmovdqu32 zmm10, permute_ctrl
	vmovdqu32 zmm11, multishift_ctrl
	loop:
	vmovdqu32 zmm1, [rsi]
	vpermb zmm2, zmm10, zmm1
	vpmultishiftqb zmm2, zmm11, zmm2
	vpandq zmm2, zmm12, zmm2
	vmovdqu32 [rdi], zmm2
	add rdi, 64
	add rsi, 40
	cmp rdi, r8
	jl loop

ベースライン: 1x	**スピードアップ: 26x**

18.18 FMA のレイテンシー

512 ビット・レジスターのポート方式で実行される場合、ポート 0 の FMA は 4 サイクルのレイテンシーを、ポート 5 の FMA は 6 サイクルのレイテンシーが科せられます。バイパスには、-2 (高速バイパス) から +1 サイクルの遅延がかかります。そのため、Skylake† マイクロアーキテクチャーの FMA ユニットで実行される命令のレイテンシーは、4-7 サイクルです。

命令は以下のグループに分類されます。

- グループ A の命令: vadd*; vfmadd*; vfnmsub*; vfnmadd*; vfnmsub*; vmax*; vmin*; vmul*; vscalef*; vsub*; vcvt*; vgetexp*; vfixupimm*; vrange*; vgetmant*; vreduce*; vcmp*; vcomi*; vdpp*; vhadd*; vhsub*; vrndscale*; vround*
- グループ B の命令: vpmaddubsw; vpmaddwd; vpmuldq; vpmulhrsw; vpmulhuw; vpmulhw; vpmullw; vpmuludq

すべての命令のソースが FMA ユニットから供給される場合、FMA ユニットは高速バイパスをサポートします。グループ A はポート 0 とポート 5 で 4 サイクルのレイテンシー、グループ B はポート 0 とポート 5 で 5 サイクルのレイテンシーを持ちます。

次の図は、すべての命令のソースが FMA ユニットから供給される高速バイパスの様子を示しています。

図 18-19 すべてのソースが FMA ユニットから供給される高速バイパス

青のボックスは計算サイクルを示します。肌色のボックスは、ポート 5 の FMA ユニット向けのデータ転送を表しています。

高速バイパスを利用せずに、すべてのソースが FMA ユニットから供給されない場合、グループ A の命令はポート 0 で 4 サイクル、ポート 5 で 6 サイクルのレイテンシーを持ちます。グループ B の命令には追加サイクルが科せられ、ポート 0 で 5 サイクル、ポート 5 で 7 サイクルのレイテンシーとなります。

表 18-7 に各種オプションにおける FMA ユニットのレイテンシーをまとめています。

表 18-7 FMA ユニットのレイテンシー

命令グループ	**高速バイパス (FMA データ再利用)**		**高速バイパスなし (FMA データの再利用なし)**	
	ポート 0	**ポート 5**	**ポート 0**	**ポート 5**
グループ A	4	4	4	6
グループ B	5	5	5	7

18.19 インテル® AVX 拡張またはインテル® AVX-512 拡張命令とインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) の混在

プロセッサー状態に影響する 2 つの命令グループがあります。

- グループ A: ベクトルレジスター 0 から 15 のビット 128 - 511 をゼロに設定するか、それらを全く変更しない命令タイプ。
  - インテル® SSE 命令
  - 128 ビット・インテル® AVX 命令、128 ビット・インテル® AVX-512 命令
  - 256 ビット (ymm16-ymm31) インテル® AVX-512 命令
  - 512 ビット (zmm16-zmm31) インテル® AVX-512 命令
  - k0 - k7 マスクレジスターへ書き込むインテル® AVX-512 命令
  - GPR 命令

- グループ B: ベクトルレジスター 0 から 15 のビット 128 - 511 を変更する命令タイプ。
  - 256 ビット (ymm0-ymm15) インテル® AVX 命令、インテル® AVX-512 命令
  - 512 ビット (zmm0-zmm15) インテル® AVX-512 命令

図 18-20 は、Skylake† Server マイクロアーキテクチャーにおける、インテル® AVX 命令またはインテル® AVX-512 命令とインテル® SSE 命令を混在させた際の遷移モデルを示しています。

実装は、Skylake† クライアントと同じですが、上位がダーティーなステート (Dirty Upper Staue) (2) でインテル® SSE 命令を実行すると、デスティネーション・レジスターの 128 - 511 ビットを保持する必要があり、命令はデスティネーション・レジスターと blend 命令のビット 128 - 511 との間に追加の依存性を持ちます。

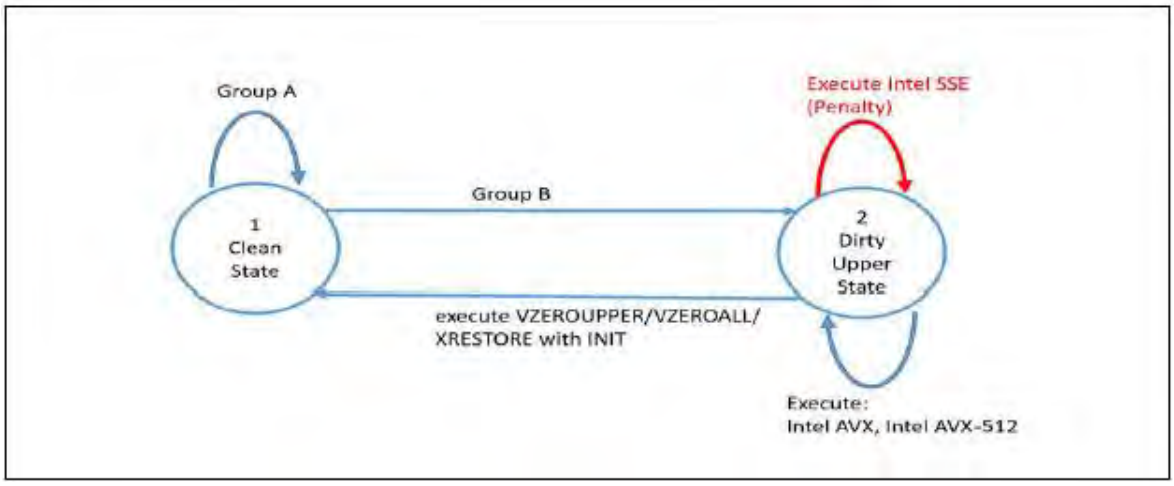


図 18-20 インテル® AVX またはインテル® AVX-512 命令をインテル® SSE 命令と混在する

推奨:

- グループ B の命令をインテル® SSE 命令と混在させる必要がある、またはその可能性がある場合、VZEROUPPER 命令を使用します。
- グループ B の命令が実行された後、さらにインテル® SSE 命令を含む関数呼び出しが行われる前に、VZEROUPPER 命令を追加します。
- グループ B の命令を使用する関数の終わりに VZEROUPPER 命令を追加します。
- スレッドが「上位ダーティーなステート」を引き継がないように、「クリーンステート」でなければスレッドを生成する前に VZEROUPPER 命令を追加します。

18.20 ZMM ベクトルコードと XMM/YMM コードの混在

Skylake† マイクロアーキテクチャーは、2 つのポート体系を持ちます。1 つは 256 ビット以下のレジスターを使用し、もう 1 つは 512 ビット・レジスターを使用するものです。.

256 ビットのレジスターを使用する FMA 操作は、ポート 0 と 1 へディスパッチされ、SIMD 操作はポート 0、1、または 5 へディスパッチされます。512 ビット・レジスターを使用する操作は、FMA および SIMD 操作の両方がポート 0 と 5 へディスパッチされます。

リザベーション・ステーション (RS) の最大レジスター幅が、256 または 512 ポート体系を決定します。

インテル® AVX-512 でエンコードされた YMM 命令を使用する場合、命令は 256 ビット幅であると想定されます。

512 ビット・ポート体系の結果は、XMM や YMM コードを 3 つのポート (0、1、5) ではなく 2 つのポート (0 と 5) へディスパッチするため、256 ビット・ポート体系と比較してスループットは低くそしてレイテンシーは長くなります。

例 18-24 256 ビット・コードと 256 ビットと 512 ビット・コードの混在

256 ビット・コードのみ	**256 ビットと 512 ビット・コードの混在**

`Loop:`
`vpbroadcastd ymm0, dword ptr [rsp]`
`vfmadd213ps ymm7, ymm7, ymm7`
`vfmadd213ps ymm8, ymm8, ymm8`
`vfmadd213ps ymm9, ymm9, ymm9`
`vfmadd213ps ymm10, ymm10, ymm10`
`vfmadd213ps ymm11, ymm11, ymm11`
`vfmadd213ps ymm12, ymm12, ymm12`
`vfmadd213ps ymm13, ymm13, ymm13`
`vfmadd213ps ymm14, ymm14, ymm14`
`vfmadd213ps ymm15, ymm15, ymm15`
`vfmadd213ps ymm16, ymm16, ymm16`
`vfmadd213ps ymm17, ymm17, ymm17`
`vfmadd213ps ymm18, ymm18, ymm18`
`vpermd ymm1, ymm1, ymm1`
`vpermd ymm2, ymm2, ymm2`
`vpermd ymm3, ymm3, ymm3`
`vpermd ymm4, ymm4, ymm4`
`vpermd ymm5, ymm5, ymm5`
`vpermd ymm6, ymm6, ymm6`
`dec rdx`
`jnle Loop`
	`Loop:`
	`vpbroadcastd zmm0, dword ptr [rsp]`
	`vfmadd213ps ymm7, ymm7, ymm7`
	`vfmadd213ps ymm8, ymm8, ymm8`
	`vfmadd213ps ymm9, ymm9, ymm9`
	`vfmadd213ps ymm10, ymm10, ymm10`
	`vfmadd213ps ymm11, ymm11, ymm11`
	`vfmadd213ps ymm12, ymm12, ymm12`
	`vfmadd213ps ymm13, ymm13, ymm13`
	`vfmadd213ps ymm14, ymm14, ymm14`
	`vfmadd213ps ymm15, ymm15, ymm15`
	`vfmadd213ps ymm16, ymm16, ymm16`
	`vfmadd213ps ymm17, ymm17, ymm17`
	`vfmadd213ps ymm18, ymm18, ymm18`
	`vpermd ymm1, ymm1, ymm1`
	`vpermd ymm2, ymm2, ymm2`
	`vpermd ymm3, ymm3, ymm3`
	`vpermd ymm4, ymm4, ymm4`
	`vpermd ymm5, ymm5, ymm5`
	`vpermd ymm6, ymm6, ymm6`
	`dec rdx`
	`jnle Loop`

ベースライン 1x	**スローダウン: 1.3x**

256 ビットのみのコード例では、FMA はポート 0 と 1 へディスパッチされ、*permd* は broadcast 命令が 256 ビットであることから、ポート 5 へディスパッチされます。256 ビットと 512 ビットが混在するコード例では、broadcast は 512 ビット幅であるため、プロセッサーは 512 ビット・ポート体系を使用して FMA をポート 0 と 5 へ、*permd* をポート 5 へディスパッチします。そのためポート 5 へのプレッシャーが高まります。

18.21 単一の FMA ユニットを備える場合

Skylake† マイクロアーキテクチャー・ベースのプロセッサーには、2 つのインテル® AVX-512 FMA ユニット (ポート 0 と 5) を備えるものがある一方、同じアーキテクチャー・ベースのプロセッサーでも単一のインテル® AVX-512 FMA ユニット (ポート 0) を持つものがあります。

2 つの FMA ユニットで実行されることを想定して最適化されたコードは、単一 FMA ユニット上での実行には最適でない可能性があります。

次のコードは、システムのインテル® AVX-512 FMA ユニットの数を判定する方法を示します。以下を含みます。

- インテル® AVX-512 のウォームアップ
- FMA 命令のみを実行する関数
- FMA 命令と shuffle 命令を実行する関数
- コードは、2 つのテスト結果を基にプロセッサーが 1 つまたは 2 つの FMA ユニットを搭載するかを特定します。

テストの精度を高めるため 3 回実行されることに注意してください。

プログラムのオーバーヘッドを軽減するため、すべての関数呼び出しでこのテストを実行しないことを強く推奨します。インストールの一環として、または起動時に一度実行してください。

2 つのプロセッサーの区別は、2 つのスループット・テスト間の比率に基づいています。2 つの FMA ユニットを備えるプロセッサーは、FMA のみのテストを FMA と shuffle テストの 2 倍高速に実行できます。しかし、単独の FMA ユニットを持つプロセッサーは、2 つのテストを同じスピードで実行します。

例 18-25 Skylake† マイクロアーキテクチャー・ベースのプロセッサーの FMA ユニットの数を特定 (1 つもしくは 2 つ)

```
#include <string.h>
#include <stdlib.h>
#include <immintrin.h>
#include <stdio.h>
#include <stdint.h>

static uint64_t rdtsc(void) {
unsigned int ax, dx;
__asm__ __volatile__ ("rdtsc" : "=a"(ax), "=d"(dx));
return ((((uint64_t)dx) << 32) | ax);
}

uint64_t fma_shuffle_tpt(uint64_t loop_cnt){
uint64_t loops = loop_cnt;
   __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
   __declspec(align(64)) int shuf_vec[16] = {0, 1, 2, 3,4, 5, 6, 7,8, 9, 10, 11,12, 13,
   14, 15};
   __asm
   {
   vmovups zmm0, [one_vec]
   vmovups zmm1, [one_vec]
   vmovups zmm2, [one_vec]
   vmovups zmm3, [one_vec]
   vmovups zmm4, [one_vec]
   vmovups zmm5, [one_vec]
   vmovups zmm6, [one_vec]
   vmovups zmm7, [one_vec]
   vmovups zmm8, [one_vec]
   vmovups zmm9, [one_vec]
   vmovups zmm10, [one_vec]
   vmovups zmm11, [one_vec]
   vmovups zmm12, [shuf_vec]
   vmovups zmm13, [shuf_vec]
   vmovups zmm14, [shuf_vec]
   vmovups zmm15, [shuf_vec]
   vmovups zmm16, [shuf_vec]
   vmovups zmm17, [shuf_vec]
   vmovups zmm18, [shuf_vec]
   vmovups zmm19, [shuf_vec]
   vmovups zmm20, [shuf_vec]
   vmovups zmm21, [shuf_vec]
   vmovups zmm22, [shuf_vec]
   vmovups zmm23, [shuf_vec]
   vmovups zmm30, [shuf_vec]
   mov rdx, loops
```

```
   loop1:
   vfmadd231pd zmm0, zmm0, zmm0
   vfmadd231pd zmm1, zmm1, zmm1
   vfmadd231pd zmm2, zmm2, zmm2
   vfmadd231pd zmm3, zmm3, zmm3
   vfmadd231pd zmm4, zmm4, zmm4
   vfmadd231pd zmm5, zmm5, zmm5
   vfmadd231pd zmm6, zmm6, zmm6
   vfmadd231pd zmm7, zmm7, zmm7
   vfmadd231pd zmm8, zmm8, zmm8
   vfmadd231pd zmm9, zmm9, zmm9
   vfmadd231pd zmm10, zmm10, zmm10
   vfmadd231pd zmm11, zmm11, zmm11
   vpermd zmm12, zmm30, zmm30
   vpermd zmm13, zmm30, zmm30
   vpermd zmm14, zmm30, zmm30
   vpermd zmm15, zmm30, zmm30
   vpermd zmm16, zmm30, zmm30
   vpermd zmm17, zmm30, zmm30
   vpermd zmm18, zmm30, zmm30
   vpermd zmm19, zmm30, zmm30
   vpermd zmm20, zmm30, zmm30
   vpermd zmm21, zmm30, zmm30
   vpermd zmm22, zmm30, zmm30
   vpermd zmm23, zmm30, zmm30
   dec rdx
   jg loop1
   }
}

uint64_t fma_only_tpt(int loop_cnt){
uint64_t loops = loop_cnt;
__declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
   __asm
   {
   vmovups zmm0, [one_vec]
   vmovups zmm1, [one_vec]
   vmovups zmm2, [one_vec]
   vmovups zmm3, [one_vec]
   vmovups zmm4, [one_vec]
   vmovups zmm5, [one_vec]
   vmovups zmm6, [one_vec]
   vmovups zmm7, [one_vec]
   vmovups zmm8, [one_vec]
   vmovups zmm9, [one_vec]
   vmovups zmm10, [one_vec]
   vmovups zmm11, [one_vec]
   mov rdx, loops
   loop1:
   vfmadd231pd zmm0, zmm0, zmm0
   vfmadd231pd zmm1, zmm1, zmm1
   vfmadd231pd zmm2, zmm2, zmm2
   vfmadd231pd zmm3, zmm3, zmm3
   vfmadd231pd zmm4, zmm4, zmm4
```

```
   vfmadd231pd zmm5, zmm5, zmm5
   vfmadd231pd zmm6, zmm6, zmm6
   vfmadd231pd zmm7, zmm7, zmm7
   vfmadd231pd zmm8, zmm8, zmm8
   vfmadd231pd zmm9, zmm9, zmm9
   vfmadd231pd zmm10, zmm10, zmm10
   vfmadd231pd zmm11, zmm11, zmm11
   dec rdx
   jg loop1
   }
}

int main()
{
   int i;
   uint64_t fma_shuf_tpt_test[3];
   uint64_t fma_shuf_tpt_test_min;
   uint64_t fma_only_tpt_test[3];
   uint64_t fma_only_tpt_test_min;
   uint64_t start = 0;
   uint64_t number_of_fma_units_per_core = 2;
   /*********************************************************/
   /* ステップ 1: ウォームアップ */
   /*********************************************************/
   fma_only_tpt(100000);

   /*********************************************************/
   /* ステップ 2: FMA と Shuffle TPT テストを実行 */
   /*********************************************************/
   for(i = 0; i < 3; i++){
       start = rdtsc();
       fma_shuffle_tpt(1000);
       fma_shuf_tpt_test[i] = rdtsc() - start;
   }

   /*********************************************************/
   /* ステップ 3: FMA のみの TPT テストを実行 */
   /*********************************************************/
   for(i = 0; i < 3; i++){
       start = rdtsc();
       fma_only_tpt(1000);
       fma_only_tpt_test[i] = rdtsc() - start;
   }

   /*********************************************************/
   /* ステップ 4: 1 つの FMA か 2 つの FMA か決定 */
   /*********************************************************/
   fma_shuf_tpt_test_min = fma_shuf_tpt_test[0];
   fma_only_tpt_test_min = fma_only_tpt_test[0];
   for(i = 1; i < 3; i++){
       if ((int)fma_shuf_tpt_test[i] < (int)fma_shuf_tpt_test_min) fma_shuf_tpt_test_min =
       fma_shuf_tpt_test[i];
       if ((int)fma_only_tpt_test[i] < (int)fma_only_tpt_test_min) fma_only_tpt_test_min =
       fma_only_tpt_test[i];
```

```
    }
    if(((double)fma_shuf_tpt_test_min/(double)fma_only_tpt_test_min) > 1.5){
        number_of_fma_units_per_core = 1;
    }
    printf("%d FMA server¥n", number_of_fma_units_per_core);
    return 0;
}
```

18.22 シャッフルのためのギャザー/スキャッター (G2S/STS)

18.22.1 ストライドロードでシャッフルするためのギャザー

メモリー上の集約された要素間にデータの局所性がある場合、gather 命令をソフトウェア・シーケンスに置き換えることでパフォーマンスを向上できます。

この節では、最も一般的なストライドロードのパターンについて考えます。ストライドロードとは、連続するロードがアクセスするメモリー上のオフセットが一定間隔のロードを指します。

次の例は、3 つの異なるコードで構造体配列 (AoS) から配列構造体 (SoA) への変換を行う様子を示しています。コードは、複素数配列内の実数と虚数要素を 2 つの異なる配列へ分離します。

次のような C コードについて考えてみます。

```
for(int i=0;i<len;i++){
  Real_buffer[i] = Complex_buffer[i].real;
  Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

例 18-26 ストライドロードでシャッフルするためのギャザーの例

代替 1: インテル® AVX-512 vpgatherdd	代替 2: インテル® AVX-512 vpermi2d を使用する G2S

loop:
vpcmpeqb k1, xmm0, xmm0
vpcmpeqb k2, xmm0, xmm0
movsxd rdx, edx
movsxd rdi, esi
inc esi
shl rdi, 0x7
vpxord zmm2, zmm2, zmm2
lea rax, [r8+rdx*8]
add edx, 0x20
vpgatherdd zmm2{k1}, [rax+zmm1*4]
vpxord zmm3, zmm3, zmm3
vpxord zmm4, zmm4, zmm4
vpxord zmm5, zmm5, zmm5
vpgatherdd zmm3{k2} [rax+zmm0*4]
vpcmpeqb k3, xmm0, xmm0
vpcmpeqb k4, xmm0, xmm0
vmovups [r9+rdi*1], zmm2
vmovups [rcx+rdi*1], zmm3
vpgatherdd zmm4{k3} [rax+zmm1*4+0x80]
vpgatherdd zmm5{k4} [rax+zmm0*4+0x80]
vmovups [r9+rdi*1+0x40], zmm4
vmovups [rcx+rdi*1+0x40], zmm5
cmp esi, r14d
jb loop
	vmovups zmm4, [rdx+r9*8]
	vmovups zmm0, [rdx+r9*8+0x40]
	vmovups zmm5, [rdx+r9*8+0x80]
	vmovups zmm1, [rdx+r9*8+0xc0]
	vmovaps zmm2, zmm7
	vmovaps zmm3, zmm7
	vpermi2d zmm2, zmm4, zmm0
	vpermt2d zmm4, zmm6, zmm0
	vpermi2d zmm3, zmm5, zmm1
	vpermt2d zmm5, zmm6, zmm1
	vmovdqu32 [rcx+r9*4], zmm2
	vmovdqu32 [rcx+r9*4+0x40], zmm3
	vmovdqu32 [r8+r9*4], zmm4
	vmovdqu32 [r8+r9*4+0x40], zmm5
	add r9, 0x20
	cmp r9, r10
	jb loop

ベースライン 1x	**スピードアップ: 4.8x**

次の定数は zmm レジスターへロードされ、gather と permute のインデックスとして使用されます。

Zmm0 (代替 1)、zmm6 (代替 2)
__declspec (align(64)) const __int32 gather_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};

Zmm1 (代替 1)、zmm7 (代替 2)
__declspec (align(64)) const __int32 gather_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};

推奨事項: 最高のパフォーマンスを得るには、短い間隔のストライドロードをロードとパーミュートのシーケンスに置き換えます。

18.22.2 ストライドストアでシャッフルするためのスキャッター

以下は、シャッフルを行うスキャッターの例であり、スキャッターをパーミュートとストア命令に置き換える C のコードです。

```
for(int i=0;i<len;i++){
  Complex_buffer[i].real = Real_buffer[i];
  Complex_buffer[i].imag = Imaginary_buffer[i];
}
```

例 18-27 ストライドストアでシャッフルするためのスキャッターの例

代替 1: インテル® AVX-512 vscatterdps	代替 2: インテル® AVX-512 vpermi2d を使用する S2S

loop:
vpcmpeqb k1, xmm0, xmm0
lea r11, [r8+rcx*4]
vpcmpeqb k2, xmm0, xmm0
vmovups zmm2, [rax+rsi*4]
vmovups zmm3, [r9+rsi*4]
vscatterdps [r11+zmm1*4]{k1} zmm2
vscatterdps [r11+zmm0*4]{k2} zmm3
add rsi, 0x10
add rcx, 0x20
cmp rsi, r10
jl loop
	loop:
	vmovups zmm4, [rax+r8*4]
	vmovups zmm2, [r10+r8*4]
	vmovaps zmm3, zmm1
	add r8, 0x10
	vpermi2d zmm3, zmm4, zmm2
	vpermt2d zmm4, zmm0, zmm2
	vmovups [r9+rsi*4], zmm3
	vmovups [r9+rsi*4+0x40], zmm4
	add rsi, 0x20
	cmp r8, r11
	jl loop

ベースライン 1x	**スピードアップ: 4.4x**

次の定数がスキャッターのインデックスとして使用されています。

Zmm1:
__declspec (align(64)) const __int32 scatter_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};

Zmm0:
__declspec (align(64)) const __int32 scatter_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};

次の定数がパーミュートのインデックスとして使用されています。

Zmm1:
__declspec (align(64)) const __int32 first_half[16] = {0, 16, 1, 17, 2, 18, 3, 19, 4, 20, 5, 21, 6, 22, 7, 23};

Zmm0:
__declspec (align(64)) const __int32 second_half[16] = {8, 24, 9, 25, 10, 26, 11, 27, 12, 28, 13, 29, 14, 30, 15, 31};

18.22.3 隣接するロードでシャッフルするためのギャザー

集約される要素が隣接するシーケンスにグループ化される場合、gather 命令をソフトウェア・シーケンスで置き換えることでパフォーマンスを向上できます。

次の例は、要素が隣接する場合どのようにベクトルをロードするかを示しています。

この場合、配列要素の順番はインデックス・バッファーに従って設定されることに注意してください。そのため、18.22.1 節「ストライドロードでシャッフルするためのギャザー」で説明したソフトウェアの最適化は、このケースには適用されません。

次のような C コードについて考えてみます。

```
typedef struct{
  double var[4];
} ElemStruct;
```

```
const int* indices = Indices;
const ElemStruct *in = (const ElemStruct*) InputBuffer;
double* restrict out = OutputBuffer;

for (int i = 0; i < width; i++){
  for (int j = 0; j < 4; j++){
    out[i*4+j] = in[indices[i]].var[j];
  }
}
```

例 18-28 隣接するロードでシャッフルするためのギャザーの例

代替 1: vgatherdpd 実装	**代替 2: ロードとマスク付きブロードキャスト**

loop:
vpbroadcastd ymm3, [r9+rsi*4]
mov r15d, esi
vpbroadcastd xmm2, [r9+rsi*4+0x4]
add rsi, 0x2
vpbroadcastd ymm3{k1}, xmm2
vpmulld ymm4, ymm3, ymm1
vpaddd ymm5, ymm4, ymm0
vpcmpeqb k2, xmm0, xmm0
shl r15d, 0x2
movsxd r15, r15d
vpxord zmm6, zmm6, zmm6
vgatherdpd zmm6{k2} [r10+ymm5*1]
vmovups [r11+r15*8], zmm6
cmp rsi, rdi
jl loop
	loop:
	movsxd r11, [r10+rcx*4]
	shl r11, 0x5
	vmovupd ymm0, [r9+r11*1]
	movsxd r11, [r10+rcx*4+0x4]
	shl r11, 0x5
	vbroadcastf64x4 zmm0{k1}, [r9+r11*1]
	mov r11d, ecx
	shl r11d, 0x2
	add rcx, 0x2
	movsxd r11, r11d
	vmovups [r8+r11*8], zmm0
	cmp rcx, rsi
	jl loop

ベースライン 1x	**スピードアップ: 2.2x**

次の定数が vgatherdpd 実装に使用されています。

ymm0:
__declspec (align(64)) const __int32 index_inc[8] = {0, 8, 16, 24, 0, 8, 16, 24};

ymm1:
__declspec (align(64)) const __int32 index_scale[8] = {32, 32, 32, 32, 32, 32, 32, 32};

K1 レジスターの値は 0xF0 です。

18.23 データ・アライメント

この節では、インテル® AVX-512 命令におけるアライメントされたデータの利点を説明し、アライメントができない場合にパフォーマンスを改善するいくつかの方法を紹介します。この節では、いくつかの SAXPY カーネル例を使用して説明を行います。SAXPY はスカラーの「Alpha * X + Y」アルゴリズムです。

以下の C コードは SAXPY の C 実装です。

```
for (int i = 0; i < n; i++)
{
  c[i] = alpha * a[i] + b[i];
}
```

18.23.1 64 バイトにデータをアライメント

ベクトル長に合わせてデータをアライメントすることを推奨します。最良の結果を得るには、インテル® AVX-512 命令を使用する場合はデータを 64 バイトにアライメントします。

インテル® AVX-512 でアライメントなしの 64 バイト・ロード/ストアを実行する場合、キャッシュラインが 64 バイトであるため、各ロード/ストアはキャッシュライン分割を引き起こします。32 バイト・レジスターを使用するインテル® AVX2 と比べて、キャッシュライン分割の頻度は倍になります。メモリー集約型のコードでキャッシュライン分割が頻発すると、パフォーマンスは低下します。

次の表は、メモリー集約型の SAXPY コードのパフォーマンスが、アライメントされていない入出力バッファーによってどのように影響されるかを示しています。表のデータは次のコードを基に算出されています。

例 18-29 データアライメント

```
__asm {
      mov rax, src1
      mov rbx, src2
      mov rcx, dst
      mov rdx, len
      xor rdi, rdi
      vbroadcastss zmm0, alpha
mainloop:
      vmovups zmm1, [rax]
      vfmadd213ps zmm1, zmm0, [rbx]
      vmovups [rcx], zmm1
      vmovups zmm1, [rax+0x40]
      vfmadd213ps zmm1, zmm0, [rbx+0x40]
      vmovups [rcx+0x40], zmm1
      vmovups zmm1, [rax+0x80]
      vfmadd213ps zmm1, zmm0, [rbx+0x80]
      vmovups [rcx+0x80], zmm1
      vmovups zmm1, [rax+0xC0]
      vfmadd213ps zmm1, zmm0, [rbx+0xC0]
      vmovups [rcx+0xC0], zmm1
      add rax, 256
      add rbx, 256
      add rcx, 256
      add rdi, 64
      cmp rdi, rdx
      jl mainloop
}
```

–

次の表は、各種オプションによる SAXPY のパフォーマンスへのデータ・アライメントの影響をまとめたものです。

表 18-8 SAXPY パフォーマンスとスピードアップへのデータ・アライメントの影響

SAXPY パフォーマンスへのデータ・アライメントの影響	スピードアップ
代替 1: ソースとデスティネーションはともに 64 バイトにアライメントされています。	ベースライン、1.0
代替 2: ソースは 64 バイトにアライメントされ、デスティネーションは 4 バイト・オフセットを持ちます。	0.66x
代替 3: ソースとデスティネーションは、アライメント境界からの 4 バイト・オフセットを持ちます。	0.59x
代替 4: 一方のソースはアライメント境界からの 4 バイト・オフセットを持ち、もう一方のソースとデスティネーションは 64 バイトにアライメントされています。	0.77x

18.24 動的メモリー割り当てとメモリーのアライメント

次の構造体について考えてみましょう。

```
float3_SOA {
  __declspec(align(64)) float x[16];
  __declspec(align(64)) float y[16];
};
```

構造体のメモリー割り当ては 64 バイトにアライメントされています。次のように使用します。

```
  float3_SOA f;
```

次のような動的なメモリー割り当てを行う場合、decspec ディレクティブは無視され、64 バイトのアライメントは保証されません。

```
  float3_SOA* stPtr = new float3_SOA();
```

この場合、動的なアライメントを保証する割り当てを使用するか、オペレーター new を再定義する必要があります。

推奨事項: 可能な限りデータを 64 バイトにアライメントし、次のガイドラインに従います。

- インテル® コンパイラーで提供される _mm_malloc 組込み関数や MIcrosoft* コンパイラーの _aligned_malloc を使用して動的なデータ割り当てをアライメントできます。次に例を示します。
  ```
  // float 2048 要素のバッファーを 64 バイトのアライメントで動的に割り当て。
  InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 64);
  ```

- __declspec(align(64)) を使用して静的なデータ割り当てをアライメントできます。次に例を示します。
  ```
  // float 2048 要素のバッファーを 64 バイトのアライメントで静的に割り当て。
  __declspec(align(64)) float InputBuffer[2048];
  ```

18.25 除算と平方根命令

VRSQRT14PS/VRSQRT14PD および VRCP14PS/VRCP14PD 命令を使用して、単精度除算と平方根の計算をスピードアップできます。これらの命令は、入力値の逆数平方根/逆数除算の近似 (14 ビット精度で) を生成します。

インテル® AVX-512 ではこれらの命令はパイプライン化され、次のような機能を持ちます。

- 256 ビットのベクトルの場合、4 サイクルのレイテンシーとサイクルごとに 1 命令のスループット。
- 512 ビットのベクトルの場合、6 サイクルのレイテンシーと 2 サイクルごとに 1 命令のスループット。

Skylake† マイクロアーキテクチャーでは、逆数平方根/逆数除算のパックド倍精度 (PD) VRSQRT14PD と VRCP14PD が導入されました。

ニュートンラフソン反復やほかの多項式近似で VRSQRT14PS/VRSQRT14PD および VRCP14PS/VRCP14PD 命令を使用することで、VDIVPS や VSQRTPS 命令 (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』を参照) と同じ精度を実現し、より高いスループットを達成できます。

完全な精度 (IEEE) が必要である場合、Skylake† マイクロアーキテクチャーの劇的なパフォーマンス向上により低レイテンシーと高スループットが達成された DIVPS と SQRTPS を使用します。表 18-11 と表 18-12 に、Broadwell† と Skylake† マイクロアーキテクチャーにおけるこれらの命令のパフォーマンスの比較を示します。

注意

除算や平方根操作が大きなアルゴリズムの一部であるためレイテンシーが隠匿されるような場合、ニュートンラフソン近似は、そのほかの命令から発行されるマイクロオペレーションでパイプが満たされることで実行速度が低下する可能性があります。

次の節では、必要とする精度レベルに合わせた推奨される計算方法と命令を示します。

注意

値の近似誤りとその近似には 2 つの定義があります。

v_{approx}

絶対誤差 $= |v - v_{approx}|$
相対誤差 $= |v - v_{approx}| / |v|$

本章では、"ビット数" エラーは相対的であり、絶対値のエラーではないとします。
ここで比較する近似値 v は、できるだけ正確で、倍精度でなければいけません。

18.25.1 除算と平方根命令の近似

表 18-9 Skylake† マイクロアーキテクチャーで推奨される DIV/SQRT ベースの操作 (単精度)

演算	精度	推奨される方法
除算	24 ビット (IEEE)	DIVPS
	23 ビット	RCP14PS + MULPS + 1 ニュートンラフソン反復
	14 ビット	RCP14PS + MULPS
逆数平方根	22 ビット	SQRTPS + DIVPS
	23 ビット	RSQRT14PS + 1 ニュートンラフソン反復
	14 ビット	RSQRT14PS
平方根	24 ビット (IEEE)	SQRTPS
	23 ビット	RSQRT14PS + MULPS + 1 ニュートンラフソン反復
	14 ビット	RSQRT14PS + MULPS

表 18-10 Skylake† マイクロアーキテクチャーで推奨される DIV/SQRT ベースの操作 (倍精度)

演算	精度	推奨される方法
除算	53 ビット (IEEE)	DIVPD
	52 ビット	RCP14PD + MULPD + 2 ニュートンラフソン反復
	26 ビット	RCP14PD + MULPD + 1 ニュートンラフソン反復
	14 ビット	RCP14PD + MULPD
逆数平方根	53 ビット (IEEE)	SQRTPD + DIVPD
	52 ビット	RSQRT14PD+2 N-R + エラー訂正または SQRTPD + DIVPD
	50 ビット	RSQRT14PD + 多項式近似
	26 ビット	RSQRT14PD+1 N-R
	14 ビット	RSQRT14PD
平方根	51 ビット (IEEE)	SQRTPD
	52 ビット	RSQRT14PD + MULPD + 多項式近似
	26 ビット	RSQRT14PD + MULPD + 1 N-R
	14 ビット	RSQRT14PD + MULPD

18.25.2 除算と平方根命令のパフォーマンス

Broadwell† と Skylake† マイクロアーキテクチャーにおけるベクトル除算と平方根操作のパフォーマンスを以下に示します。

表 18-11 インテル® AVX2 の 256 ビット除算と平方根命令のパフォーマンス

Broadwell† マイクロアーキテクチャー	DIVPS	SQRTPS	DIVPD	SQRTPD
レイテンシー	17	21	23	35
スループット	10	14	16	28
Skylake† マイクロアーキテクチャー	DIVPS	SQRTPS	DIVPD	SQRTPD
レイテンシー	11	12	14	18
スループット	5	6	16	12

表 18-12 インテル® AVX-512 の 512 ビット除算と平方根命令のパフォーマンス

Skylake† マイクロアーキテクチャー	DIVPS	SQRTPS	DIVPD	SQRTPD
レイテンシー	17	19	23	31
スループット	10	12	16	24

18.25.3 近似のレイテンシー

この節では、近似法、DIV、および SQRT 命令におけるレイテンシーとスループットについて説明します。以下に示す表から、ほとんどのケースにおいて近似法のスループット・ゲインは IEEE に対して (少なくとも) 倍以上であることが分かります。ここでは、簡単なループで除算または平方根を計算しています。

ループ反復に多くの計算 (除算と平方根以外) が含まれている場合、近似シーケンスのスループットの利点は減少します。

経験則から、ループ反復に 8-10 を超えない単精度操作、または 12-15 を超えない倍精度操作が含まれる場合、IEEE に近い精度の近似が推奨されます。次の表から、これらの正確な近似は、スループットの最適化のみに効果があることが分かります。精度を必要としない近似は、レイテンシーとスループットの両方を向上します。

ニュートンラフソン近似は、デノーマル入力、ゼロ、および無限大の特殊ケースに適用されないことに注意してください。また、デノーマルに近い入力では中間処理のアンダーフローにより、精度を失うこともあります。ゼロと無限大入力は、わずかな操作で比較的容易に解決できますが (以下のシーケンスで示すように)、デノーマル除数はパフォーマンスへの影響なしに対処することはできません。アンダーフローやオーバーフローのしきい値から離れた、「中央範囲」の入力に対し近似シーケンスは最も効率良く動作します。

次の表は、インテル® AVX-512 の除算と平方根命令の単精度のレイテンシーとスループットを、Skylake† マイクロアーキテクチャーの近似法と比較して示しています。

表 18-13 Skylake† マイクロアーキテクチャーにおける異なるベクトル幅で除算と平方根を計算する異なる手法のレイテンシー/スループット (単精度)

演算	方法	精度	256 ビット・インテル® AVX-512 組込み関数		512 ビット・インテル® AVX-512 組込み関数	
			スループット	レイテンシー	スループット	レイテンシー
除算 (a/b)	DIVPS	24 ビット (IEEE)	5	11	10	17
	RCP14PS + MULPS + 1 ニュートンラフソン反復	23 ビット	2	16	3	20
	RCP14PS + MULPS	14 ビット	1	8	2	10-12
平方根	SQRTPS	24 ビット (IEEE)	6	12	12	19
	RSQRT14PS + MULPS + 1 ニュートンラフソン反復	23 ビット	3	16	5	20
	RSQRT14PS + MULPS	14 ビット	2	9	3	12
逆数平方根	SQRTPS + DIVPS	22 ビット	11	23	22	36
	RSQRT14PS + 1 ニュートンラフソン反復	23 ビット	3.67	20	4.89	25
	RSQRT14PS	14 ビット			2	6

表 18-14 Skylake† マイクロアーキテクチャーにおける異なるベクトル幅で除算と平方根を計算する異なる手法のレイテンシー/スループット (倍精度)

演算	方法	精度	256 ビット・インテル® AVX-512 組込み関数		512 ビット・インテル® AVX-512 組込み関数	
			スループット	レイテンシー	スループット	レイテンシー
除算 (a/b)	DIVPD	53 ビット (IEEE)	8	14	16	23
	RCP14PD + MULPD + 2 ニュートンラフソン反復	22 ビット	3.2	27	4.7	28.4
	RCP14PD + MULPD + 1 ニュートンラフソン反復	26 ビット	2	16	3	20
	RCP14PS + MULPS	14 ビット	1	8	2	10-12
平方根	SQRTPD	53 ビット (IEEE)	6	12	12	19
	RSQRT14PD + MULPD + 多項式近似	22 ビット	4.82	24.54[1]	6.4	28.48[1]
	RSQRT14PD + MULPD + 1 N-R	23 ビット	3.76	17	5	20
	RSQRT14PD + MULPD	14 ビット	2	9	3	12
逆数平方根	SQRTPD + DIVPD	51 ビット	20	32	40	53
	RSQRT14PD + 2-NR + エラー訂正	52 ビット	5	29.38	6.53	34
	RSQRT14PD+2 N-R	50 ビット	3.79	25.73	5.51	30
	RSQRT14PD+1 N-R	26 ビット	2.7	18	4.5	21.67
	RSQRT14PD	14 ビット	1	4	2	6

注意:

1. コードシーケンスにはいくつかの FMA 命令 (4/6 の異なるレイテンシーを持つ) が含まれるため、丸めは行われていません。そのため、これらのシーケンスにおけるレイテンシーを考慮する必要はありません。

18.25.4 コード例

例 18-30 ベクトル化された 32 ビット浮動小数点除算

単精度除算 24 ビット (IEEE)

```
float a = 10;
float b = 5;
__asm {
   vbroadcastss zmm0, a // zmm0 に a の 16 個の要素を代入
   vbroadcastss zmm1, b // zmm0 に b の 16 個の要素を代入
   vdivps zmm2, zmm0, zmm1 // zmm2 = 16 elements of a/b
}
```

単精度除算 23 ビット

```
/* Input:
      zmm0 = vector of a's
      zmm1 = vector of b's
   Output:
      zmm3 = vector of a/b
*/
__asm {
   vrcp14ps zmm2, zmm1
   vmulps zmm3, zmm0, zmm2
   vmovaps zmm4, zmm0
   vfnmadd231ps zmm4, zmm3, zmm1
   vfmadd231ps zmm3, zmm4, zmm2
}
```

単精度除算 14 ビット

```
/* Input:
      zmm0 = vector of a's
      zmm1 = vector of b's
   Output:
      zmm2 = vector of a/b
*/
__asm {
   vrcp14ps zmm2, zmm1
   vmulps zmm2, zmm0, zmm2
}
```

例 18-31 逆数平方根

単精度逆数平方根 22 ビット

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of 1’s
   Output:
      zmm2 = vector of 1/sqrt (a)
*/
float one = 1.0;
__asm {
   vbroadcastss zmm1, one // zmm1 = vector of 16 1’s
   vsqrtps zmm2, zmm0
   vdivps zmm2, zmm1, zmm2
}
```

単精度逆数平方根 23 ビット

```
/* Input:
      zmm0 = vector of a’s
   Output:
      zmm2 = vector of 1/sqrt (a)
*/
float half = 0.5;
__asm {
   vbroadcastss zmm1, half // zmm1 =
   vector of 16 0.5’s
   vrsqrt14ps zmm2, zmm0
   vmulps zmm3, zmm0, zmm2
   vmulps zmm4, zmm1, zmm2
   vfnmadd231ps zmm1, zmm3, zmm4
   vfmsub231ps zmm3, zmm0, zmm2
   vfnmadd231ps zmm1, zmm4, zmm3
   vfmadd231ps zmm2, zmm2, zmm1
}
```

単精度逆数平方根 14 ビット

```
/* Input:
      zmm0 = vector of a’s
   Output:
      zmm2 = vector of 1/sqrt (a)
*/
__asm {
   vrsqrt14ps zmm2, zmm0
}
```

例 18-32 平方根

単精度平方根 24 ビット (IEEE)

```
/* Input:
      zmm0 = vector of a’s
   Output:
      zmm2 = vector of sqrt (a)
*/
__asm {
   vsqrtps zmm2, zmm0
}
```

単精度平方根 23 ビット

```
/* Input:
      zmm0 = vector of a’s
   Output:
      zmm0 = vector of sqrt (a)
*/
float half = 0.5;
__asm {
   vbroadcastss zmm3, half
   vrsqrt14ps zmm1, zmm0
   vfpclassps k2, zmm0, 0xe
   vmulps zmm2, zmm0, zmm1, {rn-sae}
   vmulps zmm1, zmm1, zmm3
   knotw k3, k2
   vfnmadd231ps zmm0{k3}, zmm2, zmm2
   vfmadd213ps zmm0{k3}, zmm1, zmm2
}
```

単精度平方根 14 ビット

```
/* Input:
      zmm0 = vector of a’s
   Output:
      zmm0 = vector of sqrt (a)
*/
__asm {
   vrsqrt14ps zmm1, zmm0
   vfpclassps k2, zmm0, 0xe
   knotw k3, k2
   vmulps zmm0{k3}, zmm0, zmm1
}
```

例 18-33 パックド倍精度除算

倍精度除算 53 ビット (IEEE)

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of b’s
   Output:
      zmm2 = vector of a/b
*/
__asm {
   vdivpd zmm2, zmm0, zmm1
}
```

倍精度除算 52 ビット

```
/* Input:
      zmm15 = vector of a’s
      zmm0 = vector of b’s
   Output:
      zmm0 = vector of a/b
*/
double One = 1.0;
__asm {
   vrcp14pd zmm1, zmm0
   vmovapd zmm4, zmm0
   vbroadcastsd zmm2, one
   vfnmadd213pd zmm0, zmm1, zmm2, {rn-sae}
   vfpclasspd k2, zmm1, 0x1e
   vfmadd213pd zmm0, zmm1, zmm1, {rn-sae}}
   knotw k3, k2
   vfnmadd213pd zmm4, zmm0, zmm2, {rn-sae}
   vblendmpd zmm0 {k2}, zmm0, zmm1
   vfmadd213pd zmm0 {k3}, zmm4, zmm0, {rn-
sae}
   vmulpd zmm0, zmm0, zmm15
}
```

倍精度除算 26 ビット

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of b’s
   Output:
      zmm3 = vector of a/b
*/
__asm {
   vrcp14pd zmm2, zmm1
   vmulpd zmm3, zmm0, zmm2
   vmovapd zmm4, zmm0
   vfnmadd231pd zmm4, zmm3, zmm1
   vfmadd231pd zmm3, zmm4, zmm2
}
```

倍精度除算 14 ビット

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of b’s
   Output:
      zmm2 = vector of a/b
*/
__asm {
   vrcp14pd zmm2, zmm1
   vmulpd zmm2, zmm0, zmm2
}
```

例 18-34 倍精度逆平方根

倍精度逆数平方根 51 ビット

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of 1’s
   Output:
      zmm0 = vector of 1/sqrt (a)
*/
__asm {
   vsqrtpd zmm0, zmm0
   vdivpd zmm0, zmm1, zmm0
}
```

倍精度逆数平方根 52 ビット

```
/* Input:
      zmm4 = vector of a’s
   Output:
      zmm0 = vector of 1/sqrt (a)
*/
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
// used for aligning data structures to n
bytes
#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 one[ ] =
{DUP8_DECL(0x3FF0000000000000)};
ALIGNTO(64) __int64 dc1[ ] =
{DUP8_DECL(0x3FE0000000000000)};
ALIGNTO(64) __int64 dc2[ ] =
{DUP8_DECL(0x3FD8000004600001)};
ALIGNTO(64) __int64 dc3[ ] =
{DUP8_DECL(0x3FD4000005E80001)};

__asm {
   vbroadcastsd zmm4, big_num
   vmovapd zmm0, one
   vmovapd zmm5, dc1
   vmovapd zmm6, dc2
   vmovapd zmm7, dc3
   vrsqrt14pd zmm3, zmm4
   vfpclasspd k1, zmm4, 0x5e
```

倍精度逆数平方根 50 ビット

```
/* Input:
      zmm3 = vector of a’s
   Output:
      zmm4 = vector of 1/sqrt (a)
*/
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
// used for aligning data structures to n
bytes
#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 one[ ] =
{DUP8_DECL(0x3FF0000000000000)};
ALIGNTO(64) __int64 dc1[ ] =
{DUP8_DECL(0x3FE0000000000000)};
ALIGNTO(64) __int64 dc2[ ] =
{DUP8_DECL(0x3FD8000004600001)};
ALIGNTO(64) __int64 dc3[ ] =
{DUP8_DECL(0x3FD4000005E80001)};

__asm {
vmovapd zmm5, one
vmovapd zmm6, dc1
vmovapd zmm8, dc3
vmovapd zmm7, dc2
vrsqrt14pd zmm2, zmm3
vfpclasspd k1, zmm3, 0x5e
vmulpd zmm0, zmm2, zmm3, {rn-sae}
```

```
   vmulpd zmm1, zmm3, zmm4, {rn-sae}
   vfnmadd231pd zmm0, zmm3, zmm1
   vfmsub231pd zmm1, zmm3, zmm4, {rn-sae}
   vfnmadd213pd zmm1, zmm3, zmm0
   vmovups zmm0, zmm7
   vmulpd zmm2, zmm3, zmm1
   vfmadd213pd zmm0, zmm1, zmm6
   vfmadd213pd zmm0, zmm1, zmm5
   vfmadd213pd zmm0, zmm2, zmm3
   vorpd zmm0{k1}, zmm3, zmm3
}

/* Input:
      zmm0 = vector of a's
   Output:
      zmm1 = vector of 1/sqrt (a)
*/
double half = 0.5;
__asm {
   vrsqrt14pd zmm1, zmm0
   vmulpd zmm0, zmm0, zmm1
   vbroadcastsd zmm3, half
   vmulpd zmm2, zmm1, zmm3
   vfnmadd213pd zmm2, zmm0, zmm3
   vfmadd213pd zmm1, zmm2, zmm1
}
```

```
vfnmadd231pd zmm0, zmm2, zmm5
vmulpd zmm1, zmm2, zmm0
vmovapd zmm4, zmm8
vfmadd213pd zmm4, zmm0, zmm7
vfmadd213pd zmm4, zmm0, zmm6
vfmadd213pd zmm4, zmm1, zmm2
vorpd zmm4{k1}, zmm2, zmm2
}



/* Input:
      zmm0 = vector of a's
   Output:
      zmm2 = vector of 1/sqrt (a)
*/
__asm {
   vrsqrt14pd zmm2, zmm0
}
```

例 18-35 パックド倍精度逆平方

倍精度平方根 53 ビット (IEEE)	**倍精度平方根 52 ビット**

/* Input:
zmm0 = vector of a's
Output:
zmm2 = vector of sqrt (a)
*/
 	
__asm {
vsqrtpd zmm2, zmm0
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm0 = vector of sqrt (a)
	*/
	double half = 0.5;
	__asm {
	vbroadcastsd zmm4, half
	vrsqrt14pd zmm1, zmm0
	vfpclasspd k2, zmm0, 0xe
	vmulpd zmm2, zmm0, zmm1, {rn-sae}

	vmulpd zmm1, zmm1, zmm4 knotw k3, k2 vmovapd zmm3, zmm4 vfnmadd231pd zmm3, zmm1, zmm2, {rn-sae} vfmadd213pd zmm2, zmm3, zmm2, {rn-sae} vfmadd213pd zmm1, zmm3, zmm1, {rn-sae} vfnmadd231pd zmm0 {k3}, zmm2, zmm2, {rn-sae} vfmadd213pd zmm0 {k3}, zmm1, zmm2 }
倍精度平方根 26 ビット	**倍精度平方根 14 ビット**

/* Input:
zmm0 = vector of a's
Output:
zmm0 = vector of sqrt (a)
*/
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 OneHalf[ ] = {DUP8_DECL(0X3FE0000000000000)};
 	
__asm {
vrsqrt14pd zmm1, zmm0
vfpclasspd k2, zmm0, 0xe
knotw k3, k2
vmulpd zmm0 {k3}, zmm0, zmm1
vmulpd zmm1, zmm1, ZMMWORD PTR [OneHalf]
vfnmadd213pd zmm1, zmm0, ZMMWORD PTR [OneHalf]
vfmadd213pd zmm0 {k3}, zmm1, zmm0
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm0 = vector of sqrt (a)
	*/
	
	__asm {
	vrsqrt14pd zmm1, zmm0
	vfpclasspd k2, zmm0, 0xe
	knotw k3, k2
	vmulpd zmm0 {k3}, zmm0, zmm1
	}

18.26 コンパイラーを利用するヒント

この節では、インテル® コンパイラーを使用して Skylake† Server 上で最高のパフォーマンスを得るために重要なコンパイラー・オプションについて説明します。コンパイラーのオプションとチューニングのヒントに関する詳細は、製品ページのドキュメントを参照してください: https://software.intel.com/en-us/intel-softwaretechnical-documentation (英語)。例えば、インテル® C++ コンパイラー 17.0 デベロッパー・ガイドおよびリファレンスは以下で入手できます: https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-referenceguide (英語)。

多くのオプション名は、Windows* では先頭に Q が付くことを除き、Linux*、OS X*、Windows* で同じです。この場合ドキュメントでは、[Q]option-name のように記載されます。

デフォルトの最適化レベルは O2 です (デバッグオプションが指定される場合を除く)。O2 オプションは、ベクトル化を含む多くのコンパイラーによる最適化を有効にします。O3 オプションは、ループ構造が主体な HPC アプリケーションなどに推奨され、キャッシュを効率良く利用するループ・フュージョンやループ・ブロッキングなど、より積極的なループとメモリーアクセスの最適化を有効にします。

Skylake† Server マイクロアーキテクチャー上で最高のパフォーマンスを発揮するには、プロセッサー固有のオプション [Q]xCORE-AVX512 を使用してコンパイルします。このオプションでコンパイルされた実行可能ファイルは、非インテル製プロセッサーや、下位の命令セットをサポートするインテル製プロセッサー上では実行されません。

Skylake† Server マイクロアーキテクチャーと Knights Landing† マイクロアーキテクチャー・ベースのインテル® Xeon Phi™ プロセッサーの両方で実行可能な共通バイナリーを作成する場合、[Q]xCOMMON-AVX512 オプションを使用します。このオプションは、ターゲット固有のオプション [Q]xCORE-AVX512 (Skylake† Server) と [Q]xMIC-AVX512 (Knights Landing†) で生成されたバイナリーに比べ、両方のマイクロアーキテクチャーでパフォーマンス上のコストを伴います。

また、Skylake† Server マイクロアーキテクチャー向けには、-qopt-zmm-usage=low|high (Linux*) や /Qopt-zmm-usage:low|high (Windows*) オプションを追加して zmm コードの生成を最適化できます。引数レベル low は、Skylake† Server マイクロアーキテクチャー上でインテル® AVX2 ISA からインテル® AVX-512 ISA へのスムースな移行を可能にします。これは、エンタープライズ向けのアプリケーションで効果を発揮します。ZMM 命令向けのチューニングには、#pragma omp simd simdlen() などの明示的なベクトル構文を使用することを推奨します。引数レベル high は、より広いベクトル操作を使用して命令ごとの計算量を高めるため、ベクトル計算が主体の HPC コードなどのアプリケーションに適しています。Skylake† Server マイクロアーキテクチャーをターゲットとする [Q]xCORE-AVX512 と [Q]xCOMMON-AVX512 などより上位の CORE-AVX512/MIC-AVX512 を組み合わせたコンパイルでのデフォルトは low です。

また、[Q]ax ターゲットオプションを使用して、複数の命令セットをサポートする自動ディスパッチ・バイナリーを生成することもできます (コードサイズは大きくなります)。例えば、[Q]axCORE-AVX512,CORE-AVX2 オプションでアプリケーションをコンパイルすると、コンパイラーは Skylake† Server マイクロアーキテクチャーとインテル® AVX2 をターゲットとする特殊なコードを生成します。また、このバイナリーには、インテル® SSE2 をサポートする非インテル製プロセッサーや、インテル製プロセッサーで実行可能なデフォルトのコードパスも生成されます。実行時にアプリケーションは、自動的に動作環境のプロセッサーを識別します。そして、検出されたプロセッサーに最も適したコードパスが実行されます。指定されたターゲットと一致しない場合、デフォルトのコードパス (インテル® SSE2) が実行されます。また、指定されるオプションにかかわらず、コンパイラーは実行時のプロセッサー検出に基づくコードパスのディスパッチにより、memset/memcpy など最適化されたバージョンの呼び出しに置き換えるコードを挿入する可能性があることに注意してください。

-qopt-report[n] (Linux*)、/Qopt-report[:n] (Windows*) オプションは、コンパイラーによって適用された最適化のレポートを生成します。デフォルトではレポートは .optrpt 拡張子のファイルへ書き込まれます。n には、0 (レポートなし) から 5 (最も詳しい) の詳細レベルを指定します。.-qopt-report-phase[=list] (Linux*)、/Qopt-report-phase[:list] (Windows*) オプションを使用して、コンパイラーが生成する最適化レポートに含まれる最適化フェーズを指定できます (cg、ipo、loop、offload、openmp、par、vec、pgo、tcollect、all)。デフォルトは all です。このレポートは、コンパイラーが適用した、または適用できなかったパフォーマンス最適化の詳細を知ることで、インライン展開、OpenMP* 並列

化、ループの最適化 (ループ分割やアンロールなど)、およびベクトル化など各種最適化の相互関係を理解するのに役立ちます。

レポートはコンパイラーの静的解析を基に生成されます。このレポートは、インテル® Advisor やベクトル化アドバイザー (インテル® Advisor の機能) によるホットスポット解析や他の動的解析情報と組み合わせるとさらに有用です。一度情報を入手できると、コンパイラー・レポートのホットスポット (関数/ループの入れ子) に対する最適化情報を理解できます。コンパイラーが複数バージョンのループの入れ子を生成できるため、実際に実行されるバージョンを解析する際の関連付けに役立つことを覚えておいてください。コンパイラーのループ最適化におけるフェーズの入れ替えは、最適なベクトル化を可能にします。ループ最適化のパラメーターを理解することは、将来のパフォーマンス・チューニングに役立つでしょう。多くのケースでは、プラグマ、ディレクティブおよびオプションによってループの最適化を適切に制御できます。

アプリケーション・コードが OpenMP* プラグマやディレクティブを含み、OpenMP* ベースのスレッド化とベクトル化有効にをするには、-qopenmp (Linux*)、/Qopenmp (Windows*) を指定してコンパイルします。あるいは、-qopenmp-simd (Linux*)、/Qopenmp-simd (Windows*) オプションを使用して OpenMP* の SIMD ベクトル化機能だけを有効にできます。

コンパイラーによる自動ベクトル化が有効であるかを試すには、-no-vec -no-simd -qno-openmp-simd (Linux*)、/Qvec- /Qsimd- /Qopenmp-simd- (Windows*) オプションを使用してベクトル化を完全に無効化してみると良いでしょう。

効率良くベクトル化を行うには、データのアライメントが重要です。通常これは、プログラマーまたはアプリケーションによる 2 つの手順で行われます。

- データをアライメントします。
  Fortran プログラムをコンパイルする際に、-align array64byte (Linux*)、/align:array64byte (Windows*) オプションを使用して、ほとんどの配列の先頭アドレスを 64 バイト境界のメモリーアドレスにできます。C/C++ プログラムでは、メモリー割り当てに __mm_maloc(n, 64) 関数などを使用して 64 バイトにアライメントされたポインターを要求します。データ・アライメントの詳細については、https://www.isus.jp/products/c-compilers/data-alignment-to-assist-vectorization/ を参照してください。

- 適切な句、プラグマ、およびディレクティブを使用してコンパイラーにアライメント情報を使えます。

-O3 -xcore-avx512 -qopt-prefetch[=n] (Linux*)、/O3 /QxCORE-AVX512 /Qopt-prefetch[:n] (Windows*) オプションを指定して、コンパイラーによるソフトウェア・データ・プリフェッチを有効にできます。ここで、n には 0 (プリフェッチを行わない) から 5 (最大限にプリフェッチを行う) の値を指定できます。n=5 を指定すると、コンパイラーはループ内のインデックスやストライドを使用したロード/ストアに対し、ハードウェア・プリフェッチを無視して積極的にプリフェッチを行います。n=2 を指定すると、コンパイラーによるプリフェッチの数を減らし、ハードウェア・プリフェッチャーがうまく処理できないとコンパイラーが判断した直接メモリーアクセスに対してのみプリフェッチが生成されます。特定のアプリケーションに対して最適なプリフェッチの用法を調査するため、n=2 から 5 の値を試してみることを推奨します。また、-qopt-prefetch-distance=n1[,n2] (Linux*)、/Qopt-prefetch-distance: n1[,n2] (Windows*) オプションを使用して、アプリケーションのパフォーマンスをさらに細かく調整することができます。

- n1 には次の値が有効です: 0,4,8,16,32,64。
- n2 には次の値を指定します: 0,1,2,4,8。

ループカウントをコンパイラーに伝えることができないと、ホットスポット中で実行時に判明した比較的少ないループ回数の入れ子は、インテル® AVX-512 のパフォーマンス引き出せないことがあります。多くの場合、コンパイラーはループ回数、ループのストライド、および配列範囲 (Fortran の多次元配列など) を知ることで適切なコードを生成し、高いパフォーマンスを発揮することが可能です。それができない場合、ループに対し #ptagma loop_count を追加するのが有用であることがあります。

-ipo (Linux*)、/Qipo (Windows*) オプションを使用して、プロシージャー間の最適化を有効にできます。このオプションは、アプリケーションのすべてのソースファイルに、またはホットスポットを含むソースファイルを選択して最適

化を適用できます。IPO は、複数のソースファイルにまたがるインライン展開や、その他のプロシージャー間の最適化を可能にします。特定の状況では、このオプションはコンパイル時間とコードサイズを大幅に増やすことがあります。コンパイラーによるインライン展開を制御するには、-inline-factor=n (Linux*)、/Qinline-factor:n (Windows*) オプションを利用できます。デフォルトで n は 100 です。これは、100% または 1 つのスケーリング要素を示します。例えば、200 と指定した場合、上限を定義するすべてのインライン展開オプションの値に係数 2 が掛けられ、より多くのインライン展開を有効にします。

-prof-gen と -prof-use (Linux*)、/Qprof-gen と /Qprof-use (Windows*) オプションを使用してプロファイルに基づく最適化 (PGO) を有効にできます。一般に、PGO を使用すると IPO の効率を高めます。-fp-model name (Linux*)、/fp:name (Windows*) オプションは、浮動小数点結果のパフォーマンス、精度、および一貫性間のトレードオフを制御します。name のデフォルトは fast=1 です。fast=2 に変更すると、精度と一貫性を少し犠牲にして、より積極的な最適化を可能にします。name に precise を指定すると、浮動小数点データの精度に影響する最適化を無効にします。name に double、extended、または source を指定すると、それぞれの精度で丸めを行います (double 53 ビット、extended 64 ビット、source ソースで定義)。浮動小数点の一貫性と再現性が必要な状況では、-fp-model precise -fp-model source (Linux*)、/fp:precise /fp:source (Windows*) オプションが推奨されます。

数学ライブラリー関数の精度を設定するには、-fimf-precision=name (Linux*)、/Qimf-precision=name (Windows*) オプションを使用できます。デフォルトでコンパイラーは、数学ライブラリー関数を呼び出すときに medium 精度を使用します。指定可能な name の値は、high、medium そして low です。精度を低下させると、特にベクトル化されたコードのパフォーマンスが向上する可能性がありますが、その逆もあり得ます。-[no-]prec-div および -[no-]prec-sqrt (Linux*)、/Qprec-div[-] および /Qprec-sqrt[-] (Windows*) オプションは、浮動小数点除算と平方根の精度を向上 [低下] させます。パフォーマンスが低下 [向上] することがあります。浮動小数点オプションの詳細は、https://www.isus.jp/products/c-compilers/consistency-of-floating-point-results/ を参照してください。

-[no-]ansi-alias (Linux*)、/Qansi-alias[-] (Windows*) オプションは、ANSI および ISO C 標準のエイリアシング規則を有効 [無効] にします。Linux* ではデフォルトでこのオプションが有効化されますが、Windows* では無効になっています。特に Windows* 上の C++ プログラムでは、/Qansi-alias オプションを追加してコンパイラーが最適化を適用できるようにします。この最適化には、ANSI 標準の型ベースのディスアンビグーションなどが含まれます。

最適化レポートで、コンパイル時間を短縮するために一部の最適化が無効化されていることが報告されている場合、-qoverride-limits (Linux* のみ) オプションを使用して無効化を上書きして最適化を適用できます。特に大きなボディーを持つ関数呼び出しを行うアプリケーションでは、重要になることがあります。このオプションを追加すると、コンパイル時間が増加し、コンパイル時に使用されるメモリーも劇的に増えることがあります。

以下は、ファイン・チューニング向けの最適化でループレベルの制御に使用可能な例を示しています。コンパイラーのレポートで報告された特定の変換を無効にする方法を含みます。

- #pragma simd reduction(+:sum)
  レープ内のベクトル演算で変数 sum に + にリダクション操作が行われることをコンパイラーに指示します。

- #pragma loop_count min(220) avg (300) max (380)
  Fortran シンタックス: !dir$ loop count(16)

- #pragma vector aligned nontemporal
- #pragma novector // ベクトル化を抑制
- #pragma unroll(4)
- #pragma unroll(0) // ループアンロールを抑制
- #pragma unroll_and_jam(2) // 外部ループの前で
- #pragma nofusion
- #pragma distribute_point
  for ループの制御文の直後に記述されると、そのループの分割は抑制されます。
  Fortran シンタックス: !dir$ distribute point
- #pragma prefetch *:<ヒント>:<距離>
  ループ中のすべての配列に一定のプリフェッチ距離を適用します。

- #pragma prefetch <変数>:<ヒント>:<距離>
  それぞれの配列を細かく制御します。
- #pragma noprefetch [<変数>]
  プリフェッチを無効にします [特定の配列に対し]。
- #pragma forceinline (recursive)
  関数呼び出しの前に記述すると、コンパイラーに対する呼び出しチェーン全体を (再帰的に) インライン展開するヒントとなります。

第 19 章
暗号化と有限体の算術演算の強化

Ice Lake† Client マイクロアーキテクチャーから、暗号化フローと有限フィールド演算の高速化のため追加された新しい命令拡張が利用できます。新しい ISA 拡張には、ベクトル AES、VPCLMULQDQ、ガロア体新命令 (Galois Field New Instructions - GFNI)、および VPMADD52 として知られる AVX512_IMFA が含まれます。この節では、これらの新しい命令拡張について説明と例を示し、これまでのコード実装との簡単な比較を示します。

命令の完全な定義については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』を参照してください。

インテルは、最も一般的な暗号化アルゴリズムのサポートを実装し、主なパブリック・ライブラリーや一般的に使用されるソフトウェアをサポートしています。以下の節では、これらの命令について簡単に説明します。新しい命令の適切な使用法とシステムでの応用に関する詳細は、http://goto.intel.com/cryptowp (英語) の論文を参照してください。

19.1 ベクトル AES

ベクトル AES 拡張機能は、従来のインテル® AES New Instructions (インテル® AES-NI)[30]のベクトル化をサポートします。新しい命令は、単一命令内の入力で最大 4 ブロックの並列実行をサポートします。VAESENC、VAESENCLAST、VAESDEC、および VAESDECLAST 命令の拡張 ISA は、関連する AES 操作モードと複数バッファー実装のパフォーマンスを高速化することを目的としています。これらの新しい命令セットは、インテル® AES-NI を使用する従来のコードと比較して、AES モードを最大 3.3 倍高速化します。

以下は、AT&T アセンブリーで実装された操作コードの AES-ECB モードの一部であり、従来のインテル® AES-NI とベクトル AES を比較しています。

[30] https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf (英語)

例 19-1 従来のインテル® AES-NI とベクトル AES

従来のインテル® AES-NI – AES ECB 暗号化	**ベクトル AES – AES ECB 暗号化**

// rcx = キー拡張へのポインター
movdqa 16*1(%rcx), %xmm9
// xmm1 - xmm8 – AES の 8 ブロック
// 8 ブロック AES の最初のラウンド
aesenc %xmm9, %xmm1
aesenc %xmm9, %xmm2
aesenc %xmm9, %xmm3
aesenc %xmm9, %xmm4
aesenc %xmm9, %xmm5
aesenc %xmm9, %xmm6
aesenc %xmm9, %xmm7
aesenc %xmm9, %xmm8
movdqa 16*2(%rcx), %xmm9
// 8 ブロック AES の最初のラウンド
aesenc %xmm9, %xmm1
aesenc %xmm9, %xmm2
aesenc %xmm9, %xmm3
aesenc %xmm9, %xmm4
aesenc %xmm9, %xmm5
aesenc %xmm9, %xmm6
aesenc %xmm9, %xmm7
aesenc %xmm9, %xmm8
	// rcx = キー拡張へのポインター
	// zmm0 へキーをブロードキャスト
	vbroadcasti64x2 1*16(%rcx), %zmm0
	// 8 ブロック AES の最初のラウンド
	vaesenc %zmm0, %zmm1, %zmm1
	vaesenc %zmm0, %zmm2, ,%zmm2
	vbroadcasti64x2 2*16(%rcx), %zmm0
	// 8 ブロック AES の 2 番目のラウンド
	vaesenc %zmm0, %zmm1, %zmm1
	vaesenc %zmm0, %zmm2, %zmm2

ベースライン: 1x	**スピードアップ: 3.3x**

上記のコードは、従来の AES とベクトル AES で実装された 8 つの並列バッファーを使用する ECB モードで AES を実装する方法を示しています。この方法による高速化は、AES-CTR や AES-CBC など他の操作モード、および AES-GCM などの複雑なスキームにも適用できます。後者の複雑なスキームでは、GHASH (ハッシュ関数) の高速化が必要であり、これは新しい VPCLMULQDQ 命令を使用して達成できます。詳細については次の 19.2 節で説明します。

19.2 VPCLMULQDQ

キャリーレス乗算である PCLMULQDQ は、Westmere† マイクロアーキテクチャー・ベースのインテル® Core™ プロセッサー・ファミリーで最初に導入されました。Ice Lake† Client マイクロアーキテクチャー以降の新しいアーキテクチャーでは、PCLMULQDQ がベクトル化機能を持ち VPCLMULQDQ となり、従来の 4 倍の高速化を達成します。この新しい命令は、AES-GCM など現在の暗号化アルゴリズムで使用されるバイナリーフィールドの多項式乗算に適用されます。新しい命令は BIKE などに使用され、今後ポスト量子暗号化プロジェクトにも役立つと考えられます。このような用法は、VPCLMULQDQ 命令を使用する現在の重要性を示しています。命令の典型的な用例として、GHASH 計算があります。これには、512 ビットの広いレジスターを使用して、単一命令で 4 つの異なるキャリーレス乗算を実行します。ここでは、AES の主要な動作モードである AES-GCM のパフォーマンスについて説明します。

19.3 ガロア体新命令 (GALOIS FIELD NEW INSTRUCTIONS - GFNI)

ガロア体新命令は、Ice Lake† Client マイクロアーキテクチャーで新たに導入されました。新しい命令 VGF2P8MULB、VGF2P8AFFINEQB、および VGF2P8AFFINEINVQB を使用すると、インテル® AVX512 アーキテクチャー・レジスター上で GF (2^8) のベクトルおよび行列乗算を実行できます。これらの命令は、リードソロモン符号の実装から、SM4 (中国の暗号化スキーム) など異なる暗号化スキームまで幅広く利用できます。

例 19-2 SM4 GFNI 暗号化ラウンドの例

```
LAFFINE_IN:
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34

.LAFFINE_OUT:
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.globl SM4_ENC_ECB_AVX512_GFNI
SM4_ENC_ECB_AVX512_GFNI:
  vmovdqa64 .LAFFINE_IN(%rip), %zmm10
  vmovdqa64 .LAFFINE_OUT(%rip), %zmm11
  …
/* Load data swapped LE-BE in transposed way - each block's double word is found on
different AVX512 register
*/
.Rounds:
// Initial xor between the 2nd, 3rd, 4th double word to key
  vpbroadcastd 4*0(key), %zmm6
  vpternlogd $0x96, %zmm1, %zmm2, %zmm3
  vpxorq %zmm3, %zmm6, %zmm6
/* Sbox phase */
  vgf2p8affineqb $0x65, %zmm10, %zmm6, %zmm6
  vgf2p8affineinvqb $0xd3, %zmm11, %zmm6, %zmm6
/* Done Sbox , Linear rotations start xor with 1st double word input*/
  vprold $2, %zmm6, %zmm12
  vprold $10, %zmm6, %zmm13
  vprold $18, %zmm6, %zmm7
  vprold $24, %zmm6, %zmm14
  vpternlogd $0x96, %zmm6, %zmm12, %zmm0
  vpternlogd $0x96, %zmm13, %zmm7, %zmm14
  vpxord %zmm14, %zmm0, %zmm0
/* Linear part done - round complete */
```

19.4 整数融合積和演算 (AVX512_IFMA - VPMADD52)

Ice Lake† Client マイクロアーキテクチャーから、VPMADD52 命令として知られる AVX512_IFMA が導入されました。VPMADD52LUQ と VPMADD52HUQ の新命令は、512 ビット幅のレジスターで 8x52 ビットの符号なし整数を乗算し、結果の上位と下位半分を生成して、64 ビットのアキュムレーターに加算します。入力が基数 252 を使用すると仮定すると、命令はビッグナンバー乗算に指定されます。新しい命令は、RSA で広く使用されるモジュラー指数計算の高速化に使用できます。OpenSSL[31] ではすでに利用されています。

[31] https://www.openssl.org/ (英語)

第 20 章
Knights Landing† マイクロアーキテクチャーとソフトウェアの最適化

インテル® Xeon Phi™ プロセッサー 7200/5200/3200 製品ファミリーは、Knights Landing† マイクロアーキテクチャーをベースにしています。この章では、Knights Landing† マイクロアーキテクチャーを対象としたソフトウェアのコーディング手法について説明します。Knights Landing† マイクロアーキテクチャー・ベースのプロセッサーは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3C』の第 2 章の表 2-1 に記載される CPUID の DisplayFamily_DisplayModel シグネチャーを使用して検出することができます。

この章は、Knights Landing† マイクロアーキテクチャーの概要から説明を始めます。概要に続いて、Knights Landing† マイクロアーキテクチャー上で実行するソフトウェアのパフォーマンスに影響するいくつかの重要なトピックをカバーします: インテル® AVX-512 命令、メモリー・サブシステム、マイクロアーキテクチャー固有の手法、コンパイラー・オプションとディレクティブ、数値シーケンス、MCDRAM キャッシュ、およびスカラーとベクトルコーディング。

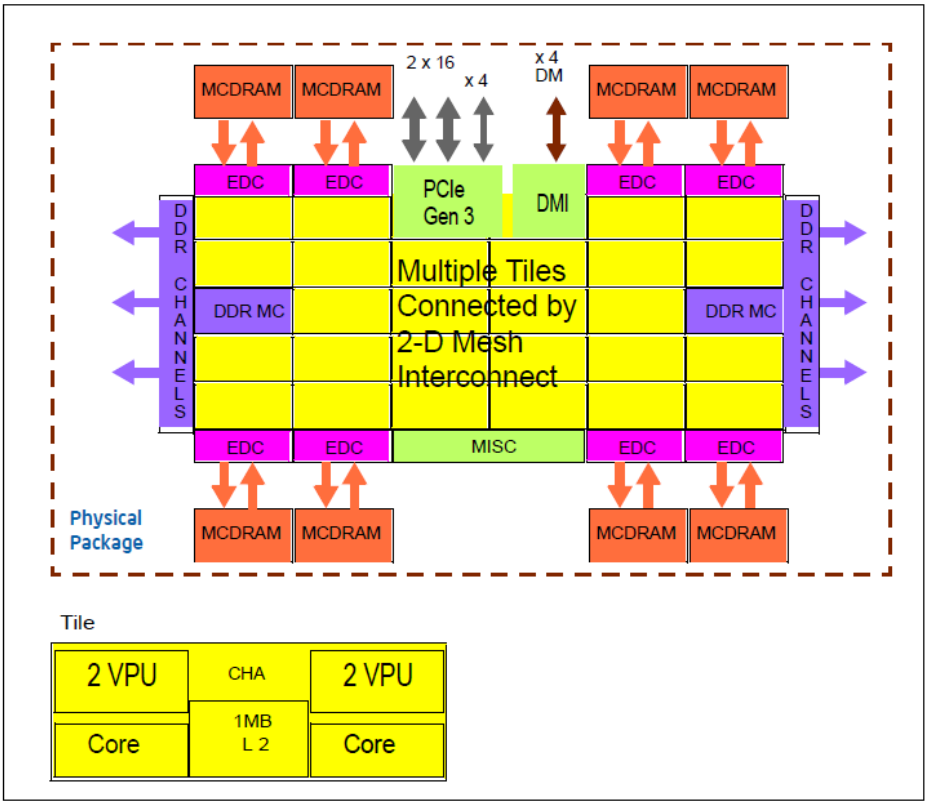


図 20-1 Knights Landing† マイクロアーキテクチャーのタイルメッシュ・トポロジー

20.1 Knights Landing† マイクロアーキテクチャー

Knights Landing† マイクロアーキテクチャーは、高度に並列化されたハイパフォーマンス・アプリケーション向けのプロセッサーおよびコプロセッサー製品ファミリーとして設計されました。Knights Landing† マイクロアーキテクチャー・ベースのインテル® Xeon Phi™ プロセッサーは以下を含みます。

- 多数のタイル。
- 2 次元 (2D) メッシュ・インターコネクトで接続されたタイル。
- IA 互換プロセッサー・コアとキャッシュ階層を含むすべてのタイルにデータを提供する高度なメモリー・サブシステム。

図 20-1 は、2 次元メッシュ・ネットワークで接続された「タイル」ユニット (プロセッサー・コアのペア) の集合、PCIe* と DMI インターフェイスを介した I/O、高帯域幅の最適化された MCDRAM をサポートするメモリー・サブシステム、および容量で最適化された DDR メモリーチャネルを示しています。

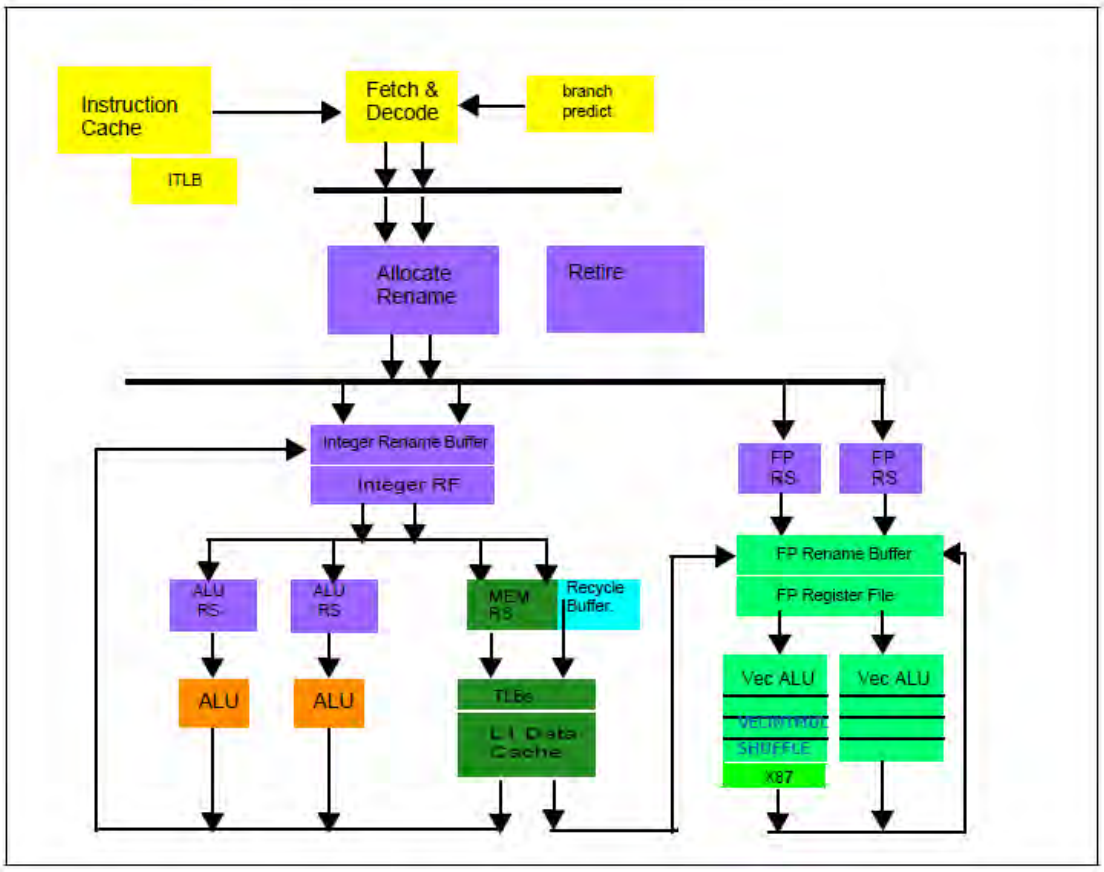


図 20-2 Knights Landing† マイクロアーキテクチャーのプロセッサー・コア・パイプラインの機能

図 20-1 はまた、各タイルの構成を示します。

- コアあたり 4 つの論理プロセッサーのインテル® ハイパースレッディング・テクノロジーをサポートする 2 つのアウトオブオーダー IA プロセッサー・コア。
- タイル内の 2 つのプロセッサー・コアで共有される 1M バイトの L2 キャッシュ。
- 2D メッシュ・インターコネクトへ各タイルを接続するキャッシング・ホーミング・エージェント (CHA)。
- 各プロセッサー・コアはまた、512 ビット、256 ビット、128 ビットのベクトルと、スカラー SIMD 命令を処理するベクトル処理ユニット (VPU) を提供します。

図 20-2 は、プロセッサー・コア内部のタイルのパイプライン (VPU パイプラインを含む) のマイクロアーキテクチャー構造を示します。

Knights Landing† マイクロアーキテクチャーのプロセッサー・コアは、次の機能を持ちます。

- 6 ワイド実行パイプライン (2 VPU、2 メモリー、2 整数) のアウトオブオーダー (OOO) 実行エンジン。具体的には、アウトオブオーダー・エンジンは次の機能によってサポートされます。
  - フロントエンドは、サイクルあたり 2 つの命令を uop (uop) にデコードできます。
  - 2 ワイドのアロケート/リネームステージ。
  - アウトオブオーダー・エンジンは、整数、メモリー、および VPU パイプラインにフィードするリザベーション・ステーション (72 エントリー) で分配されます。

- VPU は、インテル® AVX-512F、インテル® AVX-512CD、インテル® AVX-512ER、インテル® AVX-512PF、インテル® AVX、および 128 ビット SIMD/FP 命令を実行できます。
- VPU は、サイクルごとに 2 つの 512 ビット FMA 操作を行うことができ、x87 とインテル® MMX® 命令のスループットはサイクルごとに 1 つに制限されます。
- 各プロセッサー・コアは、インテル® ハイパースレッディング・テクノロジーにより 4 つの論理プロセッサーをサポートします。
- 2 つのプロセッサー・コアが、1M バイトの L2 キャッシュを共有してタイルを構成します。

20.1.1 フロントエンド

フロントエンドは、サイクルごとに命令の 16 バイトをフェッチできます。デコーダーは、サイクルごとに 24 バイト以下の 2 つの命令をデコードできます。デコーダーは、命令ごとに 1 つの uop を供給できます。命令が複数の uop にデコードされる場合 (VSVATTER* など)、マイクロコード・シーケンサー (MS) がデコーダーの命令のアライメントと MS フローの長さによって 3 ~ 7 サイクルのパフォーマンス・バブルで後続の uop を供給します。デコーダーは、分岐する分岐命令 (taken branch) に遭遇すると、若干の遅延が生じます。3 つ以上のプリフィックスを持つ命令は、複数サイクルバブルの原因になります。

フロントエンドは、アロケーション、リネーミング、およびリタイアメント・クラスターを介して OOO 実行エンジンに接続されています。uop のスケジューリングは、整数、メモリー、および VPU パイプラインにまたがって分散されたリザベーション・ステーションによって扱われます。

20.1.2 アウトオブオーダー・エンジン

リオーダーバッファー (ROB) は、72 uop の深さを持ちます。16 個のストアバッファー (アドレスとデータの両方向け) を備えています。uop の分散スケジューリングは以下を含みます (図 19-2 参照)。

- 2 つの整数リザベーション・ステーション (ディスパッチ・ポートごとに 1 つ) は、それぞれ 12 エントリーです。
- 1 つの MEC リザベーション・ステーションは 12 エントリーであり、サイクルごとに最大 2 uop をディスパッチします。
- 2 つの VPU リザベーション・ステーション (ディスパッチ・ポートごとに 1 つ) は、それぞれ 20 エントリーです。

リザベーション・ステーション、ROB、およびストア・データ・バッファーは、論理プロセッサーごとにハードウェアでパーティション化されています (プロセッサー・コアが処理する 1、2、または 4 つのアクティブな論理プロセッサー数に依存します)。リソースのハードウェア・パーティションは、論理プロセッサーがウェイクアップおよびスリープするたびに変化します。ストア・アドレス・バッファーは、論理プロセッサーごとに 2 つのエントリーが予約され、残りのエントリーは論理プロセッサー間で共有されます。

整数リザベーション・ステーションは、それぞれサイクルごとに 2 uop をディスパッチでき、アウトオブオーダーで動作します。メモリー実行リザベーション・ステーションは、スケジューラーからインオーダーで 2 uop ディスパッチされますが、任意の順番で完了できます。データキャッシュは、サイクルごとに 2 つの 64B キャッシュラインの読み込みと 1 キャッシュラインの書き込みが可能です。VPU リザベーション・ステーションは、サイクルごとにそれぞれ 2 uop ディスパッチでき、アウトオブオーダーで完了します。

Knights Landing† マイクロアーキテクチャーの OOO エンジンは、レイテンシーよりもスループットを優先した実行に最適化されています。整数レジスターへのロード (RAX など) は 4 サイクルで、VPU レジスターへのロード (XMM0、YMM1、ZMM2 または MM0) は 5 サイクルです。整数レジスターへのロードは、サイクルごとに 1 つに制限されますが、その他のメモリー操作 (ストアアドレス、ベクトルロード、プリフェッチ) は、サイクルごとに 2 つディスパッチできます。リタイアメント後のストアのコミットは、サイクルごとに 1 つの割合で行われます。データキャッシュと命令キャッシュのサイズは、それぞれ 32KB です。

最も一般的に使用される整数算術命令 (add、sub、cmp、test など) は、1 サイクルのレイテンシーでサイクルごとに 2 つのスループットです。整数パイプラインには 1 つの整数乗算器しかないため、オペランドサイズにより 3

もしくは 5 サイクルのレイテンシーがあります。整数除算器のレイテンシーは、オペランドサイズと入力値に依存します。スループットは、1 命令あたり ~20 サイクルより高速ではないと予想されます。ストアフォワードの制限に遭遇した場合、ロードフォワードへのストアには 2 サイクルのコストが掛かり、サイクルごとに 1 つフォワードできます (表 20-1 を参照)。

表 20-1 Knights Landing† マイクロアーキテクチャーのベクトル・パイプラインの特徴

Integer Instruction/operations	Latency (cycle)	Throughput (cycles per instruction)
Simple Integer	1	0.5
Integer Multiply	3 or 5	1
Integer Divide	Varies	> 20
Store to Load Forward	2	1
Integer Loads	4	1

多くの VPU 数値操作は、どちらかの VPU ポートへ 2 または 6 サイクルのレイテンシーでディスパッチできます (表 20-2 を参照)。次の命令は、単一のポートへのみディスパッチできます。

- すべての x87 数値操作。
- FP 除算または平方根。
- インテル® AVX-512ER。
- ベクトル permute/shuffle 操作。
- ベクトルから整数への移動。
- インテル® AVX-512CD 競合命令。
- インテル® AES-NI。
- ストア・セマンティクスがあるベクトル命令のストアデータ操作。

上記の操作は、2 つの VPU ディスパッチ・パイプの一方に制限されます。ベクトル・ストア・データとベクトルから整数への移動は、1 つのディスパッチ・パイプで実行されます。残りの単一パイプ命令は、その他のディスパッチ・パイプで実行されます。

表 20-2 Knights Landing† マイクロアーキテクチャーのベクトル・パイプラインの特徴

Vector Instructions	Latency (cycle)	Throughput (cycles per instruction)
Simple Integer	2	0.5
Most Vector Math (including FMA)	6	0.5
Mask Instructions (operating on opmask)	2	0.5
AVX-512ER (64-bit element)	7	2
AVX-512ER (32-bit element)	8	3
Vector Loads	5	0.5
Store to Load Forward	2	0.5
Gather (8 elements)	15	5
Gather (16 elements)	19	10
Register Move (GPR -> XMM/YMM/ZMM)	2	1
Register Move (XMM/YMM/ZMM -> GPR)	4	1
DIVSS/SQRTSS[1]	25	~20
DIVSD/SQRTSD[1]	40	~33
DIVP*/SQRTP*[1]	38	~10
Shuffle/Permute (1 source operand)[1]	2	1
Shuffle/Permute (2 source operands)[1]	3	2
Convert (from/to same width)[1]	2	1
Convert (from/to different width)[1]	6	5
Common x87/MMX Instructions[1]	6	1

注意:

1. これらの命令を実行する物理ユニットは、VPU 内のユニットの物理レイアウトの影響でスケジュールの遅延が生じることがあります。

さらに、Knights Landing† マイクロアーキテクチャーのいくつかの命令は、フロントエンドでは単一の uop としてデコードされますが、実行には 2 つのオペランドを展開する必要があります。これらの複雑な uop は、サイクルごとに 1 つのスループットでアロケーションされます。これらの命令の例を以下に示します。

- POP: 整数ロードデータ + ESP 更新
- PUSH: 整数ストアデータ + ESP 更新
- INC: レジスターへの加算 + パーシャルフラグの更新
- Gather: 2 つの VPU uop
- RET: JMP + ESP 更新
- 3 ソースの CALL、DEC、LEA

表 20-3 に Knights Landing† マイクロアーキテクチャーのキャシュリソースの特性リストを示します。

表 20-3 キャッシュリソースの特徴

	Sets	Ways	Latency	Capacity/Comments
uTLB	8	8	1	64 4KB pages (fractured)[1]
DTLB (4KB page)	32	8	4	256 4KB pages
DTLB (2M/4M page)	16	8	4	128 2MB/4MB pages
DTLB (1GB page)	1	16	4	16 1GB pages
ITLB	1	48	4	48 4KB pages (fractured)
PDE	8	4	1	Page descriptors
L1 Data Cache	64	8	4 or 5	32 KB
Instruction Cache	64	8	4	32 KB
Shared L2 Cache	1024	16	13+L1 latency	1 MB

注意:

1. μTLB と ITLB は、4KB メモリー領域のトランスレーション結果のみを保持できます。関連するページが 4KB 以上 (2MB や 1GB) の場合、バッファーはアクセスされるページの部分的なトランスレーション結果を保持します。この部分的なトランスレーションをフラクチャー・ページ (fractured page) と呼びます。

20.1.3 アンタイル

Knights Landing† マイクロアーキテクチャーでは、多くのタイルがメッシュ・インターコネクトを介して物理パッケージへ接続されています (図 20-1 を参照)。メッシュと関連するオンパッケージのコンポーネントは “アンタイル” と呼ばれます。各メッシュストップは、タイルと特定のキャッシュラインを保持する L2 キャッシュの識別子であるタグ・ディレクトリーに接続されます。物理パッケージ内に共有 L3 キャッシュはありません。タイルでミスしたメモリーアクセスは、メッシュを介して他のタイルに複製されているキャッシュを特定するため、タグ・ディレクトリーを調べる必要があります。キャッシュ・コヒーレンスには MESIF プロトコルが使用されます。他のタイルにキャッシュラインが存在しない場合、メモリーへ要求が送られます。

MCDRAM は、オンパッケージの高帯域幅メモリー・サブシステムであり、読み込みトラフィックのピーク帯域幅を提供しますが、書き込みトラフィックは (読み込みと比較して) 低い帯域幅です。MCDRAM は、オフパッケージのメモリー・サブシステム (DDR メモリーなど) より高い総帯域幅を提供します。DDR メモリーの帯域幅は、単独の書き込みや読み込みでは飽和する可能性があります。MCDRAM が達成可能な帯域幅は、読み込みと書き込みトラフィックの混在に依存しますが、DDR と同じことを行った場合およそ 4x - 6x です。

Knights Landing† マイクロアーキテクチャー でサポートされる MCDRAM の容量は、8GB または 16GB のいずれかで製品仕様に依存します。MCDRAM のピーク帯域幅は、搭載されているサイズによって異なります。MCDRAM は、DDR に比べて高い帯域幅を持っていますが、容量は少なくなります。Knights Landing† マイクロアーキテクチャーの DDR の最大容量は 384GB です。

プラットフォームの物理メモリーは MCDRAM と DDR の両方を包括します。そして、それらは複数の異なるオペレーション・モードでパーティション化できます。良く使用されるモードを次に示します。

- キャッシュモード: ダイレクト・マップ・キャッシュの MCDRAM と DDR は、ソフトウェアによってアドレス指定可能なシステムメモリーとして使用されます。
- フラットモード: MCDRAM と DDR は、個別にアクセス可能なシステムメモリーにマップされます。
- ハイブリッド・モード: MCDRAM はパーティション化され、その一部はダイレクト・マップ・キャッシュとして、そして残りの MCDRAM は直接アドレス指定可能です。DDR はアドレス指定可能なシステムメモリーにマップされます。

タイル、タグ・ディレクトリー、そしてメッシュの構成は、キャッシュ・コヒーレント・トラフィックのため、次のクラスター操作モードをサポートします。

- 全体全 (All-to-All): コア、タグ・ディレクトリーそしてメモリー・コントローラーのキャッシュライン要求は、メッシュのどこででも行うことができます。
- 4 分割 (Quadrant) : タグ・ディレクトリーとメモリー・コントローラーは同じメッシュの 4 分割をモニターし、メッシュ内のどのコアも要求することができます。
- サブ NUMA クラスター (SNC): SNC モードでは、BIOS は各 4 分割を NUMA ノードに見えるように扱います。これには、ソフトウェアが NUMA ドメインを認識し、最適なキャッシュミス・レイテンシーを実現するため、メッシュの同じ 4 分割において要求コア、タグ・ディレクトリー、およびメモリー・コントローラーを検出する必要があります。

アプリケーションのワーキングセットのクリティカルな領域が MCDRAM の容量にうまく収まる場合、それらを MCDRAM に配置してフラットまたはハイブリッド・モードを使用することで大きな利益が得られる可能性があります。一般にキャッシュモードは、Knighjts Landing† 向けにまだ最適化されておらず、ワーキングセットが MCDRAM にキャッシュできるようなコードに最も適しています。

一般に全体全モードのキャッシュミス・レイテンシーは、4 分割モードよりも長くなります。SNC モードは最良のレイテンシーを達成できます。4 分割モードはデフォルトのメッシュ構成です。SNC クラスタリングは、異なる NUMA ノードを認識するためソフトウェアから何らかのサポートを必要とします。DDR が均等に装着されていない場合 (DIMM の損失や空きなど)、メッシュは全体全クラスタリング・モードを使用する必要があります。

複数のタイルが同じキャッシュラインを読み込むと、各タイルがキャッシュラインのコピーを保持する可能性があります。同じタイル内の両方のコアがキャッシュラインを読み込むと、タイルの L2 には単一のキャッシュラインのみが保持されます。

MCDRAM がキャッシュとして構成されている場合、単一の位置でコアからアクセスされる命令とデータを保持できます。複数のタイルが同じキャッシュラインを要求すると、MCDRAM のキャッシュラインは 1 つだけ使用されます。

L1 データキャッシュは、L2 キャッシュよりも高い帯域幅と低いレイテンシーを持ちます。L2 からのキャッシュライン・アクセスは、メモリーと比べると高い帯域幅と低いレイテンシーを持ちます。

MCDRAM と DDR メモリーのレイテンシーとスループットのプロファイルは異なります。これは、キャッシュ vs フラット、またはその他のメモリーモードを選択する際に重要となります。ほとんどのメモリー構成では、DDR の容量は MCDRAM の容量よりも大幅に大きくなります。同様に、MCDRAM の容量はすべての L2 キャッシュの容量よりも大きくなります。

ワーキングセットが L2 キャッシュに収まらない場合、MCDRAM に格納する必要があります。大量のまたはアクセス頻度が低いデータ構造は、DDR に格納します。MCDRAM が “キャッシュ” または “ハイブリッド” モードに設定されている場合、Knights Landing† マイクロアーキテクチャーのハードウェアはこれを動的に行います。メモリーが “フラット” メモリーモードの場合は、データ構造は 1 つのメモリーに結合されるか、他 (MCDRAM や DDR) に割り当てられます。プログラマーは、MCDRAM へのメモリーアクセス数が最大となるよう努めるべきです。あるアルゴリズムがデータ構造を MCDRAM に配置し、それらが頻繁にアクセスされるようであれば、ワーキングセットはキャッシュに収まりきらないと考えられます。

キャッシュモードでは、最初に MCDRAM をアクセスします。キャッシュラインが MCDRAM に存在しなければ、DDR へのアクセスが開始されます。そのため、”キャッシュ” メモリーモードにおける DDR の平均アクセス・レイテンシーは、”フラット” メモリーモードよりも高くなります。

20.2 Knights Landing† マイクロアーキテクチャー向けのインテル® AVX-512 コーディングの推奨事項

インテル® AVX-512 ファミリーは、いくつかの命令セット拡張を包括しています。インテル® AVX-512 ファミリーの命令の詳細 (EVEX プリフィクス・エンコード、opmask サポートなど) と概要については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』を参照してください。Knights Landing† マイクロアーキテクチャーをベースとするインテル® Xeon Phi™ プロセッサー 7200/5200/3100 製品ファミリーは、インテル® AVX-512 基本命令 (インテル® AVX-512F)、インテル® AVX-512 指数および逆数 (インテル® AVX-512ER)、インテル® AVX-512 競合検出 (インテル® AVX-512CD)、およびインテル® AVX-512 プリフェッチ拡張をサポートしています。Knights Landing† マイクロアーキテクチャーをベースのプロセッサーでは、インテル® AVX とインテル® AVX2 命令もサポートされます。以前の世代のインテル® Xeon Phi™ コプロセッサー 7100/5100/3100 製品ファミリーでは、インテル® AVX-512、インテル® AVX2 またインテル® AVX 命令はサポートされていません。

20.2.1 Gather および Scatter 命令の利用

インテル® AVX-512F の Gather 命令は、インテル® AVX2 の Gather 命令を拡張し、512 ビット操作を行い (32 ビット・データを 16 要素または 64 ビット・データを 8 要素)、書き込みマスクとして opmask レジスターを使用して、要素を条件付きでデスティネーション ZMM レジスターに更新します。

インテル® AVX-512F の Scatter 命令は、ZMM レジスターの要素を選択的にインデックス・ベクトルで表現されるメモリー位置へストアします。デスティネーションへの条件付きストアは、opmask レジスターを使用して選択されます。インテル® AVX やインテル® AVX2 では、Scatter 命令はサポートされていません。

次のようなコードについて考えてみます。

```
for (uint32 i = 0; i < 16; i ++) {
  b[i] = a[indirect[i]];
  // ベクトル計算のシーケンス
}
```

例 20-1 インテル® AVX-512F とインテル® AVX2 における Gather の比較

AVX-512F	AVX2
vmovdqu zmm0, [rsp+0x1000] ; load indirect[] kxnor k1,k0, k0; prepare mask vpgatherdd zmm2{k1}, [rax+zmm0*4] ; compute sequence using vector register	vmovdqu ymm0, [rsp+0x1000] ; load half of index vector vmovdqu ymm3, [rsp+0x1020] ; 2nd half of indirect[] vpcmpeqdd ymm4, ymm4, ymm4 ; prepare mask vmovdqa ymm1, ymm4 vpgatherdd ymm2, [rax+ymm0*4], ymm1 vpgatherdd ymm5, [rax+ymm3*4], ymm4 ; compute sequence using vector register

VGATHER と VSCATTER を使用する場合、多くのケースですべてのマスクを 1 に設定する必要があります。これを効率良く行う命令は、自身をマスクレジスターとする KXNOR です。VSCATTER と VGATHER は、動作の最後にマスクをクリアするため、VGATHER から KXNOR へループ伝搬依存を生成します。そのため、KXNOR のソースとデスティネーションに同じマスクを使用しないようにすることが賢明です。k0 マスクがデスティネーションとして使用されることはまれであるため、"KXNORW k1, k0, k0" が "KXNOR k1, k1, k1" よりも高速である可能性があります。

インテル® AVX-512F の Gather と Scatter は、前の世代のインテル® Xeon Phi™ コプロセッサー (例 20-2 では「Previous Generation (以前の世代)」と表現されています) とは異なります。

例 20-2 インテル® AVX-512F と KNC† における Gather の比較

AVX-512F	Previous Generation Equivalent Sequence
vmovdqu zmm0, [rsp+0x1000] ; load indirect[] kxnor k1,k0, k0; prepare mask vpgatherdd zmm2{k1}, [rax+zmm0*4] ; compute sequence using vector register	vmovdqu zmm0, [rsp+0x1000] ; load indirect[] kxnor k1,k1 ; prepare mask g_loop: ; verify gathered elements are complete vpgatherdd zmm2{k1}, [rax+zmm0*4] jknzd k1, g_loop ; gather latency exposure ; compute sequence using vector register

20.2.2 拡張された逆数命令の利用

インテル® AVX-512ER 命令は、高精度の指数、逆数、および逆平方根の近似を可能にします。インテル® AVX-512ER における近似数学命令は、RCPSS の 11 ビットや VRCP14SS の 14 ビットに対し、28 ビットの精度を提供します。インテル® AVX-512ER は、ニュートンラプソンなどの反復法の実行時間を短縮します。例 20-3 に示すサンプルコードは、VRCP28SS 命令と除算を使用し単一の 32 ビット単精度浮動小数点を計算するため、ニュートンラプソン法を使用しています。両方の値はスタックから読み込まれます。いくつかの算術計算では、丸めモードのオーバーライドに注意してください。

例 20-3 32 ビット浮動小数点除算に VRCP28SS を使用する

```
vgetmantss    xmm18, xmm18, [rsp+0x10], 0
vgetmantss    xmm20, xmm20, [rsp+0x8], 0
vrcp28ss      xmm19, xmm18, xmm18
vgetexpss     xmm16, xmm16, [rsp+0x8]
vgetexpss     xmm17, xmm17, [rsp+0x10]
vsubss        xmm22, xmm16, xmm17
vmulss        xmm21{rne-sae}, xmm19, xmm20
vfnmadd231ss  xmm20{rne-sae}, xmm21, xmm18
vfmadd231ss   xmm21, xmm19, xmm20
vscalefss     xmm0, xmm21, xmm22
```

20.2.3 インテル® AVX-512CD 命令の利用

インテル® AVX-512 競合検出命令の詳細については、18.16 節「競合検出」を参照してください。

20.2.4 インテル® ハイパースレッディング・テクノロジーの利用

Knights Landing† マイクロアーキテクチャーは、それぞれのプロセッサー・コアで 4 つの論理プロセッサーをサポートします。高度にスレッド化されたソフトウェアでは、次のことを考慮する必要があります。

- 論理プロセッサーごとにコアのリソースを最大化することでスレッドのパフォーマンスを最大化します。
- プロセッサー・コアで複数の論理プロセッサーの実行を可能にすることで、コアごとのスループットを最大化します。

コアあたりのスレッド数を 2 または 4 にすると、ほとんどのアプリケーションは最も高いいパフォーマンスを発揮しますが、スレッドごとのパフォーマンスは低下します。アプリケーションが任意のスレッド数でパフォーマンスを完璧にスケールすることができるのであれば、コアあたり 4 スレッドで最も高い命令スループットを持つ可能性があります。コアあたりのスレッド数を増やすと、メモリー容量や並列処理の制約によりスケーリングの効率が制限される可能性があります。

Knights Landing† マイクロアーキテクチャーでは、いくつかのコアリソース (ROB やスケジューラーなど) は、4 つの論理プロセッサーごとにパーティション化されます。そのため、3 スレッドの構成では、スレッドが利用できる内部

リソースがコアごとに 1、2、または 4 スレッドの構成と比べ最も少なくなります。プロセッサー・コアに 3 スレッドを配置しても、2 または 4 スレッドほど良い結果は得られないでしょう。

20.2.5 フロントエンドに関する考察

フロントエンドの制限がパフォーマンスを損ねないことを保証するため、ソフトウェアは次のことを考慮すべきです。

- MSROM 命令は可能な限り回避します。典型的な例は CALL near の間接メモリー形式です。メモリーバージョンの PUSH と CALL の代わりに、レジスターへロードし、レジスターバージョンの PUSH と CALL を実行します。表 17-4 にいくつかの例を示します。
- サイクルごとにデコードできる命令バイトの合計長は、最大 16 バイトでそれぞれの命令の命令長は 8 バイト以下です。例えば、命令が 8 バイトを超えるとデコーダー 0 では、サイクルあたり 1 命令しかデコードできません。32 ビット・ディスプレースメントを使用するメモリーアドレスを持つベクトル命令は、デコーダーでパフォーマンスが制限されます。
- 複数のプリフィクスを持つ命令は、デコーダーのスループットを制限します。プリフィクスとエスケープの合計バイト数が制限に当てはまります。Knights Landing† マイクロアーキテクチャーでは、命令のプリフィクス/エスケープが 3 つを超えると、3 サイクルのペナルティーが発生します。デコーダー 0 のみがプリフィクス/エスケープ・バイトの制限を超えた命令をデコードできます。
- 各サイクルでデコード可能な分岐の最大数は 1 です。

20.2.5.1 命令デコーダー

一部の IA 命令は、複数のマイクロオペレーション (uop) フローにデコードするため、マイクロコード・シーケンサー ROM (MSROM) のルックアップが必要になります。MSROM を必要としない代替命令シーケンスを選択すると、パフォーマンスが向上します。

表 20-4 に、MSROM からデコードされる命令を置き換えることができる非 MSROM 命令のシーケンスを示します。

表 20-4 MSROM の代替命令

Instruction from MSROM	Recommendation for Knights Landing
CALL m16/m32/m64	Load + CALL reg
PUSH m16/m32/m64	Store + RSP update
(I)MUL r/m16 (Result DX:AX)	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32
(I)MUL r/m32 (Result EDX:EAX)	Use (I)MUL r32, r/m32 if extended precision not required, or (I)MUL r64, r/m64
(I)MUL r/m64 (Result RDX:RAX)	Use (I)MUL r64, r/m64 if extended precision not required

20.2.5.2 4GB 境界を超える間接分岐

フロントエンドの観点から考慮すべきの他の重要なパフォーマンス事項は、間接分岐 (間接分岐や call または ret) に対する分岐予測です。64 ビット・アプリケーションでは、分岐命令があるアドレス空間と異なる 4GB チャンクに分岐先があると (つまり、分岐命令と分岐先の仮想アドレスの上位 32 ビットが異なる場合)、間接分岐予測が失敗します。これは、アプリケーションが共有ライブラリーと分離されている場合に発生する可能性があります。レイテンシーに敏感なライブラリー呼び出しが頻繁に行われる場合、プログラマーは、コードの局所性を改善するため静的にビルドすることもできます。他の選択肢は glibc 2.23 以降を使用して、LD_PREFER_MAP_32BIT_EXEC 環境変数を設定することで、動的リンカーにすべての共有ライブラリーをアドレス空間の下位に配置することです。

20.2.6 整数実行に関する考察

20.2.6.1 フラグの利用

多くの命令には、フラグレジスターに格納される暗黙のデータがあります。これらのデータは、条件移動 (CMOVS)、分岐、さまざまな論理/算術演算 (RCL など) といった幅広い命令で利用されます。分岐条件としてよく使用される命令に比較命令 (CMP) があります。CMP 命令に依存する分岐は次のサイクルで実行できます。ADD 命令や SUB 命令に依存する分岐でも同様のことが言えます。

INC 命令と DEC 命令は一部のフラグのみ設定するため、フラグをマージする追加のマイクロオペレーション (uop) が必要になります。そのため、INC や DEC 命令は、パーシャル・フラグ・ストールを避けるため、"ADD reg, 1" や "SUB reg, 1" に置き換えるべきです。

Knights Landing† マイクロアーキテクチャーでは、8 ビットまたは 16 ビット・レジスターを操作する命令はハードウェアで最適化されていません。一般に、整数命令は 32 ビットや 64 ビット汎用レジスターオペランドを操作する方が、8 ビットや 16 ビット・レジスターを操作するよりも高速です。

20.2.6.2 整数除算

整数除算は、いくつかの数式では良く使用される操作です。しかし、ハードウェア整数除算命令の使用は、しばしばパフォーマンス上適切ではないことがあります。整数値が比較的小さいことが判明している場合 (16 ビット以下)、代替となる除算をエミュレートする高速なソフトウェア・シーケンスが知られています。除数が 2 の累乗である場合、DIV 命令の代わりに SHR (除数) と/もしくは AND (余り) 命令を使用します。定数による除算は、MUL と定数に置き換えることができます。収束しない入力値の場合、事前計算されたルックアップ・テーブルを使用することでパフォーマンスが向上する可能性があります。13.2.4 節「128 ビット整数除算を 128 ビット乗算で置き換え」と 14.5 節「ASCII 形式への数値データ変換」に、いくつかの手法の例が示されています。

コンパイラーは前述の手法を使用するか、内部ループから冗長な除算をホイストして積極的に除算命令を最小化すべきです。

20.2.7 FP およびベクトル実行の最適化

20.2.7.1 命令選択の注意事項

一般に、512 ビット命令の使用は、256 ビット命令よりも高いスループットを達成するには好都合です。これは、同様に 256 ビットと 128 ビットのベクトル命令にも当てはまります 128 ビットのインテル® SSE 命令は、等価な x87 命令を使用するよりも高いスループットを達成します。ベクトル命令拡張では x87 命令の機能 (超越関数) が提供されないことがありますが、これはベクトル命令を使用したライブラリー実装に置き換えることができます。

Knights Landing† マイクロアーキテクチャーでは、COMIS* と UCOMIS* 命令 (レガシー、VEX または EVEX エンコード) は EFLAGS を更新するため低速です。これらの命令は、インテル® AVX-512F バージョンの VCMPS* と KORTEST のより適切なシーケンスに置き換えるべきです。

例 20-4 VCOMIS* を VCMPSS/KORTEST に置き換える

```
vcmpss k1, xmm1, xmm2, imm8 ; specify imm8 according to desired primitive
kortest k1, k1
```

VCOMPRESS* などの一部の命令は、レジスターに書き込む場合は単一 uop ですが、メモリーに書き込む場合は MS フローになります。可能な限り、高速なレジスターへのストアを行う VCOMPRESS を使用してください。同様の最適化は、何らかの操作を行ってストアを実行するすべてのベクトル命令に適用されます。

Knights Landing† マイクロアーキテクチャーでは、インテル® SSE 命令とインテル® AVX 命令の混在はパフォーマンスの損失を避けるため特別な考慮が必要です。可能な限り、インテル® SSE コードを等価な 128 ビットのインテル® AVX に置き換えます。

次の条件でパフォーマンス上のペナルティーが生じます。

- 128 ビット以上のベクトル長でエンコードされているインテル® AVX 命令が、実行中のインテル® SSE 命令がリタイアする前に割り当てられる場合。
- VZEROUPPER 命令のスループットは低速です。そのため、インテル® SSE コードが実行された後にインテル® AVX へ移行することは推奨されません。VZEROALL のスループットもまた低速です。ZEROUPPER または VZEROALL 命令のどちらを使用してもパフォーマンスの損失につながります。

MASKMOVDQU と VMASKMOV のような、条件付きパックドロード/ストア命令は、要素の選択にベクトルレジスターを使用します。インテル® AVX-512F 命令は、要素の選択に opmask レジスターを使用した代替手段を提供しますが、要素選択にベクトルレジスターを使用するよりも適しています。

いくつかのベクトル数学命令は実装に VPU で複数の uop を必要とします。これによる、個々の命令のレイテンシーが、2 または 6 の標準的な計算レイテンシーを上回ります。一般に、出力/入力要素幅を変更する命令 (VCVTSD2SI など) がこのカテゴリーに分類されます。バイトとワード単位で操作を行うインテル® AVX2 命令は、32 ビットや 64 ビット単位で操作する同様の命令と比較するとパフォーマンスが低下します。

VPU のいくつかの実行ユニットは、依存関係のある uop フローのシーケンスがそれらの実行ユニットを使用する場合スケジューリングに遅延が生じます。これから分離ユニットは表 20-2 の脚注に示されています。この問題が生じると 2 サイクルバブルのコストが加算されます。VPU の分離ユニットと他のユニット間を頻繁に移行するコードは、これらのバブルによるパフォーマンスの問題が生じます。

opmask レジスターを使用する大部分のインテル® AVX-512 命令は、デスティネーションを条件付きで更新します。一般に、すべて 1 の opmask を使用する方が、ゼロ以外の値の opmask を使用するよりも高速です。ゼロ化非更新要素が選択された場合、ゼロ以外の opmask 値を使用すると、すべて 1 の opmask の命令と同等のスピードです。ゼロ以外の opmask 値とデスティネーションの非更新要素がマージされると、低速になります。

インテル® AVX の水平加算/減算命令には、等価なインテル® AVX-512 命令がありません。ソフトウェア・シーケンスを使用した水平リダクションは最良の実装です (例 20-5)。

アルゴリズムがリダクションを必要とする状況で、水平加算を行うことなくリダクションを実装する方法があります。

例 20-6 は、DGEMM 行列乗算ルーチンの内部ループのコードの一部を示し、C = A* B の密行列計算を行っています。

例 20-6 には 16 個の部分和があります。FMA 命令のシーケンスは、2 つの VPU の能力を利用して、サイクルごとに 2 FMA のスループットで、6 サイクルのレイテンシーを達成します。例 20-6 の FMA コードは、メモリーオペランドに圧縮を使用しないアドレス形式を示しています。コード・ジェネレーターが、FMA 命令長が 8 バイト未満になるように圧縮された disp8 アドレス形式を使用して最適なコードを確実に生成することが重要です。内部ループの最後で部分和は集計され、結果はメモリー上の行列 C にストアされます。

例 20-5 水平リダクションのソフトウェア・シーケンスを使用

```
    vextractf64x4 ymm1, zmm6, 1; reduction of 16
elements
    vaddps     ymm1, ymm6, ymm1
    vpermpd ymm4, ymm1,0xff
    vpermpd ymm5, ymm1,0xaa
    vpermpd ymm3, ymm1,0x44
    vaddps        xmm1, xmm1, xmm4
    vaddps        xmm3, xmm5, xmm3
    vaddps        xmm3, xmm1, xmm3
    vpsrlq     xmm1, xmm3, 32
    vaddss xmm3, xmm1, xmm3
```

```
    vextractf64x4 ymm1, zmm6, 1; reduction of 8
elements
    vaddps     ymm1, ymm6, ymm1
    valignq ymm4, ymm1,0x3
    valignq ymm5, ymm1,0x2
    valignq ymm3, ymm1,0x1
    vaddsd ymm1, ymm1, ymm4
    vaddsd ymm3, ymm5, ymm3
    vaddsd ymm3, ymm1, ymm3
```

例 20-6 Knights Landing† マイクロアーキテクチャー向けに最適化された DGEMM の内部ループ

```
    ;; matrix - matrix dense multiplication
    prefetcht0  [rdi+0x400] ;; get A matrix element into L1$
    vmovapd     zmm30, [rdi]
    prefetcht0  [rsi+0x400] ;; get B matrix element into L1$
    vfmadd231pd zmm1, zmm30, [rsi+r12]{b} ;; broadcast B elements
    vfmadd231pd zmm2, zmm30, [rsi+r12+0x08]{b} ;; displacement shown in un-compressed form
    vfmadd231pd zmm3, zmm30, [rsi+r12+0x10]{b}
    vfmadd231pd zmm4, zmm30, [rsi+r12+0x18]{b}
    vfmadd231pd zmm5, zmm30, [rsi+r12+0x20]{b}
    vfmadd231pd zmm6, zmm30, [rsi+r12+0x28]{b}
    vfmadd231pd zmm7, zmm30, [rsi+r12+0x30]{b}
    vfmadd231pd zmm8, zmm30, [rsi+r12+0x38]{b}

    prefetcht0  [rsi+0x440]     ;; pull line into the L1$
    vfmadd231pd zmm9, zmm30, [rsi+r12+0x40]{b}
    vfmadd231pd zmm10, zmm30, [rsi+r12+0x48]{b}
    vfmadd231pd zmm11, zmm30, [rsi+r12+0x50]{b}
    vfmadd231pd zmm12, zmm30, [rsi+r12+0x58]{b}
    vfmadd231pd zmm13, zmm30, [rsi+r12+0x60]{b}
    vfmadd231pd zmm14, zmm30, [rsi+r12+0x68]{b}
    vfmadd231pd zmm15, zmm30, [rsi+r12+0x70]{b}
    vfmadd231pd zmm16, zmm30, [rsi+r12+0x78]{b}
```

20.2.7.2 前世代からの組込み関数の移植

ほとんどの組込み関数はネイティブ・ハードウェアの固有の命令に対応しています。512 ビット組込み関数の中には、インテル® AVX-512F と互換性のない 以前の世代の命令セット間の違いを隠匿する構文を提供するものがあります。

しかし、以前の世代で実行するために最適化された組込み関数のコードは、マイクロアーキテクチャーが異なることから (アライメントされていないメモリーアクセス、並べ替えのコストの違い、以前の世代の制限などにより) Knights Landing† マイクロアーキテクチャー上では最適化された状態で動作しません。

以前の世代向けの組込み関数を使用してコーディングするよりも、高レベル言語 (C/Fortran) でアルゴリズムを記述して、インテル® AVX-512F をサポートするインテル® コンパイラーでコンパイルする方が最適なコードを生成できます。

20.2.7.3 ベクトル化のトレードオフを推定する

高レベル言語で記述されたループのベクトル化でインテル® AVX-512 の使用により利点が得られるかどうかは、コンパイラーおよびアセンブリーによる手動コーディングにおける最適化の重要な部分です。最もシンプルなループ構造を見積もるには、ループカウントだけをベースにします。例えば、ループカウント 4 以下ではスカラーコードを上回るパフォーマンスを得るのは難しいかもしれません。インテル® AVX-512 において、ベクトル化を考慮するべき最小のループカウントは 16 です。つまり 16 以上では利点があります。

複雑なループ構造のベクトル化のトレードオフを見積もるには、多くの考察が求められます。この節の残りのセクションでは、ループ本体の構成を調査する分析アプローチにより、表 20-5 に示す基本的な操作のコスト見積もりを使用してベクトル化とスカラーコードを比較してトレードオフを明らかにします。

表 20-5 Knights Landing† マイクロアーキテクチャー向けのベクトル化を見積もるサイクル・コスト・ビルディング・ブロック

Operation	Cost (cycles)	Example Code Construct
Simple scalar math	1	A*B+C, or A+B, or A*B
Load (split cacheline)	1 (2)	A[i] /* load reference to an array element */
Store (split cacheline)	1(2)	A[i] = 2;
Gather (Scatter) 8 elements	15 (20)	A[key[i]]
Gather (Scatter) 16elements	20 (25)	A[key[i]] ;
Horizontal reduction	30	sum += A[i]
Division or Square root	15	A/B

最初に、コスト見積もり方法を説明するため、簡単なループを考えてみます。

```
for (i=0; i<N; i++) { sum += a[i]*K + b[i]; }
```

ループ本体の基本操作は次のように構成されます。

- 反復ごとに 2 つのロード (a[i]、b[i])。
- 反復ごとに 1 つの FMA。
- スカラーバージョン: ループ反復ごとの累積。ベクトルバージョン: ループの最後で水平リダクション。

スカラーコードの N 回の総コストは 4N です。一方、メインループとリマインダー・ループ (N が 8 の倍数でない場合) の両方がベクトル化されると想定した場合、64 ビット・データ要素をインテル® AVX-512 を使用してベクトル化したコードの総コストは 3 * Ceiling(N/8) + 30 です。つまり、ベクトル化から利益を得るには少なくとも 9 回以上のトリップカウントが必要です。

GATHER 命令の利点を活用する不規則なアクセスパターンからのデータをフェッチする別の例を考えてみます。

```
for (i=0; i<N; i++) {c[i] = a[indir[i]] * K + b[i]; }
```

ループ本体の基本操作は次のように構成されます。

- 反復ごとに 2 つのロード (indir[i]、b[i])。
- 反復ごとに 1 つの FMA。
- 反復ごとに 1 つのストア。
- スカラーバージョン: 3 回の反復ごとにロード。ベクトルバージョン: 8 反復ごとに 1 つの GATHER。

スカラーコードの N 回の総コストは 5N です。一方、ベクトル化されたコードの総コストは、19* Ceiling(N/8) です。N < 4 の場合スカラーコードのほうが高速です。

前の例よりも不規則なアクセスパターンからのデータをフェッチする例を考えてみます。

```
for (i=0; i<N; i++) {c[i] = a[ind[i]]*K + b[ind[i]]; }
```

- 反復ごとに 1 つのロード (indi[i])。
- 反復ごとに 1 つの FMA。
- 反復ごとに 1 つのストア。
- スカラーバージョン: ループ反復ごとにさらに 2 つのロード。ベクトルバージョン: 8 反復ごとに 2 つの GATHER。

スカラーコード向けの N 回の総コストはまだ 5N です。一方、ベクトル化されたコードの総コストは、(15*2 + 3)* Ceiling(N/8) = 33* Ceiling(N/8) です。わずかなベクトル化の利益を得るのにかなり大きなトリップカウントを必要とします。

次の例では、1 つの不規則なアクセスパターンと水平リダクションからのデータをフェッチする状況を考えてみます。

```
for (i=0; i<N; i++) {sum += a[ind[i]]*K + b[i]; }
```

スカラーのコストはまだ 5N です。ベクトル化のコストは、19*Ceiling(N/8) + 30 になりました。N <= 13 の場合スカラーコードのほうが高速です。

除算を伴うスキャッターを例に考えてみます。

```
for (i=0; i<N; i++) {c[ind[i]] = a[i] / b[i]; }
```

スカラーコストは (15+4)*N です。ベクトル化のコストは、(15+20+3)*Ceiling(N/8) です。N > 2 の場合ベクトル化の利点があります。

スキャッターの後にギャザーが続く場合を考えてみます。

```
for (i=0; i<N; i++) {b[ind[i]] = a[ind[i]]; }
```

スカラーコードのコストは 3*N で、ベクトルコードのコストは (15+20+1)*Ceiling(N/8) です。ベクトル化の利点はありません。

miniMD として知られるさらに複雑なループ本体のコードを考えてみましょう。

```
for (int k = 0; k < numneigh; k++) {
  int j = neighs[k];
  double rsq = (xtmp - x[3*j])^2 +
    (ytmp - x[3*j+1])^2 +
    (ztmp - x[3*j+2])^2;
  if (rsq < cutforcesq) {
    double sr2 = 1.0/rsq;
    double sr6 = sr2*sr2*sr2;
    double force = sr6*(sr6-0.5)*sr2;
    res1 += delx*force;
    res2 += dely*force;
    res3 += delz*force;
  }
}
```

if 節について考える前に、1 つのロード、3 つのギャザー (x[] のストライドロード)、3 つの減算、そして 3 つの乗算があります。if 節の内部では、1 つの除算、8 つの数学計算、そして 3 つの水平リダクションが行われています。スカラーのコストは、10 * numneigh + 23 * numneigh * percent_rsq_less_than_cutforcesq です。そしてベクトルのコストは、(52 + 23) * ceiling(numneigh / 8) + 3 * 30 です。(numneigh < 6) が成立するか、コンパイラーが if 節はほとんど実行されないという高い確証があるなら、スカラーコードを使用することは理にかなっています。

多くのコンパイラーは、ベクトル化されたループを生成し、余剰操作を処理するためリマインダー・ループを使用します。言い換えると、ベクトル化されたループは floor(N/8) 回実行され、リマインダー・ループが (N mod 8) 回実行されることになります。この場合、一次ループをベクトル化するかどうか判断するため、ceiling の変わりに floor を使用するように変更します。リマインダー・ループがあり、ループの最大カウントが明白である場合、ベクトル幅は 1 つ少なくなります。N が不明である場合、ベクトル幅の半分を N に設定するのがもっとも容易です (倍精度の ZMM ベクトルでは 4)。

より洗練された分析が可能です。例えば、表 20-5 に示す 1 サイクルの簡単なビルディング・ブロックの数学操作は、依存性チェーンや長いレイテンシーの操作によってブロックされない一般的な命令シーケンスです。コスト表の内容を拡大することでより複雑な状況をカバーできます。

20.2.8 メモリー最適化

20.2.8.1 データ・アライメント

キャッシュライン境界をまたぐアドレスのデータアクセスには、わずかなパフォーマンスの利点があります。メモリーをストリームするアクセスパターンでは、64 バイト・アクセスごとに確実にキャッシュライン境界にアライメントすることで、キャッシュライン分割を避けることができます。メモリーの 32 バイトを YMM へロードする場合、opmask 値を使用してメモリーの 64 バイトにアクセスし上位 32 バイトをマスクしてはなりません。

4K バイト境界にまたがるメモリー参照は、パフォーマンス上の高いコストを被ります。512 ビット命令を使用したメモリーをストリームするアクセスパターンのスループットは、4K バイト境界を超えても高いレートをもたらします。64 バイトにアライメントすると、ページ分割のペナルティーも避けることができます。

ページ境界をまたがった次のコード空間までの距離を推測することが可能な場合、現在の読み込みストリームの数回前の反復で PREFETCH1 (L2 へ) を挿入することができます。これにより、ページ変換を事前に開始することで、L2 ハードウェアのプリフェッチャーが次のページのフェッチを開始できるようになります。

ギャザーとスキャッターのいくつかのアクセスパターンは、常に連続したアドレスのペアを持ちます。典型的な例として、実部と虚部が連続して配置される複素数があります。また、w、x、y、および z の要素が連続するのも一般的な例として上げられます。値が 32 ビットであれば、連続する 64 ビット要素の半分をギャザーまたはスキャッターする方が高速です。値が 64 ビットである場合、ロードしてギャザー操作を行う代わりに 128 ビットの値をインサートする方が高速です。

20.2.8.2 ハードウェア・プリフェッチャー

タイルには 2 タイプの HW プリフェッチャーがあります。命令ポインター・プリフェッチャー (IPP) はプロセッサー・コア内にあり、データキャッシュ内のすべてのアクセスとアクセスを要求した命令を分析します。このプリフェッチャーは、キャシュ可能なページでのストライド・アクセス・パターンを検出すると、L1 キャッシュへハードウェア・プリフェッチを挿入しようと試みます。IPP は 4K ページ境界を超えることはできません。IPP はテーブルへのインデックスを作成するため、命令アドレスと論理プロセッサーを使用します。これは、コンパイラーがメモリーをアクセスする命令が異なるテーブルのエントリーとなるように、大きなループ (> 256 バイト) に NOP を挿入する可能性があるためです。

L2 ハードウェア・プリフェッチャーは、ストリーミング・アクセス・パターンを識別して、48 のアクセスパターンを追跡します。ストリーミング・アクセス・パターンは、昇順または降順で連続したキャッシュラインを参照します。L2 で検出されるストライドは常に +/- 1 キャッシュラインです。要求を起こした論理プロセッサーにかかわりなく、48 個の検出器が割り当てられます。それぞれの検出器は、4KB 領域内で行われたアクセスを監視します。ストリームが検出さ

れると、ストリームの後の要素向けのハードウェア・プリフェッチは L2 に送られ、アクセスがミスするとメモリーを参照します。ハードウェア・プリフェッチャーは、4 KB アドレス境界をまたぐストリームにアクセスしません。同じ 4KB 領域内で複数のアクセスパターンが行われると、検出器は混乱してストリームの検出に失敗します。

20.2.8.3 ソフトウェア・プリフェッチ

Knights Landing† マイクロアーキテクチャーは、アウトオブオーダー実行をサポートします。一般に、これは 以前の世代のインオーダー・マイクロアーキテクチャーよりもキャッシュミスをうまく隠匿できます。従って、プログラマーはソフトウェア・プリフェッチを積極的に使用するのを避けるべきです。

20.2.8.2 節で示した 2 つのハードウェア・プリフェッチャーは、ほとんどのストリーミングと短いストライド・アクセス・パターンを検出します。アクセスパターンがストリーミングであるなら、4K バイト・ページ境界を超えるソフトウェア・プリフェッチには利点があります。アクセスパターンが不明でストリーミングしない場合、ソフトウェア・プリフェッチが有効である可能性があります。アクセスパターンが比較的大きなストライド (> 256 バイト) で、IPP が 4 KB 境界をまたがってフェッチしない場合は特に有効です。ソフトウェア・プリフェッチは、PMH (ページ・ミス・ハンドラー) が TLB を埋めるためウォークスルーを行い、早期にメモリー参照を開始できます。

一般的に、L2 キャッシュへのソフトウェア・プリフェッチは、L1 キャッシュへのソフトウェア・プリフェッチよりも多くの利点があります。L1 へのソフトウェア・プリフェッチは、キャッシュラインを完全にフィルするまでハードウェアの重要なリソース (フィルバッファー) を消費します。L2 へのソフトウェア・プリフェッチは、そのようなリソースを消費しないためパフォーマンスへの悪影響は少ないと推測されます。L1 ソフトウェア・プリフェッチを使用する場合、ハードウェア・リソースの占有時間を最小化するため、L1 ソフトウェア・プリフェッチが L2 キャッシュにヒットするように考慮することを強く推奨します。

無効なアドレスからのソフトウェア・プリフェッチ命令は、リタイアメント・スロットを消費するためパフォーマンスに悪影響を与えます。無効なアドレスからのプリフェッチや、ユーザコードから OS 特権レベルへの移行のパフォーマンス上のペナルティーは非常に大きくなります。パフォーマンス監視イベント NUKE.ALL は、コードに影響を与える指標を示します。

20.2.8.4 メモリー実行クラスター

MEC が uop をアウトオブオーダー実行するには制限があります。メモリー uop は、スケジューラーからインオーダーでディスパッチされますが、任意の順番で完了できます。メモリー命令の順番を再配置することで MEC の能力をうまく利用できれば、パフォーマンスが向上する可能性があります。

例 20-7 に、2 つの配列 a[] と b[] に 2 つのリードストリームでアクセスするメモリー命令シーケンスの順番の影響を示します。例 20-7 の左のリストは、最適なシーケンスであり、b[] からの 2 番目のベクトルロードはサイクル N+5 でディスパッチされ L1 キャッシュにヒットすると想定されます。右のリストは、自然なメモリー命令の順番であり、2 番目のベクトルロードはサイクル N + 8 でディスパッチされます。

右のリストは左のリストよりも多くのレジスターを使用しています。ポインターロードが L1 をミスすると、リスト中のコメントに示すように左のリストの利点が大きくなります。

例 20-7 MEC 向けメモリー命令の順番

```
movq     r15, [rsp+0x40] ; cycle N (load &a[0])
movq     r14, [rsp+0x48] ; cycle N+1 (load &b[0])
vmovups  zmm1, [r15+rax*8] ; executes in cycle N+4
vmovups  zmm2, [r14+rax*8] ; cycle N+5
```

```
movq     r15, [rsp+0x40] ; cycle N (load &a[0])
vmovups  zmm1, [r15+rax*8] ; executes in cycle N+4
movq     r15, [rsp+0x48] ; cycle N+4 (load &b[0])
vmovups  zmm2, [r15+rax*8] ; cycle N+8
```

マシン上で多くのロードが実行される場合、追加で整数レジスターの予約を要求することなく、ポインターのロードと逆参照間にいくつかのメモリー参照が存在するように、ポインターロードの巻き上げが可能であるかもしれません。

20.2.8.5 ストア・フォワーディング

Knights Landing† マイクロアーキテクチャーにおける整数実行と MEC 向けのストア・フォワーディングの制限は、Silvermont† マイクロアーキテクチャーと同じです。ここでは、VPU とフォワーディングの制限について説明します。

ベクトル、x87、およびインテル® MMX® 命令のロードとストアは、ストアとロードが同じメモリーアドレスを持ち、ロードがストア幅よりも小さい場合にフォワードできます (ZMM0、YMM1、XMM2、MM3 および ST4)。VPU ストアは整数ロードへフォワードできません。また、整数ストアを VPU ロードへフォワードすることもできません。どちらの場合も、ロードはストアのリタイアメント後メモリーから値を取得するまで待機します。

opmask を使用するベクトルストアはフォワードされません。アルゴリズムがそのような機能を必要とする場合、レジスター内の値をマージして、その後条件 opmask を使用せずにストアすることで利点が得られる可能性があります。ロードには、マージされた値がフォワードされます。

20.2.8.6 ウェイとセットの競合

メモリー階層は、アクセスされるアドレスに基づいて要求の転送を決定します。L1 データキャッシュは、使用するキャッシュのセットを特定するため、アドレスの 11:6 ビットを使用します。フォワードロジックは、アクセスのサイズからフォワードの可能性やロードとストアの衝突を識別するため、アドレスの 11:0 ビットを使用します。多くの衝突がある場合、パフォーマンスが低下する可能性があります。

多くの動的メモリー割り当てルーチン (OS とコンパイラーに依存) は、同じ底部 12 ビットで大きなメモリー領域を開始します。アクセスパターンが同一の形状 (要素のサイズや次元) と類似したインデックスを持つ多くの配列にアクセスする場合、セットの競合によりパフォーマンスが大幅に低下する可能性があります。メモリーアクセスのビット [11 .. 6] が異なっていると、セットの競合を避けることができます。例えば、以下を考えてみます。

```
a = malloc(sizeof(double) * 10000);
b = malloc(sizeof(double) * 10000);
for (i=0; i < 10000; i++) {
  a[i] = b[i] + 0.5 * b[i-1]);
}
```

ほとんどの OS では、a[] と b[] の有効アドレスは同じ下位 12 ビットを持つ可能性があります。つまり、(a & 0xfff) == (b & 0xfff) になります。ループ内で以下が発生する可能性があります。

- 反復 N の a[i] と b[i] が衝突
- 反復 N-1 の a[i] と 反復 N の b[i-1] が衝突

動的配列をオフセットするには、いくつかの方法があります。次に例を示します。

- キャッシュラインの量に応じて、malloc から得られたベースポインターをオフセットします。
- カスタム化された malloc() ルーチンを使用します。
- 異なるアライメント (2 の累乗: 64、128、256、512 など) を得るため、それぞれの動的な割り当てでアライメント・ディレクティブと posix_memalign() ルーチンを使用します。

Leslie3D として知られる HPC ワークロードは、アライメントの問題の影響を受けます。

20.2.8.7 ストリーミング・ストアと通常のストア

メモリーに書き込まれるデータがロードによってしばらくアクセスされないことが予想される場合、ストリーミング・ストアまたは通常のストア (ライトバック) のどちらを選択するか決定する良い例と言えます。Knights Landing† マイ

クロアーキテクチャーでは、"フラット" メモリーモードが選択されている場合、ストリーミング・ストアが望ましいでしょう。20.1.3 節を参照してください。

MCDRAM がキャッシュモードに設定されている場合、MCDRAM キャッシュに収まるようにデータが書き込まれていると、通常のストアがうまく動作します。実際には、両方のオプションを試すことでアプリケーションに適したパフォーマンスがもたらされるでしょう。

20.2.8.8 コンパイラー・オプションとディレクティブ

Fortran 90 構文を使用する場合、適切であれば Fortran プログラマーは CONTIGUOUS 属性を使用すべきです。さもないと、コンパイラーは受け取る配列が連続していないと仮定し、ベクトルロードとストア命令を VGATHER と VSCATTER 命令に置き換える可能性があります。これはパフォーマンスに悪影響を与えます。

インテル® コンパイラーを使用する開発者は、さまざまなプラグマやディレクティブを使用してコードに注釈を加えることができます。有効な注釈として、LOOP_COUNT、SIMD、および UNROLL が挙げられます。これらのプラグマやディレクティブのドキュメントを参照して、適切な場所で使用してください。コンパイラーは、ベクトル化のコストを評価するためより多くの情報が得られる場合に優れたコードを生成できます。

インテル® コンパイラーを使用する場合、コンパイラー・オプション "-xMIC-AVX512" で、Knights Landing† マイクロアーキテクチャーをターゲットとします。

20.2.8.9 ダイレクト割り当て MCDRAM キャッシュ

MCDRAM がキャッシュモードに設定されている場合、MCDRAM キャッシュはメモリー帯域幅を増加させる便利な方法です。メモリー側のキャッシュとして、自動的に使用されたデータをキャッシュし、DDR メモリーよりもはるかに高い帯域幅を提供します。

MCDRAM キャッシュは、ダイレクト・マップ・キャッシュです。これは、複数のメモリー位置がキャッシュ内の単一の場所にマップされることを意味します。そのため、メモリー帯域幅に影響を受けるプログラム向けの簡単な最適化は、MCDRAM をキャッシュモードとして有効にすることです。数 GB のメモリーを頻繁に使用する一部のアプリケーションでは、最大 4 倍のパフォーマンス向上を達成できました。これは、ソースコードを変更することなく、DDR からキャッシュモードの MCDRAM へ移行するだけで大幅にパフォーマンスが向上できるため、最初に試みるべき最適化です。

キャッシュを有効にすることでパフォーマンスが低下するいくつかの状況があります。1 つは、MCDRAM キャッシュがアクセスされたワーキングセットを保持できない場合です。アプリケーションが 64GB メモリーを再利用することなくストリームすると、MCDRAM キャッシュのチェック (およびミス) のため、DDR メモリーへのアクセスと比較してメモリーアクセスのコストは増加します。

MCDRAM ダイレクト・マップ・キャッシュへのデータのキャッシングでは、リニアアドレスではなく物理アドレスを使用します。リニア/仮想アドレス空間でアドレスが連続していても、OS が割り当ておよび管理する物理アドレスは連続しているとは限りません。これは、MCDRAM の大部分を使用する場合、キャッシュ競合を引き起します。この競合は、利用可能なピークメモリー帯域幅を減少させます。これは、OS のページ割り当てが実行ごとに異なるため、実行するたびに変わります。Knights Landing† マイクロアーキテクチャーのパフォーマンス・モニタリング・ハードウェアは、MCDRAM のキャッシュヒット率を計算するイベント UNC_E_EDC_ACCESS を提供しており、この問題を診断する上で役立つことがあります。

MCDRAM キャッシュが有効である場合、タイル内のキャッシュ (L1 または L2 キャッシュ) のすべての変更されたキャッシュラインは、MCDRAM キャッシュにエントリーを持つ必要があります。キャッシュラインが MCDRAM から追い出されている場合、タイル内のキャッシュの変更されたラインのデータはメモリーにライトバックされ、キャッシュ状態は共有へ移行します。頻繁にリードとライトされるラインのペアが、同じ MCDRAM セットにエイリアスされることはまれです。MCDRAM をキャッシュモードで使用すると、通常タイル内のキャッシュにヒットするライトのペアが、余分なメッシュ・トラフィックを発生させる原因となります。これは、そのスレッドのペアが、チップ内のほかのスレッ

ドよりも遅くなる原因となります。リニアから物理アドレスへのマッピングは一定でなく、実行ごとに振る舞いが異なるため診断が困難になります。

例えば、2 つのスレッドがプライベートなスタックをそれぞれリードおよびライトすると、この問題が発生します。概念的には、通常リード/ライトされるすべてのデータの場所にリード/ライトできるべきですが、スタックへのレジスター退避が最も頻繁に発生するケースです。スタックが物理メモリーで 16GB の倍数 (または、総 MCDRAM キャッシュサイズ) にオフセットされている場合、同じ MCDRAM キャッシュセットで衝突する可能性があります。実行時にすべてのスレッドのスタックを物理メモリー領域に連続して割り当てることを強制することで、この問題を回避できます。

Knights Landing† マイクロアーキテクチャーには、セット衝突の発生頻度を低減するハードウェア機能があります。特定のノード上で、このシナリオに遭遇する可能性は非常に低いでしょう。この問題を特定する手がかりは、同じチップ上のスレッドのペアは、プログラム全体を通して他のスレッドよりもかなり低速になることです。パッケージ内のどのスレッドコアが衝突を起こすかは、実行ごとに異なり、またまれにしか発生せず、"キャッシュ" メモリーモードが有効である場合のみに限定されます。ユーザーはシステム上でこの問題に遭遇しない可能性もあります。

付録 A
アプリケーション・パフォーマンス・ツール

インテルが提供するアプリケーション・パフォーマンス・ツールを活用すると、インテル® アーキテクチャー (IA) ベースのプロセッサーのパフォーマンスを引き出すことができます。この付録では、これらのツールを紹介し、その機能について説明します。これらのツールを使用すると、アセンブリー・コードを記述しなくても最も効率の高いプログラムを開発できます。

次のパフォーマンス・ツールが利用できます。

- **コンパイラー**
  - インテル® C++ コンパイラー: ハイパフォーマンスな最適化された C および C++ クロスコンパイラーはインテル® HD グラフィックスやインテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーへの計算集約型のコードのオフロード機能とインテル® Cilk™ Plus や OpenMP* を使用した複数実行ユニットでの実行機能を備えています。
  - インテル® Fortran コンパイラー: ハイパフォーマンスな最適化された Fortran コンパイラーです。

- **パフォーマンス・ライブラリー**: インテル® アーキテクチャー・ベースのプロセッサー向けに最適化されたソフトウェア・ライブラリーのセット。
  - インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP): 画像処理、信号処理、データ圧縮および暗号化向けのハイパフォーマンス・ライブラリー。
  - インテル® マス・カーネル・ライブラリー (インテル® MKL): 高度にベクトル化およびスレッド化された線形代数、高速フーリエ変換 (FFT)、ベクトル数学関数、そして統計関数のセット。
  - インテル® スレッディング・ビルディング・ブロック (インテル® TBB): 高いパフォーマンスとスケーラビリティーに優れた並列アプリケーションの開発を支援する、C と C++ 向けのテンプレート・ライブラリー。
  - インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL): データの取得からデータマイニング、機械学習まで、すべてのデータ解析フェーズに対応した最適化された解析ビルディング・ブロックを提供する C++ および Java* API ライブラリーです。ハイパフォーマンスなビッグデータ・アプリケーションには不可欠です。

- **パフォーマンス・プロファイラー**: パフォーマンス・プロファイラーは、CPU、GPU、スレッド化、ベクトル化および MPI 並列をチューニングするため、ソフトウェアのパフォーマンス・データを取集、解析、そして表示する機能を備えています。収集されたデータにより、システム全体から特定のコード行まで調査できます。
  - インテル® VTune™ プロファイラー・パフォーマンス・プロファイラー
  - インテル® Graphics Performance Analyzers (インテル® GPA): グラフィックス・アプリケーション向けのパフォーマンス・アナライザー
  - インテル® Advisor: ベクトル化の最適化とスレッド化のプロトタイプ作成
  - インテル® Trace Analyzer & Collector (トレース・アナライザー/コレクター): MPI 通信のパフォーマンス・プロファイラーと正当性検証ツール

- **デバッガー**
  - インテル® Inspector: メモリーとスレッド化向けデバッガー
  - インテル® Application Debugger
  - インテル® JTAG デバッガー
  - インテル® System Debugger

- **クラスターツール**
  - インテル® MPI ライブラリー: ハイパフォーマンス MPI ライブラリー
  - インテル® MPI Benchmarks: クラスターや MPI 実装のパフォーマンスを検証するための MPI カーネルテストのセット

上記のパフォーマンス・ツールは、次の製品スイートのいずれかに含まれています。

- **インテル® Parallel Studio XE**[32]
  - インテル® Media Server Studio
  - インテル® System Studio

A.1 コンパイラー

インテル® コンパイラーは、/O1、/O2、/O3、/fast などの汎用的な最適化の設定をサポートしています。いずれを指定した場合でも、それぞれ固有の最適化オプションが有効になります。ほとんどの場合は、/O1 オプションよりも、関数のインライン展開が有効になる /O2 オプションを使用することを推奨します。小規模な関数呼び出しの多いプログラムには、関数インラインが役立ちます。コードサイズが重要なときは、/O1 オプションが望ましい場合もあります。デフォルトでは、/O2 オプションが有効になります。

/Od (Linux* では -O0) オプションを使用すると、最適化機能がすべて無効になります。/O3 オプションを使用すると、より強力な最適化機能が有効になります。ただし、そのほとんどは、以下で説明するプロセッサー固有の最適化機能と組み合わせた場合のみ有効です。

/fast オプションは、プログラム全体の速度を最大限にします。インテル® 64 プロセッサーおよび IA-32 プロセッサー向けには、「/fast オプション」は、「/O3 /Qipo /Qprec-div- /fp:fast=2 /QxHOST (Windows*)」、「-Q3 -ipo -no-prec-div -static -fp-model fast=2 -xhost (Linus*)」、および「-O3 -ipo -mdynamic-no-pic -no-prec-div -fo-model fast=2 -xhost (OS X*)」と同じです。

上記のコマンドライン・オプションは、インテル® コンパイラーのドキュメントで説明されています。

A.1.1 インテル® 64 プロセッサーと IA-32 プロセッサーの推奨される最適化設定

表 A-1 は、インテル® プロセッサー向けにコードを生成する推奨コンパイラー・オプションを示しています。表 A-1 は、インテル® 64 プロセッサーの互換モードで動作するコードにも適用されますが、64 ビット・モードでの動作には適用されません。最新の情報については次の記事を参照してください:
https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-forsse-generation-and-processor-specific-optimizations/ (英語)

表 A-1 推奨されるプロセッサー最適化オプション

条件	推奨	説明
• インテル® AVX2 を利用するインテル® プロセッサーで最高のパフォーマンスを達成。	• /QxCORE-AVX2 (Linux* と macOS* では -x CORE-AVX2)	• 単一コードパス。
• インテル® AVX2 を利用するインテル® プロセッサーで最高のパフォーマンスを達成。	• /QaxCORE-AVX2 (Linux* と macOS* では –ax CORE-AVX2)	• 複数のコードパスが生成されます。 • 実行するすべてのシステム上でアプリケーションの検証を行う必要があります。
• インテル® SSE4.2 を利用するインテル® プロセッサーで最高のパフォーマンスを達成。	• /QxSSE4.2 (Linux* と macOS* では -x SSE4.2)	• 単一コードパス。
• インテル® SSE4.2 を利用するインテル® プロセッサーで最高のパフォーマンスを達成。	• /QaxSSE4.2 (Linux* と macOS* では -ax SSE4.2)	• 複数のコードパスが生成されます。 • 実行するすべてのシステム上でアプリケーションの検証を行う必要があります。

[32] バージョンとツールに含まれるコンポーネントの詳細については、つぎのウェブサイトを参照してください:
https://software.intel.com/en-us/intel-parallel-studio-xe

A.1.2 ベクトル化とループの最適化

インテル® C++コンパイラーとインテル® Fortran コンパイラーのベクトル化機能は、同一の命令によるシーケンシャルなデータアクセスを検出し、対象とするプロセッサー・プラットフォームに応じてインテル® SSE、インテル® SSE2、
インテル® SSE3、インテル® SSSE3、インテル® SSE4、インテル® AVX、インテル® AVX2、インテル® AVX-512 を使用するようにコードを変換できます。ベクトライザーは以下の機能をサポートしています。

- 複数のデータ型: float/double、char/short/int/long(符号付きと符号なし)、_Complex float/double をサポートします。
- ステップごとの診断: /Qopt-report (Linux* や macOS* では-qopt-report) オプションによって、ベクトライザーは、行や変数ごとに、ベクトル化されたコード、ベクトル化されなかったコード、および各コードがベクトル化されなかった理由をレポートします。このフィードバックから得られる情報に基づいて、開発者は、依存関係ディレクティブ、restrict キーワード、または OpenMP* ディレクティブを使用してベクトル化が行われるようにコードを修正または再構築できます。
- 動的なデータ・アライメント手法: アライメント手法には、ループピーリングやループアンロールが含まれます。ループピーリングは、アライメント済みロードを生成し、アプリケーションのパフォーマンスを向上させます。ループアンロールは、フル・キャッシュラインのプリフェッチと一致させ、スケジューリングを向上させます。
- 移植性のあるコード: 適切なインテル® コンパイラー・オプションを使用して新しいプロセッサーの機能を利用でき、開発者はソースコードを書き換える必要がありません。

プロセッサー固有のベクトライザーのオプションには次のものがあります: /Qx<CODE> および /Qax<CODE> (Linux* と macOS* では -x<CODE> および -xa<CODE>)。コンパイラーは、ベクトル化を制御するベクトライザーのオプションを多数備えています。それらのオプションを使用するには、/Qx<CODE> または /Qax<CODE> オプションのいずれかが有効である必要があります。デフォルトは無効です。

A.1.2.1 OpenMP* による並列化

インテル® C++ コンパイラーおよびインテル® Fortran コンパイラーは、OpenMP* プラグマやディレクティブ、ランタイム API および環境変数を使用した共有メモリー並列 (スレッド化) と分散メモリー並列 (オフロード) 機能をサポートします。OpenMP* プラグマ/ディレクティブは、/Qopenmp (Linux* と macOS* では -qopenmp) コンパイラー・オプションにより有効化されます。利用可能なプラグマ/ディレクティブについては、インテル® C++ および Fortran コンパイラーとともに提供されるデベロッパー・ガイドおよびリファレンス、または OpenMP* シンタックス・リファレンス・カードに示されています。OpenMP* に関する詳細は、http://www.openmp.org (英語) のウェブサイトを参照してください。

A.1.2.2 自動並列化

依存関係のない単純なループでは、インテル® C++ コンパイラーとインテル® Fortran コンパイラーは、マルチスレッド・コードを自動的に生成できます。この機能は、コンパイラー・オプション /Qparallel (Linux* と macOS* では -parallel) によって有効になります。

A.1.3 ライブラリー関数のインライン展開 (/Oi、/Oi-)

デフォルトでは、C、C++、数値演算ライブラリーの標準関数がコンパイラーによってインライン展開されます。通常は、これにより実行速度が高速化されます。しかし、ライブラリー関数をインライン展開すると、予期しない結果をもたらす場合があります。詳細については、インテル® コンパイラー・ドキュメントを参照してください。

A.1.4 プロシージャー間とプロファイルに基づく最適化

コードのプロファイルとプロシージャー間の依存関係に基づいてコードのパフォーマンスを改善する 2 つの方法を以下に示します。

A.1.4.1 プロシージャー間の最適化 (IPO)

ソースファイル内のプロシージャー間の最適化を行うには、/Qip (Linux* と macOS* では -ip) オプションを使用します。また、複数のソースファイル間のプロシージャー間の最適化を有効にするには、/Qipo (Linux* と macOS* では -ipo) オプションを使用します。

A.1.4.2 プロファイルに基づく最適化 (PGO)

コンパイラーはソースコードからインストルメントを行うコードを埋め込んで、プロファイルを収集可能なプログラムを作成し、リンクします。このインストルメント・コードが実行されるたびに、動的に情報ファイルが作成されます。2 度目のフィードバック・コンパイルでは、生成された動的情報がマージされてサマリーファイルが作成されます。このプロファイル情報の概要を使用して、コンパイラーはプログラム内で最も頻繁に実行されるパスの最適化を図ります。

プロファイルに基づく最適化は、特にインテル® Pentium® 4 プロセッサーとインテル® Xeon® プロセッサー・ファミリーで有効です。これは、コンパイラーによる命令キャッシュの利用率とメモリーページに関連する最適化の判断を大幅に高めます。また、PGO では最適化をガイドする実行時間情報を使用できるため、プロセッサーに適した分岐のヒント生成し、マイクロアーキテクチャーのパイプラインで最も頻繁に実行されるパスを維持するため分岐と基本ブロックを再配置することで分岐予測がかなり改善されます。

PGO を使用する際は、次のガイドラインに従ってください。

- インストルメント済みコードを実行してからフィードバック・コンパイルを行うまでの間は、プログラムに加える変更を最小限にします。フィードバック・コンパイルでは、情報が生成した後に変更された関数の動的情報は無視します。

  注意

  プログラムが変更されている場合、コンパイラーは動的情報が関数に対応していないことを示す警告を示します。

- インストルメント済みコードを実行してからフィードバック・コンパイルを行うまでの間にソースファイルに多数の変更を加えた場合、インストルメンテーション・コンパイルを繰り返します。

最適化オプションの詳細については、インテル® コンパイラーのドキュメントを参照してください。

A.1.5 インテル® Cilk™ Plus

インテル® Cilk™ Plus は、3 つのキーワードだけで簡単に単純なループとタスク並列をアプリケーションに実装できる C/C++ コンパイラー拡張です。また、ベクトル化機能と、高度なループベースのデータ並列処理およびタスク処理を組み合わせることで、優れた機能を提供します。

A.2 パフォーマンス・ライブラリー

インテル® パフォーマンス・ライブラリーは、本書を通して説明する多くの最適化を実装しています。これには、ループアンロール、命令のペアリングとスケジュールなどのアーキテクチャー固有のチューニング、および暗黙や明示的なデータ・プリフェッチとキャッシュ・チューニングを含むメモリー管理などが含まれます。

インテル® パフォーマンス・ライブラリーは、インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3) を使用した SIMD 命令レベルの並列処理の利点を活用します。これらの手法により計算集約型のアルゴリズムのパフォーマンスを向上させ、高級言語開発環境向けに手動で最適化されたパフォーマンスを提供します。

インテル® パフォーマンス・ライブラリーは、パフォーマンスが重要なアプリケーション向けに、頻繁に実行される関数のアセンブリー・レベルでの時間がかかる作業から開発者を解放します。プロトタイプ化と新しいアプリケーション機能の実装に必要な時間が大幅に短縮され、さらに市場投入までの期間を劇的に短縮できます。インテル® パフォーマンス・ライブラリーを使用して開発されたアプリケーションは、アップグレードされたバージョンのライブラリーをアプリケーションと再リンクするだけで、簡単に将来のインテル® プロセッサー世代の新しいアーキテクチャーの機能を活用できます。

ライブラリー・セットには、インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP)、インテル® マス・カーネル・ライブラリー (インテル® MKL)、およびインテル® スレッディング・ビルディング・ブロック (インテル® TBB) が含まれます。

A.2.1 インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP)

インテル® IPP for Linux* およびインテル® IPP for Windows*: インテル® IPP は、ビデオデコード/エンコード、オーディオデコード/エンコード、イメージカラー変換、コンピューター・ビジョン、データ圧縮、文字列処理、信号処理、音声認識、音声デコード/エンコード、暗号化、さらに数学ルーチンなど広範囲な分野をサポートする、クロスプラットフォーム向けソフトウェア・ライブラリーです。

これらの関数は、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) 、インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) 命令、およびインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) 命令セットを使用して高度に最適化されています。単一の API で広範囲のプラットフォームに対応できるため、互換性と開発コストの軽減を実現します。

A.2.2 インテル® マス・カーネル・ライブラリー (インテル® MKL)

インテル® MKL for Linux*、Windows*、macOS*: インテル® MKL は、インテル® プラットフォーム上で高いパフォーマンスが要求される工学、科学、および金融アプリケーション向けの高度に最適化された数学関数で構成されます。ライブラリー関数は、LAPACK や BLAS から成る線形代数、離散フーリエ変換 (DFT)、ベクトル超越関数 (ベクトル数学ライブラリー/VML)、およびベクトル統計関数 (VSL) などを含んでいます。インテル® MKL は、インテル® Xeon Phi™ 製品、インテル® Xeon® プロセッサー、インテル® Core™ プロセッサー、インテル® Core™2 プロセッサー、インテル® Pentium® 4 プロセッサー・ベースのシステムの機能を引き出すように最適化されています。マルチコアおよびメニーコア向けのマルチスレッド化によるパフォーマンスの最適化は特に配慮されています。

A.2.3 インテル® スレッディング・ビルディング・ブロック (インテル® TBB)

インテル® TBB は広く使用されている C++ テンプレート・ライブラリーであり、安定性を備え、移植性とスケーラビリティーに優れた並列アプリケーションの作成を支援します。インテル® TBB を利用することで、さまざまな環境でマルチコアおよびメニーコア・プロセッサーの能力を最大限に活用し、パフォーマンスを引き出すことができるだけでなく、保守も容易な優れたタスクベースの並列アプリケーションを簡単に短期間で開発できます。

インテル® TBB は、Windows*、Linux*、そして OS X* プラットフォーム上で、各種コンパイラーを使用して検証され、商用サポートされています。また、オープンソース・コミュニティーにより、FreeBSD*、IA ベース Solaris*、Xbox* 360 および PowerPC* ベースのシステムでも利用できます。

A.2.4 利点のまとめ

上記のライブラリーを使用することで、アプリケーション開発者は総じて次の利点が得られます。

- **開発期間の短縮** — 低レベルの各種ビルディング・ブロック関数によってアプリケーションを素早く開発でき、開発期間を短縮できます。

- **パフォーマンス** — 高度に最適化された C インターフェイスを備えた各種ルーチンによって、C/C++ 開発環境でアセンブリー・レベルのパフォーマンスが得られます。インテル® MKL は Fortran インターフェイスにも対応しています。
- **最適化されたプラットフォーム** — プロセッサー固有の最適化により、インテル® プロセッサーの世代ごとに最高のパフォーマンスを引き出すことができます。
- **互換性** — 単一のアプリケーション・プログラミング・インターフェイス (API) を使用してプロセッサー固有の最適化を行うことにより、開発コストを削減しつつ、最適なパフォーマンスを提供できます。
- **スレッド化されたアプリケーションのサポート** — インテル® MKL とインテル® IPP の関数はスレッド環境で安全に使用できるため、スレッド化されているアプリケーションで容易に利用できます。

A.3 パフォーマンス・プロファイラー

インテルのシリアルおよび並列処理プロファイル・ツールは、アプリケーションを再コンパイルすることなく低オーバーヘッドでパフォーマンスのボトルネックを特定し、高速化のためのスケーリング情報と改善のための意思決定に有用な情報を提供します。プロファイル・ツールは、組込みシステムからスーパーコンピューターまで、インテル® プロセッサー・ベースの広範囲なシステムの評価を可能にし、アプリケーションのパフォーマンス向上に役立ちます。

A.3.1 インテル® VTune™ プロファイラー

インテル® VTune™ プロファイラー[33] は、Windows* および Linux*. 向けの強力なスレッド化とパフォーマンス最適化ツールです。インテル® VTune™ プロファイラーを使用すると、プロセッサー・コアを完全に活用し、確実に新しいプロセッサーの能力を最大限に引き出して、最適なパフォーマンスに向けてファインチューニングが可能になります。

この節では、インテル® VTune™ プロファイラーの主要な機能について説明します。これらの機能の詳細については、インテル® VTune™ プロファイラーの製品ドキュメントとオンラインヘルプを参照してください。

A.3.1.1 ハードウェア・イベントベース・サンプリング解析

インテル® VTune™ プロファイラーは、イベントベース・サンプリングによるデータ収集に基づいて、インテル® Core™2 プロセッサーから最新のプロセッサーまで、マイクロアーキテクチャーの解析を可能にします。解析タイプごとに、インテル® VTune™ プロファイラーは関連するハードウェア・イベントをモニターし、収集したデータの生のイベントカウント (キャッシュミス、クロックティック、命令リタイアなど) とパフォーマンス・メトリックを表示します。それぞれのメトリックには、しきい値とイベント比率が設定されています。プログラムユニットのメトリックごとのパフォーマンスがしきい値を超えると、インテル® VTune™ プロファイラーはパフォーマンスの問題としてその値を (ピンク色で) ハイライトし、推奨される解決方法を提示します。

それぞれのインテル® プロセッサーで利用可能なイベントについては、https://perfmon-events.intel.com/ (英語) を参照してください。

A.3.1.2 アルゴリズム解析

インテル® VTune™ プロファイラーは、ユーザーモードのサンプリングとトレース収集に基づくアルゴリズム解析タイプを導入しています。

- **ホットスポット解析**は、アプリケーションの実行フローを理解し、実行に長い時間がかかっているコード領域 (ホットスポット) を特定するのに役立ちます。特定のプロセス、スレッドまたはモジュールで多数のサンプルが収集されていると、プロセッサーの利用率が高くパフォーマンスのボトルネックである可能性があります。一部のホットスポットは排除できますが、アプリケーション機能の根本にあるホットスポットは排除することはできません。インテル® VTune™ プロファイラーは、関数で費やされた時間順にアプリケーションの関数のリストを作成し

[33] 次のリソースも参照してください: https://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/ (英語)

ます。関数のコールスタックも表示するため、時間を費やしている関数がどのように呼び出されているかを確認できます。

- **ロックと待機解析**は、プロセッサー使用率が効率的ではない原因を特定します。スレッドが長時間同期オブジェクト (ロック) を待機するのは、最も一般的な問題の 1 つです。コアが十分に利用されず待機が発生すると、パフォーマンスは影響を受けます。ロックとウェイト解析により、アプリケーションに導入するそれぞれの同期オブジェクトの影響を検証し、アプリケーションが同期オブジェクト、またはスリープやブロック化 I/O などの API で待機している時間を理解できます。
- **並行性解析**は、プロセッサーの利用率が低い場所のホットスポット関数を特定するのに有効です。ホットスポットでコアがアイドル状態になっている場合、それらのコアにワークを与えることでパフォーマンスを向上できる可能性があります。

A.3.1.3 プラットフォーム解析

レンダリング、ビデオ処理、および計算向けにグラフィックス処理ユニット (GPU) を使用するアプリケーション向けのプラットフォーム全体のメトリックを収集するためインテル® VTune™ プロファイラーを使用できます。システムの各種 CPU と GPU コアにおけるコードの実行を理解するため、CPU/GPU 並行性解析を使用し、ターゲット・アプリケーションが GPU 依存であるか CPU 依存であるかを特定します。

A.4 スレッドとメモリーチェッカー

アプリケーションの信頼性、セキュリティー、および精度を向上するため、スレッド化とメモリーエラーのチェックを 1 つの強力なエラー・チェック・ツールとして統合しました。

A.4.1 インテル® Inspector

インテル® Inspector は、高い並列実行を行うアプリケーション向けのスレッドのバック解析 (データ競合、デッドロック、スレッドと同期 API の使用とスレッド間のメモリーアクセスを検出) と、シリアルおよび並列アプリケーション向けのメモリーチェック解析 (メモリーリークとメモリー破壊、メモリーの割り当てと解放 API の不一致、および一貫性のないメモリー API の利用などを検出) を提供します。

インテル® Inspector は、開発サイクルの早い段階で重大なメモリーとスレッド化の欠陥を発見することで、開発の生産性を高めてアプリケーションの信頼性を向上させます。この機能は、アプリケーションのメモリーとスレッド動作に関する詳細情報を提供し、アプリケーションの信頼性を向上できるように支援します。強力なスレッド検証ツールとデバッガーにより、実行コードパスの潜在的なエラーを簡単に見つけられます。また、エラーを引き起こすシナリオが実行されない場合でも、間欠的なエラーや非決定性のエラーを発見します。また、開発者は特殊なコードのビルドやコンパイルを行うことなく、頻繁にコードをテストできるようになります。

A.5 ベクトル化のアシスタント

A.5.1 インテル® Advisor

インテル® Advisor は、ベクトル化のアシスタントとスレッドのプロトタイプ化ツールであり、ソースコード中で最もベクトル化とスレッド化の影響がある領域を特定することで、容易にスレッド化とベクトル化が可能になります。

A.6 クラスターツール

インテル® Parallel Studio XE Cluster Edition は、IA-32、IA-64, インテル® 64 アーキテクチャーをベースとするクラスター向けの並列アプリケーションの開発、解析、およびパフォーマンス最適化に役立ちます。Cluster Edition には、クラスター向けのコード開発に有用な、インテル® Trace Analyzer & Collector、インテル® MPI ライブラリー、インテル® MPI Benchmarks が含まれています。

A.6.1 インテル® Trace Analyzer & Collector

インテル® Trace Analyzer & Collector[34] は、MPI 通信におけるパフォーマンス・ボトルネックを素早く発見することで、クラスター上のアプリケーション・パフォーマンスを理解および最適化するのに欠かせない情報を提供します。インテル® アーキテクチャー・ベースのクラスターシステムのサポート、現在の標準化と高い互換性を持つ機能、またトレースファイルの理想化と比較、カウンターデータ表示、パフォーマンス・アシスタント、MPI 正当性チェック・ライブラリーなどを含みます。MPI パフォーマンスを解析し、並列アプリケーションの実行をスピードアップし、ホットスポットとボトルネックを特定して、トレースファイルとグラフィックスを比較してタイムラインに適合した詳細な解析を提供します。

A.6.1.1 インテル® MPI パフォーマンス・スナップショット

MPI Performance Snapshot (MPS) は、MPI アプリケーション向けのスケーラブルで軽量なパフォーマンス・ツールです。通信、アクティビティー、負荷バランスなどの MPI アプリケーションの特性を収集し、分かりやすい形式で表示します。MPS はアプリケーションのハイレベルの概要を提供するため、インテル® MPI ライブラリーと OS およびハードウェア・カウンターからライトウェイト統計を組み合わせます。このツールは、インテル® Trace Analyzer & Collector の一部としてインストールされます。

A.6.2 インテル® MPI ライブラリー

インテル® MPI ライブラリーは、メッセージ・パッシング・インターフェイス 3.0 (MPI-3.0) 仕様を実装する、マルチファブリックをサポートするメッセージ・パッシング・ライブラリーです。インテル® プラットフォーム向けに標準ライブラリーを提供します。インテル® MPI ライブラリーは、InfiniBand*、Myrinet*、およびインテル® True Scale ファブリックを含む複数のハードウェア・ファブリックをサポートしています。Direct Access Programming Library (DAPL) 方式を介した高速インターコネクト向けのユニバーサルなマルチファブリック・レイヤーを提供することにより、すべての構成をカバーします。実行時にユーザーがどのファブリックを選択しても、効率良く実行できる、ファブリックに依存しない MPI コードを開発できます。

インテル® MPI ライブラリーは、必要な場合のみ動的に接続を確立し、メモリー・フットプリントを削減します。また、利用できるトランスポートの中から最も高速なものを自動で選択します。ジョブ開始時のソケットへのフォールバックにより、インターコネクトの選択に失敗した場合でも、実行の失敗を回避できます。これは、特にバッチ・コンピューティングにおいて有効です。インテル® MPI ライブラリーで開発されたアプリケーションの展開は、インテルから無料ランタイム環境キットをダウンロードできるため、ランタイム互換が保証されます。また、マルチコアノード内または SMP ノード内で DAPL をオプションで使用すると、大きなメッセージ帯域幅の利点から、アプリケーション・パフォーマンスが向上します。

A.6.3 インテル® MPI Benchmarks

インテル® MPI Benchmarks を利用すると、MPI 関数とパターンの性能比較を簡単に行うことができます。このベンチマークでは、ユーザビリティー、アプリケーション・パフォーマンス、相互運用性が強化されています。

A.7 インテル® Academic Community

インテル® Academic Community で提供されるクラスルーム・トレーニングの詳細については、https://software.intel.com/en-us/articles/intel-academic-community-showcase (英語) を参照してください。開発者向けの一般的な情報については、https://software.intel.com/en-us/ (英語) を参照してください。

[34] インテル® Trace Analyzer & Collector は、インテル® Parallel Studio XE Cluster Edition にのみ含まれます。

付録 B
パフォーマンス監視イベント

パフォーマンス監視イベントは、プログラムの命令シーケンスとマイクロアーキテクチャー・サブシステムの相互作用の特性を評価するための機能を提供します。パフォーマンス監視イベントの詳細については、https://perfmon-events.intel.com/ (英語) を参照してください。インテル

この節の前半では、インテル® マイクロアーキテクチャー向けのパフォーマンス・チューニングにおける、パフォーマンス・ボトルネックを解析するトップダウン方式のマイクロアーキテクチャー解析手法 (TMAM) について説明します。B.1 節では、最近のインテル® マイクロアーキテクチャーに適用できる一般化された手法を紹介します。Skylake† マイクロアーキテクチャー向けの TMAM の具体例が含まれています。

この節の残りのセクションでは、以前の世代のインテル® マイクロアーキテクチャーで役立つ情報をカバーしています。

B.1 トップダウン解析法

この節では、アウトオブオーダー・コアのパフォーマンス・ボトルネックを特定するトップダウン・マイクロアーキテクチャー解析方式 (TMAM) について説明します。一般的な階層フレームワークと階層的なテクニックの概念は、多くのアウトオブオーダー・マイクロアーキテクチャーに適用できます。

TMAM は、単一の階層構造で編成されるマイクロアーキテクチャーに依存しないメトリックを使用することでサイクル・アカウンティング (パフォーマンス・ボトルネックのコストを特定する過程で、CPI のブレークダウンとも呼ばれます) を簡略化します。

図 B-1 は、現代のマイクロアーキテクチャーの主要な機能ブロックに関連したパフォーマンス・ボトルネックを分類する階層的なアプローチを示しています。TMAM を使用したそれぞれのマイクロアーキテクチャーの世代に関連する高い学習曲線は、真にパフォーマンスを制限する原因を明らかにする体系的なドリルダウンによって置き換えられます。これは、マイクロアーキテクチャーの詳細をすべて理解することなく、パフォーマンス解析を行うことを可能にします。

このトップダウン階層フレームワークの利点は、調査するマイクロアーキテクチャーで考えられることをドリルダウンして開発者を導く体系的なアプローチです。あまり意味のない問題を無視し、真に解決すべき問題に解析の労力を充てることを可能にするため、ツリーのノードを重み付けします。

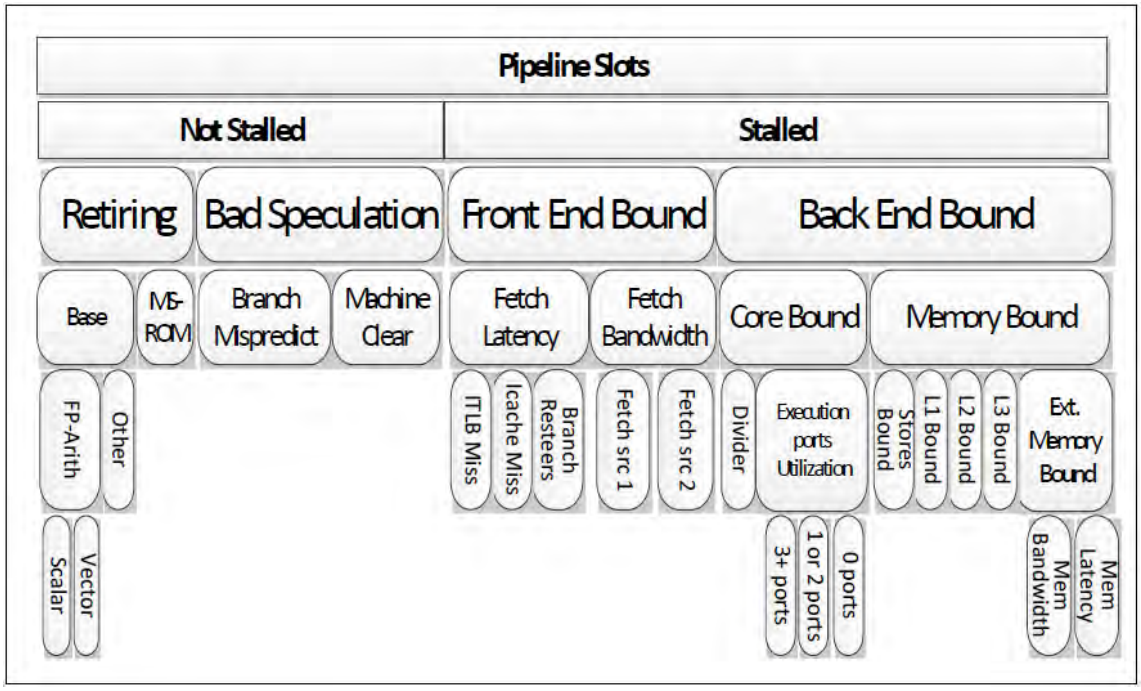


図 B-1 アウトオブオーダー・マイクロアーキテクチャー向けの一般的な TMAM 階層構造

例えば、アプリケーションに命令フェッチの問題があるなら、TMAM はフロントエンド依存としてツリーの最上位レベルでそれを分類します。ユーザーやツールは、フロントエンドのサブツリーのみに注目してドリルダウンできます。ドリルダウンは、ツリーのリーフ (樹木葉) に達するまで繰り返されます。リーフはワークロードの特定のストールを示すか、アプリケーションのパフォーマンスを制限する一般的なマイクロアーキテクチャー上の兆候と問題のサブセットを表します。

TMAM は、Sandy Bridge† マイクロアーキテクチャーのパフォーマンス監視機能に関連して開発されました。この方法論は、複数世代にわたるマイクロアーキテクチャー世代をサポートするため洗練され、その後 PMU 機能によって拡張されました。

B.1.1 トップレベル

トップレベルで、TMAM はパイプライン・スロットを 4 つの状態に分類します。

- フロントエンド依存。
- バックエンド依存。
- 投機の問題。
- リタイア。

図 B-1 に見られるように、後者 2 つは非ストールスロットを示し、前者 2 つはストールを示します。

図 B-2 にドリルダウンの簡単な決定木を示します。

- スロットが何らかの操作で利用されると、それはリタイア (コミット) されたかどうかにより、リタイアまたは投機の問題として分類されます。
- パイプラインのバックエンドが操作を受け入れることができない場合 (バックエンド・ストール)、利用されなかったスロットはバックエンド依存として分類されます。または、
  - フロントエンド依存: バックエンドはストールしていないにもかかわらず、操作 (uop) が供給されなかったことを示します。

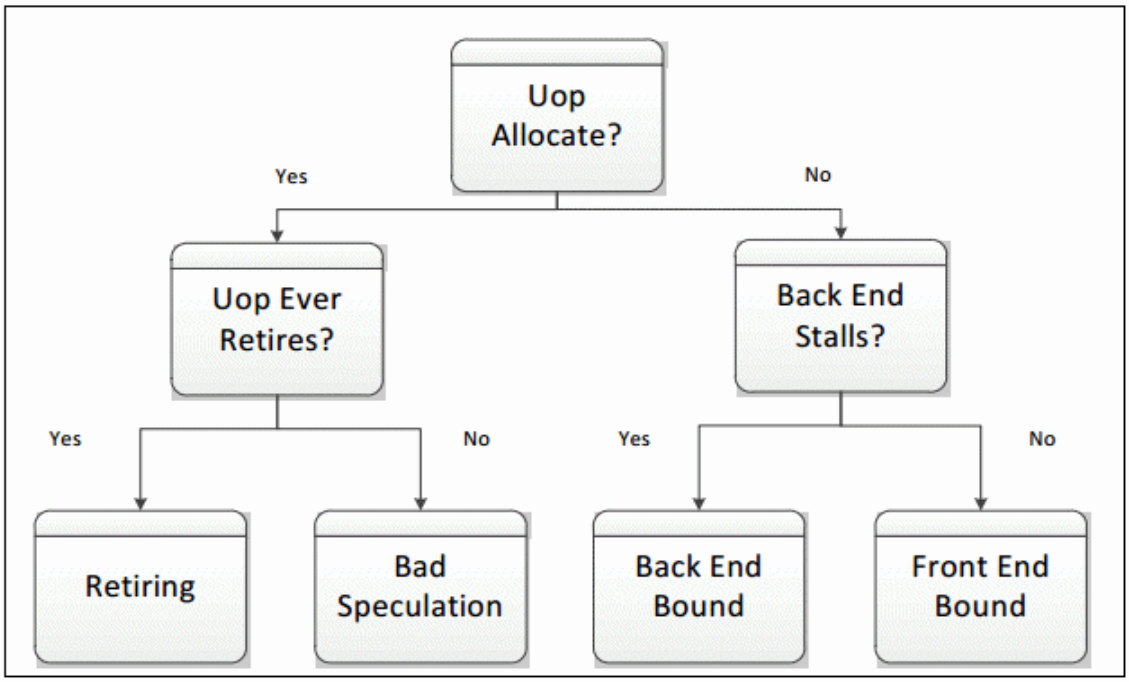


図 B-2 TMAM のトップレベルにおけるドリルダウンのフローチャート

パイプラインの問題となるステージ (アロケーション・ステージ) の単一のエントリー分岐点で、4 つのカテゴリーはスロット全体に付加可能になります。スロット粒度 (副サイクル) での分類は、トップレベルで必要なスーパースカラー・コア向けの正確で堅牢なブレークダウンを可能にします。

リタイアは、"有効な操作" で利用されたスロットを意味します。ここでサイクルあたりの命令数 (IPC) と関連付けてすべてのスロットを表示したいと思うかもしれません。高いリタイア率はスピードアップの余地がないことを意味するものではありません。

投機の問題は、誤った投機により浪費されたスロットを示します。次のものが含まれます: (a) 最終的にリタイアしなかった操作のスロット、および (b) 発行されたパイプラインが、先行する誤った投機からの回復によりブロックされたスロット。Branch_Resteers による 3 番目の問題があることを忘れてはいけません。このカテゴリーは投機のタイプごとに分類できます。例えば、分岐予測ミスとマシンクリアは、制御フローとデータの誤った投機に関連します。

フロントエンド依存は、パイプラインのフロントエンドがバックエンドに操作を十分に供給できていないことを示します。フロントエンドは、後続のバックエンドで実行される操作を提供するパイプラインの一部です。このカテゴリーはさらに、フェッチのレイテンシー (命令キャッシュや ITLB ミスなど) とフェッチの帯域幅 (部分的なデコードなど) に分類されます。

バックエンド依存は、新しい操作を受け入れるバックエンドのリソース不足により、ストールしたスロットを示します。これは、メモリー・サブシステムによる実行ストールを反映する**メモリー依存**、と実行ユニットへのプレッシャー (計算依存) や命令レベルの並列性 (ILP) 不足を反映する**コア依存**に分類されます。

以降の節では、これらのカテゴリーに関する詳細と階層の次のレベルのノードについて説明します。

B.1.2 フロントエンド依存

フロントエンドは、分岐予測器が次のフェッチアドレスを予測し、命令キャッシュからコードバイトのストリームをフェッチし、命令をバックエンドで実行されるマイクロオペレーションに解析およびデコードするパイプラインの一部です。**フロントエンド依存**は、プロセッサー・コアのフロントエンドがバックエンドに対し供給不足であることを示します。これは、バックエンドが uop (マイクロオペレーション) を受け入れる準備ができていると、フェッチバブルの原因となります。

長くバッファーされたパイプラインの初めで生じるフロントエンドの問題に TMAM なしで対処するのは困難です。一時的なこの問題は実際のパフォーマンスに影響しないこともあり、フロントエンドの問題がトップレベルで検出された場合にのみ詳細を調査すべきです。特に IPC が高い場合、フロントエンドの帯域幅はパフォーマンスに影響します。これにより、フェッチ・パイプラインのレイテンシーを隠匿し、デコード済み命令キャッシュ (DSB) と同様にループストリーム検出器 (LSD) などに必要な帯域幅を維持する専用ユニットが追加されました。

TMAM はさらに、フロントエンドにおけるストールのレイテンシーと帯域幅を識別します。

- 命令キャッシュミスは、**Fetch Latency (フェッチレイテンシー)** に分類されます。
- 命令デコーダーの非効率性は、**Fetch Bandwodth (フェッチ帯域幅)** に分類されます。

これらのメトリックはトップダウンのアプローチで定義されることに注意してください。**Fetch Latency (フェッチレイテンシー)** は、原因にかかわらずフェッチのスタベーション (uop が供給されない) を引き起こす状況をカウントします。これには命令キャッシュと命令 TLB ミスが該当しますが、それだけではありません。**Branch Resteers (分岐切り替え)** は、パイプラインのフラッシュに続くフェッチの遅延をカウントします。パイプラインのフラッシュは、分岐予測ミスやメモリーニュークなどのいクリアイベントによって発生します。**Branch Resteers (分岐切り替え)** は、投機の問題と密接に関連しています。

この方法論はさらに、フェッチユニットごとの帯域幅の問題、マイクロオペレーション・キュー (図 2-3 を参照) への uop の挿入などを分類します。命令デコーダーは、一般的に使用される x86 命令をプロセッサーが理解できる uop に変換します。これはフェッチ単位で行われます。CPUID などいくつかの x86 命令はマイクロオペレーション・フローを必要とします。これは MSROM から長いマイクロオペレーション・フローが供給され、2 番目のフェッチ単位となります。プロセッサーの世代ごとにフェッチユニットの帯域幅が異なる可能性があります。図 2-3 は、Skylake† マイクロアーキテクチャーの詳細を示しています。

B.1.3 フロントエンド依存

バックエンド依存は、バックエンドがマイクロオペレーションを受け入れるために必要なリソースが不足しているため、発行パイプラインでマイクロオペレーションが供給されないスロットがあることを示します。このカテゴリーに分類されるパフォーマンスの問題には、データ・キャッシュ・ミスや除算ユニットがオーバーロード状態であることによるストールなどが含まれます。

バックエンド依存は、**メモリー依存**と**コア依存**に分類されます。これは、サイクルごとの実行ユニットの占有率に基づいてバックエンドのストールを調査することで達成できます。IPC を最大に維持するため、実行ユニットをビジーに保つ必要があります。例えば、4 スロット幅のマシンでは、3 つ以下のマイクロオペレーションがコードの定常状態で実行される場合、最良である IPC 4 を達成することはできません。これらの部分的な最適サイクルは、ExecutionStalls と呼ばれます。

B.1.4 メモリー依存

Memory Bound (メモリー依存) は、キャッシュとメモリー・サブシステムに関連する実行ストールに相当します。通常、これらのストールは、ロードがすべてのキャッシュをミスした直後のように、実行ユニットが短時間でスタベーションを引き起こす場合に表れます。最近のインテル® Core™ プロセッサー世代の多くは、外部メモリーのレイテンシーを隠匿する 3 階層のキャッシュを備えています。最初のレベルはデータキャッシュ (L1D) です。L2 は 2 番目のレベルの命令とデータを共有する各コアのプライベート・キャッシュです。L3 は、物理パッケージ上のすべてのプロセッサー・コア間で共有されます。

アウトオブオーダーのスケジューラーは、複数の実行ユニットへマイクロオペレーションを供給することができます。マイクロオペレーションがインフライトで実行される間、保留されているメモリーアクセスに関連しない有効なマイクロオペレーションで実行ユニットをビジーに保つことで、データのメモリー・アクセス・レイテンシーを隠匿することができます。したがって、一般的には、スケジューラーが実行ユニットに対して何も提供する準備ができていない場合、メモリーアクセスは実際のペナルティーを被ります。さらにマイクロオペレーションが保留されているメモリーアクセスを待っているか、準備ができていないマイクロオペレーションに依存している可能性もあります。

実行ストールは、特定のキャッシュレベルとそれぞれのキャッシュレベルで扱われるデータ要求に依存する、いくつかのサブカテゴリーに関連しています。ある状況では、ロード要求がキャッシュレベルでミスしない場合、実行ストールはキャッシュ・レベル・レイテンシーよりも長いレイテンシーを被ることがあります。

例えば、L1D では ALU ストール (または浮動小数点加算/乗算、整数乗算などの一般的に使用される実行ユニットの完了を待機) に匹敵する短いレイテンシーがしばしば発生します。しかし、アドレスがオーバーラップした先行するストアからのデータフォワードによりブロックされているロードなどでは、ロードは L1D によって最終的にデータが返されるまで長いレイテンシーを被る可能性があります。そのような場合、インフライトのロードはしばらくの間 L1D を利用することができます。これは L1 依存にタグ付けされます。4K エイリアスによってブロックされるロードは、同様の兆候が見られるもう 1 つのケースです。

さらに、ストア操作に関連する実行ストールは、**ストア依存**のカテゴリーとして扱われます。ストア操作は、メモリーオーダーの要件からバッファリングされ、リタイア後に実行されます。通常、ストア操作は、パフォーマンスにそれほど影響しませんが、完全に無視できるほど小さくはありません。TMAM では、ストア依存を実行ポートの利用率が低く、ストアをバッファリングするためリソースの消費が高いストア数のサイクルの一部であると定義します。

データ TLB (DTLB) ミスは、各種メモリー依存のサブノードに分類されます。例えば、TLB 変換が L1D で扱われる場合、それは L1 依存にタグ付けされます。

MEM Bandwidth (メモリー帯域幅) と **MEM Latency (メモリー・レイテンシー)** を **Ext Memory Bound (外部メモリー依存)** に分類するため、簡単なヒューリスティックが使用されます。このヒューリスティックは、メモリー・コントローラーから返されるデータが保留される要求の占有期間をベースにします。占有期間が高いしきい値を超える (要求最大数の 70%) 場合、メモリー・コントローラーは同時に処理することができ、TMAM はこれをメモリー帯域幅によって潜在的に制限されるとして識別します。残りはメモリー・レイテンシーと分類されます。

B.1.5 コア依存

コア依存は、実行ユニットのプレッシャーやプログラムの命令レベルの並列性 (ILP) の欠如を示します。コア依存のストールは、短期間の命令スタベーションや、特定するのが困難である部分的な実行ポートの利用を表します。例えば、長いレイテンシーの除算操作は、実行ポートに対して特定の uop タイプがプレッシャーを与え、サイクルで利用されるポートが減少することからしばらくの間命令スタベーションを引き起こし、実行をシリアル化する可能性があります。

コア依存の問題は、より適切なコードを生成することで軽減できることがあります。例えば、依存関係のある数学計算のシーケンスは、コア依存に分類されます。コンパイラーは、より適切な命令スケジュールによってストールを軽減できる可能性があります。ベクトル化もまたコア依存の問題を緩和できます。

B.1.6 投機の問題

Bad Speculation (投機の問題) は、誤った投機により浪費されたスロットを示します。これには次の 2 つが含まれます。

- 最終的にリタイアしなかったマイクロオペレーションの発行に消費されたスロット。
- 先行する投機のミスから回復するためブロックされた発行パイプラインのスロット。

例えば、誤って予測された分岐の背後で発行されたマイクロオペレーションは、このカテゴリーに分類されます。誤った予測のペナルティーは、どれくらい迅速に正しいターゲットをフェッチできるかによります。これは、他のフロントエンドのストールをオーバーラップする可能性がある **Branch Resteers (分岐のリステア)** としてカウントされます。

トップレベルでの投機の問題のカテゴリーは、TMAM における重要な課題です。これには、解析で誤った実行パス (ほかのカテゴリーに挙げられている測定の精度に影響する) の影響を受けるワークロードの領域を明らかにします。また、低レベルのノードで、従来のパフォーマンス・カウンター (ほとんどのカウンターイベントが投機的にカウントさ

れる) の使用を可能にします。従って、高い値を示す**投機の問題**を “red flag (レッドフラグ)” として、他のカテゴリーを調査する前に最初に対応すべきです。投機の問題を最小化することは、プロセッサー・リソースの利用を改善するだけでなく、階層を通してレポートされるメトリックの確度を高めます。

TMAM はさらに、**投機の問題**を同じ兆候を示しパイプラインがフラッシュされる**分岐予測ミス**と**マシンクリア**に分類します。分岐予測ミスは、BPU が分岐方向と (または) ターゲットを誤って予測した場合に適用されます。メモリー・オーダー・クリア (例えば、メモリー・ディスアンビゲーション) による誤ったデータ投機は、マシンクリアのサブセットです。これらの問題を解析する次のステップは完全に異なることがあります。メモリーオーダーによるマシンクリアや自己修正コードなど、以降に予期しない状況を招くコードでは、最初に、プログラムの制御フローが分岐予測に適するようにします。

B.1.7 リタイア

このカテゴリーは、発行されたマイクロオペレーションがパフォーマンス・ボトルネックを発生させることなく素早くリタイアした、”適切なマイクロオペレーション” によって利用されたスロットを示します。理想的には、すべてのスロットが**リタイア**カテゴリーの属性となることが望まれます。すべてのスロットの 100% リタイアは、マイクロアーキテクチャーのサイクルごとに最大のマイクロオペレーションがリタイアしたことに相当します。例えば、1 つの命令が 1 マイクロオペレーションにデコードされる場合、1 スロットでの 50% のリタイアは、4 ワイドのマシンでは IPC 2 であることを示します。言い換えると、**リタイア**のカテゴリーを最大化することでプログラムの IPC を高めることができます。

高いリタイア値はより高いパフォーマンスの余地がないことを意味するものではありません。浮動小数点 (FP) アシストなどのマイクロコード・シーケンスはパフォーマンスを損ねますが、回避することができます。これらは注意を促すため、**MSROM** のカテゴリーに分類されます。

ベクトル化されていないコードでの高いリタイア値は、ベクトル化の候補となります。これを実現するには、単一の命令/マイクロオペレーションでより多くの操作を完了することです。その結果パフォーマンスが向上します。TMAM は、さらに**リタイア -> 基本**カテゴリーを**スカラー**と**ベクトル操作**による **FP 計算**に分類します。詳細については、行列乗算の利用例 2 を参照してください。

B.1.8 TMAM と Skylake† マイクロアーキテクチャー

Skylake† マイクロアーキテクチャーのパフォーマンス監視機能は、以前の世代と比べ大幅に拡張されています。TMAM は、カウンターイベントの多様性とプリサイス・イベントベースのサンプリング (PEBS) 機能の拡張の利点を得られます。図 B-3 は、Skylake† マイクロアーキテクチャーにおける TMAM のサポートを示しています (緑のボックスは PEBS が利用可能です)。

インテル® VTune™ プロファイラーは、多くのインテル® マイクロアーキテクチャーで TMAM を適用することを可能にします。https://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues (英語) にある技術文書で、さらに詳しい情報をご覧いただけます。

B.1.8.1 TMAM の例

15.15.1 節では、レイテンシーに関連する浮動小数点演算の最適化テクニックと、FP MUL、FP ADD、および FMA 命令のスループットに関する考慮事項を説明しています。FP_ADD と FP_MUL 命令のレイテンシーの問題を直接検出するパフォーマンス・カウンター・イベントはありません。

TMAM は、この問題がパフォーマンスを制限する可能性があることを理解するのに使用できます。

主なボトルネックがバックエンド依存 -> コア依存 -> ポート利用であり、GFLOPS メトリックにおいて顕著な測定があれば、コードはこの問題に遭遇している可能性があります。15.15.1 節に示される最適化を考慮してみても良いでしょう。

15.3.1 節では、Skylake† マイクロアーキテクチャーにおいて YMM の上位がダーティーである場合にインテル® SSE コードを実行するパフォーマンスの問題について説明しています。パーシャルレジスターの依存性と、インテル® SSE コード実行時のブレンドコストに関するパフォーマンスの問題を特定するには、ソースコードが直接インテル® AVX 命令を実行せずパフォーマンス上重要なインテル® SSE コードでのインテル® SSE 命令とインテル® AVX 命令の混在比率をモニターするのに TMAM を使用できます。

主なボトルネックが、バックエンド依存 -> コア依存である場合、Vector-MixRate (ベクトルミックス比率) メトリックに顕著な測定が見られれば、YMM レジスター上位の混在操作によるベクトル幅の不一致なベクトル操作が存在する可能性を示します。

VectorMixRate (ベクトルミックス比率) メトリックには、Skylake† マイクロアーキテクチャーで利用可能な UOPS_ISSUED.VECTOR_WIDTH_MISMATCH イベントが必要です。このイベントは、ベクトルレジスターの上位ビットを保持するため、発行ステージで挿入されたマイクロオペレーションをカウントします。このイベントは、ベクトルレジスターの上位ビットを維持するため、リソース・アロケーション・テーブル (RAT) からリザベーション・ステーション (RS) へ発行されたブレンド・マイクロオペレーションの数をカウントします。

さらに、メトリックは分母として、最近のインテル® マイクロアーキテクチャーでは一般的な UOPS_ISSUED.ANY を使用します。UOPS_ISSUED.ANY イベントは、RAT が RS へ発行したマイクロオペレーションの総数をカウントします。

VectorMixRate メトリックは、発行されたすべてのマイクロオペレーションのうち、挿入されたブレンド・マイクロオペレーションの比率を示します。通常、VectorMixRate が 5% を上回ると調査の必要があります。

VectorMixRate[%] (ベクトルミックス比率) =
100 * UOPS_ISSUED.VECTOR_WIDTH_MISMATCH / UOPS_ISSUED.ANY

実際のペナルティーは、挿入されたブレンド操作が追加するデスティネーション・レジスターとのデータ依存関係から生じるため、一定ではありません。

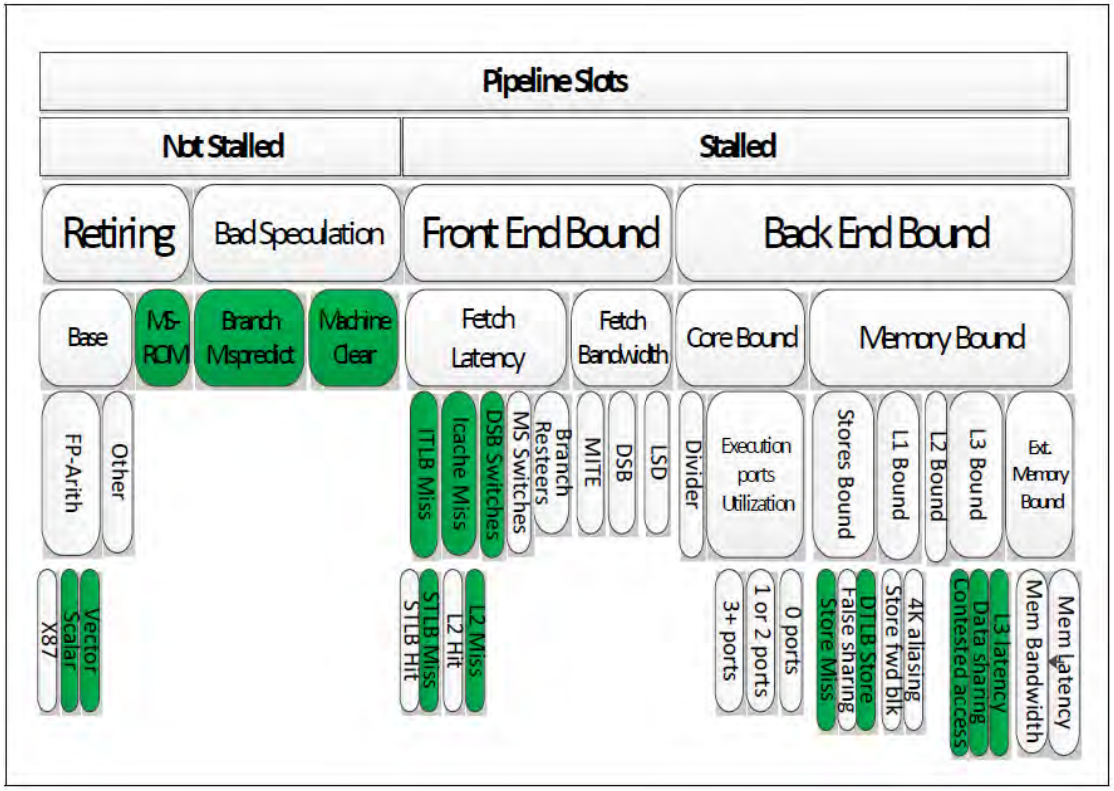


図 B-3 Skylake† マイクロアーキテクチャーでサポートされる TMAM 階層

B.2 パフォーマンス監視とマイクロアーキテクチャー

この節では、Silvermont†、Airmont† および Goldmont† マイクロアーキテクチャーに関連するパフォーマンス監視ハードウェアと用語について説明します。ここで説明する特性は、表 B-1 に示すようにマイクロアーキテクチャー固有である可能性があります。

表 B-1 パフォーマンス監視の分類

名前	説明	マイクロアーキテクチャー
L2Q、XQ	メモリー参照が L1 データキャッシュをミスした場合、L2 キュー (L2Q) へ要求が送られます。要求が L2 キャッシュでもミスすると要求は XQ へ送られ、ダイ上のインターフェイス (IDI) リンクを介してメモリーへ発行されるのを待機します。L2 キャッシュは、プロセッサー・コアのペアで共有されるため、単一の L2Q もこれら 2 つのコアで共有されることに注意してください。同様に、L2Q と IDI リンクの間に位置するプロセッサー・コアのペアには単一の XQ があります。 XQ に到達する新しい要求よりも IDI リンクからの要求が少ない場合、XQ は一杯になります。イベント L2_reject_XQ は、XQ が一杯であるため要求を L2 から XQ に送ることができないことを示し、その結果、メモリー・サブシステムがオーバーサブスクライブであることを示します。	Silvermont†、Airmont†、Goldmont†
Core Reject	core_reject イベントは、コアからの要求が L2Q によって受け入れられなかったことを示します。要求が L2Q によって拒否されるにはいくつかの理由があります。L2Q が一杯であると以降の要求は拒否されます。つまり、一方のコアの要求を拒否して、他のコアの公平性を維持することができます。つまり、あるコアが L2Q/キャッシュ/XQ/IDI リンクへの共有接続要求を占有することが許可されてないと、L2Q に余地があっても要求は拒否される可能性があります。さらに、コアからの要求がダーティーな L1 キャッシュの追い出しである場合、ハードウェアはこの追い出しが L2Q 内のすべての保留要求と競合しないことを保証する必要があります (保留要求には外部スヌープが含まれます)。イベントが競合すると、ダーティー追い出し (eviction) 要求は L2Q に余地があっても拒否される可能性があります。 そのため、L2_reject_XQ イベントは、両方のコアからメモリーへの要求比率がメモリーの応答比率を超えていることを示しますが、Core_reject イベントは微妙です。これは、L2Q への要求比率が XQ からの応答比率を超えていることを示すか、L2Q への要求比率が L2 からの応答比率を超えていることを示すか、またはコアが L2Q からの応答の妥当な共有数を超えた要求を試みていることを示します。また、それはダーティー追い出しと他の保留要求の競合を示すかもしれません。 つまり、L2_reject_XQ イベントはメモリーのオーバーサブスクリプション状態を示します。Core_reject イベントは、次のいずれかを示します。(1) メモリーのオーバーサブスクリプション、(2) L2 のオーバーサブスクリプション、(3) 他のコアへの公正性を保証するため一方のコアの要求を拒否、(4) ダーティー追い出しと他の保留要求間の競合。	Silvermont†、Airmont†、Goldmont†

Divider Busy (除算器ビジー)	すでにディスパッチされている除算 uop を処理中である場合、除算ユニットは新しい除算 uop を受け入れることができません。"CYCLES_DIV_BUSY.ANY" イベントは、除算ユニットがビジーであるサイクルをカウントします。これは、別の除算マイクロオペレーションが、除算ユニットへの割り当て (RS から) を待機しているかどうかにかかわらずカウントされます。.このイベントでは、RS が空であっても除算が処理中であればサイクルがカウントされます。	Silvermont†、Airmont†、Goldmont†
BACLEAR	命令をデコードして分岐/コール/ジャンプ/リターン命令を識別した直後、分岐アドレス計算クリア (BACLEAR) イベントの発生が可能となります。BACLEAR が発生する原因には、誤った直接分岐のターゲット予測や命令の位置で分岐が予測されていないなどが含まれます。 BACLEAR は、フロントエンドに異なる位置からフェッチを再開させます。BACLEAR は、パイプラインの一部を実行することで識別される分岐予測ミスと類似していますが、BR_MISP_RETIRED イベントはカウントされない、または LBR (LBR が予測ミスを報告) での予測ミスと示されます。分岐予測ミスと BACLEAR は、新しいターゲット位置から命令のフェッチを始めるためフロントエンドを再起動し、投機されたワークをフラッシュするという点では同じです。しかし、分岐予測ミスでは、フロントエンドとバックエンドの両方から命令を完全にフラッシュする必要があるのに対し、BACLEARは、デコード時に発生するため命令バイトをフラッシュしますが、完全にデコードされていない命令はフラッシュしません。BACLEAR後のリカバリーは、分岐予測ミス後のリカバリーよりも単純で高速です。	Silvermont†、Airmont†、Goldmont†
フロントエンドのボトルネック	フロントエンドは、命令をフェッチし、マイクロオペレーション (uop) へデコードし、バックエンドで処理するためマイクロオペレーション・キューへそれらを送出する役割を持ちます。次にバックエンドは、それらのマイクロオペレーションを取得して必要なリソースを割り当てます。すべてのリソースの準備が整うと、マイクロオペレーションが実行されます。フロントエンドのボトルネックは、マシンのフロントエンドがマイクロオペレーションをバックエンドへ供給できない場合 (バックエンドはストールせず) に発生します。バックエンドの準備ができていないためマイクロオペレーションを受け入れることができないサイクルは、フロントエンドのボトルネックとしてカウントされません。バックエンドのボトルネックはアロケーション・ユニットのストールを引き起こし、その結果フロントエンドはバックエンドがマイクロオペレーションを受け入れるようになるまで待機します。	Silvermont†、Airmont†、Goldmont†
NO_ALLOC_CYCLES	フロントエンドの問題は、このイベントクラスの各種サブイベントを使用して分析できます。	Silvermont†、Airmont†
UOPS_NOT_DELIVERED.ANY	UOPS_NOT_DELIVERED.ANY イベントは、マシンが確実にフロントエンド依存であることを特定するため、フロントエンドの効率を測定するのに使用されます。非効率なフロントエンドの例には次のものがあります: フロントエンドの帯域幅を制限する命令キャッシュミス、ITLB ミス、およびデコーダーの制限。	Goldmont†

ICache (命令キャッシュ)	命令キャッシュ (ICache) への要求は、チャンクと呼ばれる固定サイズ単位で行われます。キャッシュラインには複数のチャンクがあるため、複数のアクセスが同じキャッシュラインに集中する可能性があります。 Goldmont† マイクロアーキテクチャーでは、このイベントはキャッシュライン単位でカウントされるため、単一のキャッシュラインへの複数のフェッチは 1 つの ICACHE.ACCESS、1 つの HIT または 1 つの MISS としてカウントされます。特に、直線的なコードがキャッシュラインにまたがるか、分岐ターゲットが新しいラインである場合、イベントはカウントされます。このイベントは、本質的にかなり投機的であり、デコード、実行、およびリタイアの前に命令バイトがフェッチされます。投機は分岐と同様に直線的なコードでも行われることがあります。その結果、リタイアした命令数を調査することでは、命令キャッシュの特性は推定できません。 Silvermont† マイクロアーキテクチャーでは、ICACHE イベント (HIT、MISS) は異なる粒度でカウントされます。	Goldmont†
ICache (命令キャッシュ) アクセス	命令キャッシュのフェッチアクセスは、固定サイズのチャンクにアライメントされます。命令キャッシュからの特定チャンクのフェッチ要求は、投機実行により複数回現れる可能性があります。同じチャンクが実行中の状態で複数回要求される可能性もあります。しかし、命令フェッチミスは一度だけカウントされ、処理中のすべてのサイクルをカウントするわけではありません。 命令キャッシュミスがラインをフェッチした後、同じキャッシュラインへの別の要求が再び行われ、それはヒットと見なされます。そのため、"ヒット" + "ミス" の値は、アクセス数と等しくなりません。 ソフトウェアの観点から命令キャッシュヒットの正確な数を取得するには、命令キャッシュヒットから命令キャッシュミスの値を減算すべきです。	Silvermont†、Airmont†、Goldmont†
ラスト・レベル・キャッシュ参照、ミス	L3 を持たないプロセッサーでは、L2 がラスト・レベル・キャッシュとなります。LLC 参照とミスをカウントするアーキテクチャーのパフォーマンス・イベントは、L2_REQUESTS.ANY と L2_REQUESTS.MISS です。	Silvermont†、Airmont†、Goldmont†
マシンクリア	マシンクリアを引き起こす可能性がある多くの状態があります (割り込み、トラップ、フォルトの受信を含みます)。それらすべての条件 (MO (メモリーオーダー)) を含みますが限定はされません)、SMC (自己またはクロス修正コード)、および FP (浮動小数点アシスト) は、MACHINE_CLEAR.ANY イベントで検出されます。さらに、いくつかの条件 (SMC、MO、FP など) が明確にカウントされます。しかし、SMC、MO、および FP マシンクリアの合計が ANY 数と等しくなるわけではありません。	Silvermont†、Airmont†、Goldmont†
MACHINE_CLEAR.FP_ASSIST	ほとんどの場合、浮動小数点実行ユニットは適切な出力ビットを生成します。命令に対してマシンクリアが発生する場合など、まれにヘルプを必要とすることがあります。マシンクリアが発生すると、マシンのフロントエンドはどのような浮動小数点操作が必要であるかを特定する命令を供給し、正しい結果を生成するため特別なコードを実行します (例えば、結果が浮動小数点デノーマル値であれば、ハードウェアは IEEE に準拠する適切な丸めを行うようにアシストを要求します)。	Silvermont†、Airmont†、Goldmont†

MACHINE_CLEAR.SMC	自己修正コード (SMC) とは、マシンが実行する命令ストリームにコードの一部を書き込むことを指します。Silvermont† マイクロアーキテクチャーでは、プロセッサーが 1K でアライメントされた領域内で SMC を検出します。SMC 条件が検出されるとマシン・クリア・アシストが発生し、パイプラインがフラッシュされます。 プロセッサーが実行しているコードの 1K 以内のメモリーに書き込みを行うと、SMC 検出機能が起動されマシンクリアが発生します。マシンクリアによってストア・パイプラインが排出されるため、フロントエンドが再起動して正しい命令を (書込み後) 実行できます。	Silvermont†、Airmont†、Goldmont†
MACHINE_CLEAR.MO	メモリー・オーダー・マシン・クリアは、スヌープ要求が発生してメモリーの順番を維持できるかどうかマシンが不確実である場合に起こります。例えば、2 つのロードがあると仮定し、プログラムの順番で 1 つがアドレス X を指定し、もう 1 つがそれに続く Y を指定します。両方のロードが発行されますが、Y のロードが最初に完了し、このロードと依存関係を持つマイクロオペレーションが処理を続行します。X へのロードはデータを待機しています。同時に別のプロセッサーが Y と同じアドレスに書き込みを行い、アドレス Y へのスヌープが発生すると仮定します。 ここで問題が生じます。X のロードがまだ完了していなかったため、Y のロードは古いを取得しています。他のプロセッサーは、アドレス Y にストアされた最新の値を利用しないことで、異なる順番でロードを認識します。書込み後のデータを取得するには、アドレス Y からのロードをすべて元に戻す必要があります。 注意: 他に保留されている読み込みがなければ、ロード Y を元に戻す必要はありません。ロード X が完了していないことが、この順序付けの問題を引き起こします。	Silvermont†、Airmont†、Goldmont†
MACHINE_CLEAR.DISAMBIGUATION	ディスアンビゲーション・マシン・クリアは、先行するストアと同じアドレスを持つロードがストアを追い越し投機的に実行されたときに、ストアのアドレスが不明である場合に発生します。	Goldmont†

ページウォーク	リニアアドレスから物理アドレスへの変換の際にトランスレーション・ルックアサイド・バッファー (TLB) に変換済みのアドレスが見つからない場合、必要であれば専用ハードウェアがページテーブルと他のページ構造から物理アドレスを検索しなければいけません。これをページウォークといいます。ページウォークが完了すると、変換情報は以後の処理のため TLB に格納されます。 ページング構造はメモリーに保存されているため、ページウォークには複数のメモリーアクセスが必要となることもあります。ページウォークが命令参照を変換した場合、これらのメモリーアクセスは要求されたデータの一部であると考えられます。ページウォークのサイクル数は可変であり、必要なメモリーアクセスの回数とこれらのメモリーアクセスのキャッシュ局所性に依存します。 PAGE_WALKS イベントは、EDGE トリガービットがクリアされてからページウォークにかかった時間をカウントします。ページウォーク時間をページウォークの回数で割ると平均時間が求められます。 Goldmont† マイクロアーキテクチャーでは、ITLB.MISS と MEM_UOPS_RETIRED.DTLB_MISS イベントを使用してページウォークの回数を測定できます。 Silvermont† マイクロアーキテクチャーでは、PAGE_WALKS.WALKS によってデータと命令が混在したページウォークの回数がカウントされます。	Silvermont†、Airmont†、Goldmont†
RAT	RAT は、フロントエンドからバックエンドへマイクロオペレーションを転送する割り当てパイプラインです。割り当てパイプラインの最後で、マイクロオペレーションは 6 つのリザベーション・ステーション (RS) のいずれかに書き込まれます。各 RS は、特定の実行 (またはメモリー) クラスターへ送られるマイクロオペレーションを保持します。RS の数には制限があるため、実行クラスターにマイクロオペレーションを送出できない場合、それらは蓄積される可能性があります。RS が一杯になる典型的な理由として、除算のような長い実行レイテンシー、依存関係によりマイクロオペレーションをスケジュールできない、また多くのメモリー参照が実行中である、などがありますがこれらは限定的ではありません。RS が一杯になると、それ以上のマイクロオペレーションを受け入れることができなくなり、割り当てパイプラインをストールさせます。RS_FULL_STALL.ANY イベントは、RS のいずれかが一杯になったことで割り当てがストールするサイクルが生じるとカウントされます。割り当てパイプラインはいくつかの原因でストールする可能性がありますが、RS フルが原因でない場合 RS_FULL_STALL.ANY はカウントされません。MEC サブイベントは、MEC RS フルがさらに割り当てを妨げるかどうか判断するのを可能にします。	Silvermont†、Airmont†、Goldmont†

REHABQ	REHABQ は、1 つまたは複数の理由により、完了できないマイクロオペレーションのメモリー参照を保持する内部のキューです。マイクロオペレーションは、再発行され正常に完了するまで REHABQ に残っています。 マイクロオペレーションが REHABQ に格納されるボトルネックの原因としては、キャッシュライン分割、ブロックされたストアフォワード、およびデータの準備ができていない、などがありますが限定的ではありません。ロードやストアが REHABQ に送られるには、多くの状況が考えられます。例えば、先行するストアのストアアドレスが不明である場合、そのアドレスが判明するまで後続のストアはすべて REHABQ に送られる必要があります。	Silvermont†、Airmont†
LOAD_BLOCKS	ロードは多くの原因でブロックされる可能性があります。原因は限定的ではありませんが、UTLB ミス、ブロックされたストアフォワード、4K エイリアシングなどが上げられます。ロードが先行するストアで保存されるデータ (全体もしくは部分的) を必要とする場合、マシンのフォワード処理は次の 2 つの状況に直面します: (i) 先行するストアの完了を待機する (フォワードは制限され、ロードはブロックされる) か、(ii) ストアが完了する前にデータをロードにフォワードできる。制限される状況は次のように説明されます。 先行するストアに対しロードをチェックする場合、アドレスビットのすべてがストアアドレスと比較されるわけではありません。アドレスが未定のストアと類似 (LD_BLOCKS.4K_ALIAS) しているため、ロードはブロックされますが技術的にはブロックする必要はありません。ロードが先行するストアからデータを受け取ることができない場合、ロードは未定のストアが完了するまでブロックされます。LD_BLOCKS.STORE_FORWARD は、アドレスの不一致 (以下で説明) によってストアからフォワードされたデータの受信が制限されたロードの回数をカウントします。LD_BLOCKS.DATA_UNKOWN は、その時点でストアデータが利用可能でないため、ストアフォワードによりブロックされたロードをカウントします。ロードブロックは、LD_BLOCKS.DATA_UNKNOWN および LD_BLOCK.STORE_FORWARD としてカウントされません。表 F-17 に、ロードが先行するストアからデータを受けとることが可能な条件を示します。 これらのイベントはプリサイスイベントであるため、リタイアしない投機的なロードはカウントしません。	Goldmont†
uop リタイア	プロセッサーは、複雑なマクロ命令を単純なマイクロオペレーションのシーケンスにデコードします。ほとんどの命令は 1 もしくは 2 つのマイクロオペレーションで構成されます。いくつかの命令は長いマイクロオペレーションのシーケンスにデコードされます (浮動小数点超越関数命令、アシスト、文字列リピート命令など)。 ある状況では、マイクロオペレーションのシーケンスが単独のマイクロオペレーションに融合されるか、命令全体が融合されます。Goldmont† では、MSROM マイクロオペレーションを区分するため UOPS_RETIRED にサブイベントがあります。このサブイベントは他のマイクロアーキテクチャーとは異なります。	Silvermont†、Airmont†、Goldmont†

HW_INTERRUPTS	これらのイベントは、ハードウェア割り込みに関する情報を提供します。HW_INTERRUPTS.RECEIVED は、プロセッサーが受信したハードウェア割り込みの総数をカウントします。このイベントは ROB が認識する割り込みカウントの総数に直結します。HW_INTERRUPTS.PENDING_AND_MASKED は、EFLAGS.IF が 0 であるため割り込みが上げられず保留されているコアのサイクル数をカウントします。しかし、TPR や ISR によってマスクされている割り込みはカウントされません。これらはプリサイスイベントではありませんが、これらのイベントで非プリサイス PEBS レコードを収集することは、無反応状態を引き起こす問題を特定するのに有効です。	Goldmont†
MEM_UOPS_RETIRED	これらのイベントは、マイクロオペレーションのリタイアが有効である場合、命令がデータの読み込み (ロード)、または書き込み (ストア) を行うマイクロオペレーションをカウントします。投機的なロードとストアはカウントされません。サブイベントでは、操作を完了するために必要な付加サイクルの状態を示すことができます。特に、メモリー・マイクロオペレーションのアドレスがデータ変換ルックアサイド・バッファー (DTLB) をミスすると、要求されたデータはキャッシュラインに分割されているか、メモリー・マイクロオペレーションはロックされたロードです。これはプリサイスイベントであるため、PEBS レコードの EventingRIP フィールドはイベントの引き起こした命令を示します。	Silvermont†、Airmont†、Goldmont†
MEM_LOAD_UOPS_RETIRED	これらのイベントは、マイクロオペレーションのリタイアが有効である場合、命令がデータを読み込む (ロード) マイクロオペレーションを生成するのをカウントします。投機的なロードはカウントされません。このイベントは、要求されたデータのメモリー階層での各種状態をレポートし、データのレイテンシー・ストールの原因を特定するのに役立ちます。これはプリサイスイベントであるため、PEBS レコードの EventingRIP フィールドはイベントの引き起こした命令を示します。	Goldmont†

B.3 インテル® Xeon® プロセッサー 5500 番台

インテル® Xeon® プロセッサー5500 番台は、インテル® Core™ i7 プロセッサーと同じマイクロアーキテクチャーをベースにしています (2.6 節「インテル® ハイパースレッディング・テクノロジー」を参照)。また、インテル® Xeon® プロセッサー 5500 番台は、2 ウェイ・プロセッサーで構成される Non-Uniform Memory Access (NUMA) をサポートします (図 B-4 を参照)。図 B-4 に、各物理プロセッサーの 4 つのプロセッサー・コアと 1 つのアンコア・サブシステムを示します。

L3 アンコア・サブシステムは、統合型メモリー・コントローラー (IMC) とインテル® QuickPath インターコネクト (インテル® QPI) インターフェイスで構成されます。メモリー・サブシステムは、各 IMC にローカルに接続された 3 チャネルの DDR3 メモリーで構成されます。非ローカルな IMC に接続された物理メモリーへのアクセスは、リモート・メモリー・アクセスと呼ばれます。

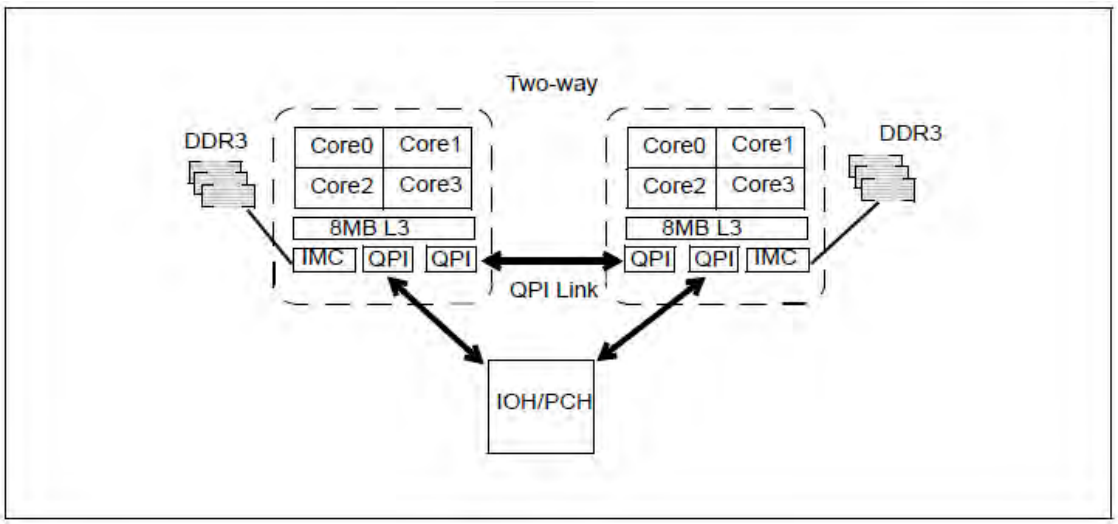


図 B-4 インテル® Xeon® プロセッサー 5500 番台でサポートされるシステムトポロジー

インテル® Xeon® プロセッサー 5500 番台のパフォーマンス監視イベントは、ソフトウェア (コードおよびデータ) とマイクロアーキテクチャー・ユニット間の相互作用を階層的に分析するために使用できます。

- コアごとの PMU: 各プロセッサー・コアは、4 つのプログラム可能なカウンターと 3 つの固定カウンターを備えています。プログラム可能なコアごとのカウンターは、フロントエンド/マイクロオペレーション (uop) のフロー問題と、プロセッサー・コア内のストールを調査するように構成できます。また、コアごとの PMU イベントのサブセットは、プリサイス・イベントベース・サンプリング (PEBS) をサポートします。ロード・レイテンシーの測定機能は、インテル® Core™ i7 プロセッサーとインテル® Xeon® プロセッサー 5500 番台の新機能です。
- アンコア PMU: アンコア PMU は、8 つのプログラム可能なカウンターと 1 つの固定カウンターを備えています。プログラム可能なアンコアのカウンターは、L3 とインテル® QPI の操作およびローカル/リモートデータのメモリーアクセスの特性評価を行うように構成できます。

インテル® Xeon® プロセッサー 5500 番台で利用できるさまざまなパフォーマンス・カウンターとプログラム可能な広範なパフォーマンス・イベントにより、ソフトウェアを最適化する開発者はパフォーマンスの問題を分析してパフォーマンスを高めることができます。パフォーマンス・イベントによるパフォーマンスに関する問題の分析は、次の分野にまとめることができます。

- サイクル・アカウンティングとマイクロオペレーション (uop) フロー
- ストールの区分とコア・メモリー・アクセス・イベント (非 PEBS)
- プリサイス・メモリー・アクセス・イベント (PEBS)
- プリサイス分岐イベント (PEBS、LBR)
- コア・メモリー・アクセス・イベント (非 PEBS)
- その他のコアイベント (非 PEBS)
- フロントエンドの問題
- アンコアイベント

B.4 インテル® Xeon® プロセッサー5500 番台のパフォーマンス分析手法

この節で説明する手法は、実行時に測定可能なパフォーマンスのボトルネックを排除または軽減する可能性を特定することに注目します。コンパイル時およびソースコード・レベルの手法は、本書のほかの章で説明します。各項では、パフォーマンス監視イベントから直接測定または計算可能なさまざまなメトリックを調べることにより、チューニングの可能性を特定するため具体的な手法について触れていきます。

B.4.1 サイクル・アカウンティングとマイクロオペレーション (uop) フロー

表 B-2 に、基本的なサイクル・アカウンティング手法の目的、パフォーマンス・メトリック、イベントを示します。

表 B-2 サイクル·アカウンティングおよびマイクロオペレーション (uop) フローの構成

概要	
目的	重大なストールが発生したコード/基本ブロックを特定
方法	「生産的」および「非生産的」領域へバイナリーのサイクルを区分
PMU とパイプラインの焦点	実行のため発行されたマイクロオペレーション (uop)
イベントコード/ Umask	マイクロオペレーション (uop) を実行する場合は、イベントコード B1H、Umask = 3FH 合計サイクル数をカウントする場合は、イベントコード 3CH、Umask = 1、CMask = 2
EvtSelc	サイクルの数をカウントし、アクティブなサイクルとストールサイクルを分離する場合は、CMask、Invert、Edge フィールドを使用
基本式	"合計サイクル数" = UOPS_EXECUTED.CORE_STALLS_CYCLES + UOPS_EXECUTED.CORE_ACTIVE_CYCLES
メトリック	UOPS_EXECUTED.CORE_STALLS_CYCLES / UOPS_EXECUTED.CORE_STALLS_COUNT
ドリルダウンの範囲	カウント: ワークロード、サンプリング: 基本ブロック
バリエーション	計算主体のマイクロオペレーション (uop) は、ポート 0、1、5 でサイクルをカウント

実行されたマイクロオペレーション(uop)のサイクル·アカウンティングは、パフォーマンス·チューニングにおけるストールサイクルの特定に効果的な手法です。マイクロアーキテクチャー·パイプライン内の、「発行された」、「ディスパッチされた」、「実行された」、「リタイアした」マイクロオペレーション (uop) の意味は厳密に定義されています。これを図 B-5 に示します。

サイクルは、マイクロオペレーション (uop) が実行ユニットにディスパッチされているサイクルと、ディスパッチされていないサイクルに分けられますが、後者は実行ストールであると考えられます。

テスト用コードに対して実行された「合計サイクル数」は、CPU_CLK_UNHALTED.THREAD (イベントコード 3CH、Umask = 1) と IA32_PERFEVTSELCn を設定 (CMask = 2、INV = 1) して直接測定できます。.

(ポート 2、3、4 で) 実行されたメモリー·アクセス·マイクロオペレーション (uop)をカウントするために使用されたシグナルは、論理プロセッサー単位で測定できないコアイベントのみです。イベントコード B1H と Umask=3FH の組み合わせでは、コア単位でのみカウントされるため、実行ストールの合計数はコア単位でのみ評価できます。HT が無効になっている場合は、マイクロオペレーション (uop) フロー·サイクル·アカウンティングの分析をスレッド単位で容易に実行できます。

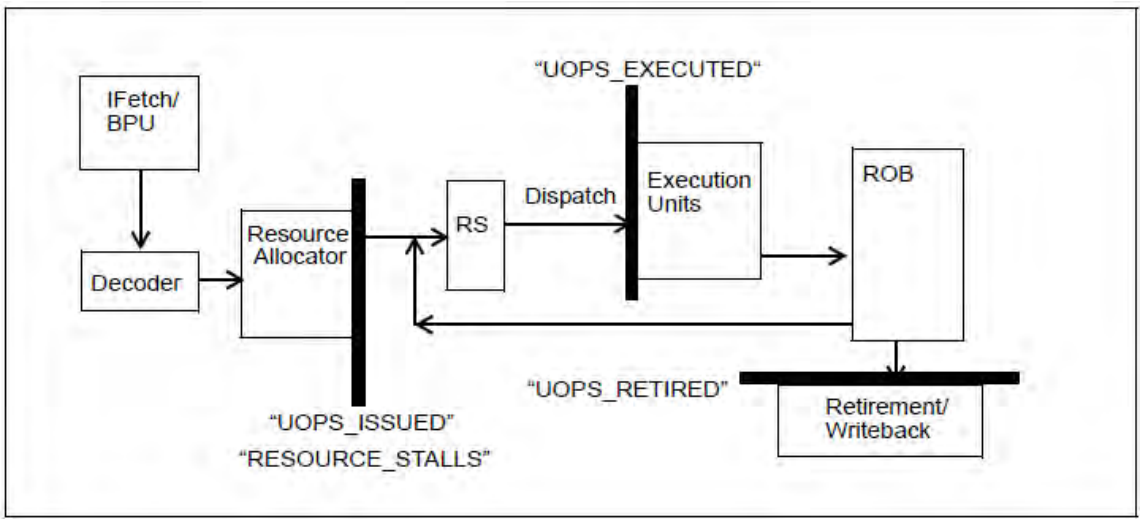


図 B-5 パイプラインにおける PMU 固有のイベントロジック

ポート 0、1、5 で UOPS_EXECUTED をカウントする PMU シグナルは、HT が有効であってもスレッド単位でカウントできます。これにより、テスト中のワークロードが HT と相互に作用する場合、代替のサイクル・アカウンティング手法が提供されます。

この代替メトリックは、適切な CMask、Inv、Edge 設定を使用して、UOPS_EXECUTED.PORT015_STALL_CYCLES から構成されます。表 B-3 にパフォーマンス・イベントの詳細を示します。

表 B-3 マイクロオペレーション (uop) フロー用イベントの CMask/Inv/Edge/スレッドの粒度

イベント名	Umask	イベントコード	Cmask	Inv	Edge	すべてのスレッド
CPU_CLK_UNHALTED.TOTAL_CYCLES	0H	3CH	2	1	0	0
UOPS_EXECUTED.CORE_STALLS_CYCLES	3FH	B1H	1	1	0	1
UOPS_EXECUTED.CORE_STALLS_COUNT	3FH	B1H	1	1	1	1
UOPS_EXECUTED.CORE_ACTIVE_CYCLES	3FH	B1H	1	0	0	1
UOPS_EXECUTED.PORT015_STALLS_CYCLES	40H	B1H	1	1	0	0
UOPS_RETIRED.STALL_CYCLES	1H	C2H	1	1	0	0
UOPS_RETIRED.ACTIVE_CYCLES	1H	C2H	1	0	0	0

B.4.1.1 サイクルのドリルダウンと分岐予測ミス

実行されたマイクロオペレーション (uop) は、供給されている実行ユニットの観点から見れば生産的であると考えられますが、このようなマイクロオペレーション (uop) がすべてプログラムの処理の進行に貢献するわけではありません。分岐予測ミスは、アウトオブオーダー (OOO) プロセッサー内で非効率な実行を行う可能性があり、通常、これは次の 3 つの要素に分類されます。

- 不適切に予測されたパスのマイクロオペレーション (uop) の実行に関連する無駄な処理。
- 不適切なマイクロオペレーション (uop) のパイプラインがフラッシュされる際に失われるサイクル。
- 不適切なマイクロオペレーション (uop) が実行ユニットに到達するのを待機している間に失われるサイクル。

Nehalem† マイクロアーキテクチャー・ベースのプロセッサーには、誤って予測されたマイクロオペレーション (uop) のパイプラインのクリア (2 番目の要素) に関連する実行ストールはありません。このようなマイクロオペレーション (uop) は、実行やディスパッチをストールさせることなく、簡単にパイプラインから排除されます。そのため、通常、予測が外れた分岐のペナルティーは低下します。また、命令スタベーション (3 番目の要素) に関連するペナルティーを測定できます。

実行されたマイクロオペレーション (uop) において無駄になる処理は、リタイアしないマイクロオペレーション (uop) です。これは分岐予測ミスに関連するコストの一部であり、パイプラインを通過するマイクロオペレーション (uop) フローを監視することによって検出できます。マイクロオペレーション (uop) フローは、図 B-5 の 3 つのポイント、イベント UOPS_ISSUED によって RS に入るとき、UOPS_EXECUTED によって実行ユニットに入るとき、そして UOPS_RETIRED によってリアイアするときに測定できます。誤って予測されたマイクロオペレーション (uop) に関連する無駄な処理は、アップストリームでの測定値とリタイア時の測定値の差によって判定できます。

UOPS_EXECUTED は、スレッド単位ではなくコア単位で測定されるため、コアあたりの無駄な処理は次のように評価されます。

Wasted Work (無駄な処理) =
UOPS_EXECUTED.PORT234_CORE + UOPS_EXECUTED.PORT015_All_Thread - UOPS_RETIRED.ANY_ALL_THREAD

上記の比率をマイクロオペレーション (uop) の平均発行比率で割るとサイクル数に変換できます。上記のイベントは、マイクロフュージョンやマクロフュージョンにおいても補正を行わず、この方法が使用できるように設計されています。

上記のイベントの最後の 2 つはスレッド単位でカウントできるため、「スレッド単位」の測定値は、発行されたマイクロオペレーション (uop) 数とリタイアしたマイクロオペレーション (uop) 数の差から求められます。これは、(実行中のサイクルを浪費する前に RS 内で排除される) 誤って予測されたマイクロオペレーション (uop) によって少し多めにカウントされますが、通常、これはわずかな収集で済みます。

Wasted Work/thread (スレッドあたりの無駄な処理) = (UOPS_ISSUED.ANY + UOPS_ISSUED.FUSED) - UOPS_RETIRED.ANY

表 B-4 予測ミスにより無駄になった処理のサイクル・アカウンティング

概要	
目的	実行されたが、予測ミスのためにリタイアしなかったマイクロオペレーション (uop) を評価
方法	実行とリタイアメント間のマイクロオペレーション (uop) フローの差を調査
PMU とパイプラインの焦点	マイクロオペレーション (uop) の実行とリタイアメント
イベントコード/ Umask	マイクロオペレーション (uop) を実行する場合は、イベントコード B1H、Umask = 3FH コア単位でカウントする場合は、イベントコード C2H、Umask = 1、AllThread = 1
EvtSelc	マイクロオペレーション (uop) をカウントする場合は、CMask、Invert、Edge フィールドを 0 に設定
基本式	“Wasted work (無駄な処理)” = UOPS_EXECUTED.PORT234_CORE + UOPS_EXECUTED.PORT015_ALL_THREAD - UOPS_RETIRED.ANY_ALL_THREAD
ドリルダウンの範囲	カウント: 分岐予測ミスのコスト
バリエーション	サイクル・アカウンティングの場合は、マイクロオペレーション (uop) の平均発行速度で除算 スレッドあたりのコストを概算する場合は、AllThread = 0 に設定

予測ミスのペナルティーの 3 番目の要素である命令スタベーションは、適切なパスに関連する命令がコアから離れており、RAT 内でのマイクロオペレーション (uop) が不足して実行がストールしている場合に発生します。マイクロオペレーション (uop) が発行されない原因は、フロントエンドのスタベーションまたはバックエンドでリソースを利用できないことのいずれかです。したがって、次の方法でリソース割り当ての出力を明示的に測定できます。

- マイクロオペレーション (uop) がアウトオブオーダー (OOO) エンジンに発行されなかったサイクルの総数をカウントします。
- 割り当てるリソース (RS、ROB エントリー、ロードバッファー、ストアバッファーなど) を利用できないサイクル数をカウントします。

HT が有効でない場合、命令スタベーションは単に次のように計算されます。

Instruction Starvation (命令スタベーション) =
UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY

HT が有効な場合、RS に供給されるマイクロオペレーション (uop) は 2 つのスレッド間で切り替えられます。発行がストールしているサイクルの 50% が他方のスレッドにマイクロオペレーション (uop) を供給している可能性があるため、理想的な状況では、上記の条件では多めにカウントされることが考えられます。その場合、他方のスレッドによってマイクロオペレーション (uop) が発行されているサイクルの数を引くように式を変更します。

Instruction Starvation (スレッドごとの命令スタベーション) =
UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY - UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD

スレッド単位の式では、命令スタベーションは若干多めにカウントされます。これは、同じコア内にある他方のスレッドがマイクロオペレーション (uop) を発行する間に、当該スレッドで RESOURCE_STALL 条件が存在している可能性があるためです。代替式は次のようになります。

CPU_CLK_UNHALTED.THREAD - UOPS_ISSUED.CORE_CYCLES_ACTIVE-RESOURCE_STALLS.ANY

表 B-5 にこれらのモデルをまとめています。

表 B-5 命令スタベーションのサイクル·アカウンティング

概要	
目的	予測ミス後にマイクロオペレーション (uop) の発行を待機しているサイクルを評価
方法	マイクロオペレーション (uop) の発行とリソース割り当て間のサイクルの差を調査
PMU とパイプラインの焦点	マイクロオペレーション (uop) の発行とリソース割り当て
イベントコード/Umask	発行されたマイクロオペレーション (uop) の場合は、イベントコード 0EH、Umask = 1 リソース割り当てがストールしているサイクルの場合は、イベントコード A2H、Umask=1
EvtSelc	マイクロオペレーション (uop) の発行がストールしているサイクルをカウントする場合は、CMask = 1、Inv = 1 に設定 マイクロオペレーション (uop) の発行がアクティブなサイクルをカウントする場合は、CMask = 1、Inv = 0 に設定 UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD に対する他方のスレッドの貢献を評価する場合は、2 つのカウンターをそれぞれ AllThread = 0、AllThread = 1 に設定
基本式	命令スタベーション(HT は無効) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY;
ドリルダウンの範囲	カウント: 分岐予測ミスのコスト
バリエーション	次の式により、スレッドあたりの値を評価 Instruction Starvation (命令スタベーション) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY - UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD

表 B-6 にパフォーマンス·イベントの詳細を示します。

表 B-6 マイクロオペレーション (uop) フロー用イベントの CMask/Inv/Edge/スレッドの粒度

イベント名	Umask	イベントコード	Cmask	Inv	Edge	すべてのスレッド
UOPS_EXECUTED.PORT234_CORE	80H	B1H	0	0	0	1
UOPS_EXECUTED.PORT015_ALL_THREAD	40H	B1H	0	0	0	1
UOPS_RETIRED.ANY_ALL_THREAD	1H	C2H	0	0	0	1
RESOURCE_STALLS.ANY	1H	A2H	0	0	0	0
UOPS_ISSUED.ANY	1H	0EH	0	0	0	0
UOPS_ISSUED.STALL_CYCLES	1H	0EH	1	1	0	0
UOPS_ISSUED.ACTIVE_CYCLES	1H	0EH	1	0	0	0
UOPS_ISSUED.CORE_CYCLES_ACTIVE	1H	0EH	1	0	0	1

B.4.1.2 基本ブロックのドリルダウン

INST_RETIRED.ANY (リタイアした命令) イベントは、命令あたりのサイクル数の比率 (CPI) を評価するため一般的に使用されます。その他の重要な利用法は、基本ブロック実行回数を評価することにより、特に高いパフォーマンスが要求される基本ブロックを決定することです。

サンプリング·ツール (インテル® VTune™ プロファイラーなど) では、サンプルが特定の IP 値周辺に集まる傾向があります。これは、INST_RETIRED.ANY またはサイクル·カウント·イベントを使用する際に該当します。ホットサンプルの逆アセンブルリストを表示すると、一部の命令は高いサンプルカウントに関連しますが、隣接する命令はサンプルに関連していないことがあります。

これは、基本ブロックの定義により、基本ブロック内のすべての命令が同じ回数だけリタイアするためです。ホットな基本ブロックのドリルダウンは、基本ブロック内の命令のサンプルカウントを平均化することで正確になります。

Basic Block Execution Count (基本ブロック実行回数) =
Sum (基本ブロック内の命令のサンプルカウント) * Sample_after_value / (基本ブロック内の命令の数)

ホットなループまたはホットでないループ構造に関連する基本ブロックの特定は、逆アセンブリー・リストを調査して、上記の手法に従って各ループ構造のトリップカウントを評価することで系統的に行うことができます。条件分岐のない単純なループでは、このトリップカウントは、ループブロックの直前/直後にあるブロックの基本ブロックの実行回数に対するループブロックの基本ブロックの実行回数の比率となります。複数ブロックの平均化を慎重に行うことで、精度を高めることができます。

これにより、高いトリップカウントを持つループを特定してチューニング作業に集中できます。この手法は、固定カウンターを使用することによって実装できます。

レイテンシーが長い命令 (fmul、fadd、imul) の依存関係チェーンでは、レイテンシーが長い命令の出力を使用できる一方で、ディスパッチがストールする可能性があります。divide/sqrt 実行ユニットを使用する命令を除き、一般にこのようなストールのカウントを支援するイベントは用意されていません。

このような場合は、ARITH イベントを使用することで、これらの命令の実行回数とそれらが実行ユニットを占有した継続期間 (サイクル数) の両方をカウントできます。ARITH.CYCLES_DIV_BUSY イベントは、divide/sqrt 実行ユニットのいずれかが占有されたサイクルをカウントします。

B.4.2 ストールサイクルの分解とコア・メモリー・アクセス

ストールサイクルの分解は、標準的な近似手法を用いて行いますが、これは、パフォーマンスに影響を与えるイベントごとにペナルティーが逐次的に発生することを前提としています。したがって、有効な処理に利用できるサイクルの総損失数は、イベント数 N_i にイベントのタイプごとの平均ペナルティー P_i を掛けたものです。

Counted_Stall_Cycles = Sum (N_i* P_i)

これは、PMU イベントによってカウントされる (またはカウントできる) パフォーマンスに影響を与えるイベントだけを考慮します。最終的には、カウントできないストールの原因は複数ありますが、それらによる影響の総量は次のように概算できます。

Unaccounted stall cycles (不明なストールサイクルの数) = Stall_Cycles - Counted_Stall_Cycles =
UOPS_EXECUTED.CORE_STALLS_CYCLES - Sum (Ni * Pi)_both_threads

このシーケンシャル・ペナルティー・モデルは、非常に単純であり、通常は個々のマイクロアーキテクチャーの問題による影響を多めにカウントするため、不明な構成要素は負数になる場合があります。

B.4.1.1 節で述べたように、UOPS_EXECUTED.CORE_STALL_CYCLES は、スレッド単位ではなくコア単位でカウントされるため、多めにカウントされる可能性があります。そのような場合、スレッド単位でカウント可能なポート 0、1、5 のマイクロオペレーション (uop) ストールを使用することが望ましいことがあります。

Unaccounted stall cycles (スレッドごとの不明なストールサイクルの数) =
UOPS_EXECUTED.PORT015_THREADED_STALLS_CYCLES - Sum (Ni* Pi)

この不明な要素は、パフォーマンス・イベントの不足によってカウントされなかったか、単にデータ収集時に無視された要素を表すために使用されます。

ストールの基準として、「リタイアメント」ポイントを適用することもできます。PEBS イベントである UOPS_RETIRED.STALL_CYCLES には、スレッド単位で評価でき、リタイアするマイクロオペレーション (uop) に関連する IP をハードウェアで取得できるという利点があります。つまり、OS カーネルのクリティカル・セクションにおいて、IP の分布が割り込みの STI/CLI 遅延による影響を受けないため、OS で処理されるプロファイルをより正確に取得できます。

B.4.2.1 マイクロアーキテクチャー条件のコストの測定

ここで説明する方法でストールサイクルの分類を行う場合、まずパフォーマンス・ペナルティーが大きい条件 (例えば、ペナルティーが 10 サイクルよりも大きいイベントなど) に注目します。アウトオブオーダー (OOO) 実行とコンパイラーの複合作用により、ペナルティーが小さいイベント (P < 5 サイクル) は隠蔽されることがあります。

アウトオブオーダー (OOO) エンジンは、命令ストリーム内の両方のタイプのストールを管理しており、命令の依存関係によっていずれかのタイプのストールが発生した場合、実行ユニットをビジー状態に保とうとします。通常、大きなペナルティーの操作はメモリーアクセスに関連し、レイテンシーが長い divide/sqrt 命令などの影響を受けます。

ペナルティーが最も大きいイベントは、キャッシュ階層の L1 または L2 内に存在しないキャッシュラインを必要とするロード操作に関連します。したがって、発生回数だけでなく、どのようなペナルティーであるかを知る必要があります。

レイテンシーの標準的な測定手法は、キューで要求されるサイクル数の平均を測定するものです。

Latency (レイテンシー) = Sum (CYCLES_Queue_entries_outstanding) /Queue_inserts

ここで、「Queue_inserts」は、そのキュー内の未処理のサイクルを引き起こしたエントリーの総数を表します。ただし、キューへの挿入に関連するペナルティー (キャッシュミス) は、レイテンシーをキューの平均占有率で割ったものです。この補正は、オーバーラップしているペナルティーに関連する過剰なカウントを回避するために必要です。

Avg_Queue_Depth (キューの深さの平均) =
Sum (CYCLES_Queue_entries_outstanding) / Cycles_Queue_not_empty

事象ごとのペナルティー (コスト) は次のようになります。

Penalty (ペナルティー) = Latency / Avg_Queue_Depth = Cycles_Queue_not_empty / Queue_inserts

別の考え方をすると、キューを占有するイベントのすべてのペナルティーの合計は、キューが空でない時間を超えることはないことを認識することです。

Cycles_Queue_not_empty (キューが空ではないサイクル) = Events * <Penalty>

前述の標準的な手法は、概念的には単純です。しかし、実際には、ワークロード内での大量のメモリー参照や、さまざまな状態/ロケーション固有のレイテンシーにより、標準的なサンプリング手法は実用的ではありません。Nehalem† マイクロアーキテクチャー・ベースのプロセッサー上では、イベントごとのプリサイス・サンプリング (PEBS) の利用が推奨されます。

サンプリングによるペナルティーのプロファイリング (IP の測定を局所化するために行われる) では、精度が問題となることがあります。L2 ミスのレイテンシーは 40 ~ 400 サイクルとさまざまであるため、必要とされる数のサンプル収集は介入的になる傾向があります。

後述するプリサイス・レイテンシー・イベントは、サンプリングを行う際の正確で柔軟性のある測定手法です。各サンプルでは、待機時間とデータソースの両方が記録されるため、データソースごとの平均レイテンシーを評価できます。また、PEBS ハードウェアは、バッファーが一杯になるまで PMI を生成しないイベントのバッファリングをサポートしているため、ワークロードに介入することなくこのような評価を効率良く行うことができます。

また、コア内で発生し、さまざまな条件、局所性、またはキャッシュ整合性要件によりトラフィックの遅延が生じたメモリーアクセスのコストを測定するため、コア PMU の多数のパフォーマンス・イベントを使用できます。メモリーアクセスのレイテンシーは、L3 の局所性、ローカル・メモリー・コントローラーまたはリモート・コントローラーに装着された DRAM、キャッシュ整合性によって異なります。表 B-7 に、レイテンシーの概算値を示します。

表 B-7 インテル® Xeon® プロセッサー5500 の概算レイテンシー

データソース	レイテンシー
L3 ヒット、ライン排他	~ 42 サイクル
L3 ヒット、ライン共有	~ 63 サイクル
L3 ヒット、別のコアで変更	~ 73 サイクル
リモート L3	100 - 150 サイクル
ローカル DRAM	~ 50 ナノ秒
リモート DRAM	~ 90 ナノ秒

B.4.3 コア PMU のプリサイスイベント

イベントごとのプリサイス・サンプリング (PEBS) により、PMU はイベントを発生させた命令の完了時にアーキテクチャー・ステートと IP を取得できます。これにより、プロファイリングとチューニングにおいて 2 つの利点が得られます。

- 命令空間内のイベント条件の位置を正確にプロファイリングできます。
- 取得したレジスター状態の PEBS レコードを使用して、後処理フェーズで命令の引数を再構成できます。

Nehalem† マイクロアーキテクチャー・ベースのプロセッサーの PEBS 機能は大幅に拡張されており、より多くのそして豊富なプリサイスイベントを使用できます。

この機構は、カウンターのオーバーフローを使用して PEBS データを取得しているため、後続のイベントでデータが取得され、割り込みが発生します。

命令が完了したときの IP 値は次の命令を指すため、取得後の IP 値は「IP +1」と表されることがあります。

プリサイスイベントは、その特性上「リタイア時」にイベントが生成されます。ここでは、プリサイスイベントを、ロードとストアのリタイアメントに関連するメモリー・アクセス・イベントと、すべての命令または特定の非メモリー命令 (分岐、FP アシスト、インテル® SSE マイクロオペレーション (uop)) のリタイアメントに関連する実行イベントに分けて説明します。

B.4.3.1 プリサイス・メモリー・アクセス・イベント

すべてのプリサイス・メモリー・アクセス・イベントには、共通する 2 つの重要な性質があります。

- イベントを生成した命令の IP がハードウェアによって取得されるため、正確に命令を特定できます。取得される IP は次の命令を示しますが、サンプルを単純に 1 つ上の命令に移動します。記録された IP が基本ブロックの最初の命令を指していてもこの方法は機能します。分岐命令はデータをロード / ストアすることがないため、イベントを生成した命令は直前の基本ブロックの最後の命令であることがあります。そのため、取得されたレジスター状態の PEBS のレコードを使用して、後処理フェーズで命令の引数を再構成できます。
- PEBS バッファーには、16 個の汎用レジスター R1 ~ R16 の値がすべて含まれます (R1 は RAX とも呼ばれます)。ロードまたはストアのアドレスは、逆アセンブルと組み合わせて再構成できるため、データ・アクセスのプロファイリングに使用できます。インテル® パフォーマンス・チューニング・ユーティリティーは、これを正確に実行し、各種の強力な分析手法を提供します。

プリサイス・メモリー・アクセス・イベントは、通常、非常に長期にわたる実行ストールの原因となるロードイベントに注目します。このイベントはデータソースごとに分類され、典型的なレイテンシーおよび NUMA 構成におけるデータの局所性を示します。プリサイス・ロード・イベントは、L2、L3、DRAM に対するロード操作のアクセスイベントのみをカウントします。その他すべてのイベントには、L1D/L2 ハードウェア・プリフェッチ要求が含まれます。多くのイベントは RFO 要求も含みますが、これはストアおよびハードウェア・プリフェッチによるものです。

4 つの汎用カウンターはすべて、プリサイス・イベント・データを収集するようにプログラムできます。ロード命令とストア命令の仮想アドレス再構成機能により、キャッシュラインとページの利用効率を分析できます。キャッシュラインとページは物理アドレスによって定義されますが、下位ビットが同じであるため、仮想アドレスを使用できます。

PEBS が命令の完了時にレジスター値を取得する場合、ポインター追跡型のロード操作が取得されないことに注意してください。これは、ロード命令を逆参照アドレスから推測できないためです。

基本的な PEBS メモリー・アクセス・イベントは、次のカテゴリーに分類されます。

- MEM_INST_RETIRED: このカテゴリーは、リタイアした命令のうち、ロード操作を含む命令をカウントします。イベントコード 0BH で選択します。
- MEM_LOAD_RETIRED: このカテゴリーは、リタイアしたロード命令のうち、Umask 値によって選択された特定の条件が発生した命令をカウントします。イベントコードは 0CBH です。
- MEM_UNCORE_RETIRED: このカテゴリーは、リタイアし、アンコア・サブシステムからデータを取得したメモリー命令をカウントします。イベントコードは 0FH です。
- MEM_STORE_RETIRED: このカテゴリーは、リタイアした命令のうち、ストア操作を含む命令をカウントします。イベントコードは 0CH です。
- ITLE_MISS_RETIRED: このカテゴリーは、リタイアした命令のうち、ITLB ミスが発生した命令をカウントします。イベントコードは 0C8H です。

上記の PEBS メモリーイベントの Umask 値および関連する名前サフィックスは、https://perfmon-events.intel.com/ (英語) に示されています。

上に示したプリサイスイベントにより、ロードによるキャッシュミスをデータソースごとに特定できます。ただし、NUMA 構成に関するキャッシュラインの "ホーム" ロケーションは特定されません。イベント MEM_UNCORE_RETIRED.LOCAL_DRAM および MEM_UNCORE_RETIRED.NON_LOCAL_DRAM は例外で、これらをインストルメント済みの malloc 呼び出しと組み合わせると、アプリケーションで使用される重要な隣接するバッファーの NUMA "ホーム" を特定できます。

すべての MEM_LOAD_RETIRED イベントの和は、MEM_INST_RETIRED.LOADS カウントと等しくなります。

L1D ミスのカウントには、MEM_LOAD_RETIRED.L1D_HIT を除くすべての MEM_LOAD_RETIRED イベントを使用します。その場合、MEM_INST_RETIRED.LOADS - MEM_LOAD_RETIRED.L1D_HIT の差ではなく、個々の MEM_LOAD_RETIRED イベントをすべて使用する方が適しています。これは、プリサイスイベントの総数は、対象のイベントを発生させた命令を正しく特定できますが、この節で後述する PEBS のシャドウ化により、イベントの分布が適切でないことがあるためです。

```
L1D_MISSES =
    MEM_LOAD_RETIRED.HIT_LFB + MEM_LOAD_RETIRED.L2_HIT +
    MEM_LOAD_RETIRED.L3_UNSHARED_HIT + MEM_LOAD_RETIRED.OTHER_CORE_HIT_HITM +
    MEM_LOAD_RETIRED.L3_MISS
```

MEM_LOAD_RETIRED.L3_UNSHARED_HIT イベントメトリックには若干説明が必要です。インクルーシブな L3 には、ラインのコピーを保持するコアを特定するビットパターンが備えられています。1 つのビットのみが要求元のコア (非共有状態のヒット) に対して設定されている場合、ほかのコアをスヌープせずに L3 からラインが返されます。複数のビットが設定されている場合、ラインは共有状態であり、L3 内のコピーは最新であるため、この場合もほかのコアをスヌープせずに L3 からラインが返されます。

ラインが別のコアによる所有権のための読み出し (RFO) 状態である場合は、L3 内のコピーは排他状態です。その後、そのコアによってラインが変更され排出されると、L3 内のライトバックされたコピーは変更状態になり、スヌープの必要はなくなります。MEM_LOAD_RETIRED.L3_UNSHARED_HIT はこれらをすべてカウントします。このイベントは、MEM_LOAD_RETIRED.L3_HIT_NO_SNOOP と呼ばれるべきです。

同じ理由で、イベント MEM_LOAD_RETIRED.L3_HIT_OTHER_CORE_HIT_HITM は、MEM_LOAD_RETIRED.L3_HIT_SNOOP と命名されるべきでしょう。

変更されたラインもまた、別のソケットから取得されると、メモリーにライトバックされます。その結果、リモート HITM アクセスは、ホーム DRAM からのアクセスのように見えます。したがって、MEM_UNCORE_RETIRED.LOCAL_DRAM および MEM_UNCORE_RETIRED.REMOTE_DRAM イベントも、リモートプロセッサーのキャッシュ内の変更されたラインによる L3 ミスをカウントします。

MEM_LOAD_RETIRED.DTLB_MISSES の動作は、インテル® Core™2 プロセッサーとは異なります。以前は、非プリサイスイベントと同様、このイベントはページに対する最初のミスだけをカウントしていました。このイベントは、ミスを発生させたすべてのロードをカウントするように変更されています (2 番目以降のミスも含む)。

B.4.3.2 ロード・レイテンシー・イベント

Nehalem† マイクロアーキテクチャー・ベースのインテル® プロセッサーは、イベントコード 0BH と Umask 値 10H (LATENCY_ABOVE_THRESHOLD) を使用して、「ロード・レイテンシー・イベント」である MEM_INST_RETIRED をサポートします。このイベントは、ロードをサンプリングして、命令の実行と実際にデータが供給されるまでのサイクルの数を記録します。測定されたレイテンシーが、MSR 0x3F6 (ビット 15:0) にプログラムされた最小レイテンシーよりも大きい場合、カウンターはカウントアップされます。

カウンターのオーバーフローによって PEBS 機構を実現しています。PMU は、レイテンシーのしきい値を満たす次のイベントで、測定されたレイテンシー、仮想アドレスまたはリニアアドレス、データソースを PEBS レコード・フォーマットで PEBS バッファーに書き込みます。仮想アドレスは既知のロケーションに取得されるため、サンプリング・ドライバーも仮想アドレスから物理アドレスへの変換を行い物理アドレスを取得する可能性があります。物理アドレスによって NUMA のホーム・ロケーションが特定され、原則としてキャッシュ占有の詳細な分析が可能になります。

また、リタイアメントの前にアドレスが取得されるため、"MOV RAX, [RAX+const]" のエンコードを追跡するポインターアドレスも取得されます。レイテンシーのしきい値を指定するには MSR_PEBS_LD_LAT_THRESHOLD MSR が必要になるため、一定の期間中に 1 つの最小レイテンシー値だけをコア上でサンプリングできます。これを可能にするため、インテル® パフォーマンス・ツールは、このイベント利用を 4 つのカウンターに制限することによってスケジューリングを簡素化しています。表 B-8 に、インテル® Performance Tuning Utilitiy (インテル® PTU) およびインテル® VTune™ プロファイラーがロード・レイテンシー・イベント向けに使用するイベント・プログラミング構成の例を示します。

さまざまな最小レイテンシーのしきい値が MSR_PEBS_LD_LAT_THRESHOLD (アドレス 0x3F6) に指定されます。

表 B-8 ロード・レイテンシー・イベント・プログラミング

ロード・レイテンシーのプリサイスイベント	MSR 0x3F6	Umask	イベントコード
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_4	4	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8	8	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_10	16	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_20	32	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_40	64	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_80	128	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_100	256	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_200	512	10H	0BH
_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8000	1024	10H	0BH

ロード・レイテンシー・イベントの PEBS アシスト機構によって各 PEBS レコードに書き込まれる 3 つのフィールドのうちの 1 つがデータソースの局所性情報をエンコードします。

表 B-9 ロード・レイテンシー PEBS レコードのデータソース・エンコード

エンコード	説明
0x0	未知の L3 キャッシュミス。
0x1	最小レイテンシーのコアキャッシュをヒット。この要求は L1 データキャッシュが対応。
0x2	保留中のコアキャッシュをヒット。同じキャッシュライン・アドレスに対して未処理のコア・キャッシュ・ミスがすでに進行しています。データはまだデータキャッシュ内に存在せず、まもなくキャッシュにコミットされるフィルバッファーがあります。
0x3	この要求は L2 が対応。
0x4	L3 ヒット。コヒーレンシー操作を必要としないアンコア内の L3 キャッシュにヒットしたローカルまたはリモートホーム要求 (スヌープ)。
0x5	L3 ヒット (ほかのコア・ヒット・スヌープ)。L3 キャッシュにヒットし、変更されたコピーが見つからなかったコア間のスヌープを持つ別のプロセッサー・コアによって処理された、ローカルまたはリモートホーム要求 (クリーン)。
0x6	L3 ヒット (ほかのコア HITM)。L3 キャッシュにヒットし、変更されたコピーが見つかったコア間のスヌープを持つ別のプロセッサー・コアによって処理された、ローカルまたはリモートホーム要求 (HITM)。
0x7	予約済み
0x8	L3 ミス (リモート・キャッシュ・フォワーディング)。L3 キャッシュをミスし、変更されたコピーが見つからなかったパッケージ間のスヌープに続く転送済みデータによって処理された、ローカルホーム要求 (リモートホーム要求はカウントされません)。
0x9	予約済み。
0xA	L3 ミス (ローカル DRAM が共有状態 (S) に遷移) L3 キャッシュをミスし、ローカル DRAM によって処理されたローカルホーム要求。
0xB	L3 ミス (リモート DRAM が共有状態 (S) に遷移) L3 キャッシュをミスし、リモート DRAM によって処理されたリモートホーム要求。
0xC	L3 ミス (ローカル DRAM が排他状態 (E) に遷移) L3 キャッシュをミスし、ローカル DRAM によって処理されたローカルホーム要求。
0xD	L3 ミス (リモート DRAM が排他状態 (E) に遷移) L3 キャッシュをミスし、リモート DRAM によって処理されたリモートホーム要求。
0xE	I/O、入出力操作の要求。
0xF	キャッシュ不可能メモリーに対する要求。

レイテンシー・イベントは、サイクル・アカウンティング分解のペナルティーを測定するための推奨される手法です。この PEBS イベントによって PMI が発生するたびに、レイテンシーに関連するロードとキャッシュラインのデータソースが PEBS バッファーに記録されます。このキャッシュラインのデータソースは、データソース・フィールドの下位 4 ビットおよび上記の表から導かれます。収集されたデータから、16 個のソースのそれぞれに対する平均レイテンシーを評価できます。最小レイテンシーは一度に 1 つずつしか収集できないため、MLC ヒットとリモートソケット DRAM のレイテンシーを評価しにくい場合があります。ただし、すべてのオフコアソースに対し適切に分散するには、32 サイクルの最小レイテンシーが科せられます。インテル® PTU バージョン 3.2 を使用すると、データ・プロファイリング・モードでレイテンシーの分布を表示し、そのイベントに対する高度なイベントフィルター処理を適用できます。

B.4.3.3 プリサイス実行イベント

コア PMU 内の PEBS 機能は、ロード命令およびストア命令を超えるものであり、選択されたタイプのリタイアした分岐および予測が外れた (その後リタイアした) 分岐に関する、分岐、near コール、条件分岐をすべてプリサイスイベントによってカウントできます。これらのイベント対し、PEBS バッファーは分岐のターゲットを含みます。最後の分岐レコード (LBR) も取得されている場合は、分岐命令のロケーションも判断できます。

分岐が処理されると、PEBS バッファー内の IP 値も LBR の最後のターゲットとして認識されます。条件分岐がジャンプしなかった場合、分岐せずにリタイアした分岐は PEBS バッファー内の IP の 1 つ前の命令となります。

near コールがリタイアした場合、イベント・ベース・サンプリング (EBS) により正確な関数呼び出しの回数を収集できます。これは、関数をインライン展開するかどうか判断する測定基準となります。呼び出し回数を測定するには、呼び出しをサンプリングする必要があり、その他のトリガーは、適切に修正されることが保証されないバイアスの原因となります。

プリサイス分岐イベントは、https://perfmon-events.intel.com/ (英語) にあるイベントコード C4H に示されます。

プリサイスイベントに関連したサンプリング結果には誤差が生じる可能性があります。これは、PMU カウンターのオーバーフローと PEBS ハードウェアを補正する間の時間的な遅延によるものです。タイミングシャドウにより、この期間中はイベントを検出できません。この影響を理解するため、関数呼び出しチェーンについて考えてみます。実行時間が長い関数「foo」は、実行時間が非常に短い 3 つの関数「foo1」、「foo2」、「foo3」を呼び出します。「foo1」は「foo2」を呼び出し、「foo2」は「foo3」を呼び出し、その後に実行時間が長い関数「foo4」が続きます。foo1、foo2、foo3 の実行時間がシャドウ期間よりも短い場合、PEBS でサンプリングされた呼び出し分布は大きくゆがんだものになります。次に例を示します。

- foo の呼び出しでオーバーフローが発生すると foo1 の呼び出しが実行されるまでに PEBS 機構が実施され、foo から foo1 の呼び出しを示すサンプルが収集されます。
- しかし、foo1、foo2、または foo3 の呼び出しによってオーバーフローが発生すると、foo4 の本体で呼び出しが実行されるまで PEBS 機構は実施されません。そのため、foo2、foo3、foo4 の呼び出しは、PEBS でサンプリングされた呼び出しとして示されません。

シャドウは、すべての PEBS イベントの分布に影響します。また、リタイアした分岐イベント(PEBS または非 PEBS)と LBR 内の最後のエントリーを組み合わせて特定される基本ブロックの実行回数の分布にも影響されます。PMU カウンターのオーバーフローと LBR の確定の間に遅延がなければ、最後の LBR エントリーから分岐したリタイア済みの分岐をサンプリングし、そこから基本ブロック実行回数を特定できます。最後に分岐した分岐と前回のターゲット間のすべての命令は一度だけ実行されます。

このようなサンプリングを使用することで、均一なサンプリングによってリタイアした「ソフトウェア」命令イベントを生成し、そのイベントを基に基本ブロックの実行回数を特定できます。残念ながら、シャドウでは短い基本ブロックの末端にある分岐は LBR の最後のエントリーにはならないため、測定結果が片寄る傾向があります。これは、基本ブロック内のすべての命令は同じ回数だけ実行されるように構成されているためです。

呼び出し回数と基本ブロックの実行回数に対するシャドウの影響は、LBR 内のエントリーを平均化することによって大幅に軽減できる可能性があります。これについては、LBR に関する節で説明します。

一般に、ワークロード内のすべての命令に分岐が占める割合は 10% を超えるため、ループの最適化では、高いトリップカウントを持つループに注目する必要があります。ループカウントに関しては、終了条件の判定において誘導変数をトリップカウントと比較するのが一般的です。これは、たとえ高度な最適化が適用されても、ループの本体内で誘導変数が使用されている場合に該当します。したがって、アンロール操作のループシーケンスは次のようになります。

```
add rcx, 8
cmp rcx, rax
jnge triad+0x27
```

この場合、2 つのレジスター rax および rcx が、トリップカウントとインダクション変数です。リタイアした条件分岐イベント向けの PEBS バッファーを取得すると、比較命令の 2 つのレジスターの平均値を評価できます。平均値が大きい方がトリップカウントです。これにより、平均値、RMS、最小値、最大値、さらに記録された値の分布まで評価できます。

B.4.3.4 最後の分岐レコード (LBR)

LBR は、分岐処理されてリタイアした各分岐のソースとターゲットを取得します。Nehalem† マイクロアーキテクチャー・ベースのプロセッサーは、ローテート・バッファー内の 16 組のソース/ターゲットアドレスを追跡できます。タイプ別および特権レベル別の分岐命令のフィルター処理は、専用機能である MSR_LBR_SELECT を使用することによって可能となります。つまり、LBR 機構は、リング 0 またはリング 3 特権レベル、あるいはその両方 (デフォルト) で発生する分岐を取得するようにプログラムできます。さらに、記録された分岐処理済み分岐をフィルター処理することもできます。MSR_LBR_SELECT を使用して指定できるフィルター処理オプションのリストについては、『インテル®

64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 17 章「Debugging, Branch Profiles and Time-Stamp Counter」で説明されています。

デフォルトでは、すべての特権レベル (すべてのビットが 0) ですべての分岐が取得されます。また、ビット 1 (リング 3 を取得)、ビット 3 (near コールを取得)、ビット 6、ビット 7 を除くすべてのビットを 1 に設定することで、リング 3 コールと無条件ジャンプだけが LBR 内に残されます。このようにプログラミングすることで、LBR は最後の 16 個の処理済みの呼び出しとリタイアした無条件ジャンプ、およびそれらのターゲットをバッファー内に保持できます。

その後、PMU サンプリング・ドライバーは、この限定された「呼び出しチェーン」を任意のイベントとともに取得することによって「コールツリー」コンテキストを提供可能になります。残念ながら、無条件ジャンプを含むと問題が生じます。これは、特にループ内に if-else 構文が存在する場合に当てはまります。

すべてのレベルで頻繁に呼び出される関数では、コンテキストを明確にするためリターンを含めて追加します。ただし、取得可能な呼び出しチェーンの入れ子数が減少します。明らかな使用方法は、レイテンシーが極めて長いロードのサンプリングをトリガーすることです。これにより、頻繁に競合するロックされた変数にアクセスするサンプルを改善し、呼び出しチェーンを取得してロックを使用するコンテキストを特定します。

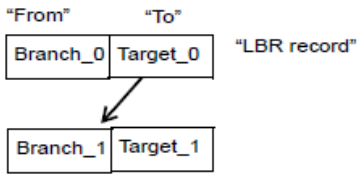
呼び出し回数と関数の引数

BR_INST_RETIRED.NEAR_CALL イベントによってトリガーされた PMI の LBR が取得された場合、LBR 内の最後のエントリーを参照するだけで呼び出し元の関数ごとに呼び出しカウントを判定できます。PEBS IP は LBR 内の最後のターゲット IP と等しいため、呼び出し元の関数のエントリーポイントを示します。同様に、LBR バッファー内の最後のソースは、呼び出し元の関数内からのコールサイトとなります。完全な PEBS レコードを取得した場合、64 ビット OS では引数の数が制限されている関数の呼び出し回数と関数の引数の両方をサンプリングできます。

LBR と基本ブロックの実行回数

もう 1 つの興味深い用法は、フィルターを用いずに BR_INST_RETIRED.ALL_BRANCHES イベントと LBR を使用して基本ブロックの実行速度を評価することです。LBR は分岐したすべての分岐を取得するため、分岐 IP (ソース) と LBR バッファー内の直前のターゲット間のすべての基本ブロックが 1 回実行されます。特定のロードモジュールの基本ブロックの実行回数を簡単に評価する方法は、すべての基本ブロックの開始位置のマップを作成することです。その後、PEBS アドレス (ターゲットであるが、おそらく分岐処理された分岐のアドレスでないために LBR バッファー内に存在する必要がない) で開始され、目的のロードモジュールと一致しないアドレスが見つかるまで LBR を逆方向に探索する BR_INST_RETIRED.ALL_BRANCHES の PEBS の集合によってトリガーされた各サンプルについて、実行された基本ブロックをすべてカウントします。この値は「number_of_basic_blocks」と呼ばれ、すべてのブロックの実行回数が 1/(number_of_basic_blocks) ずつカウントアップされます。この方法により、アクティブな分岐の分岐率と未分岐率を取得できます。

LBR にリストされる分岐は、ソース IP と (同じモジュール内の) 前回のターゲット IP の間にあるすべての分岐命令が分岐するわけではありません。これを次の図に示します。



イベントカウントごとに、Target_0 と Branch_1 の間にあるすべての命令が 1 回リタイアします。
イベントカウントごとに、Target_0 と Branch_1 の間にあるすべての基本ブロックが 1 回実行されます。
Target_0 と Branch_1 の間にあるすべての分岐命令は分岐しません。

図 B-6 LBR レコードと基本ブロック

この 16 組の LBR レコードは、サンプリング・プロセスで特定の命令が偏って集計された PEBS サンプルの誤差の修正に役立ちます。図 B-7 に、PEBS サンプルの分布が偏っている状態を示します。

正常な実行のフローにおける基本ブロックの数について考えてみます。基本ブロックには、実行に 20 サイクル要するものもあれば、2 サイクルのものもあります。シャドウは 10 サイクルを要します。オーバーフロー条件が発生するたびに、補正された PEBS の遅延が少なくとも 10 サイクル発生します。PEBS が補正されると、次のイベント生成条件で PEBS レコードが取得されます。PEBS レコードを使用してサンプリングされた命令アドレスの分布は、図 B-7 の中央に示すように偏っています。この概念上の例では、すべての分岐がこれらの基本ブロック内で分岐されることを前提としています。

PEBS サンプルが偏って分布している状態では、最後の基本ブロックの分岐 IP は、最もサンプリング数の少ない分岐 IP アドレス (2 番目の基本ブロック) の 5 倍記録されていることが分かります。

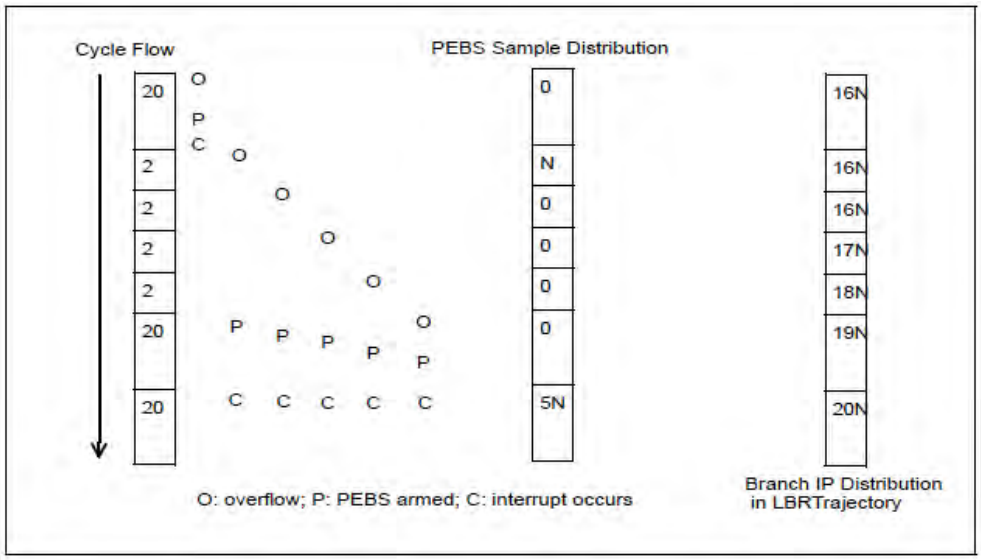


図 B-7 LBR レコードによる偏った PEBS サンプルの分布の修正

この状況では、基本ブロックによってサンプルが取得されないように見えたり、多くのサンプルが何度も取得されたりします。この例では、各エントリーに 1/(LBR の軌道内の基本ブロックの数) を上積みすることは、一番右のテーブル内の数字を 16 で割ることと同等です。そのため、PEBS サンプルを直接発生させないものを含め、すべての基本ブロックではるかに正確な実行回数 ((1.25-> 1.0) * N) が得られます。

これは、インテル® Core™2 プロセッサーには、さまざまな方法で利用できるプリサイス命令リタイアイベントが存在するためです。また、uops_retired、多様なインテル® SSE 命令クラス、FP アシストなどのプリサイスイベントもあります。FP アシストイベントは、インテル® SSE FP 命令ではなく、x87 FP アシストだけを検出することに注意してください。すべてのアシストの検出については、パイプライン・フロントエンドの節で説明します。

命令リタイアイベントには、特別な使用方法があります。分布は均一ではありませんが、それぞれの合計は正確です。基本ブロック内のすべての命令で記録された値が平均化されていれば、一定の基本ブロックの実行回数を抽出できます。上記のカウント済みループ手法を適用できない場合、基本ブロックの実行比率を用いてループのトリップカウントを算出します。

また、PEBS バージョン (汎用カウンター) の命令リタイアイベントを使用することにより、たとえ STI/CLI セマンティクスであっても、OS の実行を正確にプロファイルできます。これは、クリティカル・セクションの完了後に PEBS 割り込みが発生しますが、データはすでに適切に収集されているためです。cmask 値が非常に大きい値に設定され、invert 条件が適用されている場合、結果はすべて真であり、イベントはコアサイクル (halted および unhalted) をカウントします。

そのため、サイクル数とリタイアした命令数の両方を正確にプロファイルできます。UOPS_RETIRED.ANY イベントも正確であり、これを使用してリング 0 の実行をプロファイルすることで実行精度を改善できます。この目的に利用できるプリサイスイベントは、https://perfmon-events.intel.com/ (英語) にあるイベントコード C0H、C2H、C7H、F7H に示されています。

局所性に関連したパフォーマンスの問題やキャッシュ・コヒーレンシーの問題をドリルダウンするには、パフォーマンス監視イベントを使用する必要があります。各プロセッサー・コアには、L2 ミスによるメモリー・アクセス・トラフィックの要求をバッファーに格納するエントリーをアンコア・サブシステムに割り当てるスーパーキューが備わっています。表 B-10 に、L2 ミスに関連したパフォーマンスの問題をドリルダウンできる、コア PMU 内で利用可能な各種パフォーマンス・イベントを示します。

表 B-10 L2 ミスをドリルダウンするためのコア PMU イベント

コア PMU イベント	Umask	イベントコード
OFFCORE_REQUESTS.DEMAND.READ_DATA1	01H	B0H
OFFCORE_REQUESTS.DEMAND.READ_CODE1	02H	B0H
OFFCORE_REQUESTS.DEMAND.RFO1	04H	B0H
OFFCORE_REQUESTS.ANY.READ	08H	B0H
OFFCORE_REQUESTS.ANY.RFO	10H	B0H
OFFCORE_REQUESTS.UNCACHED_MEM	20H	B0H
OFFCORE_REQUESTS.L1D.WRITEBACK	40H	B0H
OFFCORE_REQUESTS.ANY	80H	B0H

注意:

1. *DEMAND* イベントは、L1D キャッシュ・ハードウェア・プリフェッチによって行われたすべての要求を含みます。

表 B-11 に、スーパーキュー操作に関連したパフォーマンスの問題をドリルダウンできる、コア PMU 内で利用可能な各種パフォーマンス・イベントを示します。

表 B-11 スーパーキュー操作のためのコア PMU イベント

コア PMU イベント	Umask	イベントコード
OFFCORE_REQUESTS_BUFFER_FULL	01H	B2H

また、L2 ミスは、データ移動元の属性と応答属性によってさらにドリルダウンできます。データ移動元の属性と応答属性を指定するマトリクスは、アドレス 1A6H で専用の MSROFFCORE_RSP_0 を使用します。表 B-12 および表 B-13 を参照してください。

表 B-12 オフコア応答をドリルダウンするコア PMU イベント

コア PMU イベント	OFFCORE_RSP_0 MSR	Umask	イベントコード
OFFCORE_RESPONSEL	表 B-13 を参照してください。	01H	B7H

表 B-13 OFFCORE_RSP_0 MSR のプログラミング

	位置	説明	注記
要求タイプ	0	要求データ Rd = DCU 読み出し (パーシャル、DCU プリフェッチを含む)	
	1	要求 RFO = DCU RFO	
	2	要求 Ifetch = IFU フェッチ	
	3	ライトバック = L2_EVICT/DCUWB	
	4	PF データ Rd = L2 プリフェッチ読み出し	
	5	PF RFO = L2 プリフェッチ RFO	
	6	PF Ifetch = L2 プリフェッチ命令フェッチ	
	7	その他	非テンポラルなストアを含む
	8	L3_HIT_UNCORE_HIT	排他ライン
	9	L3_HIT_OTHER_CORE_HIT_SNP	クリーンなライン
	10	L3_HIT_OTHER_CORE_HITM	変更されたライン
	11	L3_MISS_REMOTE_HIT_SCRUB	複数のコアが使用
	12	L3_MISS_REMOTE_FWD	1 つのコアが使用するクリーンなライン
	13	L3_MISS_REMOTE_DRAM	
	14	L3_MISS_LOCAL_DRAM	
	15	Non-DRAM	非 DRAM 要求

表 B-13 によれば、理論上は MSR_OFFCORE_RSP_0 の設定の組み合わせは 2^16 個許容されます。しかし、8 ビット値のサブセットを組み合わせて「要求タイプ」と「応答タイプ」を指定する方がより効果的です。表 B-14 に、より一般的な 8 ビットマスク値を示します。

表 B-14 OFFCORE_RSP_0 MSR の一般的な要求タイプと応答タイプ

要求タイプ	マスク	応答タイプ	マスク
ANY_DATA	xx11H	ANY_CACHE_DRAM	7FxxH
ANY_IFETCH	xx44H	ANY_DRAM	60xxH
ANY_REQUEST	xxFFH	ANY_L3_MISS	F8xxH
ANY_RFO	xx22H	ANY_LOCATION	FFxxH
CORE_WB	xx08H	IO	80xxH
DATA_IFETCH	xx77H	L3_HIT_NO_OTHER_CORE	01xxH
DATA_IN	xx33H	L3_OTHER_CORE_HIT	02xxH
DEMAND_DATA	xx03H	L3_OTHER_CORE_HITM	04xxH
DEMAND_DATA_RD	xx01H	LOCAL_CACHE	07xxH
DEMAND_IFETCH	xx04H	LOCAL_CACHE_DRAM	47xxH
DEMAND_RFO	xx02H	LOCAL_DRAM	40xxH
OTHER1	xx80H	REMOTE_CACHE	18xxH
PF_DATA	xx30H	REMOTE_CACHE_DRAM	38xxH
PF_DATA_RD	xx10H	REMOTE_CACHE_HIT	10xxH
PF_IFETCH	xx40H	REMOTE_CACHE_HITM	08xxH
PF_RFO	xx20H	REMOTE-DRAM	20xxH
PREFETCH	xx70H		

注意:

1. MSR_OFFCORE_RSP_0 を値 4080H に設定すると、PMU が間違ったカウントを報告する場合があります。ローカル DRAM に対する非テンポラルなストアはカウントされません。

B.4.3.5 コアごとの帯域幅を測定

個々のコアに対するすべてのメモリー・トラフィック帯域幅を測定するのは複雑ですが、コア PMU やアンコア PMU は、コアあたり帯域幅の重要な要素を測定する機能を備えています。

マイクロアーキテクチャー・レベルでは、(非テンポラルな書き込みにある程度類似した) L2 からのライトバック/排出のため L3 にはバッファリングがあります。変更されたラインを L2 から排出すると、ラインは L3 へライトバックされます。L3 内のラインは、(必要であるとしても) しばらくして L3 から排出されるときにのみメモリーに書き込まれます。また、L3 はアンコア・サブシステムの一部であり、コアの一部ではありません。

ラインの変更による L3 排出のためのメモリーへのライトバックは、アンコア PMU ロジック内の個々のコアに関連付けることができません。そのため、すべてのコアの書き込み総帯域幅をアンコア PMU 内のイベントによって測定することができません。読み出し帯域幅と非テンポラルな書き込み帯域幅は、コア単位で測定できます。2 つの物理プロセッサーが搭載されたシステムでは、メモリー帯域幅の NUMA 特性により、これらの 2 つのコンポーネントの測定値をソケットごとのコア帯域幅に区分する必要があります。

ソケットごとの読み出し帯域幅は次のイベントによって測定できます。

OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_LOCAL_DRAM
OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_REMOTE_DRAM

すべてのソケットの読み出し総帯域幅は次のイベントによって測定できます。

OFFCORE_RESPONSE_0.DATA_IFETCH.ANY_DRAM

非テンポラルなストアのソケットごとの帯域幅は次のイベントによって測定できます。

OFFCORE_RESPONSE_0.OTHER.L3_MISS_LOCAL_CACHE_DRAM
OFFCORE_RESPONSE_0.OTHER.L3_MISS_REMOTE_DRAM

非テンポラルなストアの総帯域幅は次のイベントによって測定できます。

OFFCORE_RESPONSE_0.OTHER.ANY.CACHE_DRAM

"CACHE_DRAM" エンコーディングを使用すると、表 B-14 の脚注に記載した問題を回避できます。上記のイベントにはいずれも、変更されたキャッシュ可能ラインのライトバックに関連する帯域幅は含まれません。

B.4.3.6 キャッシュミスに関するその他の L1/L2 イベント

OFFCORE_RESPONSE_0 イベントおよび後述する PEBS に加えて、同様に使用できるイベントがいくつかあります。また、offcore_response_0 イベントコードはカウンター 0 のみでサポートされるため、offcore_response_0 イベントを補助する追加のイベントも用意されています。

アーキテクチャーで定義されたイベント LONGEST_LAT_CACHE_ACCESS を使用して L2 ミスをカウントすることもできます。しかし、このイベントは L1D/L2 ハードウェア・プリフェッチによる要求も含んでいるため、その効用は限定されます。前述の OFFCORE_REQUESTS イベントに加えて、一部の L2 アクセスイベントは、L2 アクセスと L2 ミスの両方をタイプ別にドリルダウンするために使用できます。L2_RQSTS イベントと L2_DATA_RQSTS イベントを使用して、分類されたアクセスタイプを識別します。すべての L2 アクセスイベントにおいて、PREFETCH は L2 ハードウェア・プリフェッチのみを表します。DEMAND は、L1D ハードウェア・プリフェッチによるロードおよび要求を含みます。

L2_LINES_IN イベントと L2_LINES_OUT イベントは、インテル® Core™2 プロセッサーのイベントとは少し異なります。L2_LINES_OUT イベントは、クリーンおよびダーティーで排出されたラインを分離し (すなわち、ライトバック)、それらが L1D 要求または L2 ハードウェア・プリフェッチによって排出されたかどうかを判断するために使用できます。

イベント L2_TRANSACTIONS は、L2 との相互作用をすべてカウントします。

書き込みとロックされた書き込みは、複合イベント L2_WRITE でカウントされます。

L2_RQSTS、L2_DATA_RQSTS、L2_LINES_IN、L2_LINES_OUT、L2_TRANSACTIONS、L2_WRITE などの派生イベントの詳細は、https://perfmon-events.intel.com/ (英語) にあるイベントコード 24H、26H、F1H、F2H、F0H、27H に示されています。

B.4.3.7 TLB ミス

2 番目に数が多いメモリーアクセス遅延は、リニアアドレスから物理アドレスへの変換が TLB 内の限られたエントリーでマッピングされていることに関連します。第 1 レベルの TLB でのミスは、通常はアウトオブオーダー (OOO) 実行とコンパイラーのスケジューリングによって隠蔽可能な非常に小さなペナルティーです。共有状態の TLB でミスが発生するとページウォークが起動されますが、このペナルティーは実行時に表れます。

この (非 PEBS) TLB ミスイベントは、次の 3 つに分類できます。

- DTLB ミス (DTLB_MISSES) および派生イベントは、イベントコード 49H でプログラムされます。
- ロード DTLB ミス (DTLB_LOAD_MISSES) および派生イベントは、イベントコード 08H でプログラムされます。
- ITLB ミス (ITLB_MISSES) および派生イベントは、イベントコード 85H でプログラムされます。

ストア DTLB ミスは、DTLB ミスとロード DTLB ミスの差から評価できます。

それぞれ umask 値でプログラムされる多数のサブイベントがあります。上記のイベントの派生イベントの Umask に関する詳細は、https://perfmon-events.intel.com/ (英語) を参照してください。

B.4.3.8 L1 データキャッシュ

L1 データキャッシュ操作の分析に使用できるいくつかの PMU イベントがあります。これらのイベントは、4 つの汎用カウンターのうちの最初の 2 つ (すなわち、IA32_PMC0 と IA32_PMC1) によってのみカウントできます。ほとんどの L1D イベントは明らかです。

L1D の参照総数は、L1D_ALL_REF によってカウントできますが、キャッシュ可能な参照のみ、またはすべての参照がカウントされます。キャッシュ可能な参照は、L1D_CACHE_LOAD および L1D_CACHE.STORE によってロードおよびストアに分類できます。これらのイベントは、それぞれの Umask 値によってさらに MESI ステート別に分類され、I ステート参照はキャッシュミスを示します。

L1D 内で変更されたラインが排出されると L2 へライトバックが発生し、これらは、L1D_WB_L2 イベントによってカウントされます。これらは、umask 値により、さらに L2 内のキャッシュラインを MESI ステート別に分類できます。

ロックされた参照もまた、L1D_CACHE_LOCK イベントによってカウントでき、これも、L1D ラインの MES ステート別に分類されます。

L1D に取り込まれるラインの総数、つまり M ステートに達した数と、スヌープにより排出される変更済みラインの数は、L1D イベントおよびその Umask のバリエーションによってカウントされます。

L1D イベントは、https://perfmon-events.intel.com/ (英語) にあるイベントコード 28H、40H、41H、42H、43H、48H、4EH、51H、52H、53H、80H、83H に示されています。

ロードをアクティブなストアバッファーから転送できないケースがあります。これは、主に長いロードがより短いストアとオーバーラップすることに関係します。この状況を検出するイベントはありません。また、「フォルス・ストア・フォワーディング」と呼ばれるケースでは、アドレスの下位 12 ビットだけが一致します。これは、4K エイリアシングと呼

ばれることもあり、イベントコード 07H と Umask 01H を持つイベント「PARTIAL_ADDRESS_ALIAS」によって検出できます。

B.4.4 フロントエンドの監視イベント

分岐予測ミスの影響は、コードの変更や拡張インライン展開によって排除できる可能性があります。その他大部分のフロントエンドのパフォーマンス低下は、コードの生成時に対処する必要があります。そのような問題の分析はコンパイラー開発者にとって有用です。

B.4.4.1 分岐予測ミス

B.4.3.3 節で PEBS に関連して説明したリタイアした分岐イベントに加えて、分岐予測ミス機能は、LBR を使用した分岐先の分岐ロケーションと、PEBS バッファー内に取得されるターゲット・ロケーションを特定するように拡張されています。また、分岐予測に関連するその他の多くの PMU イベント (イベントコード E6、E5、E0、68、69) は、パフォーマンス・チューニングよりもハードウェア設計に関連します。

分岐予測ミスは、それ自体はパフォーマンス・ボトルネックの指標ではありません。これは、ディスパッチのストールおよび命令スタベーション条件 (UOPS_ISSUED:C1:I1 – RESOURCE_STALLS.ANY) に関連付けられます。このようなストールは、命令キャッシュミスや ITLB ミスに関連する可能性があります。このような状況には、プリサイス ITLB ミスイベントが役立つ場合があります。命令キャッシュ・ミスイベントと ITLB ミスイベントは、イベントコード 80H、81H、82H、85H、AEH に示されます。

B.4.4.2 フロントエンドのコード生成メトリック

そのほかのフロントエンド・イベントは、コード生成がアウトオブオーダー (OOO) エンジンに対する命令のデコードおよびマイクロオペレーション (uop) の発行の問題と適切に作用しない状況を特定するのに役立ちます。例えば、レングス変更プリフィクスに関連する 16 ビット即値、ROB 読み出しポートのストール、命令アライメントとループ検出の干渉、命令をデコードする帯域幅の制約の問題などがあります。LSD (Loop Stream Detector) の動作は、シグナル監視機能で CMASK 値を使用して監視します。これらのイベントのうちのいくつかは、https://perfmon-events.intel.com/ (英語) にあるイベントコード 17H、18H、1EH、1FH、87H、A6H、A8H、D0H、D2H に示されます。

一部の命令 (FSIN、FCOS、その他の超越関数命令) は、MSROM のアシストによってデコードされます。複雑なマイクロオペレーション (uop) フローをデコードするため MSROM のアシストを受ける命令が頻繁に発行されることがあります。これは、そのような状況を減らすため命令選択を改善する必要性を示します。UOPS_DECODED.MS イベントを使用して、命令選択の改善によるメリットが得られる可能性があるコード領域を特定できます。

その他、デノーマル FP 値や QNaN など非正規 FP 値に対して起動される FP アシストによっても、このイベントがトリガーされる可能性があります。そのような場合のペナルティーは、アシストに必要なマイクロオペレーション (uop) の実行と、適切な状態を確保するために行われるパイプラインのクリアです。

その結果この状況は、MACHINE_CLEAR.CYCLES による非常に明確なシグネチャーとマイクロコード・シーケンサー UOPS_DECODED.MS によって挿入されるマイクロオペレーション (uop) を持ちます。実行ペナルティーはこの 2 つの和となります。これらのイベントコードは、D1H および C3H に示されています。

B.4.5 アンコア・パフォーマンス監視イベント

アンコア・サブシステムには、図 B-4 の図に示した L3、IMC、インテル® QPI の各ユニットが含まれます。アンコア・サブシステム内のアンコア PMU は、8 つの汎用カウンターと 1 つの固定カウンターで構成されます。アンコア内の固定カウンターは、コアとは異なる周波数で動作するアンコア・クロック・ドメインの unhalted クロックサイクルを監視します。

アンコアは、PMI 割り込みを単独で生成することができません。コア PMU が論理プロセッサー単位でコア PMI を発生させるのに対して、アンコア PMU は、プロセッサー・コア内の割り込みハードウェアを使用してコア単位で PMI を発生します。アンコアのカウンターがオーバーフローすると、どのコアにシグナルを送信して PMI を発生させるか、ビットパターンが使用されます。アンコア PMU は、カウンターがオーバーフローしたイベントを発生させたコア、プロセッサー ID、またはスレッド ID を認識できません。そのため、アンコアイベントのサンプリングを行う最も合理的な手法は、パッケージ内のすべての論理プロセッサーで PMI を発生させることです。

キューの占有と挿入を監視するさまざまなイベントがあります。そのほかに、キャッシュライン転送、DRAM ページングポリシーの統計情報、スヌープのタイプ/応答などをカウントするイベントも用意されています。アンコアは、メモリーに対する総帯域幅を測定できる唯一の場所です。これについては、アンコアのすべてのコンポーネントおよびそのイベントを示した後で説明します。

B.4.5.1 グローバルキューの占有

各プロセッサー・コアは、L2 ミスによるメモリー・アクセス・トラフィックの要求をバッファーに格納するスーパーキューを持っています。アンコアは、これらのプロセッサー・コアからのトランザクション要求を処理するためグローバルキューを保持し、L3、IMC、またはインテル® QPI の各リンクから到達したデータ・トラフィックをバッファーに格納します。

グローバルキュー (GQ) 内には、3 つのトランザクション・タイプに対応する次の 3 つの「トラッカー」があります。

- パッケージ上の読み出し要求: 32 エントリーのトラッカーキュー。
- パッケージ上の書き込み要求: 16 エントリーのトラッカーキュー。
- 「ピア」から到達した要求: 12 エントリーのトラッカーキュー。

「ピア」は、インテル® QuickPath インターコネクトからの任意の要求を表します。

3 つのトラッカーすべてに対し、占有、挿入、一杯であるサイクル、空でないサイクルを監視できます。また、ロード要求がステージを通過する際に、各ステージに関連する占有と挿入を監視できます。これにより、ロードによるアンコア・メモリー・アクセスの "サイクル・アカウンティング" による詳細な調査が可能となります。

アンコア・カウンターでキューの占有率を監視する場合、最初にすべてのアンコアキューを空にする必要があります。これは、バスロックを発行するソフトウェア・ツールのドライバーによって行なわれます。この操作は、カウンターを最初にプログラムするときにのみ行います。それ以降、カウンターはキューの状態を正しく反映するため、(例えば、別のバスロックが発行されることなく) サンプリングが繰り返し実行されます。

GQ の割り当て (UNC_GQ_ALLOC) および GQ のトラッカー占有 (UNC_GQ_TRACKER_OCCUP) を監視するアンコアイベントは、https://perfmon-events.intel.com/ (英語) にあるイベントコード 03H および 02H に示されています。この 3 つのトラッカーは、Umask 値を指定して選択します。これらの派生イベントのニーモニックには、読み出しトラッカーを表す「RT」、書き込みトラッカーを表す「WT」、ピア・プローブ・トラッカーを表す「PPT」という表記が使用されます。

データが供給された直後に占有が停止する場合、キュー占有の平均継続期間によってレイテンシーを測定できます。キュー占有の平均継続期間は、UNC_GQ_TRACKER_OCCUP.X/UNC_GQ_ALLOC.X の比率によってを測定できます。ここで、「X」は特定の Umask 値を表します。読み出しトラッカーの総占有期間は、次の式によって測定されます。

Total Read Period (総読み出し期間) = UNC_GQ_TRACKER_OCCUP.RT/UNC_GQ_ALLOC.RT

この期間にはブックキーピングとクリアの時間が含まれるため、データ供給のレイテンシーよりも長くなります。次の測定値について考えます。

LLC response Latency (LLC 応答レイテンシー) =
UNC_GQ_TRACKER_OCCUP.RT_TO_LLC_RESP / UNC_GQ_ALLOC.RT_TO_LLC_RESP

これは基本的に定数です。スヌープの合計時間を含んでおらず、変更されたラインを別のコアから取得します (例えば、L3 のスキャン時間のみと、変更されたラインがこのソケット内に存在するかどうかを確認します)。

L3 ヒットの合計レイテンシーは、次の 3 つの加重平均となります。

- 要求を行ったコアによってのみラインが使用された単純なヒットのレイテンシー
- 複数のコアによるクリーンなラインへのアクセスに関するレイテンシー
- 複数のコアによってアクセスされたダーティーなラインへのアクセスに関するレイテンシー

ロードに関するこの 3 つの L3 ヒットの構成要素は、OFFCORE_RESPONSE の派生イベントを使用して分類できます。

- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_NO_OTHER_CORE
- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_OTHER_CORE_HIT
- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_OTHER_CORE_HITM

OFFCORE_RESPONSE_0.DEMAND_DATA.LOCAL_CACHE イベントは、レイテンシーを求めるための分母として使用します。個々のレイテンシーはマイクロベンチマークによって測定する必要がありますが、ロード帯域幅の影響が含まれるため、プリサイス・レイテンシー・イベントを使用する方がはるかに効果的です。

L3 ミス構成要素は次の 3 つの加重平均です。

- 別のソケット上のキャッシュにおける L3 ヒットのレイテンシー (前のセクションで説明)
- ローカル DRAM に対するレイテンシー
- リモート DRAM に対するレイテンシー

ローカル DRAM アクセスとリモート・ソケット・アクセスは、さらに多くのアンコアイベントによって分類できます。これについては後述します。

Miss to fill latency (ミスからフィルまでのレイテンシー) = UNC_GQ_TRACKER_OCCUP.RT_LLC_MISS / UNC_GQ_ALLOC.RT_LLC_MISS

RTID ニーモニックに関連する Umask 値を使用するアンコア GQ イベントにより、GQ と QHL 間の通信に関連するミスからフィルまでのレイテンシーのサブコンポーネントの監視が可能になります。

上記の 3 つのトラッカーが空でない (エントリーが 1 つ以上ある) または一杯でない場合にサイクルを監視するアンコア PMU イベントがあります。これらのイベントは、https://perfmon-events.intel.com/ (英語) にあるイベントコード 00H および 01H に示されています。

一般に、アンコア PMU は特定のイベント条件を発生させるプロセッサー・コアを区別しないため、レイテンシーをキューの平均占有率で割ることによってペナルティーを判定する手法はアンコアには適用できません。異なるコアのエントリーがオーバーラップしても、ペナルティーはオーバーラップしないためストールサイクルは減少せず、各コアは、単独で全レイテンシーの影響を受けます。

コア単位の変更を評価するには、目的のコアからのエントリーのサイクル数を取得する必要があります。これには、*NOT_EMPTY_CORE_N タイプのイベントが必要となりますが、そのようなイベントは存在しません。したがって、サイクルを分類する際に、全レイテンシーを適用してペナルティーを概算する必要があります。前述のように、個々のサンプルのレイテンシーとともにデータソースも収集されるため、最善の方法は PEBS レイテンシー・イベントを使用することです。

前述の読み出しトラッカーの個々の構成要素もまた、cmask 値を 1 または 32 に設定し、それを関連する RT 占有イベントに適用することにより、ビジーまたはフルとして監視できます。

表 B-15 占有サイクルのアンコア PMU イベント

アンコア PMU イベント	Cmask	Umask	イベントコード
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_FULL	32	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_FULL	32	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_FULL	32	08H	02H
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_BUSY	1	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_BUSY	1	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_BUSY	1	08H	02H

B.4.5.2 グローバル・キュー・ポート・イベント

GQ データバッファーのトラフィックは、個別のポートを経由して異なるサブシステムに出入りするフローを制御します。

- コア・トラフィック: 2 つのポートでデータ・トラフィックを処理します。各ポートは、1 組のプロセッサー・コア専用に使用されます。
- L3 トラフィック: 1 つのポートで L3 データ・トラフィックを処理します。
- インテル® QPI トラフィック: 1 つのポートでインテル® QPI ロジックへのトラフィックを処理します。
- IMC トラフィック: 1 つのポートで統合型メモリー・コントローラーへのデータ・トラフィックを処理します。

L3 トラフィックおよびコア・トラフィック用ポートは、サイクルごとに一定のビットを転送します。ただし、インテル® QuickPath インターコネクト・プロトコルでは、インテル® QPI および IMC 読み出しポート上で 8/16 バイトのいずれかでデータが転送されます。したがって、このイベントはデータ転送および総帯域幅を測定するために使用できません。

トラフィックのフローを区別できるアンコア PMU イベントは、https://perfmon-events.intel.com/ (英語) にあるイベントコード 04H および 05H に示されています。

B.4.5.3 グローバル・キュー・スヌープ・イベント

コアから、あるいはリモートパッケージまたは I/O ハブからのキャッシュライン要求は、GQ によって処理されます。いずれかのコアからのキャッシュライン要求をアンコアが受信すると、GQ はまず L3 をチェックして、ラインがパッケージに存在するかどうかを確認します。L3 はインクルーシブであるため、このチェックは迅速に行われます。ラインが L3 に存在し、要求元のコアによって所有されていれば、データは L3 からコアに直接返されます。

ラインが複数のコアによって所有されていれば、GQ はほかのコアをスヌープして、変更されたコピーが存在するかどうかを確認します。存在する場合は、L3 が更新され、ラインが要求元のコアに送信されます。

L3 ミスが発生している場合、GQ はローカル・メモリー・コントローラーに (またはインテル® QPI リンク経由で) ラインの要求を送信する必要があります。データがリモート L3 に存在する、またはローカル DRAM に存在しない場合、要求は、インテル® QPI 経由でリモート L3 (またはリモート DRAM) に送信されます。各物理パッケージはローカルの統合メモリー・コントローラーを持つため、GQ は、要求されたキャッシュラインの「ホーム」ロケーションを物理アドレスから特定する必要があります。ホームがローカルパッケージ上にあることがアドレスから特定されると、GQ は、同時にローカル・メモリー・コントローラーに対して要求を行います。ホームがリモートパッケージに属していることが特定されると、インテル® QPI 経由で送信された要求はリモート IMC にアクセスします。

また GQ は、インテル® QuickPath インターコネクトからのキャッシュライン要求に対するスヌープ応答を処理します。このスヌープ・トラフィックは、ピア・プローブ・トラッカー内のキューエントリーに対応しています。

スヌープ応答は、ローカルホームとリモートホームのデータの要求に分けられます。ラインが変更された状態であり、GQ が読み出し要求に応答している場合、ラインはメモリーにライトバックされる必要があります。これは、ラインが再び変更されるため RFO への応答は無駄な作業であると考えられます。そのため、RFO に対してライトバックは行われません。

アンコア PMU によって監視可能なローカル・ホーム・イベントのスヌープ応答は、https://perfmon-events.intel.com/ (英語) にあるイベントコード 06H に示されています。リモート・ホーム・イベントのスヌープ応答は、イベントコード 07H に示されます。

いくつかの関連イベントは、ほかのキャッシュ・エージェント (プロセッサーまたは IOH) からのスヌープに応答して、MESI 状態の遷移をカウントします。これらの一部は MSR のプログラミングに依存しており、MSR は 1 つしかないため、一度に 1 つずつしか測定できません。インテル® パフォーマンス・ツールは、これらのイベントを 1 つの汎用アンコア・カウンターに制限することで正しくスケジュールします。

B.4.5.4 L3 イベント

L3 ヒットおよび L3 ミスの数は GQ トラッカー割り当てイベントから特定できますが、アンコア PMU イベントの方が簡単に使用できるものがあります。これらは、https://perfmon-events.intel.com/ (英語) にあるアンコア・イベント・リストのイベントコード 08H および 09H に示されています。

割り当てられたラインと干渉されたラインの MESI ステートの分類は、イベントコード 0AH および 0BH を使用して、アンコアイベント LINES_IN、LINES_OUT で監視することもできます。詳細は、https://perfmon-events.intel.com/ (英語) に示されます。

B.4.6 インテル® QuickPath インターコネクトのホームロジック (QHL)

データが L3 をミスし、アンコアの GQ によってトランザクション要求が送信されると、インテル® QPI ファブリックは、ローカル DRAM コントローラーから、または別の物理パッケージのリモート DRAM コントローラーからの要求を処理します。GQ は、要求されたキャッシュラインの「ホーム」ロケーションを物理アドレスから特定します。ホームがローカルパッケージ上にあることが特定されると、GQ は、ローカル・メモリー・コントローラー (統合メモリー・コントローラー (IMC)) に対して要求を送ります。ホームがリモートパッケージに属していることが特定されると、インテル® QPI 経由で送信された要求はリモート IMC にアクセスします。

インテル® QPI ロジックと IMC は、アンコア・サブシステムにある異なるユニットです。インテル® QPI ロジックは、「キャッシュ・エージェント」および「ホーム・エージェント」の概念から、ローカル IMC とリモート IMC を識別します。具体的には、インテル® QPI プロトコルにより、各ソケットが「キャッシュ・エージェント」と「ホーム・エージェント」のどちらを持っているかが考慮されます。

- キャッシュ・エージェントはアンコアの GQ および L3 である (存在する場合は、IOH)。
- ホーム・エージェントは IMC である。

L3 ミスにより、すべてのキャッシュ・エージェントおよびホーム・エージェントのラインが (場所に関係なく) 同時に照会されます。

要求されたラインを別のソースがより迅速に供給できる場合、QHL 要求に優先することがあります。パッケージ上の要求によるローカル・ホーム・ラインに対する L3 ミスは、同時に QHL とインテル® QPI にも送られます。リモートのキャッシュ・エージェントがラインを最初に供給した場合、その QHL への要求に対して、トランザクションが完了したことを示すシグナルが送信されます。リモートのキャッシュ・エージェントが読み出し要求に応答して、変更されたラインを返す場合、更新されたラインをライトバックして DRAM のデータを更新する必要があります。

インテル® QPI が、ローカルのホームラインに対するスヌープ要求を GQ と QHL の両方に送信する場合にも、同様の制御シグナルフローがあります。そのラインを L3 が保持する場合、L3/GQ によってトランザクションが完了したことを QHL に知らせる必要があります。L3 (またはコア) のラインが変更され、リモートパッケージからのスヌープ要求がロードに関連する場合、QHL は、ライトバックを完了し、そのラインをインテル® QPI に転送してトランザクションを完了する必要があります。

アンコア PMU は、QHL のオペコード一致機能を使用して、アンコアのキャッシュライン・アクセスとライトバックのトラフィックを監視するイベントを提供します。インテルオペコード一致機能を使用するアンコア PMU イベントは、イベントコード 35H に示されています。表 B-16 に、QHL オペコード一致機能をプログラムに有用な設定を示します。

表 B-16 一般的な QHL オペコード一致機能のプログラミング

ロード・レイテンシーのプリサイスイベント	MSR 0x396	Umask	イベントコード
UNC_ADDR_OPCODE_MATCH.IOH.NONE	0	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDS	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPIWB	40001D00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.NONE	0	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDI	40001900_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDS	40001A00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPIWB	40001D00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.NONE	0	4H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDS	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPIWB	40001D00_00000000	1H	35H

上記の定義済みのオペコード一致機能のエンコーディングは、HITM アクセスを監視するために使用できます。これは、HITM 転送を要求するコードのプロファイルを可能にする唯一のイベントです。

図 B-8 ～図 B-15 に、キャッシュラインのローカルホームおよびリモートキャッシュの MESI ステートのさまざまな組み合わせにおける、L3 ミス発生後のデータ読み出しおよび所有権読み出し (RFO) に関連する一連のインテル® QPI プロトコル交換を示します。データがリモート L3 に存在する場合でもリモート QHL から送信されるケースに特に注意してください。それらは、M ステートのラインを持つリモート L3 による読み出しデータの場合です。

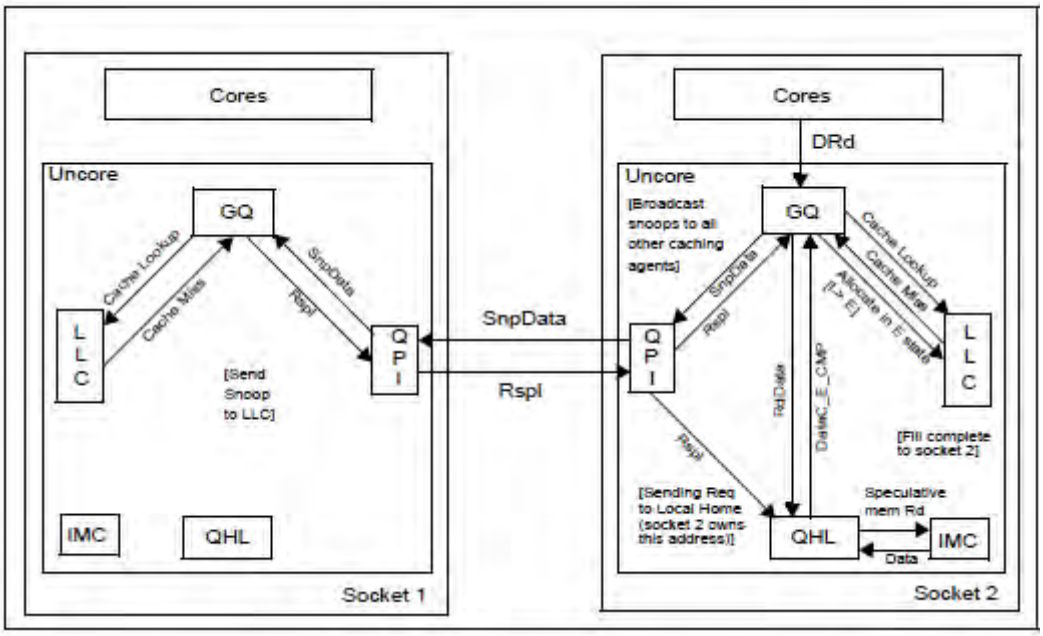


図 B-8 ローカルホームに対する LLC ミス発生後の RdData 要求 (CLEAN 応答)

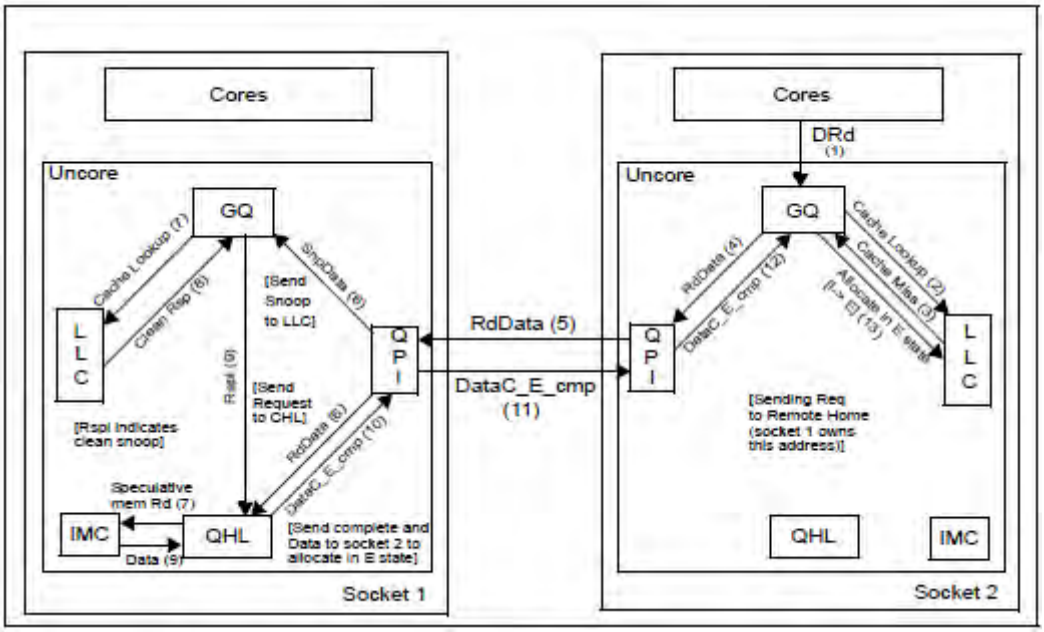


図 B-9 リモートホームに対する LLC ミス発生後の RdData 要求 (CLEAN 応答)

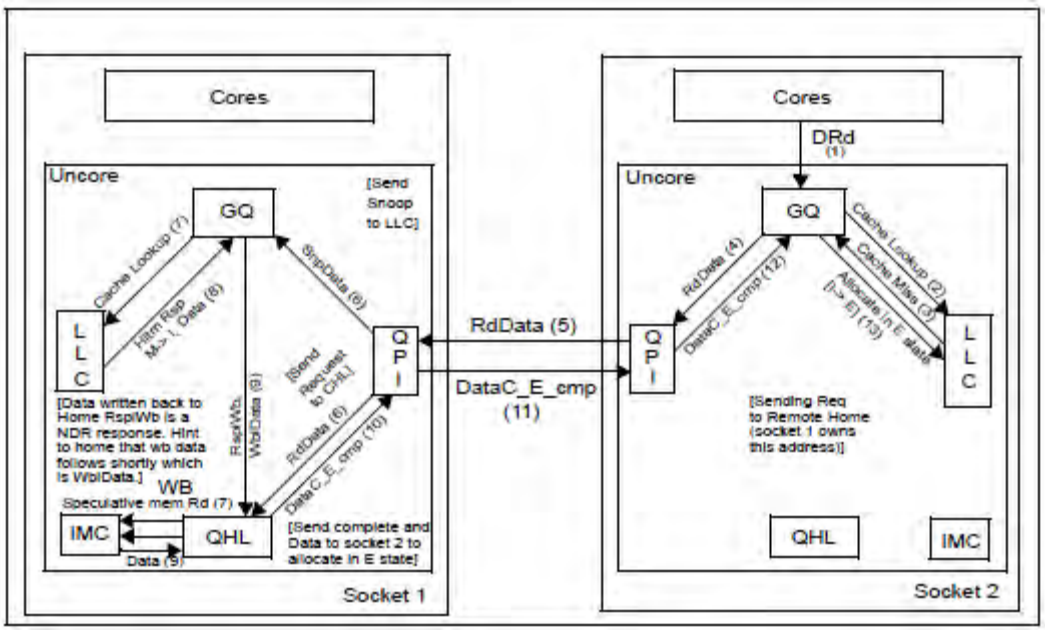


図 B-10 リモートホームに対する LLC ミス発生後の RdData 要求 (HITM 応答)

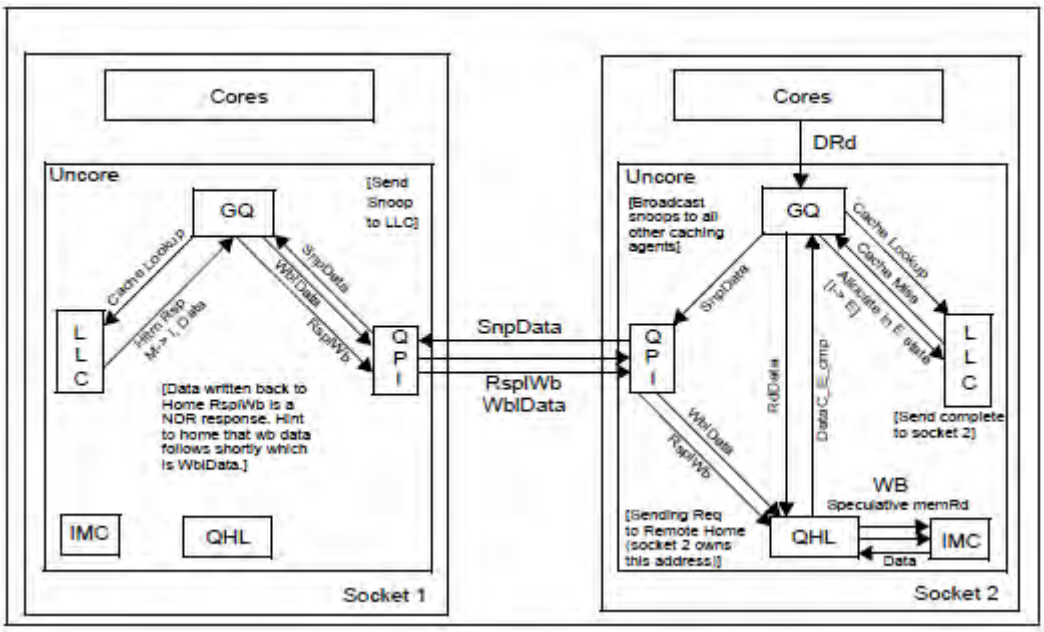


図 B-11 ローカルホームに対する LLC ミス発生後の RdData 要求 (HITM 応答)

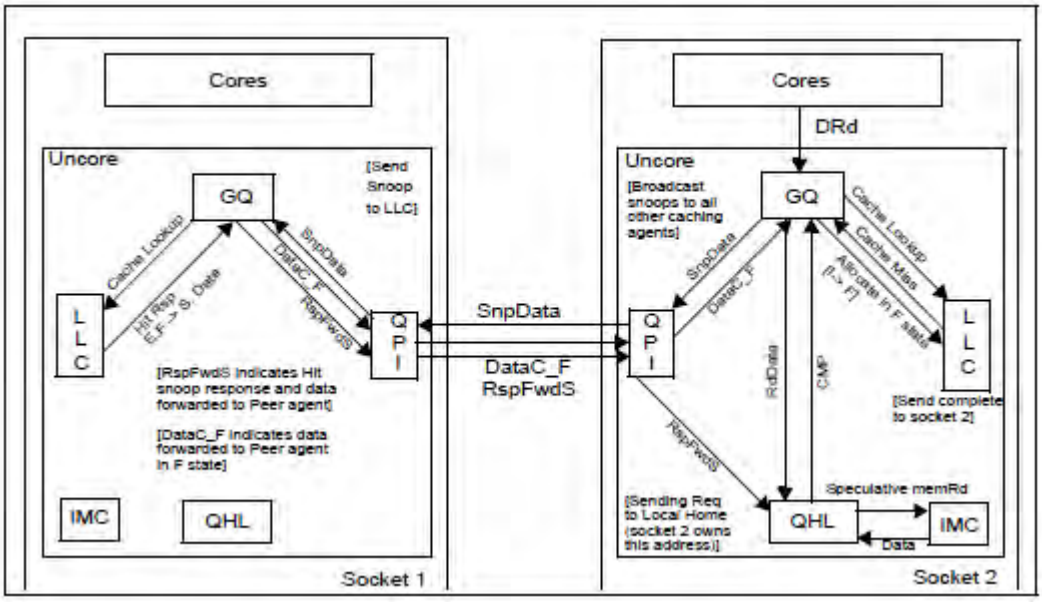


図 B-12 ローカルホームに対する LLC ミス発生後の RdData 要求 (HIT 応答)

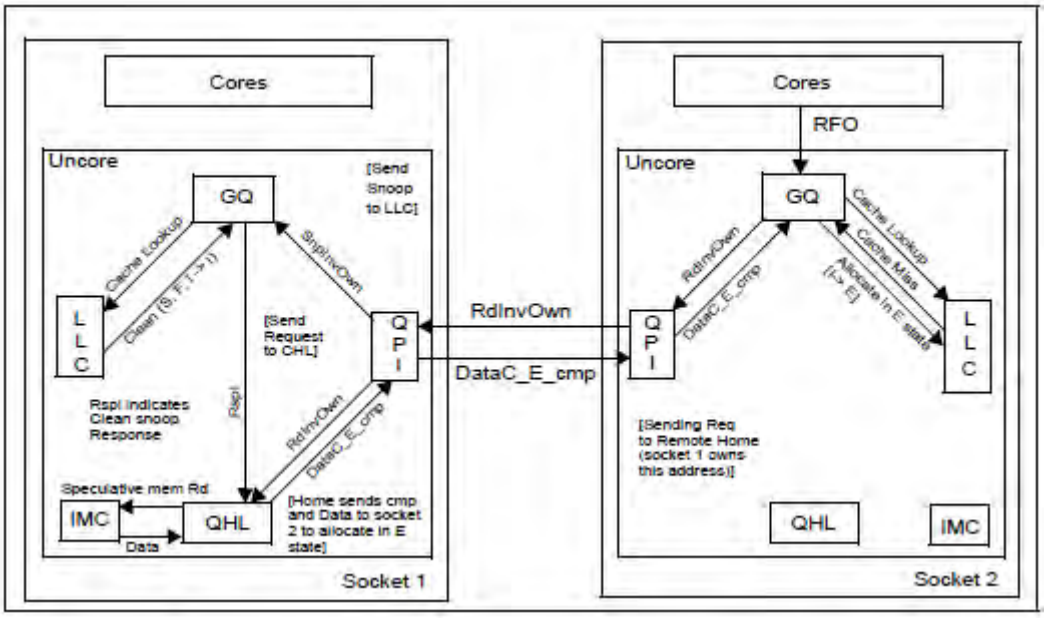


図 B-13 リモートホームに対する LLC ミス発生後の RdInvOwn 要求 (CLEAN 応答)

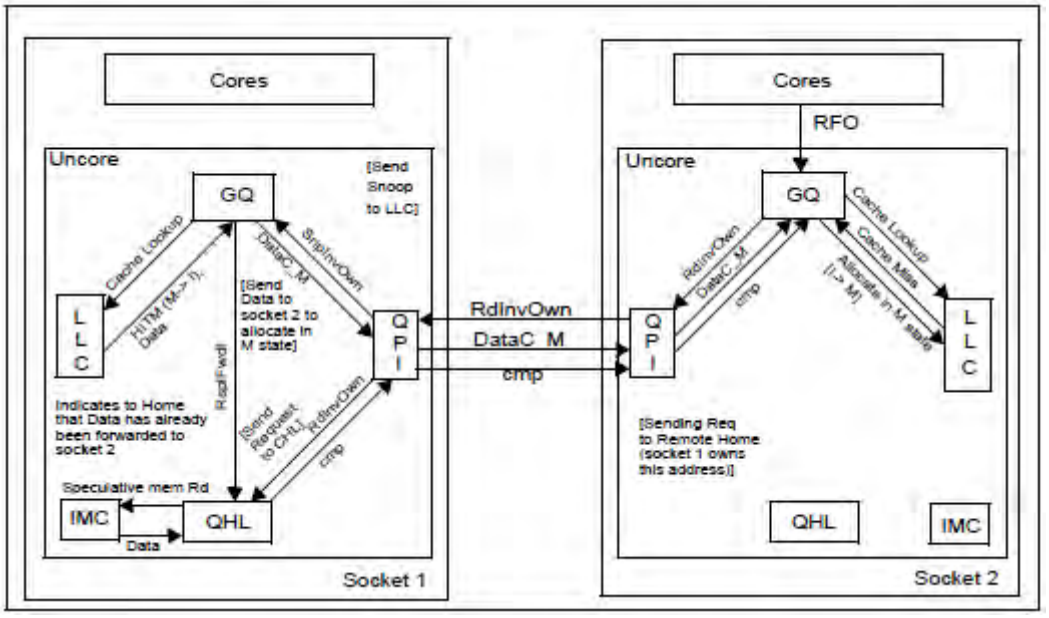


図 B-14 リモートホームに対する LLC ミス発生後の RdInvOwn要求 (HITM 応答)

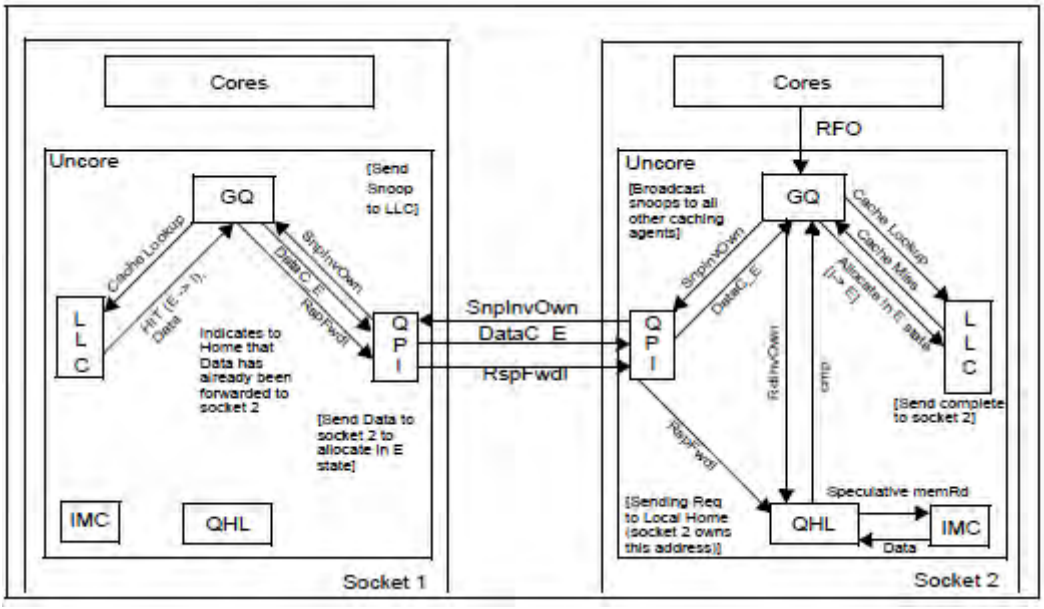


図 B-15 ローカルホームに対する LLC ミス発生後の RdInvOwn要求 (HIT 応答)

ラインは、ローカル/リモートのいずれが「ホームであった」かにかかわりなく、発生元の GQ がラインを受信する前に DRAM にライトバックされる必要があります。そのため、ラインは常に QHL から送信されたように見えます。RFO では、この処理を行いません。しかし、リモートの RFO (SnpInvOwn) に応答しており、ラインが S または F ステートである場合、キャッシュラインが無効化され、ラインは QHL から送信されます。問題は、データソースがそれほど明確でないことがあるという点です。

B.4.7 アンコアからの帯域幅測定

読み出しの帯域幅は、OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM や OFFCORE_RESPONSE_0.DATA_IN.REMOTE_DRAM などのイベントを使用して、コアごとに測定できます。総帯域幅には書き込みも含まれますが、ほとんどの場合、これらは L3 の変更されたラインの排出によって発生するため、コアから監視できません。したがって、あるコアによって使用され、変更されたラインは、無関係のタスクを実行している別のコアの書き込みによる排出のため DRAM にライトバックされる可能性があります。アンコアでは、変更されたキャッシュラインおよび (例えば、非テンポラルなストリーミング・ストアによって書き込まれた) キャッシュ不可ラインのライトバックは異なる方法で処理され、ライトバックにより各種イベントが多様な方法でカウントアップされます。

DRAM に書き込まれるラインはすべて、UNC_IMC_WRITES.FULL.* イベントによってカウントされます。これには、変更されたキャッシュラインのライトバックと (例えば、非テンポラルなインテル® SSE ストア命令によって生成された) キャッシュ不可ラインの書き込みが含まれます。リモートソケットからのキャッシュ不可ラインの書き込みは、UNC_QHL_REQUESTS.REMOTE_WRITES によってカウントされます。ローカルコアからのキャッシュ不可ラインのライトバックは、UNC_QHL_REQUESTS.LOCAL_WRITES によってカウントされません。このイベントは、ローカルにキャッシュされたラインのライトバックだけをカウントします。

UNC_IMC_NORMAL_READS.* イベントは読み込みのみをカウントします。UNC_QHL_REQUESTS.LOCAL_READS と UNC_QHL_REQUESTS.REMOTE_READS は、読み出しと「InvtoE」トランザクションをカウントします。これらは、キャッシュ不可の書き込み (USWC/UC の書き込みなど) に対して発行されます。これにより、UNC_QHL_REQUESTS.LOCAL_READS + UNC_QHL_REQUESTS.REMOTE_READS - UNC_IMC_NORMAL_READS.ANY の差を計算することによってキャッシュ不可の書き込みを算出できます。

帯域幅の評価に有用なアンコア PMU イベントは、https://perfmon-events.intel.com/ (英語) にあるイベントコード 20H、2CH、2FH に示されています。

B.5 Sandy Bridge† マイクロアーキテクチャーのパフォーマンス・チューニング手法

この節では、パフォーマンス監視イベントを使用したさまざまなパフォーマンス・チューニング手法について説明します。一般的に、一部の手法はほかのマイクロアーキテクチャーにも応用できますが、ほとんどのパフォーマンス・イベントは Sandy Bridge† マイクロアーキテクチャーに固有なものです。

B.5.1 パフォーマンスのボトルネックとソース行の関連付け

パフォーマンス分析ツールは、イベントをサンプリングして命令ポインターアドレス (IP) からホットスポットを特定することにより、パフォーマンスのボトルネックが潜在するコード領域を特定可能にします。

このサンプリング手法では、パフォーマンス・カウンターのオーバーフローにより発生するパフォーマンス監視割り込み (PMI) に応答するサービスルーチンが必要になります。イベント発生条件のパフォーマンス監視イベントの検出と実際の命令ポインターアドレスの間にはある程度のずれが生じることがあります。これは、「スキッド」と呼ばれます。言い換えれば、イベントスキッドは、問題を発生した命令とイベントがタグ付けされている命令間の距離のことです。一般的にスキッドにはいくつかの注意すべき点があります。

- プリサイスイベントでは、リタイアした次の IP に対して、定義された 1 命令分のイベントスキッドが生じます。発生した命令が分岐である場合、このイベントは分岐ターゲットでタグ付けされており、分岐命令から分離できます。そのため、プリサイスイベントを使用したサンプリングでは、ボトルネックのコード領域を正確に特定する際にノイズの影響が減少する可能性が高くなります。
- 一般に、パフォーマンス・イベントを使用する場合、イベント生成条件がパフォーマンスに与える影響が大きいほどスキッドは短くなりますが、その逆も当てはまります。次の例は、この規則を説明しています。
  - ストア・フォワーディング・ブロックの問題は、10 サイクルを超える遅延を引き起こす可能性があります。ストア・フォワーディング・ブロック・イベントをサンプリングすると、ほとんどの場合、ブロックされたロードの後に続く数個の命令にタグ付けされます。
  - 反対に、正常に転送されたロードをサンプリングすると、スキッドが長くなるため、パフォーマンス・チューニングにはあまり役立ちません。

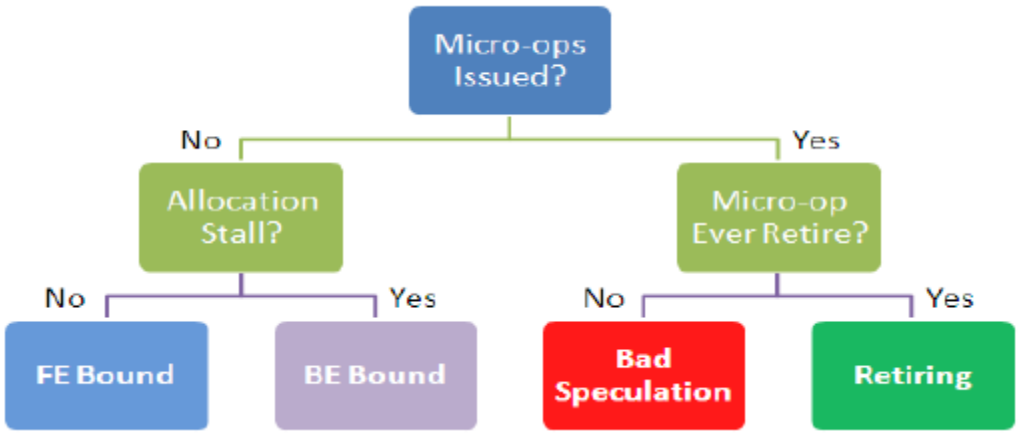
- スキッドは、イベント発生条件が命令のリタイアメントに近いほど短くなります。パイプラインのフロントエンドで発生したイベントは、実行時またはリタイア時に処理されるイベントよりも、原因となる命令から離れた命令にタグ付けされる傾向があります。
- イベント CPU_CLK_UNHALTED.THREAD によってカウントされるサイクルは、パイプライン内のより大きなボトルネックの後の命令に対して、より大きなカウントをタグ付けすることがあります。1 つの命令に対してサイクル数が累積されていれば、おそらく直前の命令でボトルネックが発生しています。
- フロントエンドで発生している低コストの問題の原因を判定することは非常に困難です。フロントエンドのイベントは、問題を発生している実際の命令の直前の IP までスキッドする可能性があります。

B.5.2 階層的なトップダウン・パフォーマンス特性方式とパフォーマンス・ボトルネックの特定

Sandy Bridge† マイクロアーキテクチャーは、マイクロアーキテクチャー・パイプラインのどの部分がストールしているか絞り込むのを支援するいくつかのパフォーマンス・イベントを導入しています。これは、階層的なアプローチにより CPU サイクルがマイクロアーキテクチャー・パイプラインで時間を費やすのワークロードを特性化するのを可能にします。上位レベルに、CPU サイクルを分類する次のような 4 つの領域があります。パイプラインのどの部分がストールしているかを判定するには、フロントエンドによって供給されたマイクロオペレーション (uop) をキューに格納し、アウトオブオーダー・バックエンド (E.2.1 節を参照) へ供給するバッファーを調べる必要があります。このバッファーは、マイクロオペレーション (uop) キューと呼ばれます。マイクロオペレーション(uop)キューに関連する、次の 4 種類のストール条件があります。

- フロントエンドのストール: パイプラインのバックエンドがマイクロオペレーション (uop) を要求しているときに、フロントエンドが 1 サイクルあたり 4 個未満のマイクロオペレーション (uop) を供給しています。このようなストールが発生すると、アウトオブオーダー (OOO) エンジンのリネーム/割り当て処理が待機状態となります。そのため、実行はフロントエンドに制約がある (制限される) と言われます。
- バックエンドのストール: パイプラインのバックエンドに追加のマイクロオペレーション (uop) を受け入れるリソースが不足しているため、マイクロオペレーション (uop) キューからマイクロオペレーション (uop) が供給されていません。このようなストールが発生する場合、実行はバックエンドに制約がある (制限される) といわれます。
- バッド・スペキュレーション: 正常にリタイアしない命令のスペキュレーティブ・エグゼキューション (投機的な実行) をパイプラインが行っています。最もよくあるケースは、パイプラインが分岐ターゲットを予測する場合の分岐予測ミスです。これは、分岐の実行を待機する代わりに、パイプラインを一杯の状態で維持するために行われます。プロセッサーの予測が誤っている場合、推測された命令をリタイアせずに、パイプラインをフラッシュする必要があります。
- リタイア: 最終的にリタイアするマイクロオペレーション (uop) をマイクロオペレーション (uop) キューが供給します。一般に、マイクロオペレーション (uop) はプログラムコードから生成されます。ただし、マイクロコード・シーケンサーがマイクロオペレーション (uop) を供給してパイプライン内の問題を処理するアシストは例外です。

次の図は、命令の実行を論理的に分類する方法を示します。



次の数式を Sandy Bridge† マイクロアーキテクチャーのコア PMU パフォーマンス・イベントとともに使用すると、各カテゴリーに要する実行スロット割合を算出できます。

%FE_Bound =
100 * (IDQ_UOPS_NOT_DELIVERED.CORE / **N**) ;
%Bad_Speculation =
100 * ((UOPS_ISSUED.ANY – UOPS_RETIRED.RETIRE_SLOTS + 4 *
INT_MISC.RECOVERY_CYCLES) / **N**) ;
%Retiring = 100 * (UOPS_RETIRED.RETIRE_SLOTS/ **N**) ;
%BE_Bound = 100 * (1 – (**FE_Bound + Retiring + Bad_Speculation**)) ;

N は、実行スロットの総数を表し、サイクル数に 4 を掛けた値となります。

- **N** = 4*CPU_CLK_UNHALTED.THREAD

次の節では、バックエンドのストール、フロントのストール、バッド・スペキュレーションという 3 つのカテゴリーのペナルティー・サイクルの原因について説明します。各節では、プロセス、モジュール、関数、命令単位に適用される数式を使用します。

B.5.2.1 バックエンド依存の特性

%BE_Bound メトリックが報告された場合、ユーザーはバックエンドで考えられる次のレベルの問題を掘り下げて調査する必要があります。ここで紹介する方法論では、サイクルごとの実行ユニットの占有率に基づいてバックエンドのストールを調査します。当然のこととして、すべての実行リソースがビジーに保たれると最適なパフォーマンスを達成することができます。現在、この方法論では**バックエンド依存**を**メモリー依存**と**コア依存**の 2 つのカテゴリーに分類しています。

“メモリー依存” は、メモリー・サブシステムに関連するストールに相当します。例えば、キャッシュミスは最終的に実行のスタベーションを引き起こす可能性があります。また、”コア依存” は実行もしくは OOO クラスターのストールに関連するため、若干扱いにくいことがあります。これらのストールは、実行のスタベーションや適切ではない実行ポートの利用が原因であることが明らかです。例えば、長いレイテンシーの除算操作は、実行ポートに対して特定の uop タイプがプレッシャーを与え、サイクルで利用されるポートが減少することからしばらくの間命令スタベーションを引き起こし、実行をシリアル化する可能性があります。

これを算出するには、実行ユニットでパフォーマンス監視イベントを使用します:

%BE_Bound_at_EXE =
(CYCLE_ACTIVITY.CYCLES_NO_EXECUTE + UOPS_EXECUTED.THREAD:c1 -
UOPS_EXECUTED.THREAD:c2) / CLOCKS

CYCLE_ACTIVITY.CYCLES_NO_EXECUTE イベントは、uop がまったく実行されなかったスタベーション・サイクルの完了をカウントします。

UOPS_EXECUTED.THREAD:c1 と UOPS_EXECUTED.THREAD:c2 は、1 サイクルで少なくとも 1 または 2 つの uop が実行されたサイクルをカウントします。したがって、イベントカウントの差から OOO バックエンドが 1 uop しか実行できなかったサイクルを測定できます。

%BE_Bound_at_EXE メトリックは、実行ユニットのパイプライン・ステージでカウントされるため、アロケーション・ステージでカウントされる Backend_Bound 比率とは一致しません。しかし、両方のカウンターは実行がバックエンド依存 (両方が高い場合) であることを確認するのに使用されるため、ここでは冗長性は許容されます。

B.5.2.2 コア依存特性

“バックエンド依存” のワークロードは、次のメトリックにより “コア依存” として分類できます。

%Core_Bound = %Backend_Bound_at_EXE - %Memory_Bound

“%Memory_Bound” メトリックについては B.5.2.3 節で説明しています。ワークロードがいったん “コア依存” として識別されると、実行ポートのプレッシャーや FP チェーンによる長いレイテンシーの算術演算など、ターゲットのパフォーマンス・カウンターを介して OOO や実行に関連する問題をドリルダウンすることができます。

B.5.2.3 メモリー依存特性

メモリー・パイプラインのパフォーマンス問題を特性化する手法は、単純な計算を使用してメモリーストールのペナルティーを推測する傾向があります。通常、特定のキャッシュ・レベル・アクセスへのミス数は、CPU 仕様ごとにキャッシュレベル向けに事前定義されたレイテンシーを掛けることで、ペナルティーを推測します。これはインオーダー・プロセッサーでは問題ないかもしれませんが、高度なアウトオブオーダー・プロセッサーでは、メモリーアクセスがオーバーラップし、スケジューラーがレイテンシーを隠匿する傾向があるため、CPU サイクルにおけるメモリーアクセスが過度に評価されることがあります。スケジューラーは、メモリー・アクセス・データを必要としない uop によって実行ストールをビジーに保つことで、メモリー・アクセス・ストールを隠匿することできることがあります。したがって、メモリーアクセスのペナルティーは、スケジューラーがディスパッチの準備ができていない場合に実行ユニットがスタベー

ションに陥ることによって生じます。uop がメモリーアクセスによるデータを待機しているか、ディスパッチされていない uop と依存関係がある可能性があります。

Ivy Bridge† マイクロアーキテクチャーでは、メモリーアクセスの調査に使用できる新しいパフォーマンス・イベント "CYCLE_ACTIVITY.STALLS_LDM_PENDING" が提供されています。

"**memory bound (メモリー依存)**" メトリックを定義するため、そのイベントを使用します。このイベントは、実行スタベーションを引き起こす完了していない続行中のメモリーロード要求のサイクルを測定します。uop がストアや HW プリフェッチの完了を (直接) 待機しない場合、ロード要求操作のみをカウントすることに注意してください。

%Memory_Bound = CYCLE_ACTIVITY.STALLS_LDM_PENDING / CLOCKS

ワークロードがメモリー依存である場合、キャッシュ階層と DRAM システムメモリーに関連するパフォーマンス特性をさらに詳しく特性化することができます。

L1 キャッシュのレイテンシーは、一般に、すべてのストールの中で最も短い ALU ユニットのストールに匹敵する短いレイテンシーです。しかし、特定の状況では、先行するストアによってブロックされるロードのように、L1 から提供されるまでロードは高いレイテンシーを被る可能性があります。L1 ヒットにはフィルバッファーが割り当てられていません。代わりに、完了しなかったロードをカウントする LDM ストール・サブイベントを使用できます。

%L1 Bound =
(CYCLE_ACTIVITY.STALLS_LDM_PENDING -CYCLE_ACTIVITY.STALLS_L1D_PENDING)/ CLOCKS

前述したように、L2 Bound (L2 依存) は次のように検出できます。

%L2 Bound =
(CYCLE_ACTIVITY.STALLS_L1D_PENDING - CYCLE_ACTIVITY.STALLS_L2_PENDING)/ CLOCKS

同様に、L3 ミスを減算することで L3 Bound (L3 依存) を計算できます。しかし、L3_PENDING を測定する等価なイベントがありません。そのため、補正係数に関連する L3_HIT と L3_MISS ロード・カウント・イベントを使用することで概算を推測できます。長いレイテンシーが L3 とメモリーにある場合、この推測は許容されます。補正係数 MEM_L3_WEIGHT は、外部メモリーから L3 キャッシュ・レイテンシーへの比率です。第 3 世代インテル® Core™ プロセッサー・ファミリーには、係数 7 を使用します。この補正係数は、CPU とメモリー周波数に依存します。

%L3 Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING * L3_Hit_fraction / CLOCKS

ここで、L3_Hit_fraction は次のとおりです。

MEM_LOAD_UOPS_RETIRED.LLC_HIT / (MEM_LOAD_UOPS_RETIRED.LLC_HIT+ MEM_L3_WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS)

第 3 世代インテル® Core™ プロセッサーの DRAM トラフィックを算出するため、L2_PENDING の残数が MEM Bound に使用されます。

%MEM Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING * L3_Miss_fraction / CLOCKS

ここで、L3_Miss_fraction は次のとおりです。

WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS / (MEM_LOAD_UOPS_RETIRED.LLC_HIT+ WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS)

場合によっては、コア外部のすべてのメモリーストールを Uncore Bound (アンコア依存) と呼べるでしょう。

%Uncore Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING / CLOCKS

B.5.3 バックエンドのストール

バックエンドのストールの主な原因は、メモリー・サブシステムのストールと実行ストールの 2 つです。最初に、バックエンド・ストールの原因を理解するため、リソースのストール・イベントを使用します。

マイクロオペレーション (uop) をスケジューラーに送る前に、リネームステージでリソースを割り当てる必要があります。アプリケーションがパイプラインのバックエンドで重大なボトルネックに遭遇すると、パイプラインがバックアップされるときに、リソースが不足します。RESOURCE_STALLS イベントは、リソースを割り当てることができなかった場合にストールサイクルを追跡します。このイベントは、割り当てに利用できないリソースを追跡できるように、各リソースを個別のサブイベントに分類します。このイベントをカウントすることで、パイプラインのバックエンドにおける問題の原因特定に役立つ可能性があります。

以下に説明するリソースのストール比率は、CPU_CLK_UNHALTED.THREAD によってカウントされるサイクルのプロセス、モジュール、関数、さらに命令単位で計測でき、同じ単位でタグ付けされたペナルティーを表します。

固有のイベントの使用

RESOURCE_STALLS.ANY: リソースが不足しているため、リネームステージがマイクロオペレーション (uop) をスケジューラーに送出できないストールサイクルをカウントします。不足しているリソースは、このステージで割り当てる必要があります。ブロックしている命令のリタイアメントに近いため、イベントスキッドは短くなる傾向があります。このイベントは、他の RESOURCE_STALL サブイベントによってカウントされるすべてのストールを示すだけでなく、RESOURCE_STALLS2 のサブイベントも含みます。この比率が高い場合、サブイベントをカウントすることによって、ストールの原因をより適切に分類できます。

%RESOURCE.STALLS.COST = 100 * RESOURCE_STALLS.ANY / CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.SB: 通常、ストア・マイクロオペレーション (uop) は割り当て可能な状態にありますが、レイテンシーが長いストアが進行中であるため、すべてのストア・バッファー・エントリーが占有されている場合に発生します。通常、このイベントは、割り当て時にストールするストア命令の後の IP にタグ付けされます。

%RESOURCE.STALLS.SB.COST = 100 * RESOURCE_STALLS.SB / CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.LB: 通常、ロード・マイクロオペレーション (uop) は割り当て可能な状態ですが、レイテンシーが長いロードが進行中であるため、すべてのロード・バッファー・エントリーが占有されているサイクルをカウントします。多くの場合、ロードバッファーが一杯になる前に、レイテンシーが長いロードに依存するマイクロオペレーション (uop) によってスケジューラーのキューが一杯になります。

%RESOURCE.STALLS.LB.COST = 100 * RESOURCE_STALLS.LB / CPU_CLK_UNHALTED.THREAD;

上記のケースでは、イベント RESOURCE_STALLS.RS は並行してカウントされることがあります。データの局所性における損失をさらに調査するため、B.5.4.2 節で説明する上位キャッシュラインの置換を検討します。最初に L1 D キャッシュの置換に集中します。

RESOURCE_STALLS.RS スケジューラー・スロットは、通常、パイプラインのバックアップ時に最初に消費されるリソースです。しかし、レイテンシーが長いロードや実行ステージでバックアップされる命令など、バックエンドにおけるボトルネックが原因であることもあります。このため、スケジューラー・エントリーの不足にタグ付けされたストールを詳しく調査する前に、他のリソースのストールを調べることを推奨します。このイベントのスキッドは短くなる傾向があります。

%RESOURCE.STALLS.RS.COST = 100 * RESOURCE_STALLS.RS/ CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.ROB: すべてのリオーダーバッファー (ROB) のエントリーが処理済みであるため割り当てがストールしているサイクルをカウントします。このイベントは、RESOURCE_STALLS.RS ほど頻繁には発生しません。新しいマイクロオペレーション (uop) は順番にリタイアする必要があるため、通常、より新しいリタイアを保留しているマイクロオペレーション (uop) によってパイプラインがバックアップされていることを示します。

%RESOURCE.STALLS.ROB.COST = 100 * RESOURCE_STALLS.ROB/ CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS2.BOB_FULL: 分岐マイクロオペレーション (uop) は割り当て可能な状態ですが、プロセッサー内で進行中の分岐の数が上限に達したため割り当てがストールしたときにカウントされます。

%RESOURCE.STALLS.BOB.COST = 100 * RESOURCE_STALLS2.BOB/ CPU_CLK_UNHALTED.THREAD;

B.5.4 メモリー・サブシステム ストール

以下の項では、Sandy Bridge† マイクロアーキテクチャー固有のパフォーマンス監視イベントを使用したメモリー・サブシステムのストールを特定する方法について説明します。

B.5.4.1 ロード・レイテンシーのアカウンティング

ロード操作の局所性の分類は、プロセス、モジュール、関数、命令など、任意の粒度で実行できます。ロード操作がボトルネックであることが判明したら、プリサイスロードをブレークダウンしてさらに詳しく調査します。この方法でボトルネックを分類できないときは、ロードに影響する可能性があるその他の問題をチェックします。

次のイベントを使用すると、ボトルネックとなっているロードのコストを算出し、メモリー階層レベルの比率 (%) をブレークダウンできます。プリサイスイベントのサンプリングは、すべてのツールでサポートされるわけではありません。これらのイベントの正確 (プリサイス) なバージョン (イベント名はサフィックス PS で終わります) が利用するツールでサポートされない場合、非プリサイスなバージョンを使用できます。

プリサイス・ロード・イベントは、このイベントを次のリタイアした命令 (IP+1) にタグ付けします。階層レベルごとのロード・レイテンシーについては、表 2-18 を参照してください。

要求イベント

MEM_LOAD_UOPS_RETIRED.L1_HIT_PS: 第 1 レベルのデータキャッシュである L1D キャッシュにヒットしたロード要求をカウントします。ロード要求は、非スペキュレーティブ・ロード・マイクロオペレーション (uop) です。

MEM_LOAD_UOPS_RETIRED.L2_HIT_PS: 第 2 レベルのキャッシュである L2 にヒットしたロード要求をカウントします。

MEM_LOAD_UOPS_RETIRED.L3_HIT_PS: 第 3 レベルの共有キャッシュである LLC にヒットしたロード要求をカウントします。

MEM_LOAD_UOPS_LLC_HIT_RETIRE.XSNP_MISS: 第 3 レベルの共有キャッシュにヒットしたロード要求のうち、他のコアのキャッシュにも存在すると想定され、キャッシュラインがすでに排出されているものをカウントします。

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS: 他のコアのキャッシュ内のキャッシュラインにヒットしたロード要求のうち、キャッシュラインが更新されていないものをカウントします。

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS: 他のコアのキャッシュ内のキャッシュラインにヒットしたロード要求のうち、そのコアによってキャッシュラインに書き込まれたものをカウントします。このイベントは、ロックの競合やフォルス・シェアリングなど、マルチスレッド・アプリケーションで発生する可能性があるパフォーマンスのボトルネックを調査する際に重要です。

MEM_LOAD_MISC_RETIRED.LLC_MISS_PS: LLC にミスしたロード要求をカウントします。これは、ロードするデータがシステムのメモリーから供給されていることを意味します。

MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS: ライン・フィル・バッファー (LFB) にヒットしたロード要求をカウントします。LFB エントリーは、L1D キャッシュ内でミスが発生するたびに割り当てられます。ロードがこのロケーションにヒットした場合、先行するロード、ストア、またはハードウェア・プリフェッチがすでに L1D キャッシュ内でミスし、データフェッチが進行中であることを意味します。したがって、LFB におけるヒットのコストはさまざまです。このイベントは、L1D キャッシュ内ではミスしますが、LLC 内ではミスしないキャッシュライン分割ロードをカウントする可能性があります。

32 バイトのインテル® AVX ロードでは、L1D キャッシュ内でミスしたロードは、すべて L1D キャッシュ内のヒットまたは LFB 内のヒットとして示されます。その他のメモリー階層のレベルでは、ヒット数は示されません。L1D キャッシュ内でインテル® AVX ロードがミスすると、ほとんどのロードはライン・フィル・バッファー (LFB) からデータを取得します。

プリサイスロードのブレークダウン

ロードソースごとの比率 (%) の分類では、単一の IP、関数、モジュール、プロセスなど、任意の粒度でタグ付けできます。これは、単一の命令のロードがキャッシュ階層のどこで見つかったブレークダウンするのに役立ちます。次の数式は、LLC からロードしたデータが供給される時間の割合を計算する方法を示しています。すべての階層レベルについて、同様の数式を使用できます。

%LocL3.HIT = 100 * MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS / **$SumOf_PRECISE_LOADS**;

$SumOf_PRECISE_LOADS =
MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS +MEM_LOAD_UOPS_RETIRED.L1_HIT_PS +
MEM_LOAD_UOPS_RETIRED.L2_HIT_PS +MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS +
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS +
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS +
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS +
MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS;

ロードペナルティーの推測

以下の数式は、特定のメモリー階層からのロードがパフォーマンスの低下にどの程度影響しているか推測するのに役立ちます。プログラム可能な CPU_CLK_UNHALTED.THREAD イベントは、同じ粒度でタグ付けされたサイクルのペナルティーを表します。プリサイスイベントと同様、命令レベルでは、長いロードのサイクルコストは 1 IP だけスキッドする傾向があります。以下の計算は、イベントが正確 (プリサイス) であるため、任意の粒度のプロセス、モジュール、関数、または命令に適用できます。目的の粒度における総クロック数の 10% 以上であれば調査する必要があります。

依存性の高いロードがコードに存在する場合、MEM_LOAD_UOPS_RETIRED.L1_HIT_PS イベントを使用して、そのロードが L1D キャッシュの 5 サイクルのレイテンシーでヒットするかどうかを判断できます。

L2 レイテンシーのコスト算出

%L2.COST = 12 * MEM_LOAD_UOPS_RETIRED.L2_HIT_PS / CPU_CLK_UNHALTED.THREAD;

L3 ヒットのコスト算出

%L3.COST = 26 * MEM_LOAD_UOPS_RETIRED.L3_HIT_PS / CPU_CLK_UNHALTED.THREAD;

他のコアのキャッシュへのヒットのコスト算出

%HIT.COST = 43* MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS / CPU_CLK_UNHALTED.THREAD;

メモリー・レイテンシーのコスト算出

%MEMORY.COST =
200 * MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS / CPU_CLK_UNHALTED.THREAD;

実際のメモリー・レイテンシーは、メモリー・パラメーターによって大きく異なります。平行に発生したメモリー・トラフィックの量によっては、特定のメモリー階層のコストが削減されることがあります。通常、上記の概算値は、(ポインター追跡を行っているような) ワーストケースであると考えられます。

多くの場合、キャッシュミスは命令のリタイアメントまで遅延し、一括して現れます。プリサイスロードをブレークダウンすることで、ロードに供給されるデータの階層レベルの分布を概算できます。

特定のキャッシュレベルの影響が大きい場合、大量のキャッシュラインの置換がコードのどの部分で発生しているかを見つけます。これは、メモリー階層レベルのブレークダウンによって検出されたコードのホットな領域と一致することもありまするが、多くの場合はそうなりません。例えば、大規模なデータ構造の定期的な走査により、キャッシュのレベルが意図せずにクリアされる可能性があります。

変更されていないデータまたは変更されたデータが別のコアでヒットし、その概算コストが高く、かつコード領域が「ホット」である場合、スレッド間のロック、共有、またはフォルス・シェアリングが原因である可能性があります。

L1D キャッシュから離れたメモリー階層レベルのロード・レイテンシーが、ロードにかかったサイクルの数を正当化するものでない場合、次のいずれかを試みます。

- 汎用レジスターから (メモリーではなく) XMM レジスターへのスピルを招く不要なロード操作を排除します。
- B.5.4.4 節で説明するロード命令に影響を与える問題を引き続き調査します。

B.5.4.2 キャッシュラインの置換の分析

アプリケーションで多数のキャッシュミスが発生する場合、どのキャッシュラインが最も頻繁に置き換えられているかを調査するとよいでしょう。置換はハードウェア・プリフェッチとストア操作によって引き起こされますが、大量のキャッシュライン置換に関連している命令とアプリケーションが時間を費やしているコード領域が常に一致するとは限りません。一般に、大きな配列やデータ構造を横断すると、大量のキャッシュライン置換が発生する可能性があります。

要求イベント

L1D.REPLACEMENT: 1 次データキャッシュを置換します。

L2_LINES_IN.ALL: L2 キャッシュに格納されるキャッシュラインです。

イベントの使用

パフォーマンスの低下を引き起こす可能性がある置換は、プロセス、モジュール、関数レベルで特定できます。これは、次の 2 つのステップで行います。

- プリサイスロードのブレークダウンにより、ロードが実行され、最大のペナルティーが発生するメモリー階層レベルを特定します。
- 以下の数式を使用して、そのレベルで大部分の置換を発生させ、さらに下位のレベルにおいてこれらの大きなペナルティ・ロードに関連するコード領域を特定します。

例えば、ロードが LLC にヒットすることで高いペナルティーが存在する場合、L2 および L1 で置換を発生しているコードをチェックします。以下の数式において、分子はモジュールまたは関数でカウントされた置換数です。分母の置換の総和は、プロセス全体のすべてのキャッシュレベルでの置換の和です。これにより、大量の置換を発生させるコード領域を特定することができます。

L1D キャッシュ置換

%L1D.REPLACEMENT = L1D.REPLACEMENT / SumOverAllProcesses(L1D.REPLACEMENT);

L2 キャッシュ置換

%L2.REPLACEMENT = L2_LINES_IN.ALL / SumOverAllProcesses(L2_LINES_IN.ALL);

B.5.4.3 ロック競合の分析

マルチスレッド・アプリケーションのスケーラビリティー分析では、ロックの競合を正確に特定することが重要です。一般的な ring3 ロックは、ほとんどの場合アトミック命令を実行します。アトミック命令とは、メモリーアドレスを含む XCHG 命令、メモリー・デスティネーションとロック・プリフィクスを含む ADD、ADC、AND、BTC、BTR、BTS、CMPXCHG、CMPXCH8B、DEC、INC、NEG、NOT、OR、SBB、SUB、XOR、または XADD 命令を指します。プリサイスイベントを使用すれば、任意のロック競合について知ることができます。多くのロック API は、ring3 でアトミックな命令を開始し、ring0 にジャンプすることで競合しているロックを迂回します。つまり、競合が少ないシナリオでは、大量のロック API はコストが非常に大きくなる可能性があります。ロックされた命令の競合回数を算出するには、アトミック命令のメモリー・デスティネーションを含むキャッシュラインが別のコア内で変更された回数を測定します。

要求イベント

MEM_UOPS_RETIRED.LOCK_LOADS_PS: IP+1 のプリサイススキッドでリタイアしたアトミック命令の数をカウントします。

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS: 別のコアの変更されたキャッシュラインにヒットしたロードの発生回数をカウントします。このイベントは、ロック競合やフォルス・シェアリングなど、マルチコアシステムで発生する可能性があるパフォーマンスのボトルネックを調査するには重要です。

イベントの使用

ロック競合の係数は、別のコアと競合するために大きなペナルティーを持つ、ロックによって実行された操作の比率 (%) を示します。通常、ロック競合の係数が 5% を超えている場合、ホットロックを調査することが望ましく、著しいロック競合は、複数スレッドのパフォーマンスに影響する場合があります。

%LOCK.CONTENTION =
    100 * MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS / MEM_UOPS_RETIRED.LOCK_LOAD_PS;

B.5.4.4 そのほかのメモリーアクセスの問題

ストアフォワードのブロック

ストア・フォワーディングが可能でない場合、依存するロードはブロックされます。ストア・フォワーディング・ブロックの平均ペナルティーは 13 サイクルです。多くのストア・フォワーディングのブロックは以前のアーキテクチャーにおいて改善されているため、今日のコードにおける最も典型的な問題は、大きなロードではなく、小さなメモリー空間へのストアに関連しています。

要求イベント

LD_BLOCKS.STORE_FORWARD: アーキテクチャーが短いストアを長いロードにフォワードできない、またはまれに発生するアライメントの問題によってストア・フォワーディングがブロックされた回数をカウントします。

イベントの使用

ストア・フォワーディング・ブロックのコストを算出するには、次の数式を使用します。イベント LD_BLOCKS.STORE_FORWARD は、ロードの後に続く IP にタグ付けされる傾向があるため、この問題は命令レベルで調査することが推奨されます。ただし、プロセス、モジュール、関数、または IP 粒度で比率を調べることはできません。

```
%STORE.FORWARD.BLOCK.COST =
    100 *LD_BLOCKS.STORE_FORWARD * 13 / CPU_CLK_UNHALTED.THREAD;
```

ストア・フォワーディングによりブロックされているロードが検出されたら、ストアの場所を特定する必要があります。一般に、ストア・フォワーディングによりブロックされる問題の 65% は、ロードの直前に実行された 10 命令以内にあるストアによって発生します。

ストア・フォワーディング・ブロックの最も一般的な問題は、長いロードにフォワードできない短いストアです。例えば、生成された次のコードは、バイト・ポインター・アドレスに書き込んでから、4 バイト (dword) のメモリー領域から読み出しています。

```
and byte ptr [ebx],7f
and dword ptr [ebx], ecx
```

ストア・フォワーディング・ブロックを回避する最善の方法は、ロードではなくストア操作を修正することです。

キャッシュライン分割

Nehalem† マイクロアーキテクチャー以降、L1D キャッシュは分割レジスターを備えているため、2 つのキャッシュラインにまたがるロードおよびストアを処理できるようになりました。これにより、以前のマイクロアーキテクチャーでは 20 サイクルかかりましたが、分割レジスターを利用できる場合、分割ロードのコストは約 5 サイクルに軽減されます。通常、分割ストアは隠蔽されますが、多数の分割ストアがある場合、ストアバッファーが一杯になり割り当てがストールしたり、分割ロードの処理に必要な分割レジスターが消費される可能性があります。キャッシュラインの分割を排除することにより、明らかな利益が得られます。

要求イベント

MEM_UOPS_RETIRED.SPLIT_LOADS_PS: 2 つのキャッシュラインにまたがるロード要求の数をカウントします。このイベントは正確 (プリサイス) です。

MEM_UOPS_RETIRED.SPLIT_STORES_PS: 2 つのキャッシュラインにまたがるストアの数をカウントします。このイベントは正確 (プリサイス) です。

イベントの使用

分割ロードは、通常、大部分のコストが実行された次の IP にタグ付けされるため、非常に簡単に検出できます。次の比率は、分割後に任意の粒度 (プロセス、モジュール、関数、IP) で使用できます。

```
%SPLIT.LOAD.COST =
    100 * MEM_UOPS_RETIRED.SPLIT_STORES_PS * 5 / CPU_CLK_UNHALTED.THREAD;
```

通常、ストアは命令のリタイアメントを遅延させないため、コストの概算により分割ストアのペナルティーを検出することは容易ではありません。大量の分割ストアを検出するには、その IP でリタイアしたストアの総数で割ります。

SPLIT.STORE.RATIO =
MEM_UOPS_RETIRED.SPLIT_STORES_PS / MEM_UOPS_RETIRED.ANY_STORES_PS;

4K エイリアシング

ロードとストア間の 4KB エイリアシングの競合は、ロードの再発行を引き起こします。以下のモデルでは、概算値として 5 サイクルが使用されています。

要求イベント

LD_BLOCKS_PARTIAL.ADDRESS_ALIAS: ロードのうち、先行するストアとアドレスの一部が一致しているために再発行された数をカウントします。

イベントの使用

%4KALIAS.COST = 100 * LD_BLOCK_PARTIAL.ADDRESS_ALIAS * 5 / CPU_CLK_UNHALTED.THREAD;

ロードとストアのアドレス変換

リニアアドレスから物理アドレスへ変換するため、2 レベルのトランスレーション・ルックアサイド・バッファー (TLB) があります。第 1 レベルの TLB である DTLB 内のミスは、第 2 レベルの TLB である STLB にヒットし、7 サイクルのペナルティーが発生します。

STLB 内でミスが発生すると、プロセッサーは、アドレス変換を含むページテーブルのすべてのエントリーを検索する必要があります。この検索 (ページウォーク) のコストは、ページ・テーブル・エントリーの位置によって異なります。ページウォークの継続期間により、STLB ミスのコストをかなり正確に概算できます。

要求イベント

DTLB_LOAD_MISSES.STLB_HIT: DTLB にミスしたロードのうち、STLB にヒットしたロードをカウントします。このイベントはスキッドが短いため、IP レベルで使用できます。

DTLB_LOAD_MISSES.WALK_DURATION: STLB ミスの後に実行されるページウォークの継続期間 (サイクル数) をカウントします。イベントスキッドは通常、1 命令であり、命令、関数、モジュール、またはプロセスの粒度で問題を検出できます。

MEM_UOPS_RETIRED.STLB_MISS_LOADS_PS: STLB 内で変換ミスを発生したロードに対するプリサイスイベント。このイベントは、ページのページウォークを開始した最初のロードのみをカウントします。

イベントの使用

ロードに対する STLB ヒットのコスト:

%STLB.HIT.COST = 100 * DTLB_LOAD_MISSES.STLB_HIT * 7/ CPU_CLK_UNHALTED.THREAD;

ページウォークのコスト:

%STLB.LOAD.MISS.WALK.COST =
100 * DTLB_LOAD_MISSES.WALK_DURATION / CPU_CLK_UNHALTED.THREAD;

頻繁に STLB ミスが発生する命令やソース行を正確に把握するには、プリサイス STLB ミスイベントを IP レベルで使用します。

%STLB.LOAD.MISS =
100 * MEM_UOPS_RETIRED.STLB_MISS_LOADS_PS/ MEM_UOPS_RETIRED.ANY_LOADS_PS;

ページウォークの継続期間が長い (何百サイクルにもなる) 場合、LLC からページテーブルが排除されている兆候を示します。ページウォークの平均コストを判定するには、次の比率を使用します。

STLB.LOAD.MISS.AVGCOST =
DTLB_LOAD_MISSES.WALK_DURATION / DTLB_LOAD_MISSES.WALK_COMPLETED;

ロードほどではありませんが、ストアの STLB ミスはボトルネックとなる可能性があります。ストアそのものがボトルネックになっている場合、サイクルはストアに続く IP にタグ付けされます。

%STLB.STORE.MISS =
100 * MEM_UOPS_RETIRED.STLB_MISS_STORES_PS / MEM_UOPS_RETIRED.ANY_STORES_PS;

DTLB/STLB ミスを減らすことで、データの局所性を高めることができます。したがって、商用グレードのメモリー・アロケーターを使用してデータの局所性を改善することを考えます。プロファイルに基づく最適化機能を提供するコンパイラーでは、モジュール全体を操作できる場合、グローバル変数をリオーダーすることでデータの局所性が改善される可能性があります。ページウォークに長い時間がかかる場合、サーバー・アプリケーションや HPC アプリケーションでは、ラージページの利用を検討してください。

B.5.5 実行ストール

以下の項では、Sandy Bridge† マイクロアーキテクチャー固有のパフォーマンス監視イベントを使用したアウトオブオーダー・エンジンのストールを特定する方法について説明します。

B.5.5.1 長い命令レイテンシー

マイクロアーキテクチャーの変更によって、既存のコード内の一部のレガシー命令のレイテンシーが長くなる場合があります。このような状況のいくつかは検出できます。

- 3 つのオペランドを持つ低速の LEA 命令 (3.5.1.2 節を参照)
- フラグとマイクロオペレーション (uop) のマージ:「shift cl」命令で必要となります (3.5.2.5 節を参照)

パイプラインの初めにイベント生成条件が検出されるため、通常、サブイベントのスキッドが最大で 10 命令になる傾向があります。

イベントの使用

このイベントを、命令レベルではなく、プロセス、モジュール、関数の粒度で使用すると、イベントスキッドを気にせず、効果的にパフォーマンスの問題を特定できます。命令 IP の粒度で問題を特定するには、イベントで特定された関数に対して静的分析を行います。これらのイベントのコード・レイテンシーへの影響を評価するには、同じ粒度のサイクル数で割ります。全体的な影響を概算するには、これらの問題による合計サイクル数から開始し、影響が大きい場合は、サブイベントを使用して正確な原因を引き続き調査します。

特定のシナリオでかかった合計サイクル数

フラフマージの比率:

```
%FLAGS.MERGE.UOP =
    100 * PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP_CYCLES / CPU_CLK_UNHALTED.THREAD;
```

低速の LEA 命令の割り当て:

```
%SLOW.LEA.WINDOW =
    100 * PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW / CPU_CLK_UNHALTED.THREAD;
```

B.5.5.2 アシスト

アシストは、通常、アシスト処理を支援するマイクロコード・シーケンサーを起動します。マイクロコード・シーケンサーによって生成されたマイクロコードのサイクル数を判定することは、多くの場合、アシストの総コストを判定する適切な方法です。アシストの総コストが高い場合、アシストを詳細なタイプに分類すると効果的です。

マイクロコード・シーケンサーのサイクル数を使用したアシストの総コストを算出:

```
%ASSISTS.COST = 100 * IDQ.MS_CYCLES / CPU_CLK_UNHALTED.THREAD;
```

浮動小数点アシスト

x87 命令のデノーマル入力は FP アシストを必要とするため、何百サイクルのコストが生じる可能性があります。

```
%FP.ASSISTS = 100 *FP_ASSIST.ANY / INST_RETIRED.ANY;
```

インテル® SSE とインテル® AVX 間の遷移

インテル® SSE コードとインテル® AVX コード間の遷移については、15.3.1 節で詳しく説明しています。標準的なコストはおよそ 75 サイクルです。

```
%AVX2SSE.TRANSITION.COST = 75 * OTHER_ASSISTS.AVX_TO_SSE / CPU_CLK_UNHALTED.THREAD;
%SSE2AVX.TRANSITION.COST = 75 * OTHER_ASSISTS.SSE_TO_AVX / CPU_CLK_UNHALTED.THREAD;
```

2 ページにまたがる 32 バイトのインテル® AVX ストア命令は、およそ 150 サイクルのコストを生じるアシストを必要とします。32 バイトのインテル® AVX ストアの後に続く IP にタグ付けされた大量のマイクロコードは、アシストが発生したことを示唆します。

```
%AVX.STORE.ASSIST.COST = 150 * OTHER_ASSISTS.AVX_STORE / CPU_CLK_UNHALTED.THREAD;
```

B.5.6 投機の問題

この節では、パイプラインのフラッシュ招く、分岐命令の予測ミスについて説明します。

B.5.6.1 分岐予測ミス

分岐予測ミスの最大の課題は、予測ミスが発生した分岐を特定することです。分岐予測ミスには、約 20 サイクルのペナルティーが課せられます。コストは、予測ミスによって、およびデコード済み命令キャッシュ内またはレガシー・デコード・パイプライン内で正しいパスが見つかるかどうかによって異なります。

要求イベント

R_MISP_RETIRED.ALL_BRANCHES_PS は、分岐ターゲットを不適切に予測した分岐をカウントするプリサイスイベントです。これは、次の命令にスキッドするプリサイスイベントであるため、分岐予測ミスの後の正しいパスの最初の命令にタグ付けされます。このイベントは、プロセス、モジュール、関数、または命令の粒度で適用できます。

イベントの使用

分岐予測ミスのコストを算出するには、次の数式を使用します。

%BR.MISP.COST = 20 * BR_MISP_RETIRED.ALL_BRANCHES_PS / CPU_CLK_UNHALTED.THREAD;

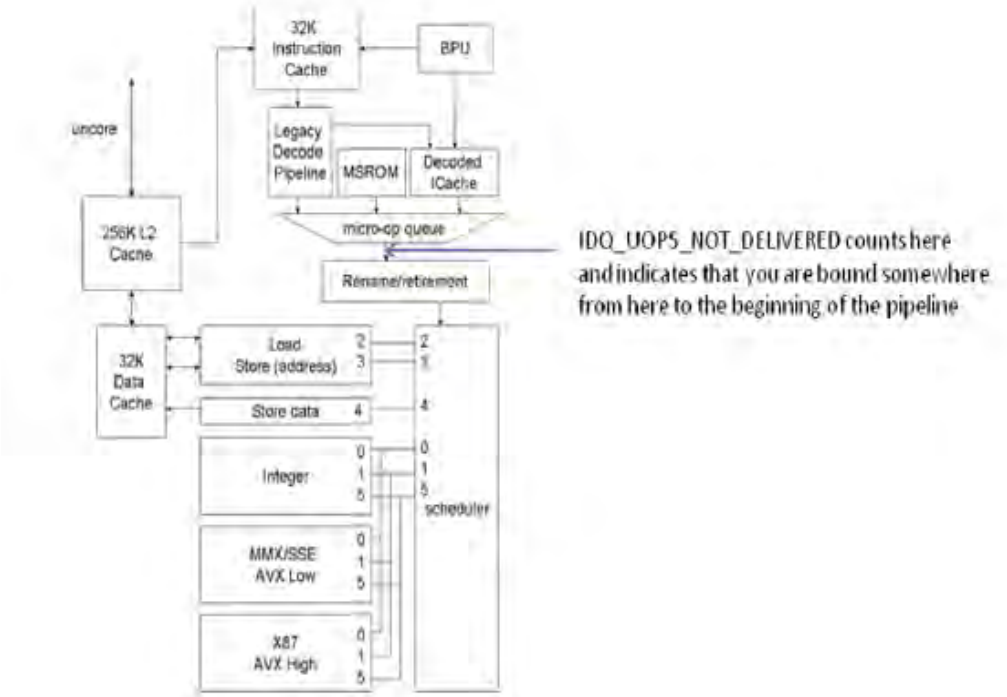
B.5.7 フロントエンドのストール

フロントエンドのストールについては、B.5.2 節で説明した解析の結果が、30% 以上フロントエンドに関連していない限り詳しく調査する必要はありません。この節では、パイプラインのフロントエンドで遅延を発生させる可能性がある問題について説明します。フロントエンドで検出されたイベントは、スキッドを予測できません。そのため、ペナルティーを IP レベルで関連付けません。このようなイベントは、関数レベル、モジュールレベル、プロセスレベルで調査します。

B.5.7.1 マイクロオペレーション (uop) の供給比率を理解する

カウンターの使用

イベント IDQ_UOPS_NOT_DELIVERED は、マイクロオペレーション (uop) を要求している間に、最大 4 つのマイクロオペレーション (uop) がリネームステージに提供されなかったときにカウントされます。パイプラインが停滞すると、リネームステージは、それ以降のマイクロオペレーション (uop) をフロントエンドに要求しません。次の図は、このイベントがマイクロオペレーション (uop) キューとリネームステージにあるマイクロオペレーション (uop) を追跡する仕組みを示します。



0、1、2、3 のマイクロオペレーション (uop) がフロントエンドから供給されている場合、IDQ_UOPS_NOT_DELIVERED イベントによりサイクルの分布を分類できます。

フロントエンドが有効であるか、実行がバックエンド依存であるサイクル数の比率 (%):
%FE.DELIVERING =
100 * (CPU_CLK_UNHALTED.THREAD - IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE) / CPU_CLK_UNHALTED.THREAD;

フロントエンドが 1 サイクルあたり 3 個のマイクロオペレーション (uop) を供給しているサイクル数の比率 (%):

%FE.DELIVER.3UOPS =
100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE) / CPU_CLK_UNHALTED.THREAD;

フロントエンドが 1 サイクルあたり 2 個のマイクロオペレーション (uop) を供給しているサイクル数の比率 (%):

%FE.DELIVER.2UOPS =
100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE) / CPU_CLK_UNHALTED.THREAD;

フロントエンドが 1 サイクルあたり 1 個のマイクロオペレーション (uop) を供給しているサイクル数の比率 (%):

%FE.DELIVER.1UOPS =
100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE) / CPU_CLK_UNHALTED.THREAD;

フロントエンドが 1 サイクルあたり 0 個のマイクロオペレーション (uop) を供給しているサイクル数の比率 (%):

%FE.DELIVER.0UOPS =
100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE) / CPU_CLK_UNHALTED.THREAD;

1 サイクルあたりに供給されるマイクロオペレーション (uop) の平均数: この比率は、バックエンド依存である場合に、フロントエンドが 1 サイクルあたり 4 個のマイクロオペレーション (uop) を供給できることを前提としています。

AVG.uops.per.cycle =
(4 * (%FE.DELIVERING) + 3 * (%FE.DELIVER.3UOPS) + 2 * (%FE.DELIVER.2UOPS) +
(%FE.DELIVER.1UOPS)) / 100

1 サイクルあたりに供給されるマイクロオペレーション (uop) の分布を確認することで、発生する可能性があるフロントエンドのボトルネックに関するヒントが得られます。デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えによる LCP やペナルティーなどの問題があると、複数のサイクルにわたってマイクロオペレーション (uop) が供給されない傾向があります。フェッチ帯域幅の問題やデコーダーのストールが発生すると、1 サイクルあたりに供給されるマイクロオペレーション (uop) が 4 個未満になります。

B.5.7.2 マイクロオペレーション (uop) キューのソースを理解する

マイクロオペレーション (uop) キューは、次のソースからマイクロオペレーション (uop) を取得します。

- デコード済み命令キャッシュ
- レガシー・デコード・パイプライン

- マイクロコード・シーケンサー (MS)

標準的な分布は、デコード済み命令キャッシュから供給されるマイクロオペレーション (uop) が約 80%、レガシー・デコード・パイプラインからの供給が 15%、マイクロコード・シーケンサーからの供給が 5% です。レガシー・デコード・パイプラインから過度なマイクロオペレーション (uop) が供給されると、デコード済み命令キャッシュが効果的に機能していないという警告される可能性があります。マイクロコード・シーケンサーから供給されるマイクロオペレーション (uop) の大部分 (複雑な命令や文字列操作など) は影響しないかもしれませんが、コードアシスト処理が望ましくない状況 (インテル® SSE コードからインテル® AVX コードへの遷移など) も考えられます。

必要なカウンターの説明

IDQ.DSB_UOPS: デコード済み命令キャッシュからマイクロオペレーション (uop) キューに供給されたマイクロオペレーション (uop)

IDQ.MITE_UOPS: レガシー・デコード・パイプラインからマイクロオペレーション (uop) キューに供給されたマイクロオペレーション (uop)

IDQ.MS_UOPS: マイクロコード・シーケンサーから供給されたマイクロオペレーション (uop)

カウンターの使用

デコード済み命令キャッシュから供給されるマイクロオペレーション (uop) の比率 (%):

%UOPS.DSB = IDQ.DSB_UOPS / **ALL_IDQ_UOPS**;

レガシー・デコード・パイプラインから供給されるマイクロオペレーション (uop) の比率 (%):

%UOPS.MITE = IDQ.MITE_UOPS / **ALL_IDQ_UOPS**;

マイクロコード・シーケンサーから供給されるマイクロオペレーション (uop) の率 (%):

%UOPS.MS = IDQ.MS_UOPS / **ALL_IDQ_UOPS**;

ALL_IDQ_UOPS = (IDQ.DSB_UOPS + IDQ.MITE_UOPS + IDQ.MS_UOPS);

アプリケーションがフロントエンド依存ではない場合、マイクロオペレーション (uop) がレガシー・デコード・パイプラインから供給されているか、デコード済み命令キャッシュから供給されているかはあまり重要ではありません。マイクロコード・シーケンサーから過度のマイクロオペレーション (uop) が供給されている場合、アシストが問題であるかどうかを調査することが推奨されます。

調査が必要になるケースを以下に示します:

- (%FE_BOUND > 30%) と (%UOPS.DSB < 70%)
  " フロントエンド依存である" ケースを定義するしきい値として 30% を適用します。このしきい値はほとんどの状況に適用できますが、ワークロードにより少し異なることがあります。
  – デコード済み命令キャッシュからマイクロオペレーション (uop) が供給されない理由を調査する。
  – レガシー・デコード・パイプラインに影響する可能性がある問題を調査する。

- (%FE_BOUND > 30%) と (%UOP_DSB > 70%)
  – 小さすぎて効果的でないコード領域が実行されている可能性があるため、デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えを調査します。
  – 分岐の予測ミスは FE のパフォーマンスに影響を与えるため、バッド・スペキュレーション数を調べます。

  - 32 バイト・チャンクのヒットごとに供給されるマイクロオペレーション (uop) の平均数を判定します。32 バイト・チャンク間で供給される処理済み分岐が多い場合、1 サイクルごとに供給されるマイクロオペレーション (uop) に影響します。
  - デコード済み命令キャッシュからのマイクロオペレーション (uop) の供給が問題である可能性もありますが、ここでは取り上げません。

- (%FE_BOUND < 20%) と (%UOPS_MS>25%)
  "フロントエンド依存ではない" ケースを定義するしきい値として 20% を適用します。このしきい値はほとんどの状況に適用できますが、ワークロードにより少し異なることがあります。

  次の手順は、マイクロコード・シーケンサーからマイクロオペレーション (uop) が供給された原因を判別する上で役立ちます。使用頻度の高い順に示します。
  - レイテンシーが長い命令 - 4 マイクロオペレーション (uop) を超えるすべての命令は、マイクロコード・シーケンサーを起動します。一部の命令 (超越関数など) は、マイクロコードから多数のマイクロオペレーション (uop) が生成されます。
  - 文字列操作 - 文字列操作は、大量のマイクロコードを生成する可能性があります。状況によっては、トリップカウントが 3 を超える文字列操作 (REP MOVSB など) により、70 サイクルを超えるコストのアシストが発生します。
  - アシスト - B.5.5.2 節を参照してください。

B.5.7.3 デコード済み命令キャッシュ

デコード済み命令キャッシュには、レガシー・デコード・パイプラインと比べて多くの利点があります。複数のマイクロオペレーション (uop) にデコードされる命令やレングス変更プリフィクス (LCP) のストールなど、レガシー・デコード・パイプラインの多数のボトルネックが排除されます。

デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えは、デコード済み命令キャッシュ内でルックアップが失敗した場合にのみ発生し、通常、パイプラインのフロントエンドで 0～ 3 サイクルのコストが生じます。

要求イベント

デコード済み命令キャッシュイベントはすべて長いスキッドを持ち、通常、イベントがタグ付けされた命令は問題の原因ではありません。したがって、この問題は、プロセス、モジュール、関数の粒度でのみ適用されます。

DSB2MITE_SWITCHES.PENALTY_CYCLES: デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えに起因するサイクルをカウントします。ただし、バックエンド依存のため、マイクロオペレーション (uop) キューがマイクロオペレーション (uop) を受け入れることができない場合を除きます。

DSB2MITE_SWITCHES.COUNT: デコード済み命令キャッシュとレガシー・デコード・パイプライン間の切り替えの数をカウントします。

DSB_FILL.ALL_CANCEL: デコード済み命令キャッシュへのフィルが取り消されたときにカウントされます。

DSB_FILL.EXCEED_DSB_LINES: 32 バイト・チャンクに対して、デコード済み命令キャッシュに割り当てられたライン数が 3 を超えたためにフィルが取り消されたときにカウントされます。

イベントの使用

ここで検討する内容はフロントエンド・イベントに関連しているため、特定の命令にイベントをタグ付けしないようにします。

デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えのコストを判定:

```
%DSB2MITE.SWITCH.COST =
    100 * DSB2MITE_SWITCHES.PENALTY_CYCLES / CPU_CLK_UNHALTED.THREAD;
```

デコード済み命令キャッシュからレガシー・フロントエンドへの切り替えあたりの平均コストを判定:

```
AVG.DSB2MITE.SWITCH.COST =
    DSB2MITE_SWITCHES.PENALTY_CYCLES / DSB2MITE_SWITCHES.COUNT;
```

デコード済み命令キャッシュにおけるミスの原因を判定

デコード済み命令キャッシュ内ではパーシャルヒットはありません。32 バイト・チャンクのルックアップの一部となっているマイクロオペレーション (uop) のいずれかが見つからない場合は、そのトランザクションに対するすべてのマイクロオペレーション (uop) で、デコード済み命令キャッシュミスが発生します。

デコード済み命令キャッシュ内でマイクロオペレーション (uop) が見つからない主な理由は 3 つあります。

- 32 バイトのコードチャンクの一部が、3 ウェイ方式のデコード済み命令キャッシュに収まりません。
- デコード済み命令キャッシュに対して、頻繁に実行されるコードセクションが大きすぎます。クライアント・アプリケーションは「ホット」な小規模のコードセットを含む傾向があるため、これは、サーバー・アプリケーションに当てはまります。
- デコード済み命令キャッシュがフラッシュされています (ITLB のエントリーが排出された場合など)。

32 バイト・コードの一部がデコード済み命令キャッシュの 3 ウェイラインに収まらない場合、プロセス、モジュール、関数、または命令の粒度で DSB_FILL.EXCEED_DSB_LINES イベントを使用します。

```
%DSB.EXCEED.WAY.LIMIT = 100 * DSB_FILL.EXCEED_DSB_LINES/ DSB_FILL.ALL_CANCEL;
```

B.5.7.4 レガシー・デコード・パイプラインにおける問題

マイクロオペレーション (uop) キューに入るマイクロオペレーション (uop) の大部分がレガシー・デコード・パイプラインから供給される場合、そのステージに影響するボトルネックがあるかどうかをチェックする必要があります。レガシー・デコード・パイプラインにおける最も一般的なボトルネックは次のとおりです。

- 十分な命令を提供しないフェッチ
  これは、ホットなコードが完全にアライメントされていない場合に発生します。例えば、実行のためにフェッチしているホットコードが 15 番目のバイトにある場合、1 バイトだけがフェッチされます。

- レングス変更プリフィクスが命令長デコーダー内でストールする場合
  2 ～ 4 個のマイクロオペレーション (uop) にデコードされた命令では、デコーダーのスループットでバブルが発生します。これは、デコーダーの直前にある命令キューが一杯になると、これらの命令によってペナルティーが生じる可能性があることを意味します。

```
%ILD.STALL.COST = 100 * ILD_STALL.LCP * 3 / CPU_CLK_UNHALTED.THREAD;
```

B.5.7.5 命令キャッシュ

大きなホット・コード・セクションを含むアプリケーションでは、命令キャッシュおよび ITLB に関する問題が発生する傾向があります。これは、サーバー・アプリケーションでより多く見られます。

要求イベント

ICACHE.MISSES: 命令キャッシュをミスした命令バイトフェッチの数をカウントします。

イベントの使用

命令キャッシュミスが問題に影響しているかどうかを判定するには、同じ粒度 (プロセス、モデル、または関数) を使用して、リタイアした命令イベントのカウント数と比較します。リタイアした命令イベントのカウントの 1% を超える場合は、重大な問題であると考えられます。

ICACHE.PER.INST.RET = ICACHE.MISSES / INST_RETIRED.ANY;

命令キャッシュミスが重大な問題を招いている場合、プロファイルに基づく最適化を使用して、ホット・コード・セクションのサイズを減らしてみます。ほとんどのコンパイラーには、テキストの並べ替えオプションが用意されています。これは、ページ数および (それほど多くありませんが) アプリケーションがターゲットとするページ数を減らすうえで役立ちます。

アプリケーションがマクロを多用している場合、関数に変換するか、インテリジェント・リンクを使用して繰り返し実行されるコードを排除します。

B.6 インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーのパフォーマンス・イベントの使用

インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーのマイクロアーキテクチャーには、固有のパフォーマンス・イベントがあります。https://perfmon-events.intel.com/ (英語) を参照してください。

B.6.1 パフォーマンス・カウンターの結果を理解する

各パフォーマンス・イベントは、コアがアクティブなときに、コアで発生したマイクロアーキテクチャー条件を検出します。コアは、以下の場合にアクティブとなります。

- コードを実行している (HALT 命令は除く)。
- 他方のコアまたはプラットフォーム上の論理プロセッサーによってスヌープされている。これはコアが HALT 状態の場合にも発生します。

一部のマイクロアーキテクチャーの条件は、複数のコアによって共有されるサブシステムに適用されます。また、一部のパフォーマンス・イベントでは、イベントマスク (またはユニットマスク) による物理プロセッサー境界またはバス・エージェント境界での条件を可能にします。

一部のイベントでは、物理プロセッサー上の全コアでのカウントではなく、特定のコアに関連したマイクロアーキテクチャー条件のカウントが可能です (https://perfmon-events.intel.com/ (英語) に記載されている L2 およびバス関連イベントを参照してください)。

マルチスレッド・ワークロードがすべてのコアを連続して使用しない場合、コア固有の条件をカウントするパフォーマンス・カウンターは、HALT したコア上である程度進行してから、カウントを中止します。または、ユニットマスクを使用すると、いずれかのプロセッサー・コアに起因する条件の発生を引き続きカウントすることもできます。通常は、ユニット・マスク・フィールドの最上位 2 ビット (IA32_PERFEVTSELx MSR のビット 15:14) を調整することによって、このような非対称な事象を識別できます (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 17 章を参照)。

HALT したコア上では、そのコアがスヌープされても進行しない 3 つのサイクル・カウント・イベントがあります。それは、Unhalted コアサイクル、Unhalted 参照サイクル、Unhalted バスサイクルです。この 3 つのイベントはすべて、イベント 3CH によって選択されたユニットで検出されます。

一部のイベントは、マイクロアーキテクチャー条件は検出できますが、開始したコアや物理プロセッサーを識別する能力は限定されます。例えば、ユニットマスク 20H を使って bus_drdy_clocks をプログラムすると、バス上のすべてのエージェントを含めることができます。この場合、各コアのパフォーマンス・カウンターはほぼ同じ値を報告します。カウントを解釈するパフォーマンス・ツールでは、バス・アクティビティーは 1 つのコア (各コアの合計は使用しない) からのイベントカウントと同一になる必要があることを、考慮しなければなりません。

上記は、11B でイベントマスク内のコア固有のサブフィールド (IA32_PERFEVTSELx MSR のビット 15:14) をプログラムした場合にも当てはまります。各コアのパフォーマンス・カウンターによって報告される結果は、ほぼ同じになります。

B.6.2 比率の解釈

ワークロードのさまざまな特性を解析するには、2 つのイベントの比率が有用です。その比率は、次のように複数の粒度で取得できます: (1) アプリケーション・スレッドごと、(2) 論理プロセッサーごと、(3) コアごと、(4) 物理プロセッサーごと。

ソフトウェア開発の観点からは (1) の比率が最も有用ですが、マルチスレッド・アプリケーション上でアプリケーション・スレッドごとにプロセッサー・アフィニティーを明示的に管理する必要があります。その他は、ハードウェア使用率に関する情報を提供します。

通常、一度の実行での測定結果 (比率にかかわるすべてのイベントの測定結果) を収集します。その理由は以下のとおりです。

- マルチスレッド・ワークロードの比率を測定する場合、一度の実行で全イベントの結果を取得すると、各スレッドにどのイベントカウンター値が属しているか把握できます。
- ライトバックなど一部のイベントは、異なる実行では動作に決定論性がない可能性があります。そのような場合、一度の実行で収集された測定結果のみが有効な比率となります。

B.6.3 特定のイベントに関する注意事項

この節では、https://perfmon-events.intel.com/ (英語) に記載されているパフォーマンス・イベントを解釈する際の、イベント固有の注意事項について説明します。

- **L2_Reject_Cycles、イベント番号 30H** — このイベントは、2 次キャッシュが新しいアクセス要求を拒否した間のサイクルをカウントします。
- **L2_No_Request_Cycles、イベント番号 32H** — このイベントは、1 次キャッシュから要求、または 2 次キャッシュへのプリフェッチが発行されていなかったサイクルをカウントします。
- **Unhalted_Core_Cycles、イベント番号 3C、ユニットマスク 00H** — このイベントは、アクティブなコアによって認識された最小時間単位をカウントします。

  ほとんどのオペレーティング・システムでは、HLT 命令によってアイドルタスクが実装されています。その場合、アイドルタスクのクロック数はカウントされません。拡張版 Intel SpeedStep® テクノロジーによる遷移によって、コアの動作周波数が変化することがあります。そのため、このイベントを利用して一定時間サンプリングを行うと、結果に差が生じる可能性があります。

- **Unhalted_Ref_Cycles、イベント番号 3C、ユニットマスク 01H** — このイベントは、カウントされるサイクルごとの間隔が一定になることが保証されます。具体的には、コアがアクティブなときにバス・クロック・サイクルに合わせてカウントされます。サイクルは、コアクロック周波数をセットするバス比率を掛けることによってコア・クロック・ドメインに変換できます。

- **Serial_Execution_Cycles、イベント番号 3C、ユニットマスク 02H** — このイベントは、一方のコアがコードをアクティブに実行し(非 HALT 状態)、物理プロセッサーの他方のコアが停止しているバスサイクルをカウントします。
- **L1_Pref_Req、イベント番号 4FH、ユニットマスク 00H** — このイベントは、データ・キャッシュ・ユニット (DCU) の 2 次キャッシュからデータ・キャッシュラインをプリフェッチする要求回数をカウントします。2 次キャッシュがビジーの場合、要求が拒否されることがあります。拒否された要求は再送信されます。
- **DCU_Snoop_to_Share、イベント番号 78H、ユニットマスク 01H** — このイベントは、他方のコアが必要とするキャッシュラインを DCU がスヌープした回数をカウントします。キャッシュラインが L1 命令キャッシュまたは他のコアのデータキャッシュ内に存在しないか、または他方のコアが書き込みを必要としているにもかかわらず読み出し専用に設定されています。このようなスヌープは、DCU ストアポートを介して行われます。DCU スヌープが頻繁に発生すると、DCU へのストアと競合するため、ストアのレイテンシーが増加したり、パフォーマンスが影響を受ける可能性があります。
- **Bus_Not_In_Use、イベント番号 7DH、ユニットマスク 00H** — このイベントは、バスの完了を待機しているトランザクションがないコアのバスサイクル数をカウントします。
- **Bus_Snoops、イベント番号 77H、ユニットマスク 00H** — このイベントは、バス上で検出された外部スヌープに対する CLEAN、HIT、または HITM 応答の数をカウントします。

  シングルプロセッサー・システムでは、CLEAN 応答と HIT 応答が発生することはありません。マルチプロセッサー・システムでは、このイベントは、一方のプロセッサーで L2 ミスが発生し、ミスしたデータを他方のプロセッサーで発見できなかったことを示します。

  シングルプロセッサー・システムでは、HITM 応答は、L1 ミス (命令またはデータ) が発生し、ミスしたキャッシュラインが変更状態にある他のコアで見つかったことを示します。マルチプロセッサー・システムでも、このイベントは、L1 ミス (命令またはデータ) が発生し、ミスしたキャッシュラインが変更状態にある他のコアで見つかったことを示します。

B.7 パフォーマンス分析のドリルダウン手法

ソフトウェア・パフォーマンスは、アプリケーション・コードとプロセッサーのマイクロアーキテクチャーの特性が密接に関連します。これらの相互の影響は、パフォーマンス監視イベントによって推測できます。各マイクロアーキテクチャーは、それぞれのサブシステムを対象とする多数のパフォーマンス・イベントを備えています。主要なパフォーマンス・イベントを選択する手法を使用すると、パフォーマンスのボトルネックをよく理解し、コード・チューニング作業の効率を向上できます。

最近のインテル® 64 および IA-32 プロセッサーは、アウトオフオーダー実行エンジンを使用するマイクロアーキテクチャーを採用しています。さらに、プログラムを順に実行するインオーダーのフロントエンドとリタイアメント・ロジックも含まれます。これらのプロセッサーはスーパースケーラー・ハードウェア、バッファリング、投機実行を利用するため、パフォーマンス・イベントとソフトウェアから認識できるパフォーマンスのボトルネックの解釈が複雑になります。

この節では、パフォーマンス・イベントを使用して、パフォーマンスのボトルネックが存在する領域を詳細に分析する手法について説明します。パフォーマンス・イベントを絞り込むことで、インテル® VTune™ プロファイラーを使用して、パフォーマンスのボトルネックとソースコード上の位置を関連付け、第 3 章～第 11 章で説明したコーディングの推奨事項を適用できます。この手法の一般原則はさまざまなマイクロアーキテクチャーに適用できますが、この節では分かりやすいように、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーで利用可能なパフォーマンス・イベントを使用します。

パフォーマンス・チューニングは通常、厳密に定義されたワークロードの完了にかかる時間を減らすことに焦点を置きます。パフォーマンス・イベントを使用して、ワークロードの開始から終了までの経過時間を測定できます。したがって、プロセッサー・サイクルの計測値が小さくなれば、ワークロード完了までの経過時間が短縮されたことになります。

このドリルダウン手法では、4 フェーズのパフォーマンス・イベントを測定することによって、主要なパイプライン・ステージまたはマイクロアーキテクチャー・サブシステムとコードの相互作用の特性を評価します。図 B-16 は、パ

フォーマンス･イベントのドリルダウン手法とソフトウェア･チューニングによるフィードバック･ループの関係を示しています。

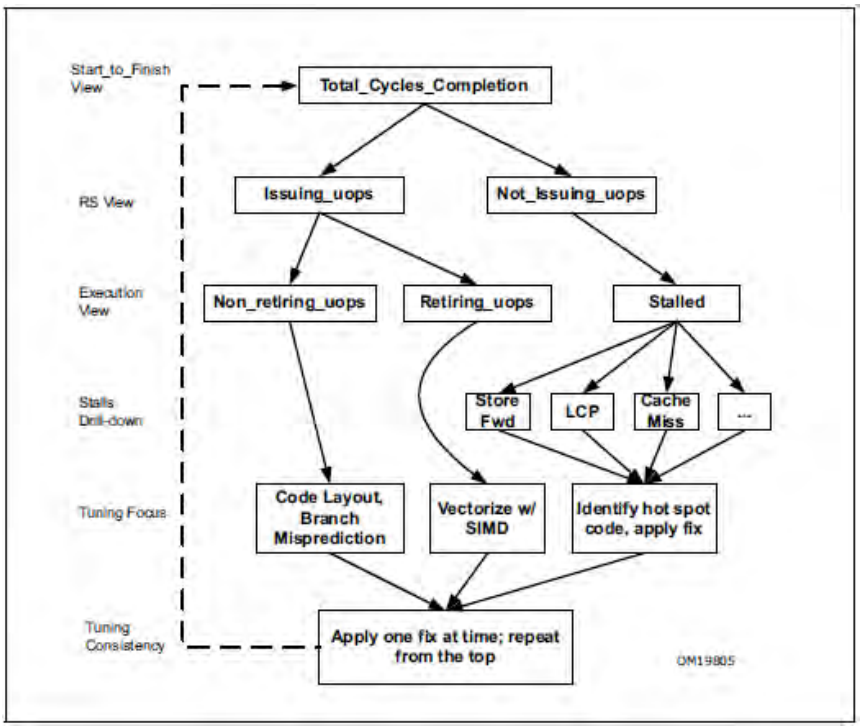


図 B-16 パフォーマンス･イベントのドリルダウンとソフトウェア･チューニングのフィードバック･ループ

通常、パフォーマンス監視ハードウェアのロジックは、サイクル、マイクロオペレーション (uop)、アドレス参照、インスタンスなどのドメインごとに異なるマイクロアーキテクチャー状態を測定します。このドリルダウン手法では、以下に説明する近似的手法によって、各フェーズで直感的に理解できるサイクルベースのビューを得られるようにします。

- **総サイクル数の測定** — 総サイクル数は、指定したアプリケーションの開始から終了までに費やされたサイクル数の合計です。一般的なパフォーマンス･チューニングでは、Total_cycles (総サイクル数) メトリックは CPU_CLK_UNHALTED.CORE で測定できます。詳細は、https://perfmon-events.intel.com/ (英語) に示されます。
- **発行ポートでのサイクル構成** — リザベーション･ステーション (RS) は、プログラムが処理を進行できるように、実行されるマイクロオペレーション (uop) をディスパッチします。したがって、メトリック Total_cycles は、2 つの排他的な構成要素である Cycles_not_issuing_uops と Cycles_issuing_uops に分類されます。Cycles_not_issuing_uops は、RS が実行されるマイクロオペレーション (uop) を発行していないサイクル数を示し、Cycles_issuing_uops は、RS が実行されるマイクロオペレーション (uop) を発行しているサイクル数を示します。Cycles_issuing_uops のサイクルには、アーキテクチャー上のコードパス、またはスペキュレーティブなコードパスのマイクロオペレーション (uop) が含まれます。
- **アウトオブオーダー(OOO)実行のサイクル構成** — アウトオブオーダー･エンジンは、複数のマイクロオペレーション (uop) を並行して実行できる複数の実行ユニットを備えています。1 つの実行ユニットがストールしても、プログラムの実行がストールするとは限りません。このドリルダウン手法では、プログラム実行の進行を近似的に示すサイクル構成ビューを作成します。関連するメトリックは、Cycles_stalled、Cycles_not_retiring_uops、Cycles_retiring_uops の 3 つです。
- **実行ストールの分析** — プログラム全体の実行サイクル構成から、パフォーマンス･イベントの選択を絞り込み、ワークロードとマイクロアーキテクチャー･サブシステム間の非生産的な相互作用をピンポイントで識別できます。

マイクロアーキテクチャー･サブシステムのストールまたは非効率なスペキュレーティブ･エグゼキューションによるサイクルの損失が特定できたら、インテル® VTune™ プロファイラーを使用して、パフォーマンスに影響する要因とソースコードの位置を関連付けることができます。ストールや予測ミスによるパフォーマンスへの影響が小さい場合

でも、インテル® VTune™ プロファイラーはソース内のホットスポット関数の位置を特定できるため、プログラマーはこれらの関数のベクトル化によって得られるメリットを評価できます。

B.7.1 発行ポートでのサイクル構成

最近のプロセッサーのマイクロアーキテクチャーは、フロントエンドでプログラム命令をマイクロオペレーション (uop) にデコードしながら、マイクロオペレーション (uop) のストリームをネイティブで実行するアウトオブオーダー・エンジンを採用しています。メトリック Total_cycles だけでは、プログラムの実行の生産的なサイクルと非生産的なサイクルの分類については不明瞭です。一貫性のあるサイクルベースの分類が行えるように、以下の 2 つのメトリックが用意されています。これらのメトリックは、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーで利用可能なパフォーマンス・イベントで測定できます。そのイベントとは次の 2 つです。

- **Cycles_not_issuing_uops** — このメトリックは、イベント RS_UOPS_DISPATCHED によって測定できます。対象とするパフォーマンス・イベント選択 (IA32_PERFEVSELx) MSR 内で INV ビットをセットし、カウンターマスク (CMASK) の値を 1 に設定します (https://perfmon-events.intel.com/ (英語) を参照)。インテル® VTune™ プロファイラーには、CMASK と INV がそれぞれ 1 に設定された、イベント RS_UOPS_DISPATCHED.CYCLES_NONE が用意されています。
- **Cycles_issuing_uops** — このメトリックは、イベント RS_UOPS_DISPATCHED によって測定できます。対象とするパフォーマンス・イベント選択 MSR 内で INV ビットをクリアし、カウンターマスク (CMASK) の値を 1 に設定します。

なお、ここで使用しているサイクル分類ビューは、基本的には近似的なものです。RS が一杯か空かを識別したり、RS は空になっているがパイプライン内のマイクロオペレーション (uop) の一部がリタイアの途中であるといった詳細な状況を把握することはできません。

B.7.2 アウトオブオーダー (OOO) 実行のサイクル構成

アウトオブオーダー (OOO) エンジン内では、スペキュレーティブ・エグゼキューション (投機実行) がプログラムの処理を進行する上で重要な要素となります。しかし、予測が外れたコードパスに含まれるマイクロオペレーション (uop) のスペキュレーティブ・エグゼキューションは、実行リソースと実行帯域幅を消費する非生産的なアクティビティーです。

Cycles_not_issuing_uops は、定義により、アウトオブオーダー・エンジンがストールしたサイクル数 (Cycles_stalled) を表します。このメトリックは、近似的にプログラムの処理が進行していないサイクル数として解釈できます。

発行されたマイクロオペレーション (uop) は、すべてリタイアメントで完了するとは限りません。リタイアメントに到達しないマイクロオペレーション (uop) は、プログラムの処理進行に貢献していません。したがって、さらなる近似的手法によって、Cycles_issuing_uops を以下の 2 つの指標に分類できます。

- **Cycles_non_retiring_uops** — リタイアしないマイクロオペレーション (uop) のサイクル数を直接測定するイベントはありませんが、いくつかの利用可能なパフォーマンス・イベントからこのメトリックを算出できます。
  - 発行ポートを通過するマイクロオペレーション (uop) の発行速度は一定とします。したがって、uops_rate = Dispatch_uops/Cycles_issuing_uops と定義します。ここで、Dispatch_uops はイベント RS_UOPS_DISPATCHED によって測定できます (INV ビットと CMASK をクリア)。
  - リタイアしない非生産的なマイクロオペレーション (uop) の数を、non_productive_uops = Dispatch_uops - executed_retired_uops によって近似的に算出します。ここで、executed_retired_uops は、実行帯域幅を消費した、処理の進行に貢献する生産的なマイクロオペレーション (uop) を表します。
  - executed_retired_uops は、(イベント UOPS_RETIRED.ANY によって測定) num_retired_uops と (イベント UOPS_RETIRED.FUSED によって測定される) num_fused_uops の 2 つのメトリックの合計によって近似的に求められます。したがって、Cycles_non_retiring_uops = non_productive_uops / uops_rate となります。

- **Cycles_retiring_uops** — このメトリックは、Cycles_retiring_uops = num_retired_uops / uops_rate から算出されます。

ここで使用するサイクル分類手法では、生産的なマイクロオペレーション (uop) と非生産的なマイクロオペレーション (uop) が同じサイクルでアウトオブオーダー・エンジンにディスパッチされる状況を識別できません。しかし、経験則によると、リタイアしないマイクロオペレーション (uop) が多数存在すると、アウトオブオーダー・エンジンが混雑し、プログラムがストールする可能性が高いため、この近似的手法は妥当であると言えます。

Total_cycles に関連する、Cycles_non_retiring_uops、Cycles_stalled、Cycles_retiring_uops の 3 つのメトリックを評価することで、チューニング作業に役立つ以下の情報が得られます。

- Cycles_non_retiring_uops が大きい場合、コードのレイアウトに注目して、分岐予測ミスを減らすことが重要です。
- Cycles_non_retiring_uops と Cycles_stalled がどちらも小さい場合、パフォーマンス・チューニングでベクトル化などの手法を重視し、ホットスポット関数のリタイアメントのスループットを向上させるべきです。
- Cycles_stalled が大きい場合、マイクロアーキテクチャー・パイプラインのさらに深い場所に潜むボトルネックを発見するため、さらなるドリルダウンが必要です。

B.7.3 パフォーマンス・ストールのドリルダウン

状況によっては、マイクロアーキテクチャー内の各種のストレスポイントを原因とするストールサイクル数を評価し、候補となるストレスポイントを合計する手法が有効です。しかし、この手法は非常に大まかな簡略化に基づくものであり、アウトオブオーダー・エンジンのスーパースケーラー構造とバッファリングを考慮することは困難です。

各種のパフォーマンス・イベントに関連するドメインは多様であるため、各ストレスポイントのパフォーマンスへの影響をサイクルベースで評価すると、影響の過大評価や過小評価によってさまざまな誤差が生じる可能性があります。

特定の原因のパフォーマンス全体に対する影響を推測する場合、インスタンスあたりのコストに対して、そのマイクロアーキテクチャー状態の発生回数を示すイベントカウントを掛けると、その影響が過大に評価される傾向があります。その結果、各種のストレスポイントが原因で失われたサイクル数の合計が、より正確な指標である Cycles_stalled の値を上回ることがあります。

しかし、個々のストレスポイントが原因で失われたサイクル数を合計する手法は、コードをチューニングして各ストレスポイントのパフォーマンスへの影響を解決する際に、コード・チューニング・ループ作業の有効性を測定する反復的な指標としては有益です。この節では、パフォーマンス・イベントによってカウントでき、本書で説明した以下のコーディングの推奨事項によって解決できる、パフォーマンスのボトルネックの一般的な原因について説明します。

以下の項目は、マイクロアーキテクチャーの典型的なストレスポイントを示します。

- **L2 ミスの影響** — L2 ロードミスがあると、メモリー・サブシステムのフルレイテンシーが明らかになります。システムメモリーのアクセス・レイテンシーは、一般にチップセットの種類によって 100 サイクル以上異なります。サーバー・チップセットはデスクトップ・チップセットより長いレイテンシーを示す傾向があります。L2 キャッシュミスの参照数は、MEM_LOAD_RETIRED.L2_LINE_MISS によって測定できます。

  システムメモリーのレイテンシーに L2 ミスの数を掛けることで L2 ミスの全体の影響を推測する手法は、アウトオブオーダー・エンジンが複数の未処理のロードミスを処理することを考慮していません。L2 ミスのレイテンシーと回数の乗算は、各 L2 ミスが逐次的に発生することを前提としています。

  L2 ミスが影響する推測の精度を高めるには、CMASK の値を 1 に設定してイベント BUS_REQUEST_OUTSTANDING を使用する代替手法も検討する必要があります。この手法では、未処理のバス読み出し要求から得られるデータについてアウトオブオーダー・エンジンが待機しているサイクル数を測定します。これによって、メモリー・レイテンシーと L2 ミス数の乗算による過大評価の問題を解決できます。

- **L2 ヒットのシャドウ** — L2 からのメモリーアクセスがあると、L2 レイテンシーのコストが発生します (表 2-28 を参照)。L2 にヒットしたキャッシュライン参照の数は、MEM_LOAD_RETIRED.L1D_LINE_MISS - MEM_LOAD_RETIRED.L2_LINE_MISS の 2 つのイベントの差によって測定できます。

  L2 ヒットのレイテンシーに L2 ヒット参照の数を掛けることで L2 ヒット全体の影響を推測する手法は、アウトオブオーダー・エンジンが複数の未処理のロードミスを処理できることを考慮していません。

- **L1 DTLB ミスのシャドウ** — DTLB ルックアップ・ミスのコストは、約 10 サイクルです。イベント MEM_LOAD_RETIRED.DTLB_MISS は、DTLB にミスしたマイクロオペレーション (uop) のロード回数を測定します。
- **LCP の影響** — LCP ストール全体の影響は、イベント ILD_STALLS によって直接測定できます。イベント ILD_STALLS は、低速デコーダーがトリガーされた回数を測定します。各インスタンスのコストは 6 サイクルです。
- **ストア・フォワーディングによるストールの影響** — ストア・フォワーディングが、ハードウェアが要求するアドレスやサイズ条件を満たさない場合、ストールが発生します。遅延は、ストア・フォワーディングのストール状況によって異なります。そのため、各種ストア・フォワーディングのストール状況をきめ細かく検出するパフォーマンス・イベントがいくつか用意されています。これには以下のものがあります。
  - 未知のアドレスへの先行するストアによってブロックされたロード: この状況は、イベント Load_Blocks.Sta によって測定できます。インスタンスあたりのコストはおよそ 5 サイクルです。
  - 先行するストアと部分的にオーバーラップしているか、ロードと先行するストアの間の 4K バイト・エイリアス・アドレスと部分的にオーバーラップしているロード。これらの 2 つの状況は、イベント Load_Blocks.Overlap_store で測定されます。
  - キャッシュラインの境界にまたがるロード: これは Load_Blocks.Until_Retire によって測定されます。インスタンスあたりのコストはおよそ 20 サイクルです。

B.8 インテル® Core™ マイクロアーキテクチャーのイベント比率

付録 B.8 には、パフォーマンス・イベントを使用したパフォーマンスのボトルネックの迅速な診断例が記載されています。

この節では、パフォーマンス・イベントを使用して各種のパフォーマンス分析、ワークロードの特性評価、パフォーマンス・チューニングに役立つメトリックを評価する方法について、さらに詳しく説明します。

なお、インテル® Core™ マイクロアーキテクチャーの多くのパフォーマンス・イベント名は、XXXX.YYY の形式で表されます。この表記の一般的な規則では、XXXX は通常はパフォーマンス・イベント選択レジスター (IA32_PERFEVSELx) 内の固有のイベント選択コードに対応し、YYY は特定のマイクロアーキテクチャー状態を個別に定義する固有のサブイベント・マスクに対応します (https://perfmon-events.intel.com/ (英語) を参照してください)。

B.8.1 命令リタイアごとのクロック比率 (CPI)

1. 命令リタイアごとのクロック比率 (CPI):
   CPU_CLK_UNHALTED.CORE / INST_RETIRED.ANY.

インテル® Core™ マイクロアーキテクチャーは、理想的な状態では CPI が 0.25 まで低くなります。しかし、実際の多くのコードの CPI の値はそれより大きくなります。特定のワークロードで CPI の値が大きいほど、コードのチューニングによってパフォーマンスを改善できる可能性は高くなります。CPI は全体的な指標であり、どのマイクロアーキテクチャー・サブシステムが高い CPI 値に影響しているかを特定するものではありません。

以下の節では、フロントエンド、実行、メモリーとの相互作用の特性評価に役立つ一連のイベント比率を定義しています。

B.8.2 フロントエンド比率

2. RS Full Ratio (RS フル比率):
   RESOURCE_STALLS.RS_FULL / CPU_CLK_UNHALTED.CORE * 100
3. ROB Full Ratio (ROB フル比率):
   RESOURCE_STALLS.ROB_FULL / CPU_CLK_UNHALTED.CORE * 100
4. Load or Store Buffer Full Ratio (ロードまたはストアバッファー・フル比率):
   RESOURCE_STALLS.LD_ST / CPU_CLK_UNHALTED.CORE * 100

ROB Full Ratio、RS Full Ratio、Load Store Buffer Full Ratio の値が小さく、CPI 値が大きい場合は、フロントエンドがアウトオブオーダー・エンジンのバッファーを一杯にするのに十分な速度で命令とマイクロオペレーション (uop) を供給できないため、アウトオブオーダー・エンジンが実行するマイクロオペレーション (uop) を待機している可能性があります。この場合は、フロントエンドにパフォーマンスの問題がないかをさらにチェックする必要があります。

B.8.2.1 コードの局所性

5. Instruction Fetch Stall (命令フェッチストール):
   CYCLES_L1I_MEM_STALLED / CPU_CLK_UNHALTED.CORE * 100

Instruction Fetch Stall 比率は、命令フェッチユニット (IFU) がキャッシュミスと命令 TLB (ITLB) ミスが原因で、デコード用のキャッシュラインを供給できない状態のサイクル数の比率 (%) です。この比率が高い場合は、コードページのワーキングセットのサイズを縮小して、実行される命令の数を減らし、コードの局所性を改善することで、パフォーマンスが向上する可能性があります。

6. ITLB Miss Rate (ITLB ミス比率):
   ITLB_MISS_RETIRED / INST_RETIRED.ANY

ITLB Miss Rate の値が大きい場合は、実行されるコードのページ数が多すぎて、多くの命令 TLB ミスが発生していることを示します。ITLB ミスを引き起こした命令がリタイアすると、パイプラインは自然に排出されます。また ITLB ミスは、多くの命令のフェッチをストールさせます。

7. L1 Instruction Cache Miss Rate (L1 命令キャッシュミス比率):
   L1I_MISSES / INST_RETIRED.ANY

L1 Instruction Cache Miss Rate の値が大きい場合、コードのワーキングセットが L1 命令キャッシュの容量を超えていることを示します。コードのワーキングセットを縮小すると、パフォーマンスは向上します。

8. L2 Instruction Cache Line Miss Rate (L2 命令キャッシュライン・ミス比率): L2_IFETCH.SELF.I_STATE / INST_RETIRED.ANY

L2 Instruction Cache Line Miss Rate が 0 より大きい場合は、L2 キャッシュからの命令キャッシュライン・ミスがプログラムのパフォーマンスに顕著な影響を与えていることを示します。

B.8.2.2 分岐とフロントエンド

9. BACLEAR Performance Impact (BACLEAR のパフォーマンスへの影響):
   7 * BACLEARS / CPU_CLK_UNHALTED.CORE

BACLEAR Performance Impact の値が大きい場合は、通常はコード内の分岐が多すぎて分岐予測ユニットが分岐を処理できていないことを示します。

10. Taken Branch Bubble (分岐した分岐のバブル):
    (BR_TKN_BUBBLE_1+BR_TKN_BUBBLE_2) / CPU_CLK_UNHALTED.CORE

Taken Branch Bubble の値が大きい場合は、連続的に分岐する分岐がコードに多数含まれ、フロントエンドにバブルを発生させていることを示します。このバブルは、後続のパイプ内の実行レイテンシーとストールによって隠蔽されない場合にのみ、パフォーマンスに影響します。

B.8.2.3 スタックポインター追跡

11. ESP Synchronization (ESP 同期):
    ESP.SYNCH / ESP.ADDITIONS

ESP Synchronization 比率は、ESP の明示的な使用 (例えば、ロード命令またはストア命令による使用) と暗黙的な使用 (例えば、PUSH 命令または POP 命令による使用) の比率を計算します。期待される比率は 0.2 以下です。この比率が 0.2 より高い場合は、コードを再編成して ESP 同期イベントを回避することを検討すべきです。

B.8.2.4 マクロフュージョン

12. Macro-Fusion (マクロフュージョン):
    UOPS_RETIRED.MACRO_FUSION / INST_RETIRED.ANY

Macro-Fusion 比率は、1 つのマイクロオペレーション (uop) に何個のリタイアした命令が結合されているかを計算します。この比率は 32 ビット・バイナリー実行ファイルでは高くなり、それと等価な 64 ビット・バイナリーではかなり低い値を示します。また、64 ビット・バイナリーの実行速度は 32 ビット・バイナリーより低速です。その理由はおそらく、32 ビット・バイナリーがマクロフュージョンから大きなメリットを得ていると考えられます。

B.8.2.5 レングス変更プリフィクス (LCP) のストール

13. LCP Delays Detected (検出された LCP 遅延):
    ILD_STALL / CPU_CLK_UNHALTED.CORE

LCP Delays Detected の値が大きい場合は、測定したコード内で多くのレングス変更プリフィクス (LCP) 遅延が発生しています。

B.8.2.6 自己修正コードの検出

14. Self Modifying Code Clear Performance Impact (自己修正コードのクリアのパフォーマンスへの影響):
    MACHINE_NUKES.SMC * 150 / CPU_CLK_UNHALTED.CORE * 100

コードセクションへの書き込みを行い、すぐにそのコードを実行するプログラムでは、大きなペナルティーが生じます。Self Modifying Code Performance Impact は、プログラムが自己修正コードのペナルティーとして要したサイクル数の比率 (%) を推測します。

B.8.3 分岐予測比率

付録 B.8.2.2 では、フロントエンドのパフォーマンスに影響を与える分岐について説明しました。この節では、分岐予測ミスの特性評価によく使用されるイベント比率について説明します。

B.8.3.1 分岐予測ミス

15. Branch Misprediction Performance Impact (分岐予測ミスのパフォーマンスへの影響):
    RESOURCE_STALLS.BR_MISS_CLEAR / CPU_CLK_UNHALTED.CORE * 100

Branch Misprediction Performance Impact を使用して、プロセッサーが分岐予測ミスからの回復に要したサイクル数の比率 (%) を計算できます。

16. Branch Misprediction per Micro-Op Retired (リタイアしたマイクロオペレーション (uop) あたりの分岐予測ミス):
    BR_INST_RETIRED.MISPRED/UOPS_RETIRED.ANY

Branch Misprediction per Micro-Op Retired 比率は、分岐の予測ミスが多すぎてコードのパフォーマンスが低下していないかを示します。低下している場合、分岐予測の精度を向上させることで、コードのパフォーマンスが著しく改善できる可能性があります。

また、個々の分岐予測ミスがパフォーマンスに大きな影響を及ぼしている場合もあります。この状況は、予測が外れた分岐に先行するコードの CPI が高く (キャッシュミスなど)、分岐予測ミスによって後続のコードがその処理を並列処理できない場合に発生します。このコードの CPI を小さくすれば、分岐予測ミスによるパフォーマンスへの影響を軽減できます。これらの状況を識別する方法については、ほかの比率を参照してください。

プリサイスイベントである BR_INST_RETIRED.MISPRED を使用して、予測が外れた分岐の実際のターゲットを検出できます。これにより、予測が外れた分岐を識別できます。

B.8.3.2 仮想テーブルと間接呼び出し

17. Virtual Table Usage (仮想テーブルの利用):
    BR_IND_CALL_EXEC / INST_RETIRED.ANY

Virtual Table Usage の値が大きい場合は、コードに多くの間接呼び出しが含まれていることを示します。間接呼び出しのデスティネーション・アドレスの予測は困難です。

18. Virtual Table Misuse (仮想テーブルの誤使用):
    BR_CALL_MISSP_EXEC / BR_INST_RETIRED.MISPRED

Branch Misprediction Performance Impact (比率 15) と Virtual Table Misuse の比率がともに高い場合は、間接関数呼び出しの予測ミスのために長い時間がかかっていることを示します。

C コード内での関数ポインターの明示的な使用以外に、間接呼び出しは C++ の継承、抽象型クラス、仮想メソッドの実装に使用されます。

B.8.3.3 リターン予測ミス

19. Mispredicted Return Instruction Rate (予測ミスしたリターン命令の比率):
    BR_RET_MISSP_EXEC/BR_RET_EXEC

プロセッサーはコールとリターンのペアを追跡する特殊なメカニズムを備えています。プロセッサーは、すべての CALL 命令がそれに対応する RETURN 命令を持つことを前提としています。RETURN 命令によって復帰したリターンアドレスが、それに対応する CALL 命令で格納されたアドレスと一致しない場合は、予測ミスのペナルティーが生じます。

B.8.4 実行比率

この節では、マイクロオペレーション (uop) と RS、ROB、実行ユニットなどの相互作用を解析するイベント比率について説明します。

B.8.4.1 リソースストール

RS Full Ratio (比率 2) の値が大きい場合は、依存関係チェーンが長いため、リザベーション・ステーション (RS) が頻繁にマイクロオペレーション (uop) で一杯になっていることを示します。RS 内のマイクロオペレーション (uop) は、先行するマイクロオペレーション (uop) によってオペランドが計算されるのを待機しているか、実行ユニットが空くのを待機しています。これにより、複数実行ユニットを使用した並列処理が妨げられます。

ROB Full Ratio (比率 3) の値が大きい場合、リオーダーバッファー (ROB) が頻繁にマイクロオペレーション (uop) で一杯になっていることを示します。これは通常、L2 キャッシュ要求ミスなどレイテンシーの長い操作を意味します。

B.8.4.2 ROB 読み出しポートのストール

20. ROB Read Port Stall Rate (ROB 読み出しポートストールの比率):
    RAT_STALLS.ROB_READ_PORT / CPU_CLK_UNHALTED.CORE

ROB Read Port Stall Rate は、ROB 読み出しポートストールを識別します。ただし、この比率は Resource Stall Ratio で示されるリソースのストールの回数が少ない場合にのみ参照するべきです。

B.8.4.3 パーシャル・レジスター・ストール

21. Partial Register Stalls Ratio (パーシャル・レジスター・ストールの比率):
    RAT_STALLS.PARTIAL_CYCLES / CPU_CLK_UNHALTED.CORE*100

パーシャルストールの原因となるレジスターに頻繁にアクセスすると、アクセスレイテンシーが増加し、パフォーマンスが低下します。Partial Register Stalls Ratio は、パーシャルストールが発生しているサイクル数の比率 (%) です。

B.8.4.4 パーシャル・フラグ・ストール

22. パーシャル・フラグ・ストール比率 (パーシャル・フラグ・ストールの比率):
    RAT_STALLS.FLAGS / CPU_CLK_UNHALTED.CORE

パーシャル・フラグ・ストールによって大きなペナルティーが生じますが、このストールは簡単に回避できます。ただし、状況によっては、実際にフラグストールが発生していないにもかかわらず、Partial Flag Stalls Ratio の値が大きくなることがあります。一部の命令は、RFLAGS レジスターを部分的に変更するため、パーシャル・フラグ・ストールを発生させる場合があります。最も典型的な例は、シフト命令 (SAR、SAL、SHR、SHL) と、INC および DEC 命令です。

B.8.4.5 実行ドメイン間のバイパス

23. Delayed Bypass to FP Operation Rate (遅延した FP 演算バイパスの比率):
    DELAYED_BYPASS.FP / CPU_CLK_UNHALTED.CORE
24. Delayed Bypass to SIMD Operation Rate (遅延した SIMD 演算バイパスの比率):
    DELAYED_BYPASS.SIMD / CPU_CLK_UNHALTED.CORE
25. Delayed Bypass to Load Operation Rate (遅延したロード操作バイパスの比率):
    DELAYED_BYPASS.LOAD / CPU_CLK_UNHALTED.CORE

ドメインバイパスがあると、命令レイテンシーが 1 サイクル増えます。コード内でドメインバイパスが頻繁に起こる操作を特定するには、上記の比率を使用します。

B.8.4.6 浮動小数点演算パフォーマンス比率

26. Floating-Point Instructions Ratio (浮動小数点命令比率):
    X87_OPS_RETIRED.ANY / INST_RETIRED.ANY * 100

浮動小数点演算の比率が高い場合は、浮動小数点演算アルゴリズム固有の最適化を適用します。

27. FP Assist Performance Impact (FP アシストのパフォーマンスへの影響):
    FP_ASSIST * 80 / CPU_CLK_UNHALTED.CORE * 100

浮動小数点演算アシストは、デノーマルや NaN などの非正規の FP 値に対して起動されます。FP アシストは、正規の FP 演算の実行に比べて極めて低速です。生じるペナルティーはアシストによって異なります。FP Assist Performance Impact は、全体的な影響を推測します。

28. Divider Busy (除算器ビジー):
    IDLE_DURING_DIV / CPU_CLK_UNHALTED.CORE * 100

Divider Busy の値が大きい場合は、除算器がビジーであり、ほかの実行ユニットやロード操作は数サイクルにわたって進行していないことを意味します。この比率は、並行して実行でき、除算器のペナルティーを隠蔽できる L1 データ・キャッシュ・ミスと L2 キャッシュミスを考慮に入れていません。

29. Floating-Point Control Word Stall Ratio (浮動小数点制御ワードストールの比率):
    ESOURCE_STALLS.FPCW / CPU_CLK_UNHALTED.CORE * 100

浮動小数点制御ワード (FPCW) が頻繁に変更されると、パフォーマンスが大幅に低下します。FPCW 変更の主な理由は、FP から整数への変換時の丸めモードの変更です。

B.8.5 メモリー・サブシステム - アクセス競合の比率

Load or Store Buffer Full Ratio (比率 4) の値が大きいと、ロードバッファーまたはストアバッファーが頻繁に一杯になるため、新しいマイクロオペレーション (uop) を実行パイプラインに送り込めなくなります。これによって実行の並列性が減少し、パフォーマンスが低下します。

30. Load Rate (ロード比率):
    L1D_CACHE_LD.MESI / CPU_CLK_UNHALTED.CORE

各コアは 1 サイクルごとに 1 つのメモリー読み出し操作を処理できます。Load Rate の値が大きい場合は、実行がメモリー読み出し操作によって制約を受けていることを示します。

31. Store Order Block (ストア・オーダー・ブロック):
    STORE_BLOCK.ORDER / CPU_CLK_UNHALTED.CORE * 100

Store Order Block 比率は、L2 キャッシュをミスしたストア操作によって、その後のストア操作によるメモリー・サブシステムへのデータ移動がブロックされたサイクル数の比率 (%) です。さらに、これによってストアバッファーが一杯になることがあります (比率 4 を参照)。

B.8.5.1 L1 データキャッシュによってブロックされたロード

32. Loads Blocked by L1 Data Cache Rate (L1 データキャッシュによってブロックされたロードの比率):
    LOAD_BLOCK.L1D/CPU_CLK_UNHALTED.CORE

Loads Blocked by L1 Data Cache Rate の値が大きい場合、多数の L1 データ・キャッシュ・ミスが同時に発生することで、リソースの不足のためロード操作が L1 データキャッシュによってブロックされていることを示します。

B.8.5.2 4K エイリアシング/ストア・フォワーディング・ブロックの検出

33. Loads Blocked by Overlapping Store Rate (オーバーラップしているストアによってブロックされたロードの比率):
    LOAD_BLOCK.OVERLAP_STORE/CPU_CLK_UNHALTED.CORE

4K エイリアシングとストア・フォワーディング・ブロックは、ロードがさまざまな理由で先行するストアによってブロックされる状況を表します。いずれの場合も、同じイベント、LOAD_BLOCK.OVERLAP_STORE で検出できます。Loads Blocked by Overlapping Store Rate の値が大きい場合、4K エイリアシングやストア・フォワーディング・ブロックがパフォーマンスに影響を与える可能性があります。

B.8.5.3 先行するストアによってブロックされたロード

34. Loads Blocked by Unknown Store Address Rate (未知のストアアドレスによってブロックされたロードの比率):
    LOAD_BLOCK.STA / CPU_CLK_UNHALTED.CORE

Loads Blocked by Unknown Store Address Rate の値が大きい場合は、未知のアドレスを持つ先行するストアによってロードが頻繁にブロックされるため、パフォーマンスのペナルティーが生じていることを示します。

35. Loads Blocked by Unknown Store Data Rate (未知のストアデータによってブロックされたロードの比率):
    LOAD_BLOCK.STD / CPU_CLK_UNHALTED.CORE

Loads Blocked by Unknown Store Data Rate の値が大きい場合、未知のデータを持つ先行するストアによってロードが頻繁にブロックされるため、パフォーマンスのペナルティーが生じていることを示します。

B.8.5.4 メモリー・ディスアンビゲーション

インテル® Core™ マイクロアーキテクチャーのメモリー・ディスアンビゲーション機能により、先行する未知のアドレスを持つストアがブロックするロードを投機的に実行できるようになります。(該当しないロード/ストアのディスアンビゲーションにより) この予測が失敗した場合は、イベント LOAD_BLOCK.STA と MEMORY_DISAMBIGUATION.RESET がカウントされます。

B.8.5.5 ロード操作のアドレス変換

36. L0 DTLB Miss due to Loads - Performance Impact (ロードによる L0 DTLB ミス - パフォーマンスへの影響):
    DTLB_MISSES.L0_MISS_LD * 2 / CPU_CLK_UNHALTED.CORE

DTLB0 ミスの数が多い場合、ワークロードが使用するデータセットが多数のページにわたり DTLB0 の容量を超えていることを示します。高いミスの数は、CPI (比率 1) が低い (約 0.8) 場合にのみ、ワークロードのパフォーマンスに影響を与えると予想されます。CPI の値が大きい場合は、DTLB0 ミスサイクルはほかのレイテンシーによって隠蔽される可能性があります。

B.8.6 メモリー・サブシステム - キャッシュミスの比率

B.8.6.1 コード内のキャッシュミスの検出

インテル® Core™ マイクロアーキテクチャーは、L1 データ・キャッシュ・ミスまたは L2 キャッシュミスを引き起こしたリタイアしたロード命令を正確にカウントするイベントを備えています。これらはプリサイスイベントであり、イベントを発生させた命令に続く命令の命令ポインターを提供します。そのため、ポインターが指す命令の直前の命令が、キャッシュミスを発生させた命令となります。パフォーマンスの問題を解決する際に注目すべきロード命令の迅速な特定に役立つ以下のイベントがあります。

MEM_LOAD_RETIRE.L1D_MISS
MEM_LOAD_RETIRE.L1D_LINE_MISS
MEM_LOAD_RETIRE.L2_MISS
MEM_LOAD_RETIRE.L2_LINE_MISS

B.8.6.2 L1 データ・キャッシュ・ミス

37. L1 Data Cache Miss Rate (L1 データ・キャッシュ・ミス比率):
    L1D_REPL / INST_RETIRED.ANY

L1 Data Cache Miss Rate の値が大きい場合、コードの L1 データ・キャッシュ・ミスの頻度が高いため、L2 キャッシュアクセスのペナルティーが生じていることを示します。Loads Blocked by L1 Data Cache Rate (比率 32) を参照してください。

ロードが原因であるキャッシュミス、ストアが原因であるキャッシュミス、ロックされた操作が原因であるキャッシュミスは、それぞれイベント L1D_CACHE_LD.I_STATE、L1D_CACHE_ST.I_STATE、L1D_CACHE_LOCK.I_STATE によって別々にカウントできます。

B.8.6.3 L2 キャッシュミス

38. L2 Cache Miss Rate (L2 キャッシュミス比率):
    L2_LINES_IN.SELF.ANY / INST_RETIRED.ANY

L2 Cache Miss Rate の値が大きい場合、実行中のワークロード内のデータセットが L2 キャッシュの容量を超えているため、データの一部が使用される前に排出されていることを示します。ハードウェア・プリフェッチまたはソフトウェア・プリフェッチ命令によって、必要なデータがすべて事前に取り込まれていない限り、メモリーからデータが取り込まれるとパフォーマンスに大きな影響を与えます。

39. L2 Cache Demand Miss Rate (L2 キャッシュ要求ミス比率):
    L2_LINES_IN.SELF.DEMAND / INST_RETIRED.ANY

L2 Cache Demand Miss Rate の値が大きい場合、このワークロードが必要とするデータの取り込みにハードウェア・プリフェッチが利用されていません。データを使用するときに、その都度メモリーから取り込まれ、メモリーアクセスごとにメモリー・レイテンシーが発生しています。

B.8.7 メモリー・サブシステム - プリフェッチ

B.8.7.1 L1 データ・プリフェッチ

イベント L1D_PREFETCH.REQUESTS は、DCU が L2 (またはメモリー) から DCU にキャッシュラインをプリフェッチしようとするたびにカウントされます。DCU プリフェッチャーが動作しており、このイベントがカウントされることが予想されるにもかかわらず、イベント MEM_LOAD_RETIRE.L1D_MISS が検出された場合は、複数のロードによるロード命令アドレスの衝突が IP プリフェッチャーの動作を妨げている可能性があります。

B.8.7.2 L2 ハードウェア・プリフェッチ

イベント L2_LD.SELF.PREFETCH.MESI を使用して、L2 ハードウェア・プリフェッチャーが L2 に発行したプリフェッチ要求の数をカウントできます。L2 にプリフェッチされたキャッシュラインの数は、イベント L2_LD.SELF.PREFETCH.I_STATE によってカウントできます。

B.8.7.3 ソフトウェア・プリフェッチ

ソフトウェア・プリフェッチに関するイベントは、それぞれのプリフェッチ・レベルを対象とします。

40. Useful PrefetchT0 Ratio (有効な PrefetchT0 の比率):
    SSE_PRE_MISS.L1 / SSE_PRE_EXEC.L1 * 100
41. Useful PrefetchT1 and PrefetchT2 Ratio (有効な PrefetchT1 と PrefetchT2 の比率):
    SSE_PRE_MISS.L2 / SSE_PRE_EXEC.L2 * 100

いずれかの有効なプリフェッチの比率が低い場合は、インテル® SSE プリフェッチ命令の一部が、すでにキャッシュ内にあるデータをプリフェッチしています。

42. Late PrefetchT0 Ratio (遅れた PrefetchT0 の比率):
    LOAD_HIT_PRE / SSE_PRE_EXEC.L1
43. Late PrefetchT1 and PrefetchT2 Ratio (遅れた PrefetchT1 および PrefetchT2 の比率):
    LOAD_HIT_PRE / SSE_PRE_EXEC.L2

遅れたプリフェッチの比率が高い場合、ソフトウェア・プリフェッチ命令の発行が遅すぎたため、プリフェッチされるデータを使用するロード操作がキャッシュラインの到着を待機していることを示しています。

B.8.8 メモリー・サブシステム - TLB ミス比率

44. TLB miss penalty (TLB ミスのペナルティー):
    PAGE_WALKS.CYCLES / CPU_CLK_UNHALTED.CORE * 100

TLB miss penalty の比率が高い場合、TLB ミスの処理に多くのサイクルを要していることを示します。TLB ミスの数を減らすことで、パフォーマンスが向上します。この比率には、DTLB0 ミスのペナルティーは含まれません (比率 37 を参照)。

以下の比率は、頻繁に TLB ミスを発生するメモリーアクセスに焦点を絞るのに役立ちます。命令フェッチが原因である TLB ミスについては、「ITLB ミス比率」(比率 6) を参照してください。

45. DTLB Miss Rate (DTLB ミス比率):
    DTLB_MISSES.ANY / INST_RETIRED.ANY

DTLB Miss Rate の値が大きい場合、コードが短時間で多くのデータページをアクセスしているため、多数のデータ TLB ミスが発生していることを示します。

46. DTLB Miss Rate due to Loads (ロードによる DTLB ミスの比率):
    DTLB_MISSES.MISS_LD / INST_RETIRED.ANY

DTLB Miss Rate due to Loads の値が大きい場合、コードが短時間で多くのページからデータをロードするため、多数のデータ TLB ミスが発生していることを示します。ロード操作による DTLB ミスは、ロード操作のレイテンシーを増加させるため、パフォーマンスに大きな影響を与えます。この比率には、DTLB0 ミスのペナルティーは含まれません (比率 37 を参照)。

DTLB ミスを発生するロード命令を正確に検出するには、プリサイスイベント MEM_LOAD_RETIRE.DTLB_MISS を使用します。

47. DTLB Miss Rate due to Stores (ストアによる DTLB ミスの比率):
DTLB_MISSES.MISS_ST / INST_RETIRED.ANY

DTLB Miss Rate due to Stores の値が大きい場合、コードが短時間で多くのデータページをアクセスしているため、ストア操作による多数のデータ TLB ミスが発生していることを示します。これらのミスは、ほかの命令と並行して行われていない場合、パフォーマンスに影響します。また、多くのストア操作が連続している場合、その一部で DTLB ミスが発生すると、フル・ストア・バッファーによるストールが発生することがあります。

B.8.9 メモリー・サブシステム - コアとの相互作用

B.8.9.1 変更されたデータの共有

48. Modified Data Sharing Ratio (変更されたデータの共有の比率):
EXT_SNOOP.ALL_AGENTS.HITM / INST_RETIRED.ANY

変更されたデータの共有が頻繁に発生するのは、2 つのスレッドが同一キャッシュラインにあるデータを変更するためです。変更されたデータの共有は、L2 キャッシュミスの原因となります。変更されたデータの共有が意図せずに発生すると (フォルス・シェアリングとも呼ばれる)、通常は要求ミスが発生し大きなペナルティーが生じます。フォルス・シェアリングを排除すると、コードのパフォーマンスは飛躍的に向上します。

49. Local Modified Data Sharing Ratio (ローカルな変更されたデータの共有の比率):
EXT_SNOOP.THIS_AGENT.HITM / INST_RETIRED.ANY

Modified Data Sharing Ratio は、システム内で測定された変更されたデータの共有の総量を示します。複数のプロセッサーを搭載したシステムでは、Local Modified Data Sharing Ratio を使用して、同じプロセッサー内の 2 つのコアで共有される、変更されたデータの共有の量を示すことができます(1 つのプロセッサーを搭載するシステムでは、2 つの比率はほぼ同じになります)。

B.8.9.2 高速同期のペナルティー

50. Locked Operations Impact (ロックされた操作の影響):
(L1D_CACHE_LOCK_DURATION + 20 * L1D_CACHE_LOCK.MESI) / CPU_CLK_UNHALTED.CORE * 100

高速同期は、通常ロックされたメモリーアクセスを使用して実装されます。Locked Operations Impact の値が大きい場合、ワークロードで使用されるロックされた操作によって、大きなペナルティーが生じていることを示します。ロックされた操作のレイテンシーは、データの位置 (L1 データキャッシュ、L2 キャッシュ、その他のコアのキャッシュ、またはメモリー) によって異なります。

B.8.9.3 同時に多発するストアミスとロードミス

51. Store Block by Snoop Ratio (スヌープによるストアブロックの比率):
(STORE_BLOCK.SNOOP / CPU_CLK_UNHALTED.CORE) * 100

Store Block by Snoop Ratio の値が大きいと、ストア操作が頻繁にブロックされ、パフォーマンスが低下していることを示します。これは、プロセッサー内の 1 つのコアが連続するストアストリームを実行し、もう 1 つのコアが、L1 データキャッシュ内に見つからないキャッシュラインを検索してそのストリームを頻繁にスヌープする場合に発生します。

B.8.10
B.8.11　メモリー・サブシステム - バスの特性
B.8.11.1 バス利用率

52. Bus Utilization (バス利用率):
    BUS_TRANS_ANY.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS * 100

Bus Utilization は、任意のタイプのバス・トランザクションの転送に使用されるバスサイクル数の比率 (%) です。シングルプロセッサー・システムでは、バス・トランザクションの大部分はデータを転送します。マルチプロセッサー・システムでは、バス・トランザクションの一部は、キャッシュの状態を調整してデータのコヒーレンシーを維持するために使用されます。

53. Data Bus Utilization (データバス利用率):
    BUS_DRDY_CLOCKS.ALL_AGENTS / CPU_CLK_UNHALTED.BUS * 100

Data Bus Utilization は、プロセッサーとメモリーを含むシステム内のすべてのバス・エージェント間のデータ転送に使用されるバスサイクル数の比率 (%) です。バス利用率が高い場合、プロセッサーとメモリーの間に大量のトラフィックが発生しています。メモリー・サブシステムのレイテンシーは、プログラムのパフォーマンスに影響することがあります。計算集約型のアプリケーションでバス利用率が高い場合、データとコードの局所性を改善する可能性を探すべきです。ほかのタイプのアプリケーション (例えば、あるメモリー領域から別のメモリー領域へ大量のデータをコピーする) では、バス利用率を最大化します。

54. Bus Not Ready Ratio (バスの準備ができていない比率):
    BUS_BNR_DRV.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS * 100

Bus Not Ready Ratio は、新しいバス・トランザクションを開始できないバスサイクル数の比率 (%) を推測します。Bus Not Ready Ratio の値が大きい場合、バスに大きな負荷がかかっています。Bus Not Ready (BNR) 信号により、新しいバス・トランザクションが遅延し、そのレイテンシーがプログラムのパフォーマンスに大きな影響を与えることがあります。

55. Burst Read in Bus Utilization (バス利用率中のバースト読み出し):
    BUS_TRANS_BRD.SELF * 2 / CPU_CLK_UNHALTED.BUS * 100

Burst Read in Bus Utilization の値が大きい場合、バースト読み出し操作によるバスとメモリーのレイテンシーがプログラムのパフォーマンスに影響を与えていることを示しています。

56. RFO in Bus Utilization (バス利用率中の RFO):
    BUS_TRANS_RFO.SELF * 2 / CPU_CLK_UNHALTED.BUS * 100

RFO in Bus Utilization の値が大きい場合、所有権読み出し (RFO) トランザクションのレイテンシーがプログラムのパフォーマンスに影響を与えていることを示します。RFO トランザクションは、ほかのバースト読み出し操作 (例えば、ロードが L2 にミスした場合など) に比べて、プログラムのパフォーマンスに大きな影響を与えます。比率 31 も参照してください。

B.8.11.2 変更されたキャッシュラインの排出

57. L2 Modified Lines Eviction Rate (変更された L2 ラインの排出比率):
    L2_M_LINES_OUT.SELF.ANY / INST_RETIRED.ANY

新しいキャッシュラインがメモリーから取り込まれると、新しいラインを格納するため、既存のキャッシュライン (おそらく変更済み) が L2 キャッシュから排出される可能性があります。変更されたラインが L2 キャッシュから頻繁に排出されると、L2 キャッシュミスのレイテンシーが増加し、バス帯域幅を消費します。

58. Explicit WB in Bus Utilization (バス利用率中の明示的ライトバック):
    BUS_TRANS_WB.SELF * 2 / CPU_CLK_UNHALTED.BUS*100

Explicit Write-back in Bus Utilization は、変更されたキャッシュラインの L2 キャッシュからの排出だけでなく、L1 データキャッシュからの排出も考慮します。この比率は、プロセッサーからメモリーへの明示的ライトバックに使用されるバスサイクル数の比率 (%) を表します。

付録 C
ラージ・コード・ページを使用するインテル® アーキテクチャーの最適化

この付録では、ラージ・コード・ページを使用してランタイム・パフォーマンスを向上する方法を示すランタイム最適化の考察を提供します。

C.1 概要

最新のプロセッサーでは、プログラムコードの複数ページサイズがサポートされています。例えば、第 2 世代のサーバー・プラットフォーム向けのインテル® Xeon® Platinum 8280 プロセッサー (Cascadelake† マイクロアーキテクチャー) は、命令向けに 4KB、2MB、4MB ページ、データ向けに 4KB、2MB、4MB、1GB をサポートしています。インテル® プラットフォームは、インテル® Xeon® プロセッサー E5 製品ファミリー (Ivy Bridge† マイクロアーキテクチャー) で、2011 年以降 4KB と 2MB ページをサポートしていました。しかしながら、大部分のプログラムはデフォルトのページサイズである 4KB のみを使用していました。Linux* では、すべてのアプリケーションはデフォルトで 4KB メモリーページにロードされます。言語ラインタイムでワークロードのパフォーマンス・ボトルネックを調査したところ、ITLB ミスによる多数のストールが検出されました。これは、ラインタイムが命令に 4KB ページのみを使用していることが原因です。

図 C-1 は、各種ワークロードにおけるインテル® Xeon® Platinum 8180 プロセッサーの ITLB ミスに起因する CPU ストールを示しています。平均してサイクルの 7% が ITLB ミスでストールしています。SPECjbb2015*[35] などのベンチマークは、高い ITLB ストール (13%) を示す SPECjEnterprise*[36] に比べると、低い ITLB ストール (2.6%) を示します。

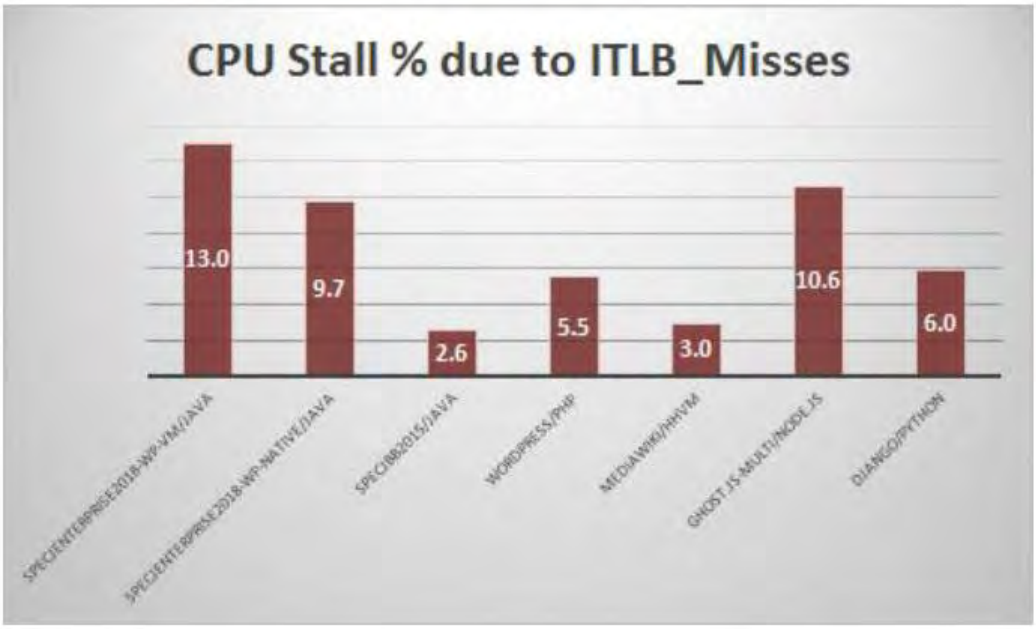


図 C-1 インテル® Xeon® Platinum 8180 プロセッサー上の言語ランタイムにおける ITLB ミスのストール

[35] SPECjbb2015 。(n.d.) 。SPECjbb2015 Design Document 。SPEC - Standard Performance Evaluation Corporation から取得: https://www.spec.org/jbb2015/docs/designdocument.pdf (英語)

[36] SPECjEnterprise 。(n.d.) 。SPECjEnterpise 2018 Web Profile 。SPEC - Standard Performance Evaluation Corporation から取得: https://www.spec.org/jEnterprise2018web/ (英語)

C.1.1 ITLB とストール

インテル® プロセッサーには、トランスレーション・ルックアサイド・バッファー (TLB) が備わっており、最近使用されたページ・ディレクトリーとページ・テーブル・エントリーが格納されます。TLB は、システムメモリー上のページテーブルを読み取るメモリーアクセス数を減らすことで、ページングが有効な場合のメモリーアクセスを高速化します。

TLB は次のグループに分類されます:

- 4KB ページの命令 TLB。
- 4KB ページのデータ TLB。
- ラージページ (2MB、4MB ページ) の命令 TLB。
- ラージページ (2MB、4MB、または 1GB ページ) のデータ TLB。

インテル® Xeon® Platinum 8180 プロセッサー (Skylake† Server マイクロアーキテクチャー) では、プロセッサー TLB は命令キャッシュ (ITLB) 専用の L1 TLB を持ちます。さらに、次に示すように、データと命令で共有されるユニファイド L2 レベル TLB (STLB) があります。

TLB:

- ITLB
  - 4 KB ページ変換:
    - 128 エントリー、8 ウェイ・セット・アソシアチブ。
    - 動的なパーティショニング。
  - 2 MB/4 MB ページ変換:
    - スレッドごとに 8 つのエントリー、フル・アソシアチブ。
    - スレッドごとに複製されます。

- STLB
  - 4 KB + 2 MB ページ変換:
    - 1536 エントリー、12 ウェイ・セット・アソシアチブ、固定パーティション。

プロセッサーが ITLB でエントリーを検出できない場合、ページウォークを行いエントリーを設定する必要があります。L1 (第 1 レベルの) ITLB のミスは、通常はアウトオブオーダー (OOO) 実行によって隠蔽可能な非常に小さなペナルティーです。STLB でミスが発生するとページウォークが起動されますが、このペナルティーは実行時に現れます。この処理が行われる間、プロセッサーは停止します。次の表に、各世代のインテル製品の TLB サイズを示します。

表 C-1 複数世代のインテル製品におけるコア TLB 構造のサイズと構成

TLB	**Sandy Bridge†/Ivy Bridge† マイクロアーキテクチャー**	**Haswell†/Broadwell† マイクロアーキテクチャー**	**Skylake†/ Cascade Lake† マイクロアーキテクチャー**
L1 命令 TLB	4K – 128、4 ウェイ 2M / 4M – 8/スレッド	4K – 128、4 ウェイ 2M / 4M – 8/スレッド	4K – 128、8 ウェイ 2M / 4M – 8/スレッド
L1 データ TLB	4K – 64、4 ウェイ 2M / 4M – 32 – 4 ウェイ 1G: 4、4 ウェイ	4K – 64、4 ウェイ 2M / 4M – 32 – 4 ウェイ 1G: 4、4 ウェイ	4K – 64、4 ウェイ 2M / 4M – 32 – 4 ウェイ 1G: 4、4 ウェイ
L2 (ユニファイド) STLB	4K – 512、4 ウェイ	4K + 2M 共有: Haswell†: 1024、8 ウェイ Broadwell†: 1536、6 ウェイ 1G: 16、4 ウェイ	4K + 2M 共有: 1536、12 ウェイ 1G: 16、4 ウェイ

表 C-1 から、Haswell† マイクロアーキテクチャー以降のユニファイド L2 TLB で 2M ページエントリーが共有されていることが分かります。

C.1.2 ラージページ

Windows* と Linux* のどちらでも、サーバー・アプリケーションはラージページのメモリー領域を確保できます。2MB のラージページを使用すると、20MB のメモリー領域をわずか 10 ページでマップできます。4KB ページを使用すると 5120 ページ必要になります。つまり、必要な TLB エントリー数が少なくなり、TLB ミスの回数が減ります。ラージページは、コードまたはデータ、あるいはその両方に使用できます。ワークロードに大きなヒープがある場合、データ向けにラージページを試すことを推奨します。ここで示す考察は、コードにラージページを使用することに注目しています。

C.2 問題の診断

C.2.1 ITLB ミス

インテルは、新しいパフォーマンス・イベントに基づく階層的な実行サイクルの区分を提案するトップダウン・マイクロアーキテクチャー解析手法 (TMAM) (付録 B「パフォーマンス監視イベント」を参照) を定義しました。TMAM は、すべての命令発行スロットを個別に調査するため、正確なストッロレベルの区分を可能にします。

フロントエンド・レイテンシーの 1 つは ITLB ミスストールであると考えられます。このメトリックは、命令 TLB ミスが原因でプロセッサーがストールしたサイクルの割合を表します。インテル® Xeon® スケーラブル・プロセッサー・ファミリー (Skylake† Server マイクロアーキテクチャー) では、ITLB ミスストールは式 1 により、2 つの PMU カウンター (ICACHE_64B.IFTAG_STALL と CPU_CLK_UNHALTED.THREAD) から求められます。

式 1: ITLB ストールメトリックの計算

$$ITLB\_Miss_{stall} = 100 * \left(\frac{ICACHE\_64B.IFTAG\_STALL}{CPU\_CLK\_UNHALTED.THREAD}\right)$$

具体的な例を見てみましょう。システム全体をクラスターモードで実行すると、Ghost.js ワークロードの ITLB_miss ストールは 10.6% になります。これら 2 つのカウンターのサンプリング値を式に適用することで、ITLB_miss ストールの % を特定できます。ITLB ミスによる 10.6% のストールは、このワークロードでは重要です。

表 C-2 Ghost.js 向けの ITLB ミスストールの計算

CPU_CLK_UNHALTED.THREAD	69838983
ICACHE_64B.IFTAG_STALL	7412534
ITLB ミスストール (%)	10.6

ランタイム・ワークロードに ITLB パフォーマンスの問題があるかどうかを判断するには、ITLB ミスストールを測定することが重要です。

C.6 節「ケーススタディー」では、シングル・インスタンス・モードで実行する場合でも、Ghost.js が ITLB ミスにより 6.47% ストールしていることが分かります。ラージページを使用すると、パフォーマンスは 5% 向上し、ITLB ミスが 30% 軽減され、ITLB ミスストールは 6.47% から 2.87% に低下しました。

その他の重要なメトリックは、ITLB Misses Per Kilo Instructions (MPKI) (1000 命令あたりの ITLB ミス) です。このメトリックは命令数に対する ITLB ミスを正規化するものであり、異なるシステム間での比較に役立ちます。式 2 に示すように、ITLB_MISSES.WALK_COMPLETED と INST_RETIRED.ANY の 2 つの PMU カウンターから計算されます。ラージページと 4KB ページには異なる PMU カウンターがあり、式 2 はそれぞれの PMU カウンターの計算を示します。

式 2: ITLB MPKI の計算

$$ITLB\_MPKI = 1000 * (\frac{ITLB\_MISSES.WALK\_COMPLETED}{INST\_RETIRED.ANY})$$

$$ITLB\_4K\_MPKI = 1000 * (\frac{ITLB\_MISSES.WALK\_COMPLETED\_4K}{INST\_RETIRED.ANY})$$

$$ITLB\_2M\_4M\_MPKI = 1000 * (\frac{ITLB\_MISSES.WALK\_COMPLETED\_2M\_4M}{INST\_RETIRED.ANY})$$

図 C-2 のランタイム・ワークロードの MPKI を見ると、ITLB MPKI と ITLB 4K MPKI がワークロード全体で近接していることが分かります。つまり、ほとんどのミスは 4KB ページウォークに起因するものであると考えられます。別の観点では、ベンチマークの ITLB MPKI は実際の大規模なソフトウェアよりも低いことが考えられます。これは、ベンチマークにおける最適化が、オープンソースのソフトウェアなどには適用されていない可能性があるということです。

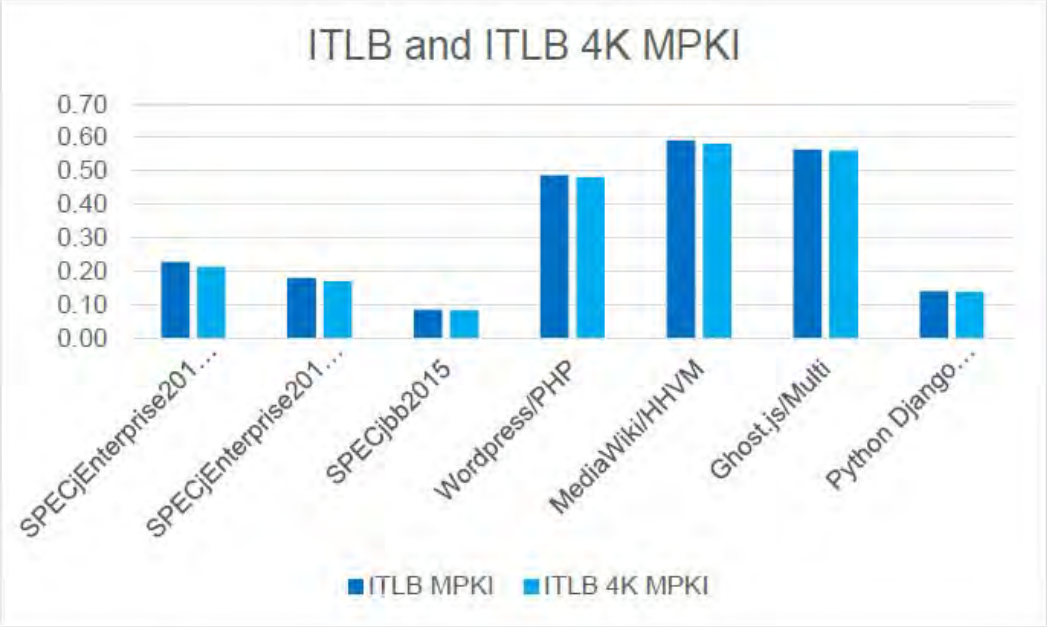


図 C-2 ランタイム・ワークロード全体の ITLB および ITLB 4K MPKI

ITLB MPKI を使用すると、さまざまなシステムやワークロードを比較できます。表 C-3 は、公開されている各種ワークロード[37,38] 全体の ITLB MPKI をまとめたものです。バイナリーサイズと ITLB MPKI には直接的な相関関係がないことが分かります。MySQL など一部の小規模なバイナリーには、大きな ITLB MPKI 値を示すものがあります。複数のスレッドがアクティブな場合、ITLB MPKI は Ghost.js (シングル・インスタンスとマルチインスタンス) と Clang (-j1 と -j4) の両方でほぼ倍になります。ITLB MPKI は、古いサーバー (インテル® Xeon® プロセッサー E5

[37] Ottoni G. & Bertrand M. (2017)。大規模データセンター向けアプリケーションの関数配置の最適化。CGO 2017。

[38] Lavaee R.、Criswell J. & Ding C. (2018 年 10 月)。コード・ステッチャー: プロシージャー間の基本ブロック配置最適化。arXiv:1810.00905v1 [cs.PL]。

製品ファミリーベース) と比べると、新しいサーバー (インテル® Xeon® Platinum 8180 プロセッサー・ベース) ではるかに低い値を示します。

表 C-3 各種ワークロードにおける ITLB MPKI と実行可能ファイルのサイズ

ワークロード	テキスト (MB)	ITLB MPKI	システムの詳細
AdIndexer	186	0.48	2 x インテル® Xeon® プロセッサーE5-2680 v2 (Ivy Bridge† マイクロアーキテクチャー・ベース) サーバー・プラットフォーム、2.80GHz、10 コア、プロセッサーごとに 25MB LLC)
HHVM	133	1.28	
Multifeed	199	0.40	
TAO	70	3.08	
MySQL	15	9.35	2 つのデュアルコア・インテル® Core™ i5-4278U プロセッサー (Haswell† マイクロアーキテクチャー・ベース)、2.60GHz、32KB 命令キャッシュと 256KB ユニファイド L2 キャッシュ (それぞれのキャッシュは、8 ウェイ・セット・アソシアチブ)、3MB ラストレベル・キャッシュ (12 ウェイ・セット・アソシアチブですべてのコアで共有)
Clang –j4	50	2.23	
Clang –j1	50	1.01	
Firefox	81	1.54	
Apache PHP (opcache あり)	16	0.33	
Apache PHP (opcache なし)	16	0.96	
Python	2	0.19	
SPECjEnterprise2018-WP-VM		0.23	インテル® Xeon® Platinum 8180 プロセッサー (Skylake† Server マイクロアーキテクチャー)、112 コア @ 2.50GHz (18 コアの SKX-D の MediaWiki/HHVM を除く)
SPECjEnterprise2018-WP-Native		0.18	
SPECjbb2015		0.09	
Wordpress/PHP		0.49	
MediaWiki/HHVM		0.59	
Ghost.js/Multi		0.56	
Ghost.js/Single		0.23	
Python Django (Instagram)		0.14	

C.2.2 ITLB ミスストールの測定

インテルには ITLB ミスストールの測定を自動化するインテル® VTune™ プロファイラー、EMON / EDP、Linux* PMU などいくつかのツールがあります。この考察では、perf ベースの便利なツール (measure-perf-metric.sh) を使用し、インテル® Xeon® スケーラブル・プロセッサーの各種ストールメトリックを収集して解析します。このツールはオープンソースであり、http://github.com/intel/iodlr (英語) からダウンロードできます。図 C-3 にプロセス ID = 69772 のアプリケーションの ITLB ミスストールを収集して解析するコマンドラインを示します。ツールの出力では、アプリケーションに 3.09% の ITLB ミスストールがあることを示しています。

```
$ git clone http://github.com/intel/iodlr
$ export PATH=`pwd`/iodlr/tools/:$PATH
$ measure-perf-metric.sh -p 69772 -t 30 -m itlb_stalls
Initializing for metric: itlb_stalls
Collect perf data for 30 seconds
perf stat -e cycles,instructions,icache_64b.iftag_stall
-------------------------------------------------
Profile application with process id: 69772
-------------------------------------------------
Calculating metric for: itlb_stalls

=================================================
Final itlb_stalls metric
-------------------------------------------------
FORMULA: metric_ITLB_Misses(%) = 100*(a/b)
         where, a=icache_64b.iftag_stall
                b=cycles
=================================================
metric_ITLB_Misses(%)=3.09
```

図 C-3 measure-perf-metric.sh でプロセス ID 69772 を 30 秒間計測

コマンド “measure-perf-metric.sh –h” を使用して、ツールの使い方を示すヘルプメッセージを表示します。ツールに新しいメトリックを追加する方法は、README.md ファイルを参照してください。

C.2.3 ITLB ミスの発生元

次の作業は ITLB ミスの発生元を見つけることです。それらは、ランタイムの .text セグメント、JIT 化されたコード、その他の動的ライブラリー、またはユーザーコード内のネイティブ・ライブラリーにある可能性があります。ITLB ミスの発生元を特定するには、perf などのパフォーマンス・ツールを利用する必要があります。

前述の Ghost.js の場合、ITLB ミスの大部分は Node.js[39] バイナリーの .text セグメントにあります。これは、Node.js の他のワークロードにもあてはまります。Node.js の現在のリリース (V12.8.0) と measure-perf-metric.sh ツールを使用して、Node.js ワークロードを調査します。図 C-4 は、ストールの 65.23% が node のバイナリーにあることを示しています。measure-perf-metric.sh の ‘-r’ オプションは、perf レコードを使用して、itlb_stalls の原因となるソースコードの場所を記録します。

```
$ measure-perf-metric.sh -p 58448 -r -t 20 -m itlb_stalls
Samples: 77K of event 'icache_64b.iftag_stall', Event count (approx.): 1558
Overhead  Shared Object        Command
  65.23%  node                 node
  20.69%  perf-56817.map       node
   4.53%  libc-2.27.so         node
```

図 C-4 measure-perf-metric.sh に -r オプションを指定して TLB ミスの発生元を調査

図 C-4 は、ストールサイクルの観点から TLB ミスのオーバーヘッドが発生している場所を示しますが、最新の Node.js* (V14.0.0-pre) を解析し “perf record -e frontend_retired.tlb_miss” コマンドを使用して ITLB ミスカウントのオーバーヘッドを検出します。perf スクリプトを使用してレポートを抽出し、ITLB ミスが発生するアドレスを基にフィルター処理します。ITLB ミスの 17.6% は JIT されたコードによるもので、72.8% は node バイナリーによるものであることが分かります。また、node バイナリーの一部であるビルトイン関数が ITLB ミスの合計 19.5% を占めていることも分かります。

[39] Node.js Foundation (2019 年 8 月)。Node.js* JavaScript* Runtime。Node.js JavaScript* Runtime の入手先: https://nodejs.org/en (英語)

Linux* と Windows* システムでは、アプリケーションはメモリーの 4KB ページにロードされます。これはほとんどのシステムのデフォルトです。ITLB ミスを減らす方法の 1 つとして、より大きなページサイズを使用することが考えられます。これには 2 つの利点があります。最初の利点は、トランスレーションが少なくなるためページウォークが少なくなることです。2 つ目の利点は、トランスレーションの結果を保存するキャッシュスペースが少なくなり、アプリケーション・コードで使用できる空間が増えることです。インテル® Xeon® プロセッサー E5-2680 v2 (Ivy Bridge† マイクロアーキテクチャー・ベース) などを搭載する、一部の古いシステムには単一レベルで構成される 8 つのヒュージ ITLB エントリーしかないため、すべてのテキストをラージページにマッピングすると回帰が発生する可能性があります。しかし、インテル® Xeon® Platinum 8180 プロセッサー (Skylake† Server マイクロアーキテクチャー・ベース) では、STLB は 4KB と 2MB の両方のページで共有される 1536 のエントリーがあります。

C.3 解決方法

C.3.1 Linux* とラージページ

Linux*では、アプリケーションでラージページを使用する 2 つの方法があります。

- **明示的なヒュージページ (hugetlbfs)。**システムメモリーの一部は、アプリケーションが mmap できるファイルシステムとして現われます。cat /proc/meminfo コマンドを使用してシステムをチェックし、HugePages_Total などの行の存在を確認できます。
- **透過なヒュージページ (THP)。**Linux* は、ラージページを自動的に管理し、アプリケーションに対して透過なヒュージページも提供します。アプリケーションは、madvise を介してラージページ・バックアップ・メモリーを使用するよう Linux* に指示できます。cat /sys/kernel/mm/transparent_hugepage/enabled コマンドでシステムをチェックできます。値が always または madvise である場合、アプリケーションは THP を利用できます。madvise では、THP は MADV_HUGEPAGE 領域内部でのみ有効になります。図 C-5 は、THP の分布を確認する方法を示します。

```
% cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
% cat /sys/kernel/mm/transparent_hugepage/defrag
always defer defer+madvise [madvise] never
```

図 C-5. THP の Linux* ディストリビューションをチェックするコマンド

C.3.2 .text のラージページ

Linux* では、.text セグメントの ITLB ミスの問題を解決するいくつかの方法があります。

- **ランタイム libhugetlbfs とリンクする:** libhugetlbfs (英語) には多数のサポート・ユーティリティーとライブラリーがパッケージ化されています。ライブラリーは、ヒュージページとテキスト、データ、ヒープ、および共有メモリーセグメントを自動的にバッキングする機能を提供します。これは、システム管理者が hugeadm などのツールによって管理する明示的なヒュージページに依存します。
- **インテルのリファレンス実装を使用する:** リファレンス実装には、透過なヒュージページを使用してプロセスを自動化する C および C++ モジュールがあります。以下で説明するいくつかの API 呼び出しは、アプリケーションの .text セグメントのサブセットを 2MB ページにマップするため、ランタイムの最初に呼び出します。
- **ランタイムで明示的なオプションやフラグを使用する:** Node.js* ランタイムには、Node.js* の実行時に --enable-largepages=on を指定して使用できる実装があります。PHP ランタイムには、.ini ファイルに追加できるフラグがあります。詳細は、https://www.php.net/manual/en/opcache.configuration.php (英語) を参照してください。

C.3.3 リファレンス・コード

ここでは、アプリケーションの実行時にラージページを利用できるようにするリファレンス実装を示します。オープンソースのリファレンス実装は、http://github.com/intel/iodlr (英語) からダウンロードできます。C と C++ 向けの実装があります。

以下にリファレンス実装と API の概要を示します。

1. メモリーの .text 領域を探します。
    a. /proc/self/maps を調査して、マップされている .text 領域を特定し、開始アドレスと終了アドレスを取得します。
    b. .text セグメントの先頭を指すように開始アドレスを変更します。
    c. 開始アドレスと終了アドレスをラージページの境界に合わせます。
2. .text 領域をラージページに移動します。
    a. 一時領域をマッピングし、そこに元のコードをコピーします。
    b. MAP_FIXED の開始アドレスを使用して mmap でその仮想アドレスを取得します。
    c. madvise で MADV_HUGE_PAGE を指定して匿名の 2MB ページを使用します。
    d. 成功したら、一時領域からコードをコピーしてマップを解除します。

リファレンス実装には、図 C-6 で示すように 5 つの API 呼び出しがあります。最初のリリース以降では、DSO をマップする機能が追加されました。

```
/* Performs a platorm-dependent check to determine whether it is possible to map
   to large pages and stores the result of the check in result. */
map_status IsLargePagesEnabled(bool* result);

/* Attempts to map an application's .text region to large pages.
   If the region is not aligned to 2 MiB then the portion of the page that lies below
   the first multiple of 2 MiB remains mapped to small pages. Likewhise, if the region
   does not end at an address that is a multiple of 2 MiB, the remainder of the region
   will remain mapped to small pages. The portion in-between will be mapped to large
   pages. */
map_status MapStaticCodeToLargePages();

/* Retrieves an address range from the process' maps file associated with a DSO
   whose name matches lib_regex and attempts to map it to large pages */
map_status MapDSOToLargePages(const char* lib_regex);

/* Attempts to map the given address range to large pages. */
map_status MapStaticCodeRangeToLargePages(void* from, void* to);

/* A string containing the textual error message. The string is owned by the
   implementation and must not be freed. */
const char* MapStatusStr(map_status status, bool fulltext);
```

図 C-6 インテルのリファレンス実装で提供される API 呼び出し

C.3.4 ヒープ向けのラージページ

ジャストインタイム (JIT) コンパイラーはオンデマンドでメソッドをコンパイルし、JIT されたコードのメモリーはヒープから割り当てられるため、ガベージ・コレクションの対象になります。

mmap とフラグ引数 MAP_HUGETLB (Linux* 2.6.32 以降で利用可能) または MAP_HUGE_2MB/MAP_HUGE_1GB (Linux* 3.8 以降で利用可能) を使用して、ランタイムがラージページにヒープを割り当てるようにできます。または、madvise システムコールと MADV_HUGE_PAGES を使用して、Linux* で透過なヒュージページを使用するようにヒープ領域を設定できます。madvise を使用する場合、ランタイムは OS で transparent_hugepage が madvise または always に設定されており、never に設定されていないことを確認する必要があります。

Java* VM には、Java* ヒープをラージページ[40] にマッピングするいくつかのオプションがあります。JIT されたコードもヒープ上にあるため、コードとデータの両方をラージページに割り当てます。

-XX:+UseHugeTLBFS は、Java ヒープを hugetlbfs に mmap します。これは個別に準備する必要があります。
-XX:+UseTransparentHugePages madvise-s は、Java* ヒープが THP を使用する必要があることを示します。

C.4 解決策の統合

解決策を新しいランタイムに統合するには次の変更が必要です。

1. ランタイムのスタイルガイドに従ってリファレンス・コードを更新します。
2. ランタイムで .text セグメントを再マップするため API を呼び出す場所を決定します。
3. 新しいファイル/ライブラリーとリンクするようビルド環境を変更します。
4. この機能を有効にするには、ビルドまたはランタイムオプションを使用します。

C.4.1 リファレンス実装と V8 の統合

V8[41] は、C++ で記述された Google* オープンソースのハイパフォーマンス JavaScript* エンジンです。上記の手順に従って、インテルのラージページ・リファレンス実装を V8 に統合しました。

以下は、V8 にリファレンス実装を統合するのに使用した具体的な手順です。

1. https://v8.dev/docs/build-gn (英語) から V8 をチェックアウトし、設定およびビルドを行います。
2. d8.cc の Shell::Main() の先頭に MapStaticCodeToLargePages() 呼び出しを追加します。ソースファイルで huge_page.h をインクルードします。
3. 次のコマンドでビルドファイルを生成します。
   gn gen out/foo –args='is_debug=false target_cpu="x64" is_clang=false'
4. 次のビルドファイルを更新します:

   a. out/foo/obj/d8.ninja を更新します
      include_dirs 変数に –Ipath/to/huge_page.h を追加します
      ldflags 変数に -Wl,-T path/to/ld.implicit.script を追加します
   b. out/foo/toolchain.ninja を更新します
      リンクコマンドの –Wl,–endgroup の前に、path/to/libhuge_page.a –lstdc++ を追加します

5. 次のコマンドで V8 をコンパイルします。
   ninja -C out/bar/ d8

[40] Aleksey Shipilev、Redhat (2019、03 03)。Transparent Huge Pages。JVM Anatomy Quarks から取得: https://shipilev.net/jvm/anatomy-quarks/2-transparent-huge-pages/ (英語)。
[41] Google* V8 JavaScript (2019 年 8 月)。V8 JavaScript* Engine。V8 JavaScript* Engine から取得: https://v8.dev/ (英語)。

C.4.2 リファレンス実装と JAVA* JVM の統合

OpenJDK* は、C と C++ で記述された Java* Platform Standard Edition の無料のオープンソース実装です。V8 や Node.js* とは異なり、Java* 実行形式ファイルは dlopen を使用して libjvm をロードする軽量の C ラッパーであると考えられます。

インテルの C ラージページ・リファレンス実装を OpenJDK* と統合しました。リファレンス実装を統合する手順を以下に示します。

1. http://cr.openjdk.java.net/~ihse/demo-new-build-readme/common/doc/building.html（英語）の手順に従って、OpenJDK* をチェックアウトし、設定およびビルドします。
2. src/java.base/unix/native/libjli/java_md_solinux.c を変更して libjvm.so を 2M ページにロードします。
    - API を使用して LargePages がサポートされているか確認します。
    - API MapDSOToLargePages(const char* lib_regex) を使用して、libjvm.so を 2M ページにロードします。

3. Java* ラッパーをコンパイルして再ビルドします。

C.5 制限事項

ラージページに使用に際して注意すべき制限事項があります。

- フラグメンテーション（断片化）は、ラージページを使用する際に発生する問題です。ラージページを配置するのに十分な連続したメモリーがない場合、オペレーティング・システムはラージページを配置できるようにメモリーを再編成しようとします。これにより、レイテンシーが増加する可能性があります。これは、事前にラージページを明示的に割り当てておくことで軽減できますが、リファレンス・コードでは明示的なラージページをサポートしていません。
- もう一つの問題は、インテルのリファレンス・コードでアルゴリズムを実行するのに追加される時間です。実行時間が短いプログラムでは、実行時間が長くなりスローダウンする可能性があります。
- 同じアプリケーションを複数のインスタンスで実行すると、LLC ミスが増加するという問題を確認できました。これは、再マッピング後にカーネルがコードを共有しないためであると考えられます。現在解決策を調査しています。

.text が再マップされた後、perf ツールがシンボルを追尾できなくなり（図 C-7）、perf の出力にはシンボルが表示されなくなります。起動時に /tmp/perf-PID.map の静的シンボルを perf に提供する必要があります。

```
1.35%  node  perf-12142.map          [.] 0x0000562eb19e86aa
1.19%  node  perf-12142.map          [.] 0x0000562eb19e8803
0.71%  node perf-12142.map           [.] 0x0000562eb19e891f
```

図 C-7 ラージページのマッピング後 perf 出力にシンボルが出力されない場合

C.6 ケーススタディー

ここでは、最適化がパフォーマンス向上にどのように役立ち、3 つの環境の 3 つのワークロードで ITLB ミスがどのように軽減されるか詳しく説明します。次のワークロードを使用します。

- Ghost[42] は、最新のオンライン出版を構築および実行する、オープンソースで適用可能なプラットフォームです。
- Web Tooling[43] は、JavaScript* 関連のワークロードを測定するために設計されたスイートです。

[42] Ghost Team (n.d.)。Ghost: プロレベルの出版プラットフォーム。Ghost Non Profit ウェブサイトから取得: https://ghost.org/ (英語)。
[43] Google Web Tooling (2019 年 8 月)。Web Tooling Benchmark。Web Tooling Benchmark から取得:

- MediaWiki[44] は、PHP で記述された無料のオープンソース wiki エンジンです。

このケーススタディーでは、インテル® Xeon® Platinum 8180 プロセッサー (Skylake† Server マイクロアーキテクチャー・ベース) で実行される Ghost.js と Web Tooling のデータを使用し、MediaWiki にインテル® Xeon® D-2100 プロセッサー[45] を使用してラージページのメリットを示します。ケーススタディーの最後に、視覚化ツールを使用してデータのパターンを識別する方法を紹介します。

C.6.1 Ghost.js ワークロード

Ghost は JavaScript で記述された Node.js で実行されるオープンソースのブログ・プラットフォームです。サーバーとして Node.js* で実行される単一インスタンスの Ghost.js を持ち、リクエストするクライアントとして Apache* Bench を使用するワークロードを作成しました。1 秒あたりの要求数 (RPS) メトリックでパフォーマンスを測定します。

インテルが開発して提供した 2MB コード PR は、現在 Node.js* マスターにマージされています。最新の Node.js* ビルドでは、実行時に --enable-largepages=on スイッチを使用してラージページを有効にできます。古いビルドでは、--use-largepages オプションを指定してビルドすることでラージページを有効にできます。次に、デフォルトの Node.js* ビルドとラージページが有効にされたビルドを比較します。

表 C-4 に、デフォルトの Node.js* とラージページが有効な Node.js* の主要メトリックを示します。RPS は 5% 向上しています。ITLB ミスによるストールは 56% 減少しています。ITLB MPKI は 57% 向上しています。パフォーマンス・ゲインは、主に ITLB ページウォークの 55% 減少から得られます。2MB ページを使用するため、2MB ウォーク数が増えることに注意してください。

表 C-4 ラージページあり/なしでの Ghost.js の主要メトリック

メトリック	**Node.js* とラージページ**	**ラージページなしの Node.js* (デフォルト)**	**ラージページ/デフォルト**
スループット 1 秒あたりの要求数 (RPS)	134.32	127.23	1.05
metric_ITLB_Misses(%)	2.87	6.47	0.44
txn ごとのメトリックサイクル	30048182	31426008	0.95
txn ごとのメトリック命令	46703106	47153431	0.99
ITLB_MISSES.WALK_COMPLETED	36799580	82504511	0.45
ITLB_MISSES.WALK_COMPLETED_2M_4M	938969	250959	3.74
ITLB_MISSES.WALK_COMPLETED_4K	35842004	82166894	0.44
メトリック ITLB MPKI	0.098	0.230	0.43
メトリック ITLB 4K MPKI	0.096	0.229	0.43
メトリック ITLB 2M_4M MPKI	0.002	0.0007	3.55

C.6.2 Web Tooling ワークロード

C.6.2.1 Node バージョン

Clear Linux* は、ラージページのサポート向けにコンパイルされた Node 10.16.0 を配布しています。Ubuntu* 18.04 にオフィシャルの Node 10.16.0 (ラージページは未サポート) をインストールして、同一バージョンで比較できるようにしました。Ubuntu* には、apt のリポジトリーの一部として Node 8.10.0 が含まれています。

---

https://github.com/v8/web-tooling-benchmark (英語)。

[44] WikiMedia Foundation (2019 年 8 月)。Mediawiki Software。Mediawiki Software から取得: http://mediawiki.org/wiki/MediaWiki (英語)。

[45] Xeon-D、I (n.d.)。Xeon-D。intel.com から取得: https://www.intel.com/content/www/us/en/products/processors/xeon/d-processors.html (英語)。

C.6.2.2 Web Tooling

これは、Babel や TypeScript のような一般的なツールの主なワークロードなど、ウェブ開発者がよく利用する JavaScript* に関するワークロードを測定するために設計されたスイートです。多くのサブコンポーネントがあり、スループットのスコアをレポートします。

C.6.2.3 Clear Linux* と Ubuntu* を比較

表 C-5 は、Web Tooling のワークロード実行の主なメトリックを示しています。Ubuntu* と比較すると、Clear Linux* にはスループットとサイクルにわずかな改善が見られます。Clear Linux* では ITLB ミスストールが 59%、ITLB MPKI が 51%、そして 4KB MPKI が 52% 軽減されています。ここでは、--use-largepages オプションでコンパイルされた Clear Linux* が配布する Node.js* を使用しています。ITLB ストールがそれほど多くないため、スループットにも大きな影響はありません。

表 C-5 Clear Linux* と Ubuntu* 18.04 における Web Tooling の主要メトリック

メトリック	**Clear Linux***	**Ubuntu* 18.04**	**Clear Linux*/ Ubuntu***
スループット	10.91	10.80	1.01
metric_ITLB_Misses(%)	0.91	2.21	0.41
txn ごとのメトリックサイクル	531,821,553.08	537,631,089.06	0.98
txn ごとのメトリック命令	836,955,459.53	879,834,649.97	0.95
ITLB_MISSES.WALK_COMPLETED	8,145,008	17,230,161	0.47
ITLB_MISSES.WALK_COMPLETED_2M_4M	241,215	142,356	1.69
ITLB_MISSES.WALK_COMPLETED_4K	7,909,985	17,070,769	0.46
メトリック ITLB MPKI	0.0298	0.0604	0.49
メトリック ITLB 4K MPKI	0.0289	0.0598	0.48
メトリック ITLB 2M_4M MPKI	0.0009	0.0005	1.76

C.6.3 MediaWiki ワークロード

MediaWiki は PHP で記述された無料のオープンソース wiki エンジンです。MediaWiki を実行するには HHVM 3.25 を使用します。HHVM はホットなテキストページを 2MB ページ[46] にマップし、4KB と 2MB の両方のページを使用します。HHVM は、-vEval.MaxHotTextHugePages と -vEval.MapTCHuge コマンドライン・オプションを提供し、ホットなテキストページとトランスレーション・キャッシュ・ページ (JIT で生成されたコードを保持) 向けにラージページを有効にします。さらに、TLB ミスを減らすコードの順序付けに依存します。表 C-6 は、ラージページで改善されたメトリックを示しています。ITLB ミスストールが 16% 減少し、完了したページウォークが 29% 減少し、共有 TLB へのヒットが 66% 減少しています。Skylake† Server マイクロアーキテクチャーは、新しいプリサイス・フロントエンド・イベントを導入しているため (例えば、FRONTEND_RETIRED.ITLB_MISS は実際に ITLB (命令 TLB)ミスが生じたリタイアした命令をカウント)、STLB_MISS が 23% 軽減され、ラージページではすべてが少なくなっていることが分かります。

[46] Ottoni,G. & Bertrand, M (2017 年)。Optimizing Function Placement for Large-Scale Data-Center Applications。CGO 2017。

表 C-6. HHVM 上の MediaWiki ワークロード向けの主要メトリック

メトリック	ラージページ	ラージページなし	ラージページ/ラージページなし
metric_ITLB_Misses(%)	2.86	3.42	0.84
txn ごとのメトリックサイクル	44339066	44372158	0.99
txn ごとのメトリック命令	39129166	39168226	0.99
ITLB_MISSES.WALK_COMPLETED	7735.06	10850.57	0.71
ITLB_MISSES.WALK_COMPLETED_2M_4M	241,215	142,356	1.69
ITLB_MISSES.WALK_COMPLETED_4K	7,909,985	17,070,769	0.46
FRONTEND_RETIRED.ITLB_MISS	160403.87	162401.23	0.99
FRONTEND_RETIRED.L1I_MISS	160403.87	162401.23	0.99
FRONTEND_RETIRED.L2_MISS	19566.98	20075.91	0.97
FRONTEND_RETIRED.STLB_MISS	3820.17	4961.98	0.77
メトリック ITLB MPKI	0.2426	0.351	0.69
メトリック ITLB 4K MPKI	0.2310	0.341	0.67
メトリック ITLB 2M_4M MPKI	0.0116	0.009	1.18

C.6.4 利点の可視化

C.6.4.1 プリサイスイベント

インテル® PMU は、キャッシュミスの正確な命令アドレスなどをレポートするプリサイス・イベントベースのサンプリング (PEBS) を備えています。インテル® Xeon® Platinum 8180 プロセッサー (Skylake† Server マイクロアーキテクチャー・ベース) では、PEBS イベントはソースコードでの検出が困難な追加のフロントエンド・イベントをサポートしています。表 C-7 に示すように、それらの 2 つは ITLB ミス向けです。

表 C-7 ITLB ミス向けのプリサイス・フロントエンド・イベント

イベント	説明
FRONTEND_RETIRED.ITLB_MISS (第 1 レベル)	実際の ITLB ミスに続いてリタイアした命令。
FRONTEND_RETIRED.STLB_MISS (第 2 レベル)	ITLB および STLB ミス (第 2 レベル) の後にリタイアした命令。

C.6.4.2 プリサイス TLB ミスの可視化

Linux* の perf を使用して frontend_retired.itlb_miss イベントを記録し、Netflix* からオープンソースのツールである FlameScope を入手してイベントを可視化できます。FlameScope は、時間範囲のヒートマップと FlameGraph を調査する視覚化ツールです。FlameScope では、入力データをインタラクティブな subsecond-offset ヒートマップを視覚化することから開始します。

```
$ perf record -o /tmp/webtooling.node12.14.1.callgraph.itlb_miss.orig.out.b -b  -e
frontend_retired.itlb_miss -- /home/dslo/ssuresh1/node-v12.14.1/node.orig --perf-
basic-prof dist/cli.js
[ perf record: Woken up 8 times to write data ]
[ perf record: Captured and wrote 1.944 MB
/tmp/webtooling.node12.14.1.callgraph.itlb_miss.orig.out.b (1539 samples) ]

$ perf script -i /tmp/webtooling.node12.14.1.callgraph.itlb_miss.orig.out --header >
webtooling.node12.14.1.itlb_miss.orig
```

図 C-8 -e frontend_retired.itlb_miss で perf レコードの ITLB ミスを特定し、perf スクリプトを実行して FlameScope にインポートするデータを取得

perf スクリプトの出力は、FlameScope にインポートして ITLB ミスを視覚化できます。ワークロードの一部には、他の部分よりも多くの ITLB ミスがあることが分かります。図 C-9 と 図 C-10 を比較すると、node.js がラージページを使用している場合、ヒートマップが ITLB ミスに対してまばらであることが見てとれます。

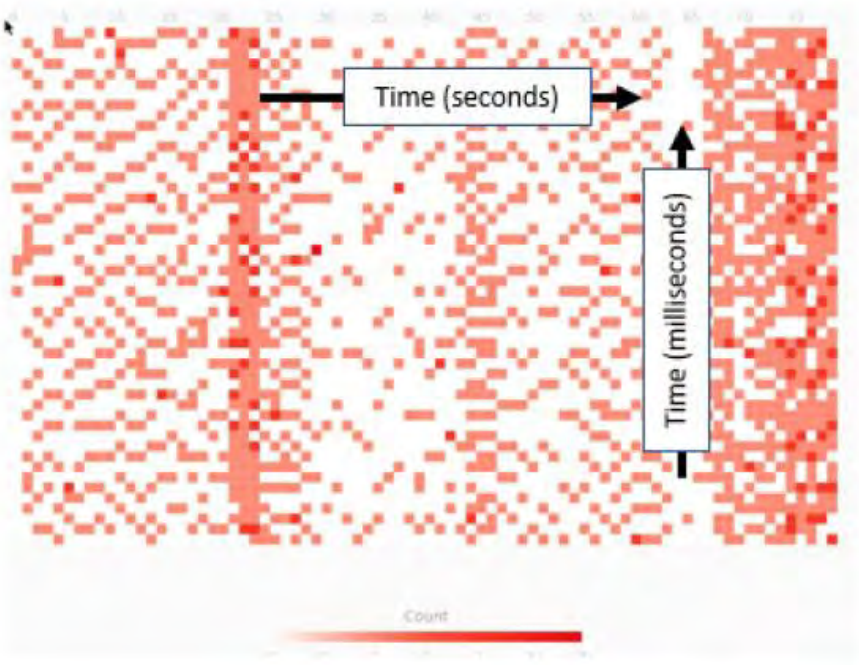


図 C-9 FlameScope を使用して WebTooling ワークロードの ITLB ミスのヒートマップを可視化

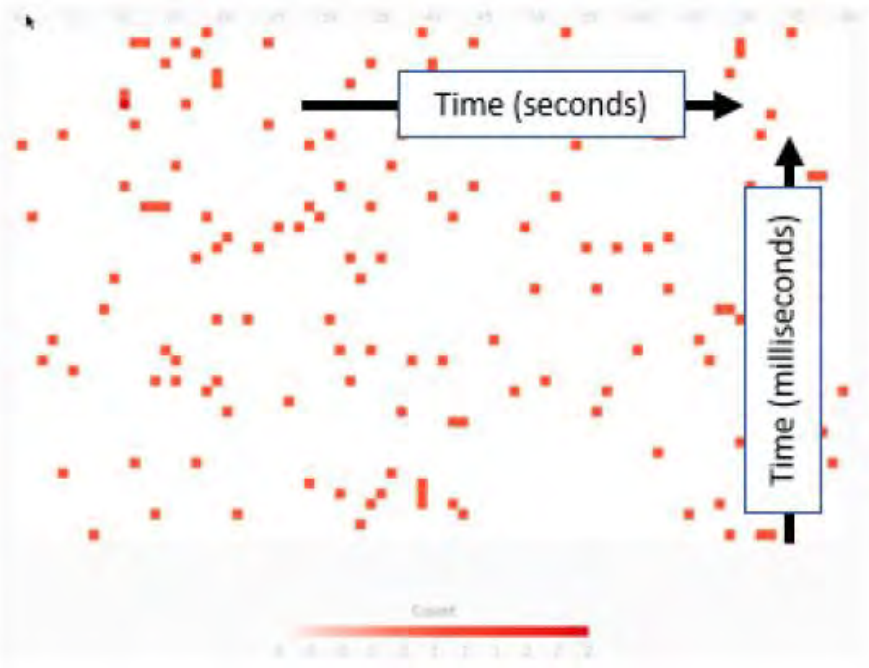


図 C-10 FlameScope を使用してラージページを使用する WebTooling ワークロードの ITLB ミスのヒートマップを可視化

さらに、perf スクリプトの出力からビルトイン関数に関連する ITLB ミスを抽出することで、V8 ビルトイン関数の ITLB ミスカウントを視覚化できます。y 軸にビルトイン関数の仮想アドレス、x 軸に時間を適用してグラフをプロットします (図 C-11 と図 C-12)。FlameScope のグラフと同様に、Node.js* がラージページを使用する場合、ITLB ミスはまばらになります。

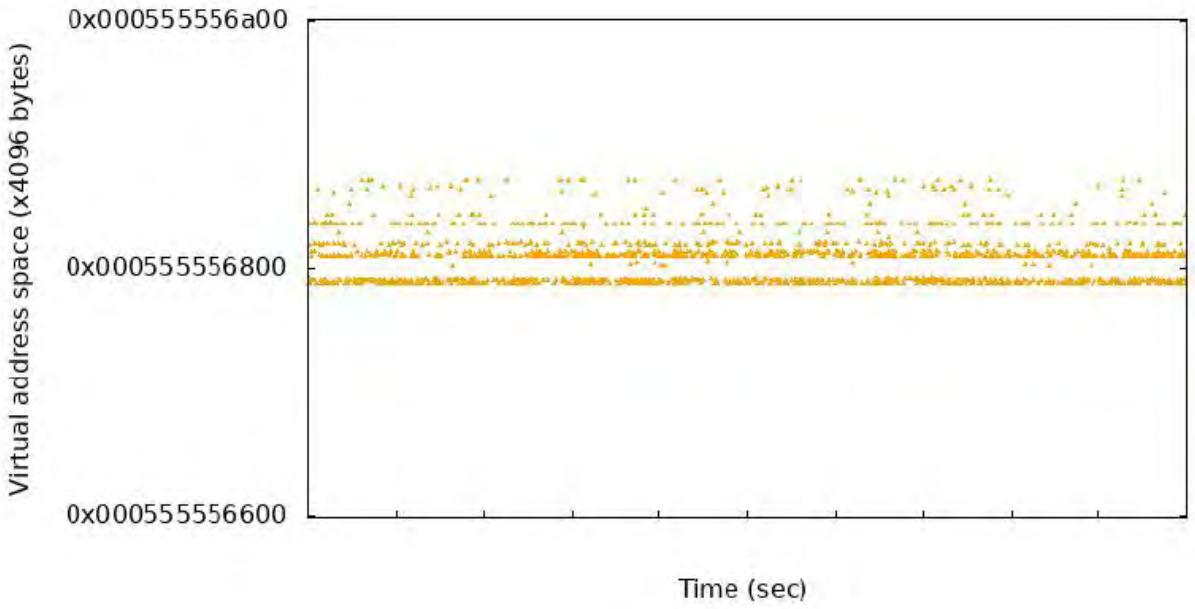


図 C-11 Ghost.js ワークロードのビルトイン関数の ITLB ミスの傾向を可視化

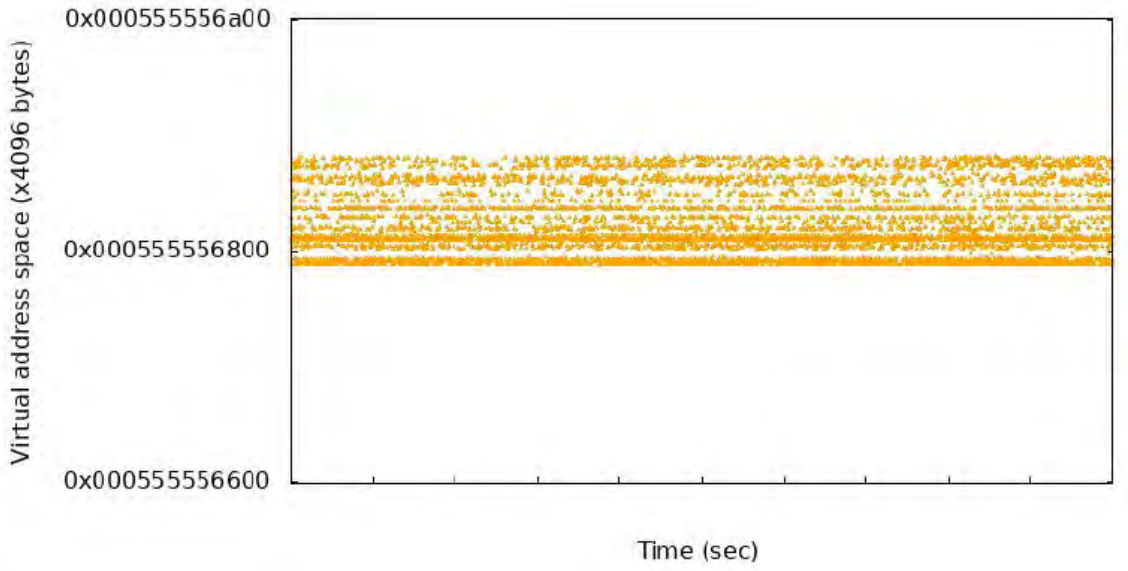


図 C-12 ラージページを使用する Ghost.js ワークロードのビルトイン関数の ITLB ミスの傾向を可視化

C.7 まとめ

このランタイム最適化の考察では、ランタイムの ITLB ミスストールが高い場合に発生する問題について説明し、問題の診断と解決方法を示すリファレンス実装について説明しました。ケーススタディーでは、解決方法をランタイムに統合する利点を示しました。3 つのケーススタディーの例により、2M ページを使用することで ITLB ミスストールが 43%、ITLB ページウォークが 45%、そして ITLB MPKI が 46% 軽減する可能性があることを示しました。

C.8 テスト構成の詳細

テスト構成の詳細を以下に示します。

表 C-8 システムの詳細

システム情報	DSLOHost011
製造元	Intel Corporation
製品名	S2600WFT
BIOS バージョン	SE5C620.86B.0X.01.0115.012820180604
OS	Ubuntu* 18.04.3 LTS
カーネル	4.15.0-58-generic
マイクロコード	0x200005e

表 C-9 プロセッサー情報

モデル名:	インテル® Xeon® Platinum 8180 CPU@2.50GHz
ソケット	2
ハイパースレッディング有効	はい
合計 CPU 数	112
NUMA ノード	2
NUMA cpu リスト	0-27,56-83 :: 28-55,84-111
L1d キャッシュ	32K
L1i キャッシュ	32K
L2 キャッシュ	1024K
L3 キャッシュ	39424K
プリフェッチ有効	DCU HW、DCU IP、L2 HW、L2 Adj。
ターボモード有効	True
電力とパフォーマンスのポリシー	Balanced
CPU 周波数ドライバー	intel_pstate
CPU 周波数管理	powersave
現在の CPU 周波数 (MHz)	1000
AVX2 利用可能	True
AVX512 利用可能	True
AVX512 テスト	passed
PPIN (CPU0)	c6aa1d2bcbba4d86

表 C-10 カーネルの脆弱性状態

脆弱性	**DSLOHost011**
CVE-2017-5753	OK (緩和: usercopy/swapgs バリアと __user pointer 機密性)
CVE-2017-5715	OK (完全な retpoline + IBPB で脆弱性を緩和)
CVE-2017-5754	OK (緩和: PTI)
CVE-2018-3640	OK (CPU マイクロコードにより脆弱性を緩和)
CVE-2018-3639	OK (緩和: 投機ストアバイパスは、prctl と seccomp により無効化)
CVE-2018-3615	OK (CPU ベンダーは CPU モデルに脆弱性がないと報告)
CVE-2018-3620	OK (緩和: PTE 反転)
CVE-2018-3646	OK (システムはハイパーバイザーを実行していません)
CVE-2018-12126	OK (緩和: クリア CPU バッファー; SMT 脆弱性)
CVE-2018-12130	OK (緩和: クリア CPU バッファー; SMT 脆弱性)
CVE-2018-12127	OK (緩和: クリア CPU バッファー; SMT 脆弱性)
CVE-2019-11091	OK (緩和: クリア CPU バッファー; SMT 脆弱性)

C.9 関連情報

この付録で引用した参考文献に加え、次の文献を参照しました。

- Ahmad Yasin, Intel Corporation. (2014). A Top-Down method for performance analysis and counters
- architecture. In IEEE International Symposium on Performance Analysis of Systems and Software,

- ISPASS , (pp. 35-44).
- Performance Monitoring Event List. Retrieved from 01.org: https://download.01.org/perfmon/SKX/ (英語)
- Panchenko, M. (2017). Building Binary Optimizer with LLVM. Retrieved from LLVM.ORG: https://llvm.org/devmtg/2016-03/Presentations/BOLT_EuroLLVM_2016.pdf (英語)
- Panchenko, M., Auler, R., Nell, B., Ottoni, & Guilherme. (n.d.). BOLT: A Practical Binary Optimizer for
- Datacenters and Beyond.

付録 D
命令レイテンシーとスループット

注意

最新のプロセッサーのレイテンシーとスループットの情報は、次のサイトで公開されています:
https://software.intel.com/en-us/articles/intel-sdm#optimization (英語)

この付録には、よく使用される命令に関連したレイテンシーとスループットを記載した表を収録しています。[47] 命令タイミングデータは、プロセッサーのファミリー/モデルによって異なります。付録は、以下の節で構成されています。

- **D.1 節「概要」** — 命令の選択とスケジューリングに関係する問題の概要を説明しています。
- **D.2 節「用語説明」** — 本付録で使用する用語の定義を示します。
- **D.3 節「レイテンシーとスループット」** — よく使用される命令のスループット、レイテンシーを各表にまとめています。

D.1 概要

この付録には、アセンブリー言語のプログラマーやコンパイラー開発者向けの情報が含まれています。ここで示す情報を参照することにより、依存関係チェーンに起因するレイテンシーが最も小さくなる命令シーケンスを選ぶ作業が容易になります。パフォーマンスに関する以下の要素にアプリケーションが影響されていなければ、ここで示す情報を活用すると、パフォーマンス面で数パーセントの効果が現れることが分かっています。

- キャッシュ・ミス・レイテンシー
- バス帯域幅
- I/O 帯域幅

第 2 章では以下のパフォーマンス問題について議論しましたが、プログラマーがこれらの問題を解決した後に問題となるのが、命令の選択とスケジューリングです。

- ストア・フォワーディングの制約事項を守る。
- キャッシュラインの分割やメモリー・オーダー・バッファーの分割を避ける。
- 分岐予測を妨げない。
- メモリー・ロケーションへの xchg 命令の使用回数を最小限にする。

上記の項目は、適切な命令を選択することに関連しますが、本付録では次の問題点に焦点を当てています。重要な順番に記載していますが、どの項目がパフォーマンスに大きく影響するかはアプリケーションによって異なります。

- 実行コアへのマイクロオペレーション (uop) のフローを最大化します。5 つ以上のマイクロオペレーション (uop) で構成される命令は、マイクロコード ROM (MSROM) から供給されるため追加のステップが必要となります。長いフローのマイクロオペレーション (uop) の命令では、フロントエンドが遅延し、実行コアへのマイクロオペレーション (uop) 供給が停止します。

  インテル® Pentium® 4 プロセッサー、インテル® Xeon® プロセッサーでは、マイクロコード ROM からマイクロオペレーションを実行すると、多くの場合、複数のマイクロオペレーション (uop) をトレースキャッシュにパックする効率が低下します。可能な場合は、4 つ以下のマイクロオペレーション (uop) で構成される命令を選択します。例えば、単一のメモリーオペランドの 32 ビットの整数乗算を実行する場合、マイクロコード ROM を参照せずにトレースキャッシュに収まりますが、メモリーオペランドを持つ 16 ビットの整数乗算は収まりません。

[47] 命令レイテンシーはいくつかの限定された状況 (命令レイテンシーが顕著に現れる依存性チェーンがあるタイトなループなど) では役立つかもしれませんが、スーパースカラー、アウトオブオーダー・マイクロアーキテクチャー上のソフトウェア最適化では、一般に大きなスケールのコードパスの有効スループットをより高める方が有益です。命令のスケジューリングに影響する命令レイテンシーのみに注目するコーディング・テクニックは、アウトオブオーダー・マシンに悪影響を与えたり、命令レベルの並列性を損ねる可能性があるため、最適とはならないことがあります。

- リソースの競合を避けます。同じポートや同じ実行ユニットの奪い合いが起きないように複数の命令をインターリーブすると、スループットが改善できます。例えば、PADDQ 命令と PMULUDQ 命令を交互に配置した場合、どちらの命令も 2 クロックサイクルにつき発行は 1 回というスループットになります。インターリーブを適用すると、1 クロックサイクルにつき命令 1 つという高いスループットが得られます。これは、ポートが同じでも異なる実行ユニットが使用されるためです。発行ポートの帯域幅が狭くならないようにしたり、レイテンシーを隠蔽したり、ソフトウェアのパフォーマンスをより向上するには、スループットの高い命令を選択するのも有効です。

- クリティカル・パスで生じる依存関係チェーンのレイテンシーを最小にします。例えば、2 ビット左へシフトする操作では、2 つの加算命令としてエンコードする方が、1 つのシフト命令としてエンコードするよりも実行速度が速くなります。レイテンシーが問題でない場合、シフト命令を実行するとバイト・エンコーディングの密度が高まります。

本書には、一般的な規則、特殊な規則、コーディングに関するガイドライン、命令に関するデータを収録しましたが、そのほかにも、http://developer.intel.com/software/products/index.htm (英語) で入手できるソフトウェア解析およびチューニング・ツール群が利用できます。このツール群には、さまざまなパフォーマンス監視機能を備えたインテル® VTune™ プロファイラーが含まれます。

D.2 用語説明

データをいくつかの表にまとめています。表には次の情報が含まれます。

- **命令名** — 各命令のアセンブリー・ニーモニック。
- **レイテンシー** — 命令を構成しているすべてのマイクロオペレーション (uop) の実行が実行コアで完了するのに要するクロックサイクル数。
- **スループット** — 発行ポートが同じ種類の命令を再度受け入れられるようになるまで待たなければならないクロックサイクル数。多くの命令は、命令のスループットの方がレイテンシーよりもかなり小さくなります。
- RDRAND 命令のレイテンシーとスループットは上記の定義の例外です。RDRAND 命令を実行するハードウェア機能はアンコアにあるため、物理パッケージ内のすべてのプロセッサー・コアと論理プロセッサーに依存します。シングルスレッドで実行される “rdrand に続く jnc” シーケンスを使用するソフトウェアのレイテンシーとスループットは、~100 サイクル程度に抑えることができます。Ivy Bridge マイクロアーキテクチャー・ベースの第 3 世代インテル® Core™ プロセッサーでは、アンコアの RDRAND による乱数送出の総帯域幅は、およそ 500MB/秒です。同一プロセッサー・コアのマイクロアーキテクチャーと異なるアンコア実装では、RDRAND のレイテンシーとスループットは、インテル® Core™ プロセッサーとインテル® Xeon® プロセッサーでは異なります。

D.3 レイテンシーとスループット

この節には、使用頻度の高い命令のレイテンシーとスループットに関する情報が記載されています。ここで扱われる命令には、インテル® MMX® テクノロジー命令、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、および以降の世代のインテル® SSE 命令が含まれるほか、整数命令や x87 浮動小数点命令についても、よく使用されるほとんどの命令が含まれます。

ダイナミック・エグゼキューションは複雑であり、また命令シーケンスの実行順序にかかわらず実行するという特性 (アウトオブオーダー) が実行コアに備わっているため、ここに示したレイテンシー・データを用いるだけでは実際のコードシーケンスのパフォーマンスを正確に予測するには十分ではありません。

- 命令のレイテンシー・データは、依存関係チェーンをチューニングする際に役立ちます。ただし、依存関係チェーンは、アウトオブオーダー・コアがマイクロオペレーション (uop) を並列実行する能力を制限します。命令のスループット・データは、依存関係チェーンによって妨げられない並列コードをチューニングする際に役立ちます。
- 数値はすべて概算値です。
  - マイクロアーキテクチャー (uop) の実装方式が将来変われば、この数値も変わる可能性があります。
  - この数値は、命令レベルでのパフォーマンス評価基準として使用するものではありません。異なるマイクロアーキテクチャー・ベースのマイクロプロセッサーで実行される命令レベルのパフォーマンスの比較は、複雑な話題であり、本書以外の情報を必要とします。

異なるマイクロアーキテクチャー間でレイテンシーとスループットのデータを比較すると、誤解を招きます。

D.3.1 節では、レジスター-レジスター方式の命令タイプのレイテンシーとスループット・データを示します。D.3.3 節では、レジスター-メモリー方式の命令タイプと、メモリー-レジスター方式の命令タイプのレイテンシーとスループットの仕様を調整する方法について説明します。

レイテンシーやスループットの数値が半クロックとなることがあります。これは倍速 ALU にのみ適用されます。

D.3.1 レジスターオペランドのレイテンシーとスループット

命令のレイテンシーとスループットのデータを表 C-4 から表 C-15 に示します。インテル® AES-NI、インテル® SSE4.2、インテル® SSE4.1、インテル® SSSE3、インテル® SSE3、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジー命令を収録し、頻繁に使用されるインテル® 64 命令および IA-32 命令のほとんどを収録しています。異なるプロセッサー・マイクロアーキテクチャーの命令レイテンシーとスループットは、別々の列に示されます。

プロセッサーの命令タイミングデータはプロセッサー固有であり、同じファミリーのエンコーディングでモデル・エンコーディング値が 3 の場合もあれば、2 未満の場合もあります。CPUID シグネチャーが 0xF2n の列と 0xF3n の列には、命令レイテンシーとスループットのデータセットが別々に示されます。0xF3n の列のデータは、CPUID シグネチャーが 0xF4n と 0xF6n のインテル® プロセッサーにも適用されます。0xF2n の表記は、入力値が EAX = 1 の CPUID 命令によって報告される EAX レジスターの下位 12 ビットの 16 進値を表しています。「F」はファミリー・エンコーディングが 15、「2」はモデル・エンコーディングが 2、「n」は任意のステッピング・エンコーディングを示します。

インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーは、06_0EH として示されます。65nm プロセスを採用したインテル® Core™ マイクロアーキテクチャ・ベースのプロセッサーは、06_0FH です。

拡張版インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーは、06_17H と 06_1DH で示されます。Nehalem† マイクロアーキテクチャー・ベースのプロセッサーの CPUID ファミリー/モデル・シグネチャーは、06_1AH、06_1EH、6_1FH、および 06_2EH です。

Westmere† マイクロアーキテクチャー・ベースのプロセッサーは、06_25H、6_2CH、および 06_2FH です。

Sandy Bridge† マイクロアーキテクチャー・ベースのプロセッサーは、06_2AH および 06_2DH です。

Ivy Bridge† マイクロアーキテクチャー・ベースのプロセッサーは、06_3AH と 06_3EH です。

Haswell† マイクロアーキテクチャー・ベースのプロセッサーは、06_3CH、06_45H、および 06_46H です。

表 D-1 近年のマイクロアーキテクチャー (CPUIDシグネチャー) による SIMD 拡張命令のサポート

DisplayFamily_DisplayModel	近年のマイクロアーキテクチャー
06_4EH, 06_5EH	Skylake† マイクロアーキテクチャー
06_3DH, 06_47H, 06_56H	Broadwell† マイクロアーキテクチャー
06_3CH, 06_45H, 06_46H, 06_3FH	Haswell† マイクロアーキテクチャー
06_3AH, 06_3EH	Ivy Bridge† マイクロアーキテクチャー
06_2AH, 06_2DH	Sandy Bridge† マイクロアーキテクチャー
06_25H, 06_2CH, 06_2FH	Westmere† マイクロアーキテクチャー
06_1AH, 06_1EH, 06_1FH, 06_2EH	Nehalem† マイクロアーキテクチャー
06_17H, 06_1DH	拡張版インテル® Core™ マイクロアーキテクチャー
06_0FH	インテル® Core™ マイクロアーキテクチャー

命令レイテンシーはマイクロアーキテクチャーによって異なります。表 D-1 は、近年のマイクロアーキテクチャーにおけるインテル® ストリーミング SIMD 拡張命令を示しています。各マイクロアーキテクチャーは、CPUID 命令の「ファミリー」と「モデル」によって示される複数のシグネチャーに関連付けられている場合があります。特定のファミリー/モデルに関連付けられた、すべてのプロセッサーのすべての命令セット拡張が有効となるわけではありません。特定の命令セット拡張がサポートされているかどうかを知るには、ソフトウェアは『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の説明に従って適切な CPUID 機能フラグを使用する必要があります。

表 D-2 マイクロアーキテクチャー (CPUID シグネチャー) で挿入された命令拡張

SIMD 命令拡張	DisplayFamily_DisplayModel							
	06_4EH, 06_5EH	06_3DH, 06_47H, 06_56H	06_3CH, 06_45H, 06_46H, 06_3FH	06_3AH, 06_3EH	06_2AH, 06_2DH	06_25H, 06_2CH, 06_2FH	06_1AH, 06_1EH, 06_1FH, 06_2EH	06_17H, 06_1DH
CLFLUSHOPT	はい	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ
ADX, RDSEED	はい	はい	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ
AVX2, FMA, BMI1,BMI2	はい	はい	はい	いいえ	いいえ	いいえ	いいえ	いいえ
F16C, RDRAND, RWFSGSBASE	はい	はい	はい	はい	いいえ	いいえ	いいえ	いいえ
AVX	はい	はい	はい	はい	はい	いいえ	いいえ	いいえ
AESNI, PCLMULQDQ	はい	はい	はい	はい	はい	はい	いいえ	いいえ
SSE4.2, POPCNT	はい	はい	はい	はい	はい	はい	はい	いいえ
SSE4.1	はい	はい	はい	はい	はい	はい	はい	はい
SSSE3	はい	はい	はい	はい	はい	はい	はい	はい
SSE3	はい	はい	はい	はい	はい	はい	はい	はい
SSE2	はい	はい	はい	はい	はい	はい	はい	はい
SSE	はい	はい	はい	はい	はい	はい	はい	はい
MMX	はい	はい	はい	はい	はい	はい	はい	はい

表 D-3 BMI1、BMI2 および汎用命令

命令	レイテンシー[1]		スループット	
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**
ADCX	1	1	1	1
ADOX	1	1	1	1
RESEED	RDRAND と同様	RDRAND と同様	RDRAND と同様	RDRAND と同様

表 D-4 256 ビットのインテル® AVX 命令

命令	レイテンシー 1			スループット		
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**
VEXTRACTI128 xmm1, ymm2, imm	1	1	1	1	1	1
VMPSADBW	4	6	6	2	2	2
VPACKUSDW/SSWB	1	1	1	1	1	1
VPADDB/D/W/Q	1	1	1	0.33	0.5	0.5
VPADDSB	1	1	1	0.5	0.5	0.5
VPADDUSB	1	1	1	0.5	0.5	0.5
VPALIGNR	1	1	1	1	1	1
VPAVGB	1	1	1	0.5	0.5	0.5
VPBLENDD	1	1	1	0.33	0.33	0.33
VPBLENDW	1	1	1	1	1	1
VPBLENDVB	1	2	2	1	2	2
VPBROADCASTB/D/SS/SD	3	3	3	1	1	1
VPCMPEQB/W/D	1	1	1	0.5	0.5	0.5
VPCMPEQQ	1	1	1	0.5	0.5	0.5
VPCMPGTQ	3	5	5	2	2	2
VPHADDW/D/SW	3	3	3	2	2	2
VINSERTI128 ymm1, ymm2, xmm, imm	3	3	3	1	1	1
VPMADDWD	5[b]	5	5	0.5	1	1
VPMADDUBSW	5[b]	5	5	0.5	1	1
VPMAXSD	1	1	1	0.5	0.5	0.5
VPMAXUD	1	1	1	0.5	0.5	0.5
VPMOVSX	3	3	3	1	1	1
VPMOVZX	3	3	3	1	1	1
VPMULDQ/UDQ	5[b]	5	5	0.5	1	1
VPMULHRSW	5[b]	5	5	0.5	1	1
VPMULHW/LW	5[b]	5	5	0.5	1	1
VPMULLD	10[b]	10	10	1	2	2
VPOR/VPXOR	1	1	1	0.33	0.33	0.33
VPSADBW	3	5	5	1	1	1
VPSHUFB	1	1	1	1	1	1
VPSHUFD	1	1	1	1	1	1
VPSHUFLW/HW	1	1	1	1	1	1
VPSIGNB/D/W/Q	1	1	1	0.5	0.5	0.5

命令レイテンシーとスループット

命令	レイテンシー 1			スループット		
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**
VPERMD/PS	3	3	3	1	1	1
VPSLLVD/Q	2	2	2	0.5	2	2
VPSRAVD	2	2	2	0.5	2	2
VPSRAD/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLDQ ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm, ymm, ymm	4	4	4	1	1	1
VPUNPCKHBW/WD/DQ/QDQ	1	1	1	1	1	1
VPUNPCKLBW/WD/DQ/QDQ	1	1	1	1	1	1
ALL VFMA	4	5	5	0.5	0.5	0.5
VPMASKMOVD/Q mem, ymm[d], ymm				1	2	2
VPMASKMOVD/Q NUL, msk_0, ymm				>200[e]	2	2
VPMASKMOVD/Q ymm, ymm[d], mem	11	8	8	1	2	2
VPMASKMOVD/Q ymm, msk_0, [base+index][f]	>200	~200	~200	>200	~200	~200

b: バイパスによる 1 サイクルのバブルを含みます
c: バイパスによる 2 つの 1 サイクルバブルを含みます
d: L1 参照で測定された MASKMOV 命令のタイミングと、少なくとも 1 つ以上の要素を選択するマスクレジスター。
e: 0 要素を選択するマスク値の MASKMOV ストア命令と、アシストによる遅延を被る不正アドレス (NULL または非 NULL)
f: 0 要素を選択するマスク値の MASKMOV ロード命令と、アシストによる遅延を被る正規のアドレス

表 D-5 L1D* からタイミングデータを収集

命令	レイテンシー[1]			スループット		
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**
VPGATHERDD/PS xmm, [vi128], xmm	~20	~17	~14	~4	~5	~7
VPGATHERQQ/PD xmm, [vi128], xmm	~18	~15	~12	~3	~4	~5
VPGATHERDD/PS ymm, [vi256], ymm	~22	~19	~20	~5	~6	~10
VPGATHERQQ/PD ymm, [vi256], ymm	~20	~16	~15	~4	~5	~7

* ギャザー命令はメモリーを参照してデータ要素をフェッチ。タイミングデータは、L1 データキャッシュに存在するメモリー参照、および選択されたすべてのマスク要素に適用されます。

表 D-6 BMI1、BMI2 および汎用命令

命令	レイテンシー[1]			スループット		
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**
ANDN	1	1	1	0.5	0.5	0.5
BEXTR	2	2	2	0.5	0.5	0.5
BLSI/BLSMSK/BLSR	1	1	1	0.5	0.5	0.5
BZHI	1	1	1	0.5	0.5	0.5
MULX r64, r64, r64	4	4	4	1	1	1
PDEP/PEXT r64, r64, r64	3	3	3	1	1	1
RORX r64, r64, r64	1	1	1	0.5	0.5	0.5
SALX/SARX/SHLX r64, r64, r64	1	1	1	0.5	0.5	0.5
LZCNT/TZCNT	3	3	3	1	1	1

表 D-7 F16C、RDRAND 命令

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
RDRAND* r64	それぞれ異なる	それぞれ異なる	それぞれ異なる	<200	<300	~250	~250	<200
VCVTPH2PS ymm1, xmm2	7	6	6	7	1	1	1	1
VCVTPH2PS xmm1, xmm2	5	4	4	6	1	1	1	1
VCVTPS2PH ymm1, xmm2, imm	7	6	6	10	1	1	1	1
VCVTPS2PH xmm1, xmm2, imm	5	4	4	9	1	1	1	1

D.2 節を参照してください。

表 D-8 256 ビットのインテル® AVX 命令

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
VADDPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VADDSUBPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VANDNPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.5	1	1	1
VANDPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VBLENDPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	0.33	0.33	0.33	0.5
VBLENDVPD/PS ymm1, ymm2, ymm3,	1	2	2	1	1	2	2	1

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
ymm								
VCMPPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VCVTDQ2PD ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTDQ2PS ymm1, ymm2	4	3	3	3	0.5	1	1	1
VCVT(T)PD2DQ ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTPD2PS ymm1, ymm2	7	6	6	4	1	1	1	1
VCVT(T)PS2DQ ymm1, ymm2	4	3	3	3	1	1	1	1
VCVTPS2PD ymm1, xmm2	7	4	4	2	1	1	1	1
VDIVPD ymm1, ymm2, ymm3	14	16-23	25-35	27-35	8	16	27	28
VDIVPS ymm1, ymm2, ymm3	11	13-17	17-21	18-21	5	10	13	14
VDPPS ymm1, ymm2, ymm3	13	12	14	12	1.5	2	2	2
VEXTRACTF128 xmm1, ymm2, imm	3	3	3	3	1	1	1	1
VINSERTF128 ymm1, xmm2, imm	3	3	3	3	1	1	1	1
VMAXPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMINPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMOVAPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMOVDDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVDQA/U ymm1, ymm2	1	1	1	1	0.25	0.25	0.25	0.5
VMOVMSKPD/PS ymm1, ymm2	2	2	2	1	1	1	1	1
VMOVQ xmm1, xmm2	1	1	1	1	0.33	0.33	0.33	0.33
VMOVD/Q xmm1, r32/r64	2	1	1	1	1	1	1	1
VMOVD/Q r32/r64, xmm	2	1	1	1	1	1	1	1
VMOVNTDQ/PS/PD					1	1	1	1
VMOVSHDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVSLDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVUPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMULPD/PS ymm1, ymm2, ymm3	4	3	5	5	0.5	0.5	0.5	1
VORPD/PS ymm1, ymm2,	1	1	1	1	0.33	1	1	1

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
ymm3								
VPERM2F128 ymm1, ymm2, ymm3, imm	3	3	3	2	1	1	1	1
VPERMILPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VRCPPS ymm1, ymm2	4	7	7	7	1	2	2	2
VROUNDPD/PS ymm1, ymm2, imm	8	6	6	3	1	2	2	1
VRSQRTPS ymm1, ymm2	4	7	7	7	1	2	2	2
VSHUFPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	1	1	1	1
VSQRTPD ymm1, ymm2	<18	19-35	19-35	19-35	<12	16-27	16-27	28
VSQRTPS ymm1, ymm2	12	18-21	18-21	18-21	<6	13	13	14
VSUBPD/PS ymm1, ymm2, imm	4	3	3	3	0.5	1	1	1
VTESTPS ymm1, ymm2	3	2	2	2	1	1	1	1
VUNPCKHPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VUNPCKLPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VXORPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VZEROUPPER	0	0	0	0	1	1	1	1
VZEROALL					12	8	8	9
VEXTRACTPS reg, xmm2, imm	3	2	2	2	1	1	1	1
VINSERTPS xmm1, xmm2, reg, imm	1	1	1	1	1	1	1	1
VMASKMOVPD/PS mem[a], ymm, ymm					1	2	2	2
VMASKMOVPD/PS NUL, msk_0, ymm					>200[b]	2	2	2
VMASKMOVPD/PS ymm, ymm[a], mem	11	8	8	9	1	2	2	2
VMASKMOVPD/PS ymm, msk_0, [base+index][c]	>200	~200	~200	~200	>200	~200	~200	~200

CPUID シグネチャー 06_3AH のレイテンシーとスループットは、一般に 06_2AH と同じであり、06_2AH と異なるもののみが 06_3AH のカラムに示されます。

a: L1 参照で測定された MASKMOV 命令のタイミングと、少なくとも 1 つ以上の要素を選択するマスクレジスター。

b: 0 要素を選択するマスク値の MASKMOV ストア命令と、アシストによる遅延を被る不正アドレス (NULL または非 NULL)

c: 0 要素を選択するマスク値の MASKMOV ロード命令と、アシストによる遅延を被る正規のアドレス

VEX.128 エンコードされたインテル® AVX 命令のレイテンシーは、対応するレガシー 128 ビット命令に相当します。

表 D-9 インテル® AES-NI と PCLMULQDQ 命令

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
AESDEC/AESDECLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESENC/AESENCLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESIMC xmm1, xmm2	8	14	14	14	2	2	2	2
AESKEYGENASSIST xmm1, xmm2, imm	12	10	10	10	12	8	8	8
PCLMULQDQ xmm1, xmm2, imm	7[b]	5	7	14	1	1	2	8

b: バイパスによる 1 サイクルのバブルが含まれます。

表 D-10 インテル® SSE4.2 命令

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
CRC32 r32, r32	3	3	3	3	1	1	1	1
PCMPESTRI xmm1, xmm2, imm	15	10	10	11	5	4	4	4
PCMPESTRM xmm1, xmm2, imm	10	10	10	11	6	5	5	4
PCMPISTRI xmm1, xmm2, imm	15	10	10	11	3	3	3	3
PCMPISTRM xmm1, xmm2, imm	15	11	11	11	3	3	3	3
PCMPGTQ xmm1, xmm2	3	5	5	5	0.33	1	1	1
POPCNT r32, r32	3	3	3	3	1	1	1	1
POPCNT r64, r64	3	3	3	3	1	1	1	1

表 D-11 インテル® SSE4.1 命令

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
BLENDPD/S xmm1, xmm2, imm	1	1	1	1	0.33	0.33	0.33	0.5
BLENDVPD/S xmm1, xmm2	1	2	2	2	1	2	2	1
DPPD xmm1, xmm2	9	7	9	9	1	1	1	1
DPPS xmm1, xmm2	13	12	14	13	2	2	2	2
EXTRACTPS xmm1, xmm2, imm	3	2	2	2	1	1	1	1
INSERTPS xmm1, xmm2, imm	1	1	1	1	1	1	1	1
MPSADBW xmm1, xmm2, imm	4	6	6	6	2	2	2	2
PACKUSDW xmm1, xmm2	1	1	1	1	1	1	1	0.5
PBLENVB xmm1, xmm2	2	2	2	2	2	2	2	1
PBLENDW xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PCMPEQQ xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRB/W/D reg, xmm1, imm	3	3	3	3	1	1	1	1
PHMINPOSUW xmm1,xmm2	4	5	5	5	1	1	1	1
PINSRB/W/D xmm1,reg, imm	2	2	2	2	1	1	1	1
PMAXSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMAXUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMOVSXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVSXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVZXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVZXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMULDQ xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PMULLD xmm1, xmm2	10[c]	10	10	5	2	2	2	1
PTEST xmm1, xmm2	3	2	2	2	1	1	1	1
ROUNDPD/PS xmm1, xmm2, imm	6	6	6	3	2	2	2	1
ROUNDSD/SS xmm1, xmm2, imm	6	6	6	3	2	2	2	1

b: バイパスによる 1 サイクルのバブルを含みます

c: バイパスによる 2 つの 1 サイクルバブルを含みます

表 D-12 インテル® SSSE3 命令

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
PALIGNR xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PHADDD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PMADDUBSW xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PMULHRSW xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PSHUFB xmm1, xmm2	1	1	1	1	1	1	1	0.5
PSIGNB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PABSB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5

b: バイパスによる 1 サイクルのバブルを含みます

表 D-13 インテル® SSE3 浮動小数点命令

命令	レイテンシー[1]				スループット			
DisplayFamily_DisplayModel	**06_4E,, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E 06_2A 06_2D**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E 06_2A 06_2D**
ADDSUBPD/ADDSUBPS	4	3	3	3	0.5	1	1	1
HADDPD xmm1, xmm2	6	5	5	5	2	2	2	2
HADDPS xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPD xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPS xmm1, xmm2	6	5	5	5	2	2	2	2
MOVDDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSHDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSLDUP xmm1, xmm2	1	1	1	1	1	1	1	1

表 D-14 インテル® SSE2 の 128 ビット整数命令

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E 06_2A 06_2D**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E 06_2A 06_2D**
CVTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
CVTTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
MASKMOVDQU xmm, xmm					7	6	6	6
MOVD xmm, r64/r32	2	1	1	1	1	1	1	1
MOVD r64/r32, xmm	2	1	1	1	1	1	1	1
MOVDQA xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVDQU xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVQ xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PACKSSWB/PACKSSDW/ PACKUSWB xmm, xmm	1	1	1	1	1	1	1	0.5
PADDB/PADDW/PADDD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PADDSB/PADDSW/PADDUSB /PADDUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PADDQ/ PSUBQ[3] xmm, xmm1	1	1	1	1	0.33	0.5	0.5	0.5
PAND xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PANDN xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PAVGB/PAVGW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPEQB/PCMPEQD/PCMP EQW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPGTB/PCMPGTD/PCMP GTW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRW r32, xmm, imm8	3	3	3	3	1	1	1	1
PINSRW xmm, r32, imm8	2	2	2	2	2	2	2	1
PMADDWD xmm, xmm	5[b]	5	5	5	0.5	1	1	1
PMAX xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMIN xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMOVMSKB[3] r32, xmm	2	2	2	2	1	1	1	1
PMULHUW/PMULHW/PMUL LW xmm, xmm	5[b]	5	5	5	0.5	1	1	1
PMULUDQ xmm, xmm	5[b]	5	5	5	0.5	1	1	1
POR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PSADBW xmm, xmm	3	5	5	5	1	1	1	1
PSHUFD xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFHW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFLW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLDQ xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLW/PSLLD/PSLLQ xmm, imm8	1	1	1	1	1	1	1	1
PSLL/PSRL xmm, xmm	2	2	2	2	1	1	1	1
PSRAW/PSRAD xmm, imm8	1	1	1	1	1	1	1	1
PSRAW/PSRAD xmm, xmm	2	2	2	2	1	1	1	1

命令レイテンシーとスループット

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E 06_2A 06_2D**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E 06_2A 06_2D**
PSRLDQ xm m, imm8	1	1	1	1	1	1	1	0.5
PSRLW/PSRLD/PSRLQ xmm, imm8	1	1	1	1	1	1	1	1
PSUBB/PSUBW/PSUBD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PSUBSB/PSUBSW/PSUBUSB /PSUBUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PUNPCKHBW/PUNPCKHWD /PUNPCKHDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKHQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLBW/PUNPCKLWD/ PUNPCKLDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PXOR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33

b: バイパスによる 1 サイクルのバブルを含みます

表 D-15 インテル® SSE2 の浮動小数点命令

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E (06_3A 06_3E)**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_2A 06_2D (06_3A 06_3E)**
ADDPD xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSD xmm, xmm	4	3	3	3	0.5	1	1	1
ANDNPD xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPD xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
CMPSD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
COMISD xmm, xmm	2	2	2	2	1	1	1	1
CVTDQ2PD xmm, xmm	5	4	4	4	1	1	1	1
CVTDQ2PS xmm, xmm	4	3	3	3	1	1	1	1
CVTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1
CVTPD2PS xmm, xmm	5	4	4	4	1	1	1	1
CVT[T]PS2DQ xmm, xmm	4	3	3	3	1	1	1	1
CVTPS2PD xmm, xmm	5	2	2	2	1	1	1	1
CVT[T]SD2SI r64/r32, xmm	6	4	4	5	1	1	1	1
CVTSD2SS xmm, xmm	5	4	4	4	1	1	1	1
CVTSI2 SD xmm, r64/r32	5	3	3	4	1	1	1	1
CVTSS2SD xmm, xmm	5	2	2	2	1	1	1	1
CVTTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1
CVTTSD2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPD xmm, xmm[1]	14	<14	14-20	16-22 (15-20)	4	8	13	22 (14)

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E (06_3A 06_3E)**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_2A 06_2D (06_3A 06_3E)**
DIVSD xmm, xmm	14	<14	14-20	16-22 (15-20)	4	5	13	22 (14)
MAXPD xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSD xmm, xmm	4	3	3	3	0.5	1	1	1
MINPD x mm, xmm	4	3	3	3	0.5	1	1	1
MINSD xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MOVMSKPD r64/r32, xmm	2	2	2	2	1	1	1	1
MOVSD xmm, xmm	1	1	1	1	1	1	1	1
MOVUPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MULPD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
MULSD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
ORPD xmm, xmm	1	1	1	1	0.33	1	1	1
SHUFPD xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPD xmm, xmm[2]	18	20	20	22 (21)	6	13	13	22 (14)
SQRTSD xmm, xmm	18	20	20	22 (21)	6	7	13	22 (14)
SUBPD xmm, xmm	4	3	3	3	0.5	1	1	1
SUBSD xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISD xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPD xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPD xmm, xmm	1	1	1	1	1	1	1	1
XORPD[3] xmm, xmm	1	1	1	1	0.33	1	1	1

注意:

1. DIVPD/DIVSD のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは ~6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。
2. SQRTPD/SQRTSD のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは ~6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。

表 D-16 インテル® SSE の浮動小数点命令

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_2A 06_2D (06_3A 06_3E)**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_2A 06_2D (06_3A 06_3E)**
ADDPS xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSS xmm, xmm	4	3	3	3	0.5	1	1	1
ANDNPS xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPS xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPS xmm, xmm	4	3	3	3	0.5	1	1	1
CMPSS xmm, xmm	4	3	3	3	0.5	1	1	1
COMISS xmm, xmm	2	2	2	2	1	1	1	1
CVTSI2SS xmm, r32	6	4	4	5	1	1	1	1
CVTSS2SI r32, xmm	6	4	4	5	1	1	1	1
CVT[T]SS2SI r64, xmm	6	4	4	5	1	1	1	1
CVTTSS2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPS xmm, xmm[1]	11	<11	<13	10-14	3	4	6	14 (6)
DIVSS xmm, xmm	11	<11	<13	10-14	3	2.5	6	14 (6)
MAXPS xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSS xmm, xmm	4	3	3	3	0.5	1	1	1
MINPS xmm, xmm	4	3	3	3	0.5	1	1	1
MINSS xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MOVHLPS xmm, xmm	1	1	1	1	1	1	1	1
MOVLHPS xmm, xmm	1	1	1	1	1	1	1	1
MOVMSKPS r64/r32, xmm	2	2	2	2	1	1	1	1
MOVSS xmm, xmm	1	1	1	1	1	1	1	1
MOVUPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MULPS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
MULSS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
ORPS xmm, xmm	1	1	1	1	0.33	1	1	1
RCPPS xmm, xmm	4	5	5	5	1	1	1	1
RCPSS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTPS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTSS xmm, xmm	4	5	5	5	1	1	1	1
SHUFPS xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPS xmm, xmm[2]	13	13	13	14	3	7	7	14 (7)
SQRTSS xmm, xmm	13	13	13	14	3	4	7	14 (7)
SUBPS xmm, xmm	4	3	3	3	0.5	1	1	1
SUBSS xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISS xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPS xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPS xmm, xmm	1	1	1	1	1	1	1	1
XORPS xmm, xmm	1	1	1	1	1	1	1	1
LFENCE[3]					6	5	5	4
MFENCE[3]					~40	~35	~35	~35

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_2A 06_2D (06_3A 06_3E)**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_2A 06_2D (06_3A 06_3E)**
SFENCE[3]					7	6	6	5
STMXCSR[3]					1	1	1	1
FXSAVE3					~90	~71	~75	~78

注意:

1. DIVPS/DIVSS のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは ~6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。
2. SQRTPS/SQRTSS のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは ~6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。
3. FXSAVE/LFENCE/MFENCE/SFENCE/STMXCSR のスループットは、デスティネーションが L1 データキャッシュにある状態で計測されます。

表 D-17 汎用命令

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
ADC/SBB reg, reg	1	2	2	2	0.5	1	1	1
ADC/SBB reg, imm	1	2	2	2	0.5	1	1	1
ADD/SUB	1	1	1	1	0.25	0.25	0.25	0.33
AND/OR/XOR	1	1	1	1	0.25	0.25	0.25	0.33
BSF/BSR	3	3	3	3	1	1	1	1
BSWAP	2	2	2	2	0.5	0.5	0.5	1
BT	1	1	1	1	0.5	0.5	0.5	0.5
BTC/BTR/BTS	1	1	1	1	0.5	0.5	0.5	0.5
CBW/CWDE/CDQE	1	1	1	1	1	1	1	1
CDQ	1	1	1	1	1	1	1	1
CQO	1	1	1	1	0.5	0.5	0.5	0.5
CLC					0.5	0.5	0.5	0.5
CMC					0.25	0.33	0.33	0.33
STC					0.25	0.33	0.33	0.33
CLFLUSH[12]					~2-50	~3-50	~3-50	~5-50
CLFLUSHOPT[13]					~2-10	NA	NA	NA
CMOVE/CMOVcc	1	1	2	2	0.5	0.5	0.5	0.5
CMOVBE/NBE/A/NA	2	2	3	3	1	1	1	1
CMP/TEST	1	1	1	1	0.25	0.25	0.25	0.33
CPUID (EAX = 0)					~100	~100	~100	~95
CPUID (EAX != 0)					>200	>200	>200	>200
CMPXCHG r64, r64	5	5	5	5	5	5	5	5
CMPXCHG8B m64	15	8	8	8	15	8	8	8

命令レイテンシーとスループット

命令	レイテンシー[1]				スループット			
CPUID	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_3A 06_3E**
CMPXCHG16B m128	19	10	10	10	19	10	10	10
Lock CMPXCHG8B m64	22	19	19	24	22	19	19	24
Lock CMPXCHG16B m128	32	28	28	29	32	28	28	29
DEC/INC	1	2	2	2	0.25	0.25	0.25	0.33
IMUL r64, r64	3	3	3	3	1	1	1	1
IMUL r64[10]	4,5	3,4	3,4	3,4	1	1	1	1
IMUL r32	5	4	4	4	1	1	1	1
IDIV r64 (RDX!= 0)[8]					~85-100	~85-100	~85-100	~85-100
IDIV r32[9]					~20-26	~20-26	~20-26	~19-25
LEA	1	1	1	1	0.5	0.5	0.5	0.5
LEA [base+index]disp	3	3	3	3	1	1	1	1
MOVSB/MOVSW	1	1	1	1	0.25	0.25	0.25	0.33
MOVZB/MOVZW	1	1	1	1	0.25	0.25	0.25	0.33
DIV r64 (RDX!= 0)[8]					~85-95	~85-95	~85-95	~85-95
DIV r32[9]					~20-26	~20-26	~20-26	~19-25
MUL r64[10]	4,5	3,4	3,4	3,4	1	1	1	1
NEG/NOT	1	2	2	2	0.25	0.25	0.25	0.33
PAUSE					~140	~10	~10	~10
RCL/RCR reg, 1	2	2	2	2	2	1.5	1.5	1.5
RCL/RCR	6	6	6	6	6	6	6	6
RDTSC					~13	~10	~10	~20
RDTSCP					~20	~30	~30	~30
ROL/ROR reg 1	1(2flg)	1(2flg)	1(2flg)	1(2flg)	1	1	1	1
ROL/ROR reg imm1	1	1	1	1	0.5	0.5	0.5	0.5
ROL/ROR reg, cl	2	2	2	2	1.5	1.5	1.5	1.5
LAHF/SAHF	3	2	2	2				
SAL/SAR/SHL/SHR reg, imm	1	1	1	1	0.5	0.5	0.5	0.5
SAL/SAR/SHL/SHR reg, cl	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
SETBE	2	2	2	2	1	1	1	1
SETE	1	1	1	1	0.5	0.5	0.5	0.5
SHLD/RD reg, reg, cl	6	4	4	2(4flg)	1.5	1	1	1.5
SHLD/RD reg, reg, imm	3	3	3	1	0.5	0.5	0.5	0.5
XSAVE[11]					~98	~100	~100	~100
XSAVEOPT[11]					~86	~90	~90	~90
XADD	2	2	2	2	1	1	1	1
XCHG reg, reg	1	1	1	2	1	1	1	1
XCHG reg, mem	22	19	19	19	22	19	19	19

D.3.2 表の脚注について

この付録に収録されている表を参照する際は、以下の点に注意してください。

1. 5 つ以上のマイクロオペレーション (uop) から構成される複合命令の多くは、(最悪の場合を考慮して) 控えめに算出した推測値を示しています。そのような命令をアウトオブオーダー・コア実行ユニットで実行したときの実際のパフォーマンスは、各表に示したレイテンシー・データよりも若干高速なものから、著しく高速なものまで幅があります。
2. コードシーケンスでインテル® 64 命令と IA-32 命令をいくつか繰り返して呼び出すようにすると、この表に列挙した数値よりも 1 ～ 2 サイクルだけレイテンシーが短くなることがあります。
3. 超越関数命令のレイテンシーとスループットは、ダイナミック・エグゼキューション環境では大きく変わることがあります。したがって、超越関数命令については、概算値か、ある程度幅を持った数値のいずれを示しています。
4. コードシーケンスにおける FXCH 命令のレイテンシーはゼロです。ただし、1 クロックサイクルあたり 1 命令しか発行できません。
5. ロード定数命令 FINCSTP と FDECSTP は、コードシーケンスではレイテンシーが 0 です。
6. 分岐の予測的中率を高くするため、条件分岐命令を選択するときは、3.4.1 節「分岐予測の最適化」の推奨事項に従ってください。分岐予測に成功すると、jcc のレイテンシーは事実上ゼロになります。
7. シフト回数が 1 回の RCL/RCR は最適化されています。1 以外のシフト回数の RCL/RCR を使用すると、実行速度が遅くなります。これは、インテル® Pentium® 4 プロセッサーとインテル® Xeon® プロセッサーに適用されます。
8. “DIV/IDIV r64” のレイテンシーとスループットは、RDX:RAX の入力値の有効桁数によって異なります。RDX の入力が 0 の場合、スループットはかなり高くなります。これは、”DIV/IDIV r32” と同様です。RDX が 0 でない場合は、示される範囲よりもかなり低くなります。入力 RDX:RAX (除数の有効ビット数に比例して) または出力商の有効ビット数の増加に従って、スループットは減少 (サイクル数の増加) します。”DIV/IDIV r64” のレイテンシーは、入力値の有効ビットによって異なります。特定セットの入力値では、レイテンシーはサイクルのスループットとほぼ同等です。
9. “DIV/IDIV r32” のスループットは、入力 EDX:EAX および/または、除数 r32 の有効ビットサイズに対する除算の商の有効数値数によって異なります。入力 EDX:EAX または出力商の有効ビット数の増加に従って、スループットは減少 (サイクル数の増加) します。”DIV/IDIV r32” のレイテンシーは、入力値の有効ビットによって異なります。特定セットの入力値では、レイテンシーはサイクルのスループットとほぼ同等です。
10. 128 ビットの結果を生成する MUL r64 のレイテンシーには 2 組の数字がありますが、下位 64 ビットの結果 (RAX) の読み取りから利用までのレイテンシーはより小さくなります。128 ビットの結果の上位 64 ビットのレイテンシー (RDX) の方が大きくなります。
11. XSAVE と XSAVEOPT のスループットは、デスティネーションが L1 データキャッシュにある状態で測定され、YMM ステートを含みます。
12. CLFLUSH のスループットは、バッファーサイズ範囲に対するクリーンなキャッシュラインである場合を示します。CLFLUSH のスループットは、次の要因によって劇的に減少します: (a) 連続して実行される CLDLUSH の数、(b) 変更されたキャッシュラインの排出による、他のコヒーレント状態のキャシュラインに関連する追加のコスト。9.4.6 節を参照してください。
13. CLFLUSHOPT のスループットは、バッファーサイズ範囲に対するクリーンなキャッシュラインである場合を示します。CLFLUSHOPT のスループットは、次の要因によって劇的に減少します。(a) 変更されたキャッシュラインの排出による、他のコヒーレント状態のキャシュラインに関連する追加のコスト、(b) 連続するキャッシュラインの数。9.4.7 節を参照してください。

D.3.3 メモリーオペランドを持つ命令

メモリーオペランドを持つ命令のレイテンシーは、メモリー/キャッシュ階層内のデータの局所性や各マイクロアーキテクチャーに固有の特性などの多くの要因によって異なります。一般に、ソフトウェアにおける局所性のチューニングと命令の選択は、互いに独立したアプローチが可能です。このため、表 C-4 ～ 表 C-18 は命令を選択する目的に使用できます。メモリー/キャッシュ階層内のデータ移動レイテンシーとスループットは、命令のレイテンシーとスループットとは独立して扱うことができます。キャッシュ階層のレイテンシーのデータは、第 2 章に記載されています。

D.3.3.1 ソフトウェアから観察できるメモリー参照のレイテンシー

個別の命令のメモリー参照レイテンシーを測定する場合、観察されるレイテンシーは多くの要因に影響されます。アクセスパターン、キャッシュの局所性、ハードウェア・プリフェッチの効果を除き、異なるマイクロアーキテクチャーでは、命令エンコードに関するデスティネーションやメモリーアドレス形式のレジスタードメインなどの違いが表れる可能性があります。

次の表は、近年のインテル® マイクロアーキテクチャーにおけるメモリー参照のエンコードによる、ポインター追尾構文を使用するソフトウェアの L1D キャッシュヒットのレイテンシーのいくつかを示しています。

表 D-18 ソフトウェアが計測可能なポインター追尾における L1 データ・キャッシュ・レイテンシーの変化

ポインター追尾構文	**観察された L1D レイテンシー**
MOV rax, [rax]	4
MOV rax, disp32[rax] , disp32 < 2048	4
MOV rax, [rcx+rax]	5
MOV rax, disp32[rcx+rax] , disp32 < 2048	5

付録 E
旧世代のインテル® 64 および IA-32 プロセッサー・アーキテクチャー

E.1 Haswell† マイクロアーキテクチャー

Haswell† マイクロアーキテクチャーは、Sandy Bridge† および Ivy Bridge† マイクロアーキテクチャーの成功の上に構築されています。図 E-1 に Haswell† マイクロアーキテクチャーの基本パイプライン機能を示します。一般に、E.1.1 節から E.1.4 節で示される多くの機能は、Broadwell† マイクロアーキテクチャーにも適用されます。Broadwell† マイクロアーキテクチャーの拡張は、E.1.7 節にまとめられています。

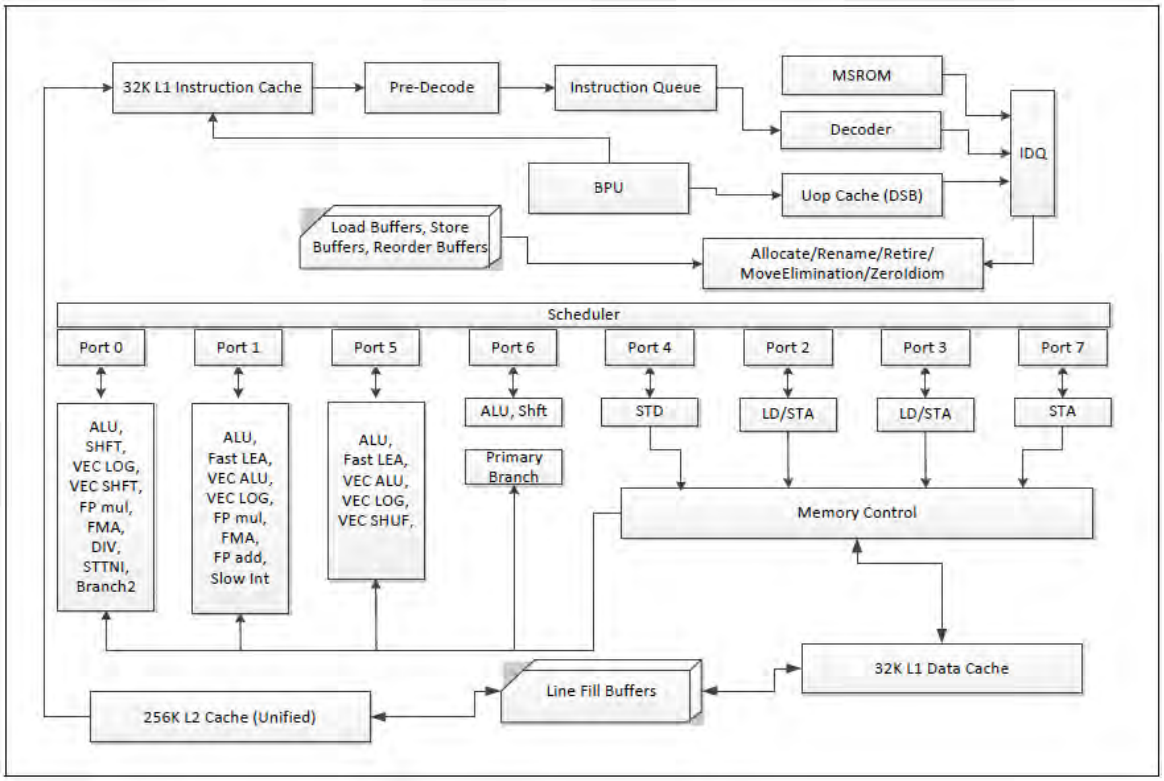


図 E-1 インテル® マイクロアーキテクチャー Haswell† の CPU コア・パイプライン

Haswell† マイクロアーキテクチャーの基本パイプラインは、以下の革新的な機能を提供します。

- インテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2)、FMA のサポート
- 整数値演算と暗号化を高速化する新しい汎用命令
- インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) のサポート
- 各コアでサイクルごとに最大 8 マイクロオペレーション (uop) をディスパッチ可能
- メモリー操作、FMA、インテル® AVX 浮動小数点実行ユニット、インテル® AVX2 整数実行ユニット用の 256 ビット・データ・パス
- L1 データキャッシュと L2 キャッシュの帯域幅が増加
- 2 つの FMA 実行パイプライン
- 4 つの数値演算ユニット (ALU)
- 3 つのストア・アドレス・ポート
- 2 つの分岐実行ユニット
- IA プロセッサー・コアおよびアンコア・サブシステム向けの高度な電力管理機能
- オプションの L4 キャッシュをサポート

インテル® マイクロアーキテクチャー Haswell† は、L3 (オプションでオフダイの L4 も) の複数のスライスへのリング・インターコネクト、プロセッサー・グラフィックス、統合型メモリー・コントローラー、インターコネクト・ファブリックなどを含むいくつかの要素で構成される共有アンコア・サブシステムと、複数のプロセッサー・コアとの柔軟な統合をサポートしています。図 E-2 に、4 CPU コアとアンコア要素で構成されるシステム統合の例を示します。

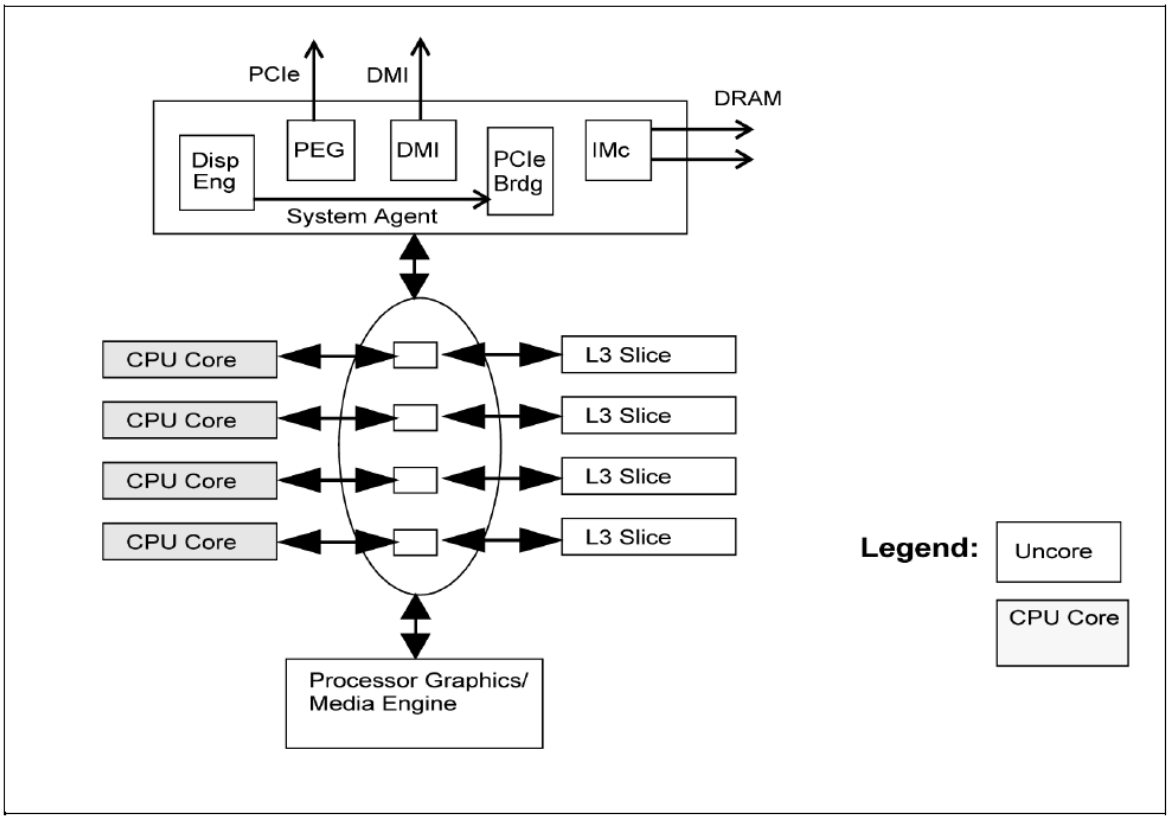


図 E-2 インテル® マイクロアーキテクチャー Haswell† の 4 コアのシステム統合

E.1.1 フロントエンド

インテル® マイクロアーキテクチャー Haswell† のフロントエンドは、インテル® マイクロアーキテクチャー Sandy Bridge† (E.2.2 節) とインテル® マイクロアーキテクチャー Ivy Bridge† (E.2.7 節) をベースに開発され、次の点が拡張されています。

- マイクロオペレーション (uop) キャッシュ (またはデコード済み命令キャッシュ) は、2 つの論理プロセッサー間で均等に分割されます。
- 命令デコーダーは、アクティブな論理プロセッサー間で交互に使用される。1 つの論理プロセッサーがアイドル状態の場合は、もう一方のアクティブな論理プロセッサーがデコーダーを続けて使用されます。
- ループストリーム検出器 (LSD)/マイクロオペレーション (uop) は、56 マイクロオペレーション (uop) までの小さなループを検出できます。56 エントリーのマイクロオペレーション (uop) キューは、インテル® ハイパースレッディング・テクノロジーが有効な場合、2 つの論理プロセッサーによって共有されます (インテル® マイクロアーキテクチャー Sandy Bridge† では、各コアに 28 エントリーのマイクロオペレーション (uop) キューの複製が提供されます)。

E.1.2 アウトオブオーダー・エンジン

以下に、アウトオブオーダー・エンジンの主要構成要素と主な改善点を示します。

リネーマー: リネーマーは、マイクロオペレーション (uop) キューからスケジューラーのディスパッチ・ポートへマイクロオペレーション (uop) を移動し、実行リソースにバインドする。ゼロイディオム、1 イディオム、ゼロレイテンシーのレジスター移動命令はリネーマーによって実行され、スケジューラーと実行コアを解放することでパフォーマンスを向上できます。

スケジューラー: スケジューラーは、ディスパッチ・ポートへのマイクロオペレーション (uop) のディスパッチを制御します。アウトオブオーダー実行コアをサポートするため 8 つのディスパッチ・ポートがあり、そのうち 4 つは計算処理用の実行リソースを提供し、残り 4 つは 1 サイクルで最大 2 つの 256 ビット・ロード操作と 1 つの 256 ビット・ストア操作をサポートします。

実行コア: スケジューラーは、各ポートで 1 つずつ、サイクルごとに最大 8 つのマイクロオペレーション (uop) をディスパッチできます。計算リソースを提供する 4 つのポートには ALU が 1 つずつあり、実行パイプのうち 2 つは FMA ユニット専用です。除算/平方根を除き、STTNI (String and Text New Instructions)/インテル® AES-NI (Advanced Encryption Standard New Instructions) ユニット、ほとんどの浮動小数点および整数 SIMD 実行ユニットは 256 ビット幅です。メモリー操作用の 4 つのディスパッチ・ポートは、2 つのロード/ストアアドレス操作用のデュアルユース・ポート、ストアアドレス専用のポート、1 つのストアデータ専用ポートで構成されており、すべてのポートで 256 ビットのメモリー・マイクロオペレーション (uop) を処理できます。浮動小数点のピーク・スループットは、FMA を使用した場合、単精度では 1 サイクルあたり 32 マイクロオペレーション (uop)、倍精度では 16 マイクロオペレーション (uop) であり、インテル® マイクロアーキテクチャー Sandy Bridge† の 2 倍です。

アウトオブオーダー・エンジンは、同時に 192 マイクロオペレーション (uop) を処理できます (インテル® マイクロアーキテクチャー Sandy Bridge† では 168 マイクロオペレーション (uop) です)。

E.1.3 実行エンジン

表 E-1 に、各ポートでディスパッチ可能なマイクロオペレーション (uop) を示します。

表 E-1 ディスパッチ・ポートと実行スタック

ポート 0	ポート 1	ポート 2、3	ポート 4	ポート 5	ポート 6	ポート 7
ALU、Shift	ALU、 Fast LEA、	Load_Addr、 Store_addr	Store_data	ALU、Fast LEA	ALU、 Shift、 JEU	Store_addr、 Simple_AGU
SIMD_Log、 STTNI、 SIMD_Shifts	SIMD_ALU、 SIMD_Log			SIMD_ALU、 SIMD_Log		
FMA/FP_mul、 Div	FMA/FP_mul、 FP_add			FP/Int Shuffle		
2nd_Jeu	slow_int					

表 E-2 は、実行ユニットとこれらのユニットに関連する代表的な命令を示します。表 E-2 はまた、Broadwell† マイクロアーキテクチャー・ベースのプロセッサーのみで利用可能な命令をいくつか含んでいます。

表 E-2 Haswell† マイクロアーキテクチャーの実行ユニットと主要な命令

Execution Unit	# of Ports	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa
SHFT	2	sal, shl, rol, adc, sarx, (adcx, adox)[1] etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
SIMD Log	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)blendp*, vpblendd
SIMD_Shft	1	(v)psl*, (v)psr*
SIMD ALU	2	(v)padd*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)pcmpgt*
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)pblendw
SIMD Misc	1	(v)pmul*, (v)pmadd*, STTNI, (v)pclmulqdq, (v)psadw, (v)pcmpgtq, vpsllvd, (v)bendv*, (v)plendw,
FP Add	1	(v)addp*, (v)cmpp*, (v)max*, (v)min*,
FP Mov	1	(v)movap*, (v)movup*, (v)movsd/ss, (v)movd gpr, (v)andp*, (v)orp*
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

注意:

1. Broadwell† マイクロアーキテクチャー・ベースと CPUID ADX 機能フラグをサポートするプロセッサーのみで利用可能。

リザベーション・ステーション (RS) が 60 エントリーに拡大され (インテル® マイクロアーキテクチャー Sandy Bridge† では 54 エントリー)、マイクロオペレーション (uop) の実行準備ができている場合、サイクルごとに最大 8 つのマイクロオペレーション (uop) をディスパッチできます。RS でマイクロオペレーション (uop) は特定のデータ型やデータの粒度を処理するスタックに分けられ、発行ポートから特定の実行クラスターにディスパッチされます。

あるスタックで実行されるマイクロオペレーション (uop) のソースが、別のスタックで実行されるマイクロオペレーション (uop) から取得される場合、遅延が生じる可能性があります。インテル® SSE 整数操作とインテル® SSE 浮動小数点操作の間の遷移でも遅延が発生します。これは、命令フローに追加されるマイクロオペレーション (uop) によって、データ遷移が行われるためです。実行後にライトバックされるデータを、後続のマイクロオペレーション (uop) 実行にバイパスする方法とその遅延サイクル数を表 2-30 に示します。

表 E-3 マイクロオペレーション (uop) 間のバイパスによる遅延 (サイクル数)

遷移元/遷移先	整数	SSE-INT/ AVX-INT	SSE-FP/ AVX-FP_LOW	X87/ AVX-FP_High
整数		• uOP (ポート 5) • uOP (ポート 6) + 1 サイクル遅延	• uOP (ポート 5) • uOP (ポート 6) + 1 サイクル遅延	uOP (ポート 5) + 3 サイクル遅延
SSE-INT/ AVX-INT	uOP (ポート 1)		1 サイクル遅延	
SSE-FP/ AVX-FP_LOW	uOP (ポート 1)	1 サイクル遅延		uOP (ポート 5) + 1 サイクル遅延
X87/ AVX-FP_High	uOP (ポート 1) + 3 サイクル遅延		uOP (ポート 5) + 1 サイクル遅延	
ロード		1 サイクル遅延	1 サイクル遅延	2 サイクル遅延

E.1.4 キャッシュとメモリー・サブシステム

キャッシュ階層は前世代と類似しており、各コアに L1 命令キャッシュ、L1 データキャッシュ、L2 ユニファイド・キャッシュがあります。さらに、L3 ユニファイド・キャッシュもあり、そのサイズは製品構成に依存します。L3 キャッシュは複数のキャッシュスライスで構成されており、各スライスのサイズはリング・インターコネクトで接続される製品構成に依存します。キャッシュトポロジーの詳細は、CPUID leaf 4 で確認できます。L3 キャッシュは、すべてのプロセッサー・コアで共有される “アンコア” サブシステムにあります。一部の製品構成では L4 キャッシュもサポートされています。表 E-23 にキャッシュ階層の詳細を示します。

表 E-4 インテル® マイクロアーキテクチャー Haswell† のキャッシュ・パラメーター

レベル	容量/アソシアティブ (ウェイ)	ラインサイズ (バイト)	最小レイテンシー[1]	スループット (クロック数)	ピーク帯域幅 (バイト/サイクル数)	アップデート方式
L1 データ	32KB/8	64	4 サイクル	0.5[2]	64 (ロード) + 32 (ストア)	ライトバック
命令	32KB/8	64	なし	なし	なし	なし
L2	256KB/8	64	11 サイクル	それぞれ異なる	64	ライトバック
L3 (共有)	それぞれ異なる	64	~ 34	それぞれ異なる		ライトバック

注意:

1. ソフトウェアから検知できるレイテンシーは、アクセスパターンやその他の要因により異なります。
2. L1 データキャッシュは、最大 32 バイトのデータをフェッチ可能なロード操作を各サイクルで 2 つ処理できます。

TLB (Translation Lookaside Buffer) 階層は、L1 命令キャッシュ用の TLB、L1 データキャッシュ用の TLB、L2 ユニファイド・キャッシュ用の TLB で構成されます。

表 E-5 インテル® マイクロアーキテクチャー Haswell† の TLB パラメーター

レベル	ページサイズ	エントリー	アソシアティブ (ウェイ)	パーティション
命令	4KB	128	4 ウェイ	動的
命令	2MB/4MB	スレッドあたり 8		固定
L1 データ	4KB	64	4	固定
L1 データ	2MB/4MB	32	4	固定
L1 データ	1GB	4	4	固定
L2	4KB、2MB/4MB ページで共有	1024	8	固定

E.1.4.1 ロード操作とストア操作の拡張

L1 データキャッシュは各サイクルで 2 つの 256 ビット・ロード操作と 1 つの 256 ビット・ストア操作を処理でき、L2 ユニファイド・キャッシュは 各サイクルで 1 つのキャッシュライン (64 バイト) を処理できます。さらに、マイクロオペレーション (uop) の同時実行をサポートするため、72 のロードバッファーと 42 のストアバッファーが装備されています。

E.1.5 アンラミネーション

いくつかのマイクロフューズされた命令は、単一の uop として割り当てできないため、マイクロ・オペレーション・キューでは 2 つの uop に分解されます。フューズ (融合) された命令を uop に分解する処理は、アンラミネーションと呼ばれます。

フューズされた命令のソースが 3 つ以上である場合、アンラミネーションが行われます。

アンラミネーションされるコンテキストの命令ソースは、メモリー・アドレス・ベース、メモリー・アドレス・インデックス、ソースレジスター、デスティネーション・レジスター (フラグを含む)、またはソースとデスティネーション・レジスターのいずれかであると考えられます。

アンラミネーション・コンテキストのメモリーオペランドは、最大 2 つのソースを持ちます。x86 命令セットのメモリーアドレスは、base + index * scale + displacement で構成されます。

ベース (base) とインデックス (index) のみが命令ソースとしてカウントされます。インデックスが使用される場合、ベースが指定されていなくともベースとしてカウントされることに注意してください。

さらに、ソースとデスティネーション・レジスターは 2 つのソースとしてカウントされます。これもまた、ソースとデスティネーション・レジスターが同じである場合にも当てはまります。

次の表は、マイクロフューズ命令の例とアンラミネーションの詳細を示します。

表 E-6 フロントエンドの構成要素

命令の例	ソース	ディスティネーション	ソースとディスティネーション	インデックス	ベース	ソース[1]の数	アンラミネーション
mulss xmm1, [4*rax+100]	-	-	xmm1	rax	0	3	なし
vmulss xmm1, xmm1, [rax +100]	xmm1	xmm1	-	-	rax	3	なし
vmulss xmm1, xmm1, [4*rax+100]	xmm1	xmm1	-	rax	0	4	あり
cmp rax, [rbx+4*rax+4]	rax	フラグ	-	rax	rbx	4	あり
cmp rax, [rbx+4]	rax	フラグ	-	-	rbx	3	なし

注意:

1. 推奨事項: アンラミネーションを避けるには、マイクロフューズ命令のソースを 4 以下に保ちます。

E.1.6 Haswell-E† マイクロアーキテクチャー

Haswell-E† マイクロアーキテクチャーは、Haswell† マイクロアーキテクチャーで説明した同じプロセッサー・コアを包括するインテル® プロセッサーをベースとしていますが、より高度なアンコアと統合 I/O 能力を提供します。Haswell-E† マイクロアーキテクチャー・ベースのプロセッサーは、複数ソケットのプラットフォームをサポートします。

Haswell-E† マイクロアーキテクチャーは、スケーラビリティーと高いパフォーマンス向けに多面的なプロセッサー・アーキテクチャーをサポートします。Haswell-E† マイクロアーキテクチャーのアンコアと統合 I/O サブシステムで提供される機能を以下に示します。

- 複数ソケット構成で複数のインテル® QuickPath インターコネクトをサポートします。
- 物理コアごとに最大 2 つのメモリー・コントローラーが統合されました。
- 物理プロセッサーごとに最大 40 レーンの PCI Express* 3.0 リンクを提供します。
- それぞれの物理プロセッサーで、最大 18 個のプロセッサー・コアが 2 つのリング・インターコネクトによって L3 接続されます。

図 E-3 に、Haswell-E† マイクロアーキテクチャーを使用する 12 コアのプロセッサー実装の例を示します。アンコアと統合 I/O サブシステムの能力は、Haswell-E† マイクロアーキテクチャーを実装するプロセッサー・ファミリーごとに異なります。詳細については、インテル® Xeon® プロセッサー E5 v3 ファミリーのデータシートを参照してください。

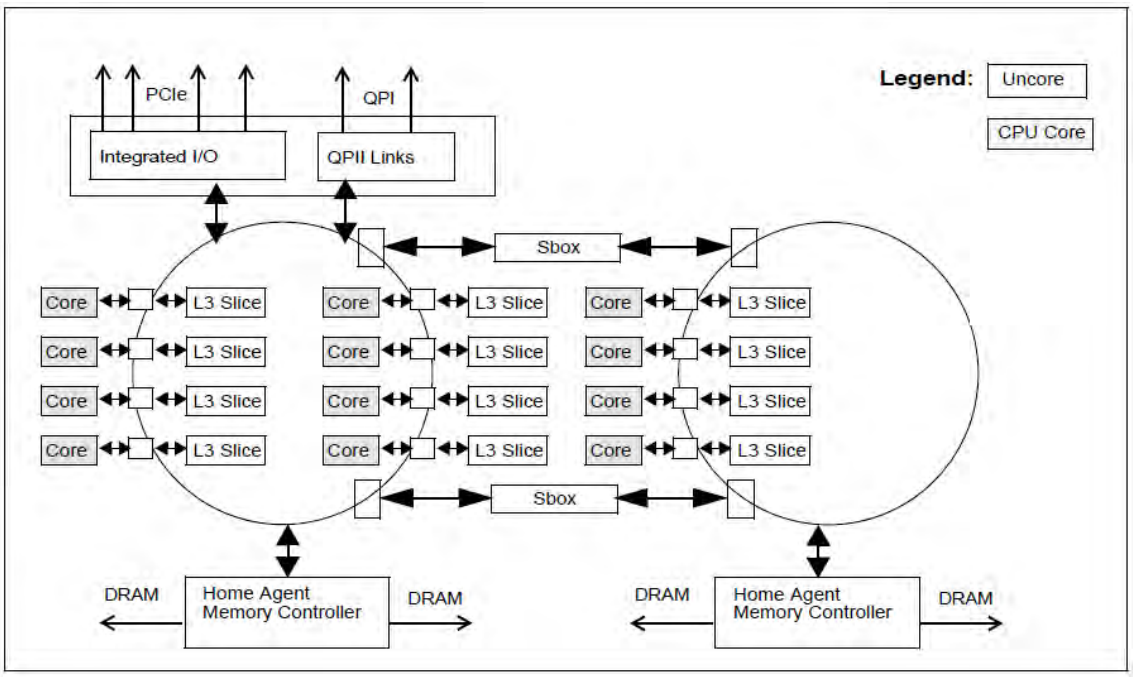


図 E-3 12 個のプロセッサー・コアをサポートする Haswell-E† マイクロアーキテクチャーの例

E.1.7 Broadwell† マイクロアーキテクチャー

インテル® Core™ M プロセッサーは、Broadwell† マイクロアーキテクチャーをベースにしています。Broadwell† マイクロアーキテクチャーは、Haswell† マイクロアーキテクチャーから派生し、いくつかの拡張を実装しています。このセクションでは、Broadwell† マイクロアーキテクチャーの拡張機能について説明します。

- 浮動小数点乗算命令のレイテンシーが、前の世代の 5 サイクルから Broadwell† マイクロアーキテクチャーでは 3 サイクルに改善されています。これは、インテル® AVX、インテル® SSE、および FP 命令セットに適用されます。
- ギャザー命令のスループットが大幅に向上しました。表 D-5 を参照してください。
- Broadwell† マイクロアーキテクチャーでは、PCLMULQDQ 命令が単一 uop で実装されており、レイテンシーとスループットが改善されました。

TLB の階層は、命令キャッシュ向けの TLB、L1D 向けの TLB、さらに L2 向けのユニファイド TLB から成ります。

表 E-7 Broadwell† マイクロアーキテクチャーの TLB パラメーター

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2MB pages	1536	6	fixed
Second Level	1GB pages	16	4	fixed

E.2 インテル® マイクロアーキテクチャー Sandy Bridge†

インテル® マイクロアーキテクチャー Sandy Bridge† は、インテル® Core™ マイクロアーキテクチャーおよびインテル® マイクロアーキテクチャー Nehalem† の成功を受けて開発されました。以下の革新的な機能を提供しています。

- インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX)
  - 128 ビットのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) の 256 ビット浮動小数点命令セットへの拡張により、128 ビットのコードと比べパフォーマンスが最大 2 倍向上
  - 非破壊デスティネーションの採用により、より柔軟なコーディング手法を実現
  - 256 ビットのインテル® AVX コード、128 ビットのインテル® AVX コード、128 ビットのインテル® SSE レガシーコード間の柔軟な移行および共存をサポート

- 機能が強化されたフロントエンドおよび実行エンジン
  - 新たなデコード済み命令キャッシュの実装により、フロントエンドの帯域幅を向上および分岐の予測ミスのペナルティーを軽減
  - 高度な分岐予測
  - マクロフュージョンの追加サポート
  - ダイナミック・エグゼキューション範囲の拡大
  - マルチ精度整数算術の拡張 (ADC/SBB、MUL/IMUL)
  - LEA 帯域幅の向上
  - 一般的な実行ストール (読み出しポート、ライトバック競合、バイパス・レイテンシー、パーシャルストール) の削減
  - 高速な浮動小数点例外処理
  - XSAVE/XRSTORE 命令のパフォーマンスの向上、および新しい XSAVEOPT 命令の追加

- キャッシュ階層の改善によるデータパスの拡大
  - メモリー操作のための 2 つのシンメトリックなポートにより、帯域幅が倍増
  - バッファー増加により、より多くの実行中のロードおよびストアを同時に操作
  - 各サイクルで 2 つのロードと 1 つのストアを実行可能な内部帯域幅
  - プリフェッチの改善
  - 高帯域幅および低レイテンシーの LLC アーキテクチャー
  - オンダイ・インターコネクトの高帯域幅リング・アーキテクチャー

- システムオンチップのサポート
  - 第 2 世代インテル® Core™ プロセッサーでグラフィックス・エンジンとメディアエンジンの統合
  - PCIe* コントローラーの統合
  - メモリー・コントローラーの統合

- インテル® ターボ・ブースト・テクノロジー 2.0
  - TDP ヘッドルームの改善により、CPU コアおよび内蔵グラフィックス・ユニットのパフォーマンスを向上

E.2.1 インテル® マイクロアーキテクチャー Sandy Bridge† のパイプライン概要

図 E-4 に、インテル® マイクロアーキテクチャー Sandy Bridge† をベースとするプロセッサー・コアのパイプラインおよび主要構成要素を示します。パイプラインは以下で構成されています。

- 命令をフェッチし、マイクロオペレーション (uop) にデコードするインオーダー発行フロントエンド。フロントエンドは、プログラムが実行する可能性の最も高いパスからマイクロオペレーション (uop) の連続的なストリームを次のパイプライン・ステージに供給します。

- サイクルあたり最大 6 つのマイクロオペレーション (uop) をディスパッチするアウトオブオーダー・スーパースケーラー実行エンジン。入力ソースが準備でき、実行リソースが利用可能になり次第実行できるように、割り当て/リネームブロックがマイクロオペレーション (uop) を「データフロー」の順序に並べ替えます。
- 検出された例外など、マイクロオペレーション (uop) の実行結果が元のプログラム順序になることを確実にするインオーダー・リタイアメント・ユニット。

パイプライン中の命令の流れは以下に要約できます。

1. 分岐予測ユニットは、プログラムから次に実行するコードブロックを選択します。プロセッサーは、次のリソース内で (順番どおりに) コードを検索します。

    a. デコード済み命令キャッシュ
    b. レガシーのデコード・パイプラインによってデコードされた命令キャッシュ
    c. 必要に応じて、L2 キャッシュ、ラスト・レベル・キャッシュ (LLC) およびメモリー

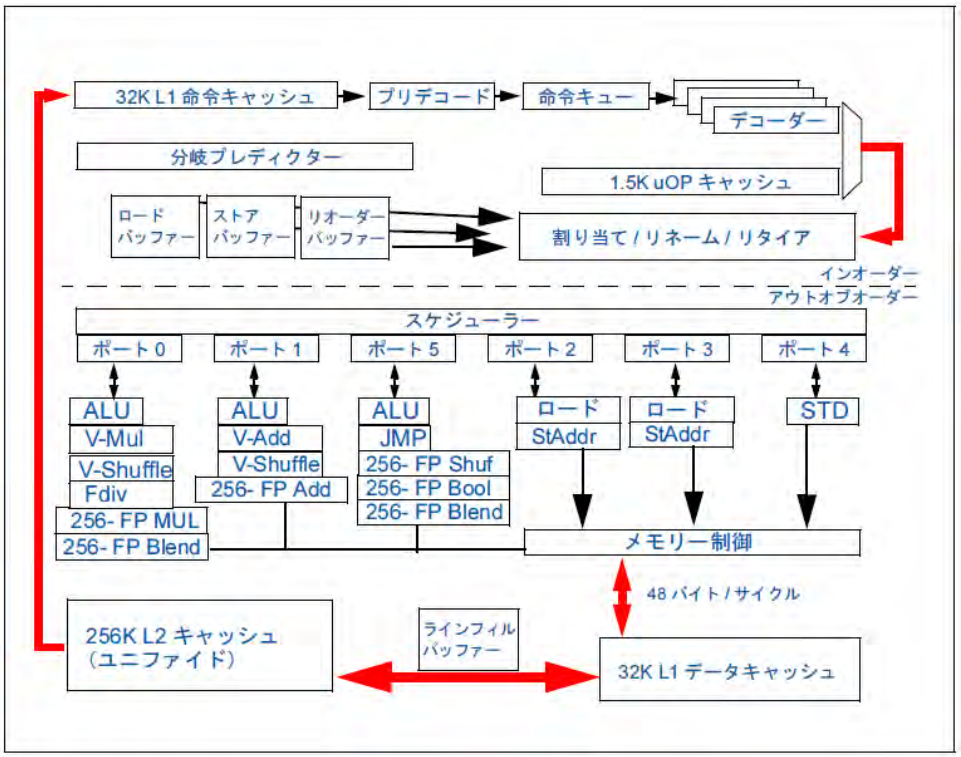


図 E-4 インテル® マイクロアーキテクチャー Sandy Bridge† のパイプラインの構造

2. コードに対応するマイクロオペレーション (uop) がリネーム/リタイアメント・ブロックに送信されます。プログラムの順番どおりにスケジューラーに入りますが、データフロー順序に従ってスケジューラーから実行そして割り当て解除されます。同時に準備できたマイクロオペレーション (uop) は、一般に FIFO 順序で操作されます。3 つのスタックに配置されている実行リソースを使用して、マイクロオペレーション (uop) が実行されます。各スタックの実行ユニットは、命令のデータ型に関連付けられます。

    分岐の予測ミスは分岐実行時に通知されます。正しいパスからマイクロオペレーション (uop) を供給するようにフロントエンドに指示されます。プロセッサーは、分岐の予測ミスよりも前の処理と正しいパスからの処理をオーバーラップできます。

3. 並列性および最高のパフォーマンスを実現するため、メモリー操作が管理およびリオーダーされます。L1 データキャッシュにミスがあった場合、L2 キャッシュを参照します。データキャッシュはノンブロッキングで、複数のミスを同時に処理できます。
4. 例外 (フォルト、トラップ) は、フォルトが発生した命令のリタイアメント (または、リタイアメントの試行) 時に通知されます。

インテル® ハイパースレッディング・テクノロジーが有効であれば、インテル® マイクロアーキテクチャー Sandy Bridge† をベースとする各プロセッサー・コアは、2 つの論理プロセッサーをサポートできます。

E.2.2 フロントエンド

この節では、フロントエンドの主な特性について説明します。表 E-8 に、フロントエンドの構成要素、その機能、扱う問題の一覧を示します。

表 E-8 インテル® マイクロアーキテクチャー Sandy Bridge† のフロントエンドの構成要素

構成要素	機能	パフォーマンスの課題
命令キャッシュ	命令バイトの 32K バイトのバッキングストア	ホットな命令バイトへの高速アクセス
レガシー・デコード・パイプライン	マイクロオペレーション（μOP）キューおよびデコード済み命令キャッシュに供給された命令をマイクロオペレーション（μOP）にデコードする。	レガシーのインテル® プロセッサーと同じデコード・レイテンシーおよび帯域幅を提供する。 デコード済み命令キャッシュのウォームアップ
デコード済み命令キャッシュ	マイクロオペレーション（μOP）キューにマイクロオペレーション（μOP）のストリームを供給する。	レガシー・デコード・パイプラインよりも低いレイテンシーおよび低い消費電力でより高いマイクロオペレーション（μOP）帯域幅を提供する。
MSROM	レガシー・デコード・パイプラインとデコード済み命令キャッシュの両方からアクセス可能な、複雑な命令のマイクロオペレーション（μOP）のセットを保持	
分岐予測ユニット（BPU）	次に実行されるコードブロックを判断し、デコード済み命令キャッシュとレガシー・デコード・パイプラインの参照を推進する。	分岐の予測ミスを軽減することで、パフォーマンスおよび電力効率を向上する。
マイクロオペレーション（μOP）キュー	デコード済み命令キャッシュとレガシー・デコード・パイプラインからマイクロオペレーション（μOP）をキューに格納する。	フロントエンド・バブルの隠蔽。一定の速度で実行マイクロオペレーション（μOP）を提供する。

E.2.2.1 レガシー・デコード・パイプライン

レガシー・デコード・パイプラインは、命令トランスレーション・ルックアサイド・バッファー (ITLB)、命令キャッシュ (ICache)、命令プリデコーダー、命令デコードユニットで構成されます。

命令キャッシュと ITLB

命令フェッチは、ITLB を通じて命令キャッシュに対して 16 バイト境界で参照します。命令キャッシュは、命令プリデコーダーにサイクルごとに 16 バイトを供給できます。表 E-9 に、命令キャッシュと ITLB の前世代との比較を示します。

表 E-9 インテル® マイクロアーキテクチャー Sandy Bridge† の命令キャッシュと ITLB

構成要素	インテル® マイクロアーキテクチャー Sandy Bridge	インテル® マイクロアーキテクチャー Nehalem
命令キャッシュのサイズ	32K バイト	32K バイト
命令キャッシュの連想性（ウェイ）	8	4
ITLB 4K ページエントリー数	128	128
ITLB ラージページ（2M または 4M）エントリー数	8	7

ITLB でミスが生じた場合、DTLB と ITLB に共通の第 2 レベルの TLB (STLB) をルックアップします。ITLB ミスおよび STLB ヒットのペナルティーは 7 サイクルです。

命令プリデコーダー

プリデコード・ユニットは、命令キャッシュから 16 バイトを受け入れて、命令長を判断します。

以下のレングス変更プリフィクス (LCP) は、デフォルトの命令長と異なる命令長を意味します。そのため、命令長のデコード時に LCP あたり 3 サイクルの追加ペナルティーが発生します。以前のプロセッサーでは、1 つまたは複数の LCP を含む各 16 バイト・チャンクに対して 6 サイクル・ペナルティーが発生します。通常、16 バイト・チャンクに含まれる LCP は 1 つだけであるため、ほとんどの場合、インテル® マイクロアーキテクチャー Sandy Bridge† では、以前のプロセッサーと比べペナルティーは改善されています。

- オペランド・サイズ・オーバーライド (66H): ワード/ダブルワード即値データを持つ命令の先頭に付きます。このプリフィクスは、コードが 32 ビットより小さい 16 ビット・データ型、Unicode 処理、画像処理を使用している場合に出現します。
- アドレス・サイズ・オーバーライド (67H): リアルモード、ビッグリアル・モード、16 ビット保護モード、または 32 ビット保護モードで、mod r/m を持つ命令の先頭に付きます。このプリフィクスは、ブート・コード・シーケンスに出現します。
- インテル® 64 命令セットの REX プリフィクス (4xh) は、2 つの命令 (MOV オフセットと MOV 即値) のサイズを変更できます。ただし、LCP のペナルティーは発生しないので、LCP とはみなされません。

命令デコード

命令をマイクロオペレーション (uop) にデコードする 4 つのデコードユニットで構成されます。最初のデコーダーは、最大で 4 つのマイクロオペレーション (uop) で構成されるすべての IA-32 命令とインテル® 64 命令をデコードします。残りの 3 つのデコードユニットは、単一マイクロオペレーション (uop) 命令を処理します。4 つすべてのデコードユニットは、マイクロフュージョンやマクロフュージョンなど、一般的なケースにおける単一マイクロオペレーション (uop) のフローをサポートしています。

デコーダーから発行されたマイクロオペレーション (uop) は、マイクロオペレーション (uop) キューとデコード済み命令キャッシュに送られます。4 マイクロオペレーション (uop) よりも長い命令は、MSROM からマイクロオペレーション (uop) が生成されます。MSROM からの帯域幅は 1 サイクルあたり 4 マイクロオペレーション (uop) です。MSROM からマイクロオペレーション (uop) が生成される命令は、レガシー・デコード・パイプラインまたはデコード済み命令キャッシュから開始できます。

マイクロフュージョン

マイクロフュージョンでは、同一命令の複数のマイクロオペレーション (uop) が単一の複雑なマイクロオペレーション (uop) に融合されます。複雑なマイクロオペレーション (uop) は、マイクロフュージョンされなかった場合と同じ回数、アウトオブオーダー実行コアにディスパッチされます。

マイクロフュージョンを利用すると、デコード帯域幅を損ねることなく、複合命令セット・コンピューター (CISC) 命令セットを使いメモリーとレジスター間の操作を行い、実際のプログラム動作を表現できます。マイクロフュージョンでは、デコードからリタイアメントに供給される命令帯域幅が向上し、消費電力が減少します。

単一マイクロオペレーション (uop) 命令を使用して命令シーケンスをコーディングすると、コードサイズが大きくなり、レガシー・パイプラインからのフェッチ帯域幅を圧迫します。

すべてのデコーダーで処理が可能なマイクロフュージョンされたマイクロオペレーション (uop) の例を以下に示します。

- 即値のストアを含めた、メモリーに対するすべてのストア。ストアは、内部でストア-アドレスとストア-データの 2 つの異なる機能として実行されます。

- ロード操作と演算操作を組み合わせたすべての命令 (ロード + op)。
  例:
  – ADDPS XMM9, OWORD PTR [RSP+40]
  – FADD DOUBLE PTR [RDI+RSI*8]
  – XOR RAX, QWORD PTR [RBP+32]

- 「ロードおよびジャンプ」形式のすべての命令。
  例:
  – JMP [RDI+200]
  – RET

- 即値とメモリーオペランドを持つ CMP および TEST 命令。

RIP 相対アドレス指定を行う命令は、以下の場合にはマイクロフュージョンの対象になりません。

- 即値の追加が必要な場合。
  例:
  – CMP [RIP+400], 27
  – MOV [RIP+3000], 142

- 命令が、RIP 相対アドレス指定を使用して間接ターゲットが指定された制御フロー命令である場合。
  例:
  – JMP [RIP+5000000]

このような場合、マイクロフュージョンできない命令により、デコーダー 0 は 2 つのマイクロオペレーション (uop) を発行する必要があり、デコード帯域幅の損失を招きます。

64 ビット・コードでは、グローバルデータに RIP 相対アドレス指定を使用するのが一般的です。このようなケースではマイクロフュージョンできないため、32 ビット・コードを 64 ビット・コードに移植する際にパフォーマンスが低下することがあります。

マクロフュージョン

マクロフュージョンでは、2 つの命令が単一のマイクロオペレーション (uop) にマージされます。インテル® Core™ マイクロアーキテクチャーでは、このハードウェアによる最適化は、マクロフュージョン可能な命令ペアの組み合わせ (1 番目と 2 番目) に特定の条件があります。

- マクロフュージョンされるペアの 1 番目の命令はフラグを変更します。以下の命令はマクロフュージョン可能です。
  – インテル® マイクロアーキテクチャー Nehalem†: CMP, TEST
  – インテル® マイクロアーキテクチャー Sandy Bridge†: CMP, TEST, ADD, SUB, AND, INC, DEC
  – 以下の場合に、これらの命令をマクロフュージョンできます。
    ・ 1 番目のソース/ デスティネーション・オペランドがレジスターである。
    ・ 2 番目のソースオペランド (存在する場合) が即値、レジスター、非 RIP 相対メモリーのいずれかである。

- マクロフュージョン可能なペアの 2 番目の命令は条件分岐です。表 3-1 にマクロフュージョン可能な分岐との組み合わせを示します。

1 番目の命令がキャッシュラインのバイト 63 で終了し、2 番目の命令が次のキャッシュラインのバイト 0 で始まる条件分岐である場合、マクロフュージョンは行われません。

このような命令ペアは多くのアプリケーションに含まれる可能性があるため、再コンパイルされない既存のバイナリーでも、マクロフュージョンによってパフォーマンスが向上します。

マクロフュージョンされたそれぞれの命令は、単一のディスパッチで実行されます。これによって、レイテンシーが減少し、実行リソースが解放されます。また、リネーム/リタイア帯域幅の向上、仮想ストレージの拡大、少ないビット数で処理量を増加することによる省電力も実現できます。

E.2.2.2 デコード済み命令キャッシュ

デコード済み命令キャッシュは、基本的にはレガシー・デコード・パイプラインのアクセラレーターです。デコード済み命令キャッシュでは、デコードされた命令を格納することにより、以下の機能が可能になります。

- 分岐予測ミスでのレイテンシーの減少
- アウトオブオーダー・エンジンへのマイクロオペレーション (uop) 供給帯域幅の拡大
- フロントエンドの消費電力の軽減

デコード済み命令キャッシュは、命令デコーダーの出力をキャッシュに蓄えます。実行のためにマイクロオペレーション (uop) が次に要求されると、デコード済み命令キャッシュからデコードされたマイクロオペレーション (uop) が取り出されます。これにより、マイクロオペレーション (uop) のフェッチステージとデコードステージをスキップでき、フロントエンドの消費電力およびレイテンシーが軽減されます。デコード済み命令キャッシュはマイクロオペレーション (uop) の 80% 以上の平均ヒット率を達成し、さらに「Hotspot」のヒット率は通常 100% 近くになります。

一般的な整数プログラムの平均命令長は、1 命令あたり 4 バイト未満で、フロントエンドがバックエンドに先立って、スケジューラーが命令レベルの並列処理を見つけやすいように広い範囲を満たします。ただし、インテル® SSE メディア・アルゴリズムや過度にアンロールされたループなど、基本ブロックが多くの命令で構成されるパフォーマンスが高いコードの場合は、1 サイクルあたり 16 命令バイトが制約になることがあります。デコード済み命令キャッシュの 32 バイト指向はそのようなコードに役立ち、この制約が回避されます。

デコード済み命令キャッシュは、時間的局所性と空間的局所性でプログラムのパフォーマンスを自動的に向上させます。ただし、デコード済み命令キャッシュの潜在的能力を十分に利用するには、内部構成を理解する必要があります。デコード済み命令キャッシュは 32 のセットで構成されます。各セットには 8 つのウェイが含まれます。各ウェイは最大 6 つのマイクロオペレーション (uop) を保持できます。デコード済み命令キャッシュは最大 1536 のマイクロオペレーション (uop) を保持できます。

以下は、デコード済み命令キャッシュにマイクロオペレーション (uop) が格納される際のルールの一部です。

- ウェイのすべてのマイクロオペレーション (uop) は、コード内で静的に隣接し、同じアライメントされた 32 バイト領域内の EIP を持ちます。
- 同じ 32 バイトにアライメントされたチャンクに最大 3 つのウェイを割り当てることができるため、オリジナルの IA プログラムの 32 バイト領域あたり、合計 18 のマイクロオペレーション (uop) をキャッシュに格納できます。
- 複数のマイクロオペレーション (uop) をウェイに分割することはできません。
- 1 ウェイあたり、最大 2 つの分岐が許容されます。
- MSROM を使用する命令があると、ウェイ全体が消費されます。
- 条件分岐以外の分岐は、ウェイの最後のマイクロオペレーション (uop) です。
- マイクロフュージョンされたマイクロオペレーション (uop) (ロード + op およびストア) は 1 つのマイクロオペレーション (uop) として保持されます。
- マクロフュージョンされた命令のペアは 1 つのマイクロオペレーション (uop) として保持されます。
- 64 ビット即値を伴う命令では、即値を保持するのに 2 つのスロットを必要とします。

こうした制約によりマイクロオペレーション (uop) をデコード済み命令キャッシュに格納できない場合、レガシー・デコード・パイプラインから供給されます。レガシー・パイプラインからマイクロオペレーション (uop) が供給された場合、デコード済み命令キャッシュからのマイクロオペレーション (uop) のフェッチは次の分岐のマイクロオペレーション (uop) までは再開できません。頻繁に切り替えると、ペナルティーが発生する可能性があります。

デコード済み命令キャッシュは、事実上、命令キャッシュと ITLB に含まれます。つまり、デコード済み命令キャッシュにマイクロオペレーション (uop) がある命令は、オリジナルの命令バイトが命令キャッシュに存在します。命令キャッシュの排出はデコード済み命令キャッシュからも排出される必要があり、この場合、必要なラインだけが排出されます。

デコード済み命令キャッシュ全体がフラッシュされることがあります。この 1 つの理由として ITLB エントリーの排出が考えられます。それ以外の理由は通常アプリケーション・プログラマーからは見えず、例えば CR3 でのマッピングや CR0 および CR4 での機能やモードの有効化など、重要な制御が変更になったときに発生します。また、例えば CS ベースアドレスがゼロに設定されていないなど、デコード済み命令キャッシュが無効になっているケースもあります。

E.2.2.3 分岐予測

分岐予測は分岐ターゲットを予測し、実際の分岐の実行パスが判明する手前から命令の実行を開始できます。すべての分岐が予測に分岐予測ユニット (BPU) を利用します。このユニットは、分岐の EIP だけでなく、この EIP に実行が到達した実行パスに基づいて、ターゲットアドレスを予測します。BPU は以下の分岐タイプを効率良く予測できます。

- 条件分岐
- 直接コールおよびジャンプ
- 間接コールおよびジャンプ
- リターン

E.2.2.4 マイクロオペレーション (uop) キューおよびループストリーム検出器 (LSD)

マイクロオペレーション (uop) キューはフロントエンドとアウトオブオーダー・エンジンを分離します。図 2-7 に示すように、マイクロオペレーション (uop) の生成とリネーマーの間にあります。このキューは、フロントエンドのマイクロオペレーション (uop) の各種のソースで発生するバブルを隠蔽するのに有効であり、各サイクルで実行に向けて 4 つのマイクロオペレーション (uop) を確実に供給します。

マイクロオペレーション (uop) キューは、特定の命令タイプに対してポストデコード機能を提供します。特に、演算処理とすべてのストアを組み合わせたロードでは、インデックス付きアドレス指定が使用されると、デコーダーやデコード済み命令キャッシュでは単一のマイクロオペレーション (uop) として示されます。マイクロオペレーション (uop) キューでは、これはアンラミネーションと呼ばれる過程で 2 つのマイクロオペレーション (uop) (1 つはロード、もう一方は演算) に細分化されます。一般的な例は、以下の「ロード + 演算」命令です。

```
ADD RAX, [RBP+RSI] ; rax := rax + LD( RBP+RSI )
```

同様に、以下のストア命令にはレジスターソースが 3 つあり、「ストアアドレス生成」と「ストアデータ生成」のサブコンポーネントに分割されます。

```
MOV [ESP+ECX*4+12345678], AL
```

アンラミネーションによって生成される追加のマイクロオペレーション (uop) は、リネーム帯域幅とリタイアメント帯域幅を消費します。ただし、これによって全体的な消費電力を削減できるという利点があります。インデックス付きアドレス指定 (一般に配列処理の場合に生じる) に影響されるコードでは、ベース (またはベース + ディスプレースメント) アドレス指定を使用するリコード・アルゴリズムで、ロード + 演算命令とストア命令をフュージョンさせることにより、パフォーマンスを向上できることがあります。

ループストリーム検出器(LSD)

ループストリーム検出器は、インテル® Core™ マイクロアーキテクチャーで導入されました。LSD は、マイクロオペレーション (uop) キューにある小さなループを検出し、ロックします。分岐予測ミスで終了するまで、どのキャッシュからもマイクロオペレーション (uop) のフェッチ、デコード、または読み取りがない状態となり、マイクロオペレーション (uop) キューからループストリームが送出されます。

以下の属性のループが LSD/マイクロオペレーション (uop) キューの再実行の対象となります。

- 32 命令バイトの最大 8 つのチャンクフェッチ
- 最大 28 のマイクロオペレーション (uop) (28 以下の命令)
- すべてのマイクロオペレーション (uop) がデコード済み命令キャッシュにも存在する
- 実行される分岐が 8 つを超えてはならず、いずれも CALL や RET ではない場合
- 不一致のスタック操作は許可されません。例えば、POP 命令よりも PUSH 命令が多い場合などです

大量の計算を行うループ、検索、文字列の移動は、上記の特性に当てはまることが多く、状況に応じてループキャッシュ機能を使用します。パフォーマンスが高いコードを作成するには、LSD 機能をオーバーフローしても、通常はループアンロールを行う方がパフォーマンス的に望ましいでしょう。

E.2.3 アウトオブオーダー・エンジン

アウトオブオーダー・エンジンは、消費電力特性に優れ、以前の世代と比べ、パフォーマンスが向上しています。依存関係の連鎖を検出し、正しいデータフローを保持したまま、アウトオブオーダーで実行に送出します。依存関係の連鎖が 2 次データ・キャッシュラインなどのリソースを待機している場合、別の連鎖からマイクロオペレーション (uop) を実行コアに送ります。このため、1 サイクルあたりに実行される命令数 (IPC) の全体的なレートが増加します。

アウトオブオーダー・エンジンは、図 E-4 のコア機能図に示すように、リネーム/リタイアメント・ブロックとスケジューラーの 2 つのブロックから構成されます。

アウトオブオーダー・エンジンには、以下の主要構成要素が含まれます。

リネーマー: マイクロオペレーション (uop) をフロントエンドから実行コアに移動します。マイクロオペレーション (uop) 間の不正な依存関係を排除し、マイクロオペレーション (uop) のアウトオブオーダー実行を可能にします。

スケジューラー: すべてのソースオペランドの準備が整うまで、マイクロオペレーション (uop) をキューに格納します。できる限り先入れ先出し (FIFO) 順序に従い、準備のできたマイクロオペレーション (uop) をスケジュールし、利用可能な実行ユニットにディスパッチします。

リタイアメント: 命令およびマイクロオペレーション (uop) を順番にリタイアさせ、フォルトおよび例外を処理します。

E.2.3.1 リネーマー

リネーマーは、図 E-4 のインオーダー部分とスケジューラーのデータフローの橋渡しを行います。サイクルごとに最大 4 つのマイクロオペレーション (uop) をマイクロオペレーション (uop) キューからアウトオブオーダー・エンジンに移動します。リネーマーはサイクルごとに最大 4 つのマイクロオペレーション (uop) (マイクロおよびマクロフュージョンされていない状態、マイクロフュージョンされた状態、またはマクロフュージョンされた状態) を送出できますが、これは発行ポートがサイクルごとに 6 つのマイクロオペレーション (uop) をディスパッチするのに相当します。この過程では、アウトオブオーダー・コアは以下のステップを実行します。

- マイクロオペレーション (uop) のソースとデスティネーションを、マイクロアーキテクチャー上のソースとデスティネーションにリネームします。
- マイクロオペレーション (uop) にリソースを割り当てます。例えば、ロードバッファーやストアバッファーなど。
- マイクロオペレーション (uop) を適切なディスパッチ・ポートにバインドします。

マイクロオペレーション (uop) によってはリネーム中に実行が完了するものがあり、その場合、実行が完了した時点でパイプラインから削除され、実行帯域幅に影響しません。例えば、以下のようなものがあります。

- ゼロイディオム (依存関係解消イディオム)
- NOP
- VZEROUPPER
- FXCHG

以前のマイクロアーキテクチャーでは各サイクルで 1 つの分岐しか割り当てることができなかったのに対し、リネーマーは各サイクルで 2 つの分岐を割り当てることができます。これによって、実行時の一部のバブルを解消できます。

インデックス・レジスターを使用する、マイクロフュージョンされたロード操作とストア操作は 2 つのマイクロオペレーション (uop) に分解され、リネーマーが各サイクルで使用可能な 4 つのスロットのうち 2 つが消費されます。

ゼロイディオム(依存関係解消イディオム)

通常の命令を使用してレジスターの内容をゼロにクリアすることにより、命令の並列性を向上できます。リネーマーは、デスティネーション・レジスターのゼロ評価時にこれを検出します。

可能な場合は以下のいずれか 1 つの依存関係解消イディオムを使用して、レジスターをクリアします。

- XOR REG,REG
- SUB REG,REG
- PXOR/VPXOR XMMREG,XMMREG
- PSUBB/W/D/Q XMMREG,XMMREG
- VPSUBB/W/D/Q XMMREG,XMMREG
- XORPS/PD XMMREG,XMMREG
- VXORPS/PD YMMREG, YMMREG

ゼロイディオムはリネーマーによって検出および解決されるため、実行レイテンシーはありません。

もう 1 つ別に「1 イディオム」という依存関係解消イディオムがあります。

- CMPEQ XMM1, XMM1; "1 イディオム" はすべての要素をすべて「1」に設定します

この場合、マイクロオペレーション (uop) の実行が必要ですが、入力データに関係なく、出力データは常に「すべて 1」であることがわかっているため、ゼロイディオムの場合と同様、マイクロオペレーション (uop) のソースへの依存関係は存在せず、空いている実行ポートが見つかり次第実行できます。

E.2.3.2 スケジューラー

スケジューラーは、実行ポートへのマイクロオペレーション (uop) のディスパッチを制御します。そのため、どのマイクロオペレーション (uop) が準備でき、そのソースが何であるか (レジスター・ファイル・エントリーなのか、それとも実行ユニットから直接のバイパスなのか) を特定する必要があります。ディスパッチ・ポートおよびライトバック・バスの利用状況、準備ができたマイクロオペレーション (uop) の優先度に応じて、スケジューラーはサイクルごとにどのマイクロオペレーション (uop) をディスパッチするのかを選択します。

E.2.4 実行コア

実行コアはスーパースケーラーであり、命令をアウトオブオーダーで処理できます。潜在的な遅延を抑えながら、最も一般的な操作を効率良く処理することで、全体的なパフォーマンスを最適化します。

アウトオブオーダー実行コアでは、以前の世代と比べ、以下の点で、実行ユニットの編成が改善されています。

- 読み出しポートのストールの軽減
- ライトバックの競合および遅延の軽減
- 消費電力の軽減
- デノーマル入力とアンダーフロー出力処理の SIMD FP アシストの軽減

FTZ=0 および DAZ=0 における操作では、ある種の高精度 FP アルゴリズムが必要です。つまり、SIMD FP アシストによる以前の世代のマイクロアーキテクチャーのパフォーマンスの低下を犠牲にして、より高い数値精度を実現するために、アンダーフローの即値結果とデノーマル入力を許容します。インテル® マイクロアーキテクチャー Sandy Bridge† では、次のインテル® SSE 命令 (および派生するインテル® AVX) における SIMD FP アシストを削減します: ADDPD/ADDPS、MULPD/MULPS、DIVPD/DIVPS、および CVTPD2PS。

アウトオブオーダー・コアは 3 つの実行スタックから構成され、それぞれのスタックに特定のタイプのデータがカプセル化されます。実行コアは、以下の実行スタックを備えています。

- 汎用整数
- SIMD 整数および浮動小数点
- x87

実行コアは、キャッシュ階層との接続も備えています。ロードされたデータはキャッシュからフェッチされ、いずれかのスタックに書き戻されます。

スケジューラーは各ポートで 1 つずつ、サイクルごとに最大 6 つのマイクロオペレーション (uop) をディスパッチできます。次の表は、どの操作をどのポートにディスパッチできるかをまとめたものです。

表 E-10 ディスパッチ・ポートと実行スタック

	ポート 0	ポート 1	ポート 2	ポート 3	ポート 4	ポート 5
整数	ALU、Shift	ALU、Fast LEA、Slow LEA、MUL	Load_Addr、Store_addr	Load_Addr Store_addr	Store_data	ALU、Shift、Branch、Fast LEA
SSE-Int、AVX-Int、MMX	Mul、Shift、STTNI、Int-Div、128b-Mov	ALU、Shuf、Blend、128b-Mov			Store_data	ALU、Shuf、Shift、Blend、128b-Mov
SSE-FP、AVX-FP_low	Mul、Div、Blend、256b-Mov	Add、CVT			Store_data	Shuf、Blend、256b-Mov
X87、AVX-FP_High	Mul、Div、Blend、256b-Mov	Add、CVT			Store_data	Shuf、Blend、256b-Mov

実行後、ディスパッチ・ポートと結果のデータ型に応じて、ライトバック・バスに書き戻されます。同じポートでディスパッチされ、レイテンシーが異なるマイクロオペレーション (uop) は同じサイクルでライトバック・バスが必要となることがあります。このような場合、ライトバック・バスが使用可能になるまで、いずれか 1 つのマイクロオペレーション (uop) の実行が遅延されます。例えば、MULPS (5 サイクル) と BLENDPS (1 サイクル) の両方がポート 0 で実行準備が整った場合、衝突が発生し、最初に MULPS が実行され、4 サイクル後に BLENDPS が実行されます。Sandy Bridge† マイクロアーキテクチャーでは、マイクロオペレーション (uop) が結果を異なるスタックに書き出す限り、このような衝突は解消されます。例えば、整数 ADD (1 サイクル) は整数スタックを使用し、MULPS (5 サイクル) は FP スタックを使用するため、MULPS の 4 サイクル後に整数 ADD をディスパッチできます。

あるスタックで実行されるマイクロオペレーション (uop) のソースが、別のスタックで実行されるマイクロオペレーション (uop) からのものである場合、1 サイクルまたは 2 サイクルの遅延が生じる可能性があります。インテル® SSE 整数操作とインテル® SSE 浮動小数点操作の間の遷移でも遅延が発生します。このような場合、命令フローに追加されるマイクロオペレーション (uop) によって、データ遷移が行われることがあります。以下の表に、実行後にライトバックされるデータが以下のサイクルでどのようにマイクロオペレーション (uop) 実行にバイパスできるかを示します。

表 E-11 実行コアのライトバック・レイテンシー (サイクル数)

	整数	SSE-Int、AVX-Int、MMX	SSE-FP、AVX-FP_low	X87、AVX-FP_High
整数	0	micro-op（ポート 0）	micro-op（ポート 0）	micro-op（ポート 0）+1 サイクル
SSE-Int、AVX-Int、MMX	micro-op（ポート 5）または micro-op（ポート 5）+1 サイクル	0	1 サイクル遅延	0
SSE-FP、AVX-FP_low	micro-op（ポート 5）または micro-op（ポート 5）+1 サイクル	1 サイクル遅延	0	micro-op（ポート 5）+1 サイクル
X87、AVX-FP_High	micro-op（ポート 5）+1 サイクル	0	micro-op（ポート 5）+1 サイクル	0
ロード	0	1 サイクル遅延	1 サイクル遅延	2 サイクル遅延

E.2.5 キャッシュ階層

キャッシュ階層には、1 次命令キャッシュ、1 次データキャッシュ (L1D キャッシュ)、2 次 (L2) キャッシュがそれぞれのコアの中に含まれます。L1D キャッシュは、インテル® ハイパースレッディング・テクノロジーをサポートしていれば、2 つの論理プロセッサーで共有されます。L2 キャッシュは命令とデータで共有されます。物理プロセッサー・パッケージ内のすべてのコアは、リング接続により、共有されるラスト・レベル・キャッシュ (LLC) と接続します。

キャッシュは、命令トランスレーション・ルックアサイド・バッファー (ITLB)、データ・トランスレーション・ルックアサイド・バッファー (DTLB)、共有トランスレーション・ルックアサイド・バッファー (STLB) の各種機能を使用して、リニアアドレスを物理アドレスに変換します。すべてのキャッシュレベルのデータ・コヒーレンシーは MESI プロトコルを使用して維持されます。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3C』を参照してください。キャッシュ階層の詳細については、実行時に CPUID 命令を使用して取得できます。『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』を参照してください。

表 E-12 キャッシュ・パラメーター

Level	Capacity	Associativity (ways)	Line Size (bytes)	Write Update Policy	Inclusive
L1 Data	32 KB	8	64	Writeback	-
Instruction	32 KB	8	N/A	N/A	-
L2 (Unified)	256 KB	8	64	Writeback	No
Third Level (LLC)	Varies, query CPUID leaf 4	Varies with cache size	64	Writeback	Yes

E.2.5.1 ロード操作とストア操作の概要

この節では、ロード操作とストア操作の概要を示します。

ロード

ライトバック (WB) 方式のメモリー・ロケーションから命令がデータを読み出す場合、プロセッサーはキャッシュおよびメモリー内を検索します。表 E-13 に、アクセス・ルックアップ順序およびベストケースのレイテンシーを示します。実際のレイテンシーはキャッシュキューの空き具合、LLC リングの空き具合、メモリー・コンポーネント、そのパラメーターによって変わることがあります。

表 E-13 ルックアップ順序とロード・レイテンシー

レベル	レイテンシー（サイクル数）	帯域幅（1 コアあたり、1 サイクルあたり）
L1 データ	4[1]	2x16 バイト
L2（ユニファイド）	12	1x32 バイト
3 次（LLC）	26-31	1x32 バイト
その他のコア内の L2 キャッシュおよび L1 D キャッシュ（該当する場合）	43- クリーンヒット 60- ダーティーヒット	

注意:

1. 表 2-16 に示すように、実行コアのバイパス制約を受けます。
2. L3 のレイテンシーは製品セグメントと SKU によって異なります。値は、第 2 世代インテル® Core™ プロセッサー・ファミリーに適用されます。

LLC はそれよりも上位のすべてのキャッシュレベルを含みます。コアキャッシュに含まれるデータは LLC にもなければいけません。LLC の各キャッシュラインは、L2 キャッシュと L1 キャッシュにこのラインを含む可能性があるコアの存在を示します。他のコアに目的のラインが含まれることが LLC で示され、その状態を変更する必要がある場合、それらのコアの L1D キャッシュと L2 キャッシュのルックアップを行います。他のコアキャッシュからのデータのフェッチを必要としない場合、このルックアップは「クリーン」ルックアップと呼ばれます。変更されたデータを他のコアキャッシュからフェッチし、ローディング・コアに転送する必要がある場合は、このルックアップを「ダーティー」ルックアップと呼びます。

上記に示すレイテンシーはベストケースのシナリオです。変更されたキャッシュラインを排出して、新しいキャッシュライン用にスペースを確保しなければならないことがあります。変更されたキャッシュラインは新しいデータの取得と並行して排出されるので、追加のレイテンシーは必要としません。ただし、データがメモリーにライトバックされる際は、キャッシュ帯域幅のほか、メモリー帯域幅も排出に使用される可能性があります。したがって、変更されたキャッシュラインの排出を伴うキャッシュミスが短時間に複数発生した場合、キャッシュ応答時間が全体的に低下します。メモリー・アクセス・レイテンシーは、メモリー・コントローラー・キューの空き具合、DRAM コンフィグレーション、DDR パラメーター、DDR ページング動作 (要求されたページがページヒット、ページミス、ページ・エンプティーのいずれかの場合) によって異なります。

ストア

命令がライトバック方式のメモリー・ロケーションにデータを書き込む際、プロセッサーはまず L1D キャッシュに排他または変更 MESI 状態でこのメモリー・ロケーションを含むラインがあることを確認します。キャッシュラインがそこに正しい状態で存在しない場合、プロセッサーは所有権読み出し要求を使用して、次のメモリー階層レベルからフェッチします。プロセッサーは、指定された順序で以下の場所からキャッシュラインを検索します。

1. L1D キャッシュ
2. L2

3. ラスト・レベル・キャッシュ
4. その他のコア内の L2 キャッシュおよび L1D キャッシュ (存在する場合)
5. メモリー

キャッシュラインが L1D キャッシュにあると、そこに新しいデータが書き込まれ、そのラインが変更済みとしてマークされます。

所有権読み出しとデータのストアは、命令のリタイアメントの後に発生し、ストア命令のリタイアメントの順序に従います。したがって、ストア・レイテンシーは、通常、ストア命令自体には影響を与えることはありません。ただし、L1D キャッシュをミスする一部の連続したストアは、レイテンシーを累積してパフォーマンスに影響を与えることがあります。ストアが完了しない間は、ストアバッファー内のエントリーが消費されます。ストアバッファーがいっぱいになると、新しいマイクロオペレーション (uop) が実行パイプに入ることができず、実行がストールする可能性があります。

E.2.5.2 L1D キャッシュ

L1D キャッシュは、1 次データキャッシュです。内部データ構造を通過するすべてのタイプからのすべてのロード要求とストア要求を管理します。
L1D キャッシュ:

- ロードとストアを投機的にアウトオブオーダー発行することができます。
- リタイアしたロードとストアがリタイア時に正確なデータを得られます。
- ロードとストアが IA-32 アーキテクチャーとインテル® 64 命令セット・アーキテクチャーのメモリーの順序付け規則に従うことを確実にします。

表 E-14 L1 データキャッシュの構成要素

構成要素	インテル® マイクロアーキテクチャー Sandy Bridge	インテル® マイクロアーキテクチャー Nehalem
データ・キャッシュ・ユニット (DCU)	32KB、8 ウェイ	32KB、8 ウェイ
ロードバッファー	64 エントリー	48 エントリー
ストアバッファー	36 エントリー	32 エントリー
ライン・フィル・バッファー (LFB)	10 エントリー	10 エントリー

DCU は 32KB の 8 ウェイ・セット・アソシアティブとして編成されます。キャッシュライン・サイズは 8 つのバンクに配置された 64 バイトです。

内部的には、アクセスは最大 16 バイトで、256 ビットのインテル® AVX 命令が 2 つの 16 バイト・アクセスを使用します。各サイクルで、2 つのロード操作と 1 つのストア操作を処理できます。

L1D キャッシュは、すぐに処理できない要求を完了するまで保持します。要求を遅延する理由としては、キャッシュミス、キャッシュライン間で分割されるアライメントされていないアクセス、先行するストアからフォワードされる準備が整っていないデータ、バンクの衝突が発生するロード、キャッシュラインの置換によるロードブロックなどがあります。

L1D キャッシュは、割り当てからリタイアメントまで最大 64 のロード・マイクロオペレーション (uop) を保持できます。割り当てからストア値がキャッシュにコミットされるまで、または非テンポラルなストアの場合はライン・フィル・バッファー (LFB) に書き込まれるまで、最大 36 のストア操作を保持できます。

L1D キャッシュは、複数の未処理のキャッシュミスを処理し、後続のストアとロードの処理を続けることができます。LFB を利用することにより、最大 10 のキャッシュライン・ミスを同時に管理できます。

L1D キャッシュはライトバック、ライトアロケート・キャッシュです。DCU にヒットするストアは、より下位のメモリー階層を更新しません。DCU が見つからないストアはキャッシュラインを割り当てます。

ロード

L1D キャッシュ・アーキテクチャーは、1 サイクルで 2 つのロードを処理でき、それぞれのロードは最大 16 バイトまで可能です。アウトオブオーダー・エンジンでの割り当てから、ロードされた値が実行コアに戻されるまで、さまざまな進捗段階で最大 32 のロードを保持できます。ロードでは以下を実行できます。

- ロードアドレスとストアアドレスの範囲が競合していないことが明らかな場合は、先行するストアの前にデータを読み取ることができます。
- 先行する分岐が解決される前に、投機実行できます。
- キャッシュミスにアウトオブオーダーでオーバーラップして対応します。

ロードでは以下を実行できません。

- あらゆるフォルトやトラップに対し投機的に対応すること。
- キャッシュできないメモリーに投機的にアクセスすること。

一般的なロード・レイテンシーは 5 サイクルです。シンプルなアドレス指定モードを使用している場合、2048 以下のベース + オフセットでは、4 サイクルのロード・レインテンシーが可能です。この手法は特にポインター追跡コードに便利です。ただし、スタックバイパスにより、ターゲットレジスターのデータ型に応じて、全体的なレイテンシーが変わります。詳細については、2.4.4 節を参照してください。

次の表は、全体的なロード・レイテンシーの一覧です。これらのレイテンシーではフラットセグメントの一般的なケース (つまり、セグメントのベースアドレスがゼロであること) を前提としています。セグメントベースがゼロでない場合、ロード・レイテンシーは増大します。

表 E-15 ロード・レイテンシーに対するアドレス指定モードの影響

データタイプ / アドレス指定モード	ベース + オフセット >2048; ベース + インデックス [+ オフセット]	ベース + オフセット <2048
整数	5	4
MMX、SSE、128 ビット AVX	6	5
X87	7	6
256 ビット AVX	7	7

ストア

メモリーへのストアは、2 つのフェーズで実行されます。

- 実行フェーズ: ストアバッファーをリニアアドレス、物理アドレス、およびデータでいっぱいにします。ストアアドレスとデータが判明していれば、そのストアデータを必要とする以降のロード操作に転送できます。
- 完了フェーズ: ストアのリタイア後、L1D キャッシュはストアバッファーから DCU に、1 サイクルあたり最大 16 バイトのデータを移動できます。

アドレス変換

DTLB はサイクルごとに、ロードアドレスに 2 つ、ストアアドレスに 1 つの合計 3 つのリニアアドレスから物理アドレスへの変換を行うことができます。DTLB でアドレスが見つからない場合、プロセッサーはデータおよび命令アドレス変換を保持する STLB を検索します。STLB にヒットする DTLB ミスのペナルティーは 7 サイクルです。ラージページ・サポートには、4K ページと 2M/4M ページに加え、1G バイト・ページが含まれます。

DTLB および STLB は 4 ウェイ・セット・アソシアティブです。以下の表に、DTLB および STLB のエントリーの数を示します。

表 E-16 DTLB および STLB のパラメーター

TLB	ページサイズ	エントリー
DTLB	4KB	64
	2MB/4MB	32
	1GB	4
STLB	4KB	512

ストア・フォワーディング

ストアに続いてロードを行い、ストアによってメモリーに書き込まれたデータを再読み込みする場合、ストア操作からロードにデータを直接転送できます。これはストア-ロード・フォワーディングと呼ばれ、ロードがメモリーを介さずにストア操作から直接データを取得できるので、サイクル数の節約になります。ストア・フォワーディングを利用すると、サブフィールドを転送する能力を犠牲にすることなく、複雑な構造を移動できます。メモリー制御ユニットは、以前のマイクロアーキテクチャーと比べ、より少ない制約でストア・フォワーディングを処理できます。

ストア-ロード・フォワーディングを行うには、以下の規則に従わなければなりません。

- ストアは、ロードに先行する、該当アドレスへの最後のストアでなければなりません。
- ストアに、ロードされているすべてのデータが含まれていなければなりません。
- ロードはライトバック・メモリーからで、ロードもストアも非テンポラルアクセスであってはなりません。

以下の場合、ストアはロードに転送できません。

- 先行する 16 バイトまたは 32 バイトのストアに対して、8 バイト境界をまたぐ 4 バイトおよび 8 バイトのロード。
- 32 バイト・ストアの 16 バイト境界をまたぐロード。

表 E-17 から表 E-20 に、ストア-ロード・フォワーディング動作の詳細を示します。一定のストアサイズに対して、オーバーラップする可能性があるすべてのロードが示され、「F」記号で示されています。32 バイト・ストアからの転送は、半分の各 16 バイト・ストアからの転送と類似します。転送できないケースは「N」として示されます。

表 E-17 ストア・フォワーディング条件 (1 バイト・ストアおよび 2 バイト・ストア)

		ロード・アライメント															
ストアサイズ	ロードサイズ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
2	1	F	F														
	2	F	N														

表 E-18 ストア・フォワーディング条件 (4 - 16バイト・ストア)

		ロード・アライメント															
ストア サイズ	ロード サイズ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	F	F	F												
	2	F	F	F	N												
	4	F	N	N	N												
8	1	F	F	F	F	F	F	F	F								
	2	F	F	F	F	F	F	F	N								
	4	F	F	F	F	F	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

表 E-19 ストア・フォワーディング条件 (0 – 15 バイト・ストア)

		ロード・アライメント															
ストア サイズ	ロード サイズ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

表 E-20 ストア・フォワーディング条件 (16 – 31 バイト・ストア)

		ロード・アライメント															
ストア サイズ	ロード サイズ	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

メモリー・ディスアンビゲーション

ロード操作は先行するストアに依存する可能性があります。多くのマイクロアーキテクチャーでは、先行するストアのアドレスがすべて判明するまで、ロードがブロックされます。メモリー・ディスアンビゲーション機能は、どのロードが先行するアドレスが不明なストアに依存しないかを予測します。ロードにそのような依存関係がないことをディスアンビゲーション機能が予測した場合、先行するストアと同じアドレスからデータを取得します。ロードが実行された時点でアドレスが不明なストアに実際に依存する場合、競合が検出され、ロードとすべての後続命令が再実行されます。

以下のロードはディスアンビゲーションの対象外です。先行するすべてのストアのアドレスが判明するまで、これらのロードの実行はストールします。

- 16 バイト境界にまたがるロード。
- 32 バイトにアライメントされていない 32 バイトのインテル® AVX ロード。

メモリー・ディスアンビゲーション機能では、ロードと同じアドレスビット 0:11 を持つ先行するストアの間との依存関係を常に前提とします。

バンクの競合

16 バイト・ロードは最大 3 つのバンクをカバーし、1 サイクルで 2 つのロードが可能であるため、ロードに対して各サイクルで 8 つのバンクのうち 6 つにアクセスできます。異なるセットで 2 つのロードアクセスが同じバンクを同時に必要とした (アドレスが同じ 2 ~ 4 ビット値を持つ) 場合にバンクの競合が発生します。バンクの競合が発生すると、ロードアクセスの一方が内部で再試行されます。

多くの場合、スタックからオペランドを取り出す場合やシーケンシャル・アクセスの場合に、2 つのロードは同じキャッシュラインの全く同じバンクにアクセスします。このような場合、競合は発生せず、同時にロードが処理されます。

E.2.5.3 リング・インターコネクトとラスト・レベル・キャッシュ

システムオンチップ設計により、IA コアとアンコアの各種サブシステム間を接続するための高帯域幅の双方向リングバスを提供します。第 2 世代インテル® Core™ プロセッサー 2xxx 番台では、アンコア・サブシステムとして、システム・エージェント、グラフィックス・ユニット (GT)、ラスト・レベル・キャッシュ (LLC) が含まれます。

LLC は複数のキャッシュスライスで構成されます。スライスの数は IA コアの数と同じです。各スライスには、ロジック部分とデータ配列部分があります。ロジック部分は、データ・コヒーレンシー、メモリーアクセス順序、データ配列部分へのアクセス、LLC ミス、メモリーへのライトバックなどを処理します。データ配列部分はキャッシュラインを格納します。各スライスには、32 バイト/サイクルを供給できるフル・キャッシュ・ポートが含まれます。

LLC データ配列に保持されるデータの物理アドレスは、アドレスが均一に分散されるように、ハッシュ関数によってキャッシュスライス間に分散されます。キャッシュブロック内のデータ配列は、0.5M/1M/1.5M/2M のブロックサイズに合わせ、4/8/12/16 のウェイの構成が可能です。ただし、キャッシュブロック間でのアドレス分散がソフトウェアの観点からであるため、これは通常の N ウェイキャッシュとして見えることはありません。

プロセッサー・コアおよび GT から見ると、LLC は複数のポートと、コアの数に応じてスケーリングする帯域幅を備えた 1 つの共有キャッシュとして機能します。LLC ヒット・レイテンシー (26 ~ 31 サイクル) は、LLC ブロックに対するコアの位置、およびその要求がリング上をどこまで伝わる必要があるかによって左右されます。

キャッシュブロックの数はコアの数に応じて増えるため、リングおよび LLC がコア操作に対して帯域幅を制限する可能性はありません。

GT も同じリング・インターコネクト上にあり、データ操作に LLC を使用します。この点では、IA コアと非常に類似しています。そのため、キャッシュ帯域幅と大きなキャッシュ領域を使用する高帯域幅のグラフィックス・アプリケーションの場合、多少コア操作に干渉するおそれがあります。

LLC ミス、ダーティーなライン・ライトバック、キャッシュ不能な操作、MMIO/IO 操作など、LLC で満足させることができないすべてのトラフィックが、キャッシュブロックのロジック部分およびリングを通り、システム・エージェントに伝わります。

インテル® Xeon® プロセッサー E5 ファミリーのアンコア・サブシステムは、グラフィックス・ユニット (GT) を含みません。代わりに、アンコア・サブシステムは、大きな容量と複数のプロセッサーをサポーするスヌーピング機能を備えた LCC、複数ソケットのプラットフォームをサポートするンテル® QuickPath インターコネクト・ファブリック、電力管

理制御ハードウェア、およびメモリーと I/O デバイスからの高帯域幅のトラフィックをサポートするシステム・エージェント機能など、多くのコンポーネントを備えています。

インテル® Xeon® プロセッサー E5 2xxx または 4xxx 製品ファミリーでは、LLC の容量は 1 コアあたり 2.5M バイトでプロセッサーのコア数によって異なります。

E.2.5.4 データ・プリフェッチ

ソフトウェア・プリフェッチ、ハードウェア・プリフェッチ、またはその 2 つの組み合わせを使用して、L1D キャッシュに投機的にデータをロードできます。

4 つのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) プリフェッチ命令を使用して、ソフトウェア制御プリフェッチを行うことができます。このような命令は、データを含むキャッシュラインを要求されるレベルのキャッシュ階層に移動する際のヒントとして機能します。ソフトウェア制御プリフェッチは、コードのプリフェッチではなく、データのプリフェッチを対象とします。

この節の残りでは、Sandy Bridge† マイクロアーキテクチャーによって提供される各種ハードウェア・プリフェッチ・メカニズム、およびこれまでのプロセッサーに対する改善点について説明します。プリフェッチの目的は、プログラムが消費しそうなデータを事前に予測することです。このデータが実行コアまたは内部キャッシュの近くにない場合、プリフェッチでは、次のレベルのキャッシュ階層およびメモリーからデータを取り込みます。プリフェッチは以下の影響を及ぼします。

- データがプログラムで使用される順序で連続的に配置されている場合、パフォーマンスが向上します。
- アクセスパターンが局所的でなくまばらな場合、帯域幅の問題によりわずかにパフォーマンスが低下することがあります。
- まれに、アルゴリズムのワーキングセットがキャッシュの大部分を占めるようにチューニングされ、プログラムから要求されたキャッシュラインが不要なプリフェッチによって排出されている場合、1 次キャッシュの容量が原因でハードウェア・プリフェッチが重大なパフォーマンス低下をもたらすことがあります。

L1 データキャッシュに対するデータ・プリフェッチ

データ・プリフェッチは、以下の条件が満たされた場合にロード操作によりトリガーされます。

- ライトバック・メモリーからのロードである。
- プリフェッチされるデータが、トリガーしたロード命令と同じ 4K バイト・ページ内にある。
- パイプライン内でフェンスが発生していない。
- その他のロードミスがあまり発生していない。
- 連続的なストアのストリームが発生していない。

L1D キャッシュにデータをロードするハードウェア・プリフェッチは 2 つあります。

- **データ・キャッシュ・ユニット(DCU)プリフェッチャー:** このプリフェッチはストリーミング・プリフェッチとも呼ばれ、最後にロードされたデータへのユニット・ストライド・アクセスによってトリガーされます。プロセッサーは、このアクセスがストリーミング・アルゴリズムの一部であるとみなし、次のラインを自動的にフェッチします。
- **間隔を空けた命令ポインター(IP)ベースのプリフェッチャー:** このプリフェッチは、個々のロード命令を追跡します。ロード命令に規則的なアドレススキップがあることが検出された場合、現在のアドレスとこの間隔の合計である次のアドレスがプリフェッチされます。前方または後方へのプリフェッチが可能であり、最大で 2K バイトの間隔に対処できます。

L2 キャッシュおよびラスト・レベル・キャッシュに対するデータ・プリフェッチャー:

メモリーから L2 キャッシュおよびラスト・レベル・キャッシュへデータをプリフェッチするハードウェア・プリフェッチは以下の 2 つです。

空間的プリフェッチャー: L2 キャッシュにフェッチされたすべてのキャッシュラインを、128 バイトにアライメントされたチャンクとなるペアラインにしようとします。

ストリーマー: 昇順および降順のアドレスシーケンスに対して、L1 キャッシュからの読み込み要求を監視します。監視される読み込み要求には、ロード操作とストア操作およびハードウェア・プリフェッチによって開始された L1D キャッシュ要求、およびコードフェッチに対する L1 命令キャッシュ要求が含まれます。前方または後方の要求ストリームが検出されると、予想されるキャッシュラインがプリフェッチされます。プリフェッチされるキャッシュラインは同じ 4K ページになければいけません。

ストリーマーおよび空間的プリフェッチでは、ラスト・レベル・キャッシュにデータをプリフェッチします。見つからない要求が大量に L2 キャッシュにロードされていない限り、通常、L2 にもデータが取り込まれます。

ストリーマーの改善点として、以下の機能が挙げられます。

- ストリーマーは、L2 ルックアップごとに 2 つのプリフェッチ要求を発行できます。ストリーマーはロード要求に先立ち、最大 20 のラインを処理できます。
- 1 コアあたりの未処理の要求の数に合わせて動的に調整を行います。未処理の要求がそれほど多くない場合、ストリーマーはさらに先立ってプリフェッチを行います。未処理の要求が多い場合は、LLC にのみ低頻度のプリフェッチを行います。
- キャッシュラインがはるか先にある場合、ラスト・レベル・キャッシュにのみプリフェッチを行い、L2 に対してはプリフェッチを行いません。この方法により、L2 キャッシュ内の有用なキャッシュラインが置換されなくなります。
- 最大 32 のデータ・アクセス・ストリームを検出および保持できます。4K バイト・ページごとに、1 つの前方ストリームと 1 つの後方ストリームを保持します。

E.2.6 システム・エージェント

第 2 世代インテル® Core™ プロセッサー・ファミリーのシステム・エージェントには、以下の構成要素が含まれます。

- アービター: リングドメインおよび I/O (PCIe* および DMI) からのすべてのアクセスを処理し、所定の場所にルーティングします。
- PCIe* コントローラー: 外部 PCIe* デバイスに接続します。PCIe* コントローラーは可能なコンフィグレーションがさまざまで、製品セグメントの詳細に応じて異なります (x16+x4、x8+x8+x4、x8+x4+x4+x4)。
- DMI コントローラー: PCH チップセットに接続します。
- 内部のグラフィック操作のための統合ディスプレイ・エンジン、フレキシブル・ディスプレイ・インターコネクト、およびディスプレイ・ポート。
- メモリー・コントローラー

メイン・メモリー・トラフィックは、すべてアービターからメモリー・コントローラーにルーティングされます。第 2 世代インテル® Core™ プロセッサー 2xxx 番台のメモリー・コントローラーは、2 つの DDR チャネルをサポートし、ユニットタイプ、システム構成、DRAM に応じて、データレートは 1066MHz、1333MHz、1600MHz、および 1 サイクルあたり 8 バイトです。最高の帯域幅、最小の Hotspot 衝突を実現するために、チャネル間のロードのバランスを取るローカルなハッシュ関数に基づき、メモリーチャネル間でアドレスが分散されます。

最高のパフォーマンスを実現するために、両方のチャネルに同じメモリー量、できれば全く同じタイプの DIMM を設定することが推奨されます。さらに、同じメモリー量にさらに多くのランクを使用すると、さらに多くの DRAM ページを同時に開くことができるため、メモリー帯域幅が多少向上します。最高のパフォーマンスを実現するために、最高の DRAM タイミングを備えたサポート速度が最高の DRAM (最高のサポート周波数に応じて、1333MHz または 1600MHz のデータレート)をシステムに設定します。

2 つのチャネルは固有のリソースを持ち、メモリー要求を個別に処理します。メモリー・コントローラーには、レイテンシーを最低限に抑えながら最高のメモリー帯域幅を実現する高性能のアウトオブオーダー・スケジューラーが内蔵されています。各メモリーチャネルには、32 キャッシュラインのライトデータバッファーがあります。メモリー・コント

ローラーへの書き込みは、ライトデータバッファーへ書き込まれた時点で完了とみなされます。ライトデータバッファーは、その後ライト・レイテンシーに影響を及ぼさなくなった時点で、メインメモリーにフラッシュされます。

パーシャル書き込みはメモリー・コントローラーでは効率的に処理されず、パーシャル書き込みが時間内にキャッシュライン全体を完了できない場合、DDR チャネルは [読み出し-変更-書き込み] 操作となります。ソフトウェアではできる限りパーシャル書き込みトランザクションを行わないようにし、パーシャル書き込みをフル・キャッシュライン書き込みにバッファリングするなど、別の方法を考慮する必要があります。

メモリー・コントローラーは優先度の高い等時性要求 (USB 等時性要求、ディスプレイ等時性要求など) もサポートします。統合ディスプレイ・エンジンからの高帯域幅メモリー要求は、メモリー帯域幅の一部を占有し、コア・アクセス・レイテンシーに多少影響が生じます。

E.2.7 Ivy Bridge† マイクロアーキテクチャー

Ivy Bridge† マイクロアーキテクチャーをベースとした第 3 世代インテル® Core™ プロセッサーです。E.2.1 節から E.2.6 節で示される多くの機能は、Ivy Bridge† マイクロアーキテクチャーにも適用されます。この節ではコーディングとパフォーマンスに影響するマイクロアーキテクチャーの違いについて説明します。

新しく追加された命令には次のサポートが含まれます。

- 半精度浮動小数値との間の数値変換
- NIST SP 800-90A 準拠のハードウェア・ベースの乱数生成器
- ユーザーモードのスレッド化のサポートを強化するため、すべての特権リングレイヤーで FS/GS ベースレジスターの読み書きが可能になりました。

ハードウェア・ベースの乱数生成器命令 RDRAND の使い方の詳細は、インテル® デベロッパー・ゾーンで公開されている記事を参照してください:https://software.intel.com/en-us/articles/inteldigital-random-number-generator-drng-software-implementation-guide/ (英語)

ソフトウェアに利点がある若干のマイクロアーキテクチャーの拡張が行われています。

- ハードウェア・プリフェッチャーの拡張: Ivy Bridge† マイクロアーキテクチャーでは、ネクスト・ページ・プリフェッチャー (NPP) が追加されました。NPP は、ページ境界へ近づく上方または下方の連続したキャッシュライン・アクセスでトリガーされます。
- ゼロレイテンシーのレジスター移動操作: レジスター-レジスター間の MOV 命令のサブセットはフロントエンドで実行され、アウトオブオーダー・エンジンのスケジュールと実行リソースを使用しません。
- フロントエンドの拡張: Sandy Bridge† マイクロアーキテクチャーでは、マイクロオペレーション (uop) キューは、ソフトウェアがシングルスレッドまたはマルチスレッドで実行されるのにかかわりなく、それぞれの論理プロセッサーへ 28 個のエントリーを提供するため静的に分割されていました。Ivy Bridge† マイクロアーキテクチャーでは、一方の論理プロセッサーがアクティブでない場合、プロセッサー上でシングルスレッドが実行されるとマイクロオペレーション・キューを 56 エントリー使用することができます。この場合、LSD は 28 エントリー以上を必要とする大きなループ構造を処理できます。
- いくつかの命令のレイテンシーとスループットが、Sandy Bridge† マイクロアーキテクチャーに比べ改善されています。例えば、256 ビットのパックド浮動小数点除算と平方根命令が高速化されています。また、ROL と ROR 命令も改善されています。

E.3 インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャー

インテル® Core™ マイクロアーキテクチャーでは、シングルスレッドのワークロードだけでなくマルチスレッドにおけるワークロードでも優れたパフォーマンスと電力効率を実現する以下の機能が採用されています。

- **インテル® ワイド・ダイナミック・エグゼキューション:** 各プロセッサー・コアが 1 サイクルあたり最大 4 命令のフェッチ、ディスパッチ、高帯域幅での実行、リタイアを行います。これには以下の機能が含まれます。
  - 14 ステージの効率的なパイプライン
  - 3 つの演算論理ユニット
  - 1 サイクルあたり最大 5 命令をデコード可能な 4 つのデコーダー
  - フロントエンドのスループットを高めるマクロフュージョンとマイクロフュージョン
  - 1 サイクルあたり最大 6 マイクロオペレーション (uop) をディスパッチ可能な発行能力
  - 1 サイクルあたり最大 4 マイクロオペレーション (uop) のリタイアメント帯域幅
  - 高度な分岐予測
  - 関数やプロシージャーの開始/終了を効率化するスタックポインター追跡機能

- **インテル® アドバンスト・スマート・キャッシュ:** シングルスレッド・アプリケーションとマルチスレッド・アプリケーションのいずれの場合でも、2 次キャッシュからコアへの帯域幅を拡大し、最適なパフォーマンスと柔軟性を提供します。これには以下の機能が含まれます。
  - マルチコアおよびシングルスレッドの実行環境向けに最適化
  - 2 次キャッシュから 1 次データキャッシュへの帯域幅を拡大する 256 ビットの内部データパス
  - 4MB、16 ウェイ (または 2MB、8 ウェイ) のユニファイド型共有 2 次キャッシュ

- **インテル® スマート・メモリー・アクセス:** データ・アクセス・パターンに応じてメモリーからデータをプリフェッチし、アウトオブオーダー実行によりキャッシュミス発生を軽減します。これには以下の機能が含まれます。
  - ハードウェア・プリフェッチによって 2 次キャッシュミスの実効レイテンシーを軽減
  - ハードウェア・プリフェッチによって 1 次データ・キャッシュ・ミスの実効レイテンシーを軽減
  - メモリー・ディスアンビゲーションによってスペキュレーティブ・エグゼキューション・エンジンを効率化

- **インテル® アドバンスト・デジタル・メディア・ブースト:** ほとんどの 128 ビット SIMD 命令のパフォーマンスを 1 サイクルのスループットと浮動小数点演算により改善します。これには以下の機能が含まれます。
  - ほとんどの 128 ビット SIMD 命令を 1 サイクルのスループットで実行 (128 ビットのシャッフル/パック/アンパック操作を除く)
  - 1 サイクルあたり最大 8 個の浮動小数点演算を実行可能
  - 3 つの発行ポートを利用して SIMD 命令をディスパッチし、実行することが可能

拡張版インテル® Core™ マイクロアーキテクチャーは、インテル® Core™ マイクロアーキテクチャーの機能をすべてサポートしているほか、以下のような包括的な拡張機能を提供します。

- **インテル® ワイド・ダイナミック・エグゼキューション:** 以下の拡張機能が追加されています。
  - 従来の基数 4 の除算器に代わり基数 16 の除算器を採用することで、除算や平方根などレイテンシーが大きい演算を高速化しています。
  - 内部構造の改善によって、RDTSC、STI、CLI、VM 終了の遷移など長いレイテンシーの操作を高速化しました。

- **インテル® アドバンスト・スマート・キャッシュ:** 2 つのプロセッサー・コア間で共有される最大 6MB の 2 次キャッシュ (クアッドコア・プロセッサーは最大 12MB の 2 次キャッシュを搭載)。最大 24 ウェイのセット・アソシアティビティーで構成されます。
- **インテル® スマート・メモリー・アクセス:** 最大 1600MHz の高速システムバスをサポートし、キャッシュライン分割されたロードやストアおよびロード・フォワーディングなどのメモリー操作の処理を効率化します。
- **インテル® アドバンスト・デジタル・メディア・ブースト:** シャッフル/パック/アンパック操作を高速化する 128 ビットのシャッフルユニットを提供します。これに伴い 47 個のインテル® SSE4.1 命令が追加されました。2.5 節のインテル® Core™ マイクロアーキテクチャーに関する説明の大部分は、拡張版インテル® Core™ マイクロアーキテクチャーにも適用されます。両アーキテクチャーの違いについては、明確に記載しています。

E.3.1 インテル® Core™ マイクロアーキテクチャーのパイプラインの概要

インテル® Core™ マイクロアーキテクチャーのパイプラインは以下で構成されます。

- インオーダー発行フロントエンド: メモリーから命令ストリームをフェッチし、4 つの命令デコーダーによってデコードされた命令 (マイクロオペレーション: uop) をアウトオブオーダー実行コアに供給します。
- アウトオブオーダー・スーパースケーラー実行コア: 1 サイクルあたり最大 6 マイクロオペレーション (uop) を発行できます (表 2-27 を参照)。入力ソースが準備でき実行リソースが利用可能になり次第、実行できるようにマイクロオペレーション (uop) をリオーダーします。
- インオーダー・リタイアメント・ユニット: マイクロオペレーション (uop) の実行結果が反映され、プログラム順序に従ってアーキテクチャー・ステートが更新されるようにします。

インテル® Core™2 Extreme プロセッサー X6800、インテル® Core™2 Duo プロセッサー、インテル® Xeon® プロセッサー 3000/5100 番台には、インテル® Core™ マイクロアーキテクチャー・ベースの 2 つのプロセッサー・コアが実装されています。また、インテル® Core™2 Extreme クアッドコア・プロセッサー、インテル® Core™2 Quad プロセッサー、インテル® Xeon® プロセッサー 3200/5300 番台には、4 つのプロセッサー・コアが実装されています。これらのクアッドコア・プロセッサーの各物理パッケージには 2 つのプロセッサー・ダイが搭載され、各ダイには 2 つのプロセッサー・コアが搭載されています。図 E-5 に、各コア内のサブシステムの機能を示します。

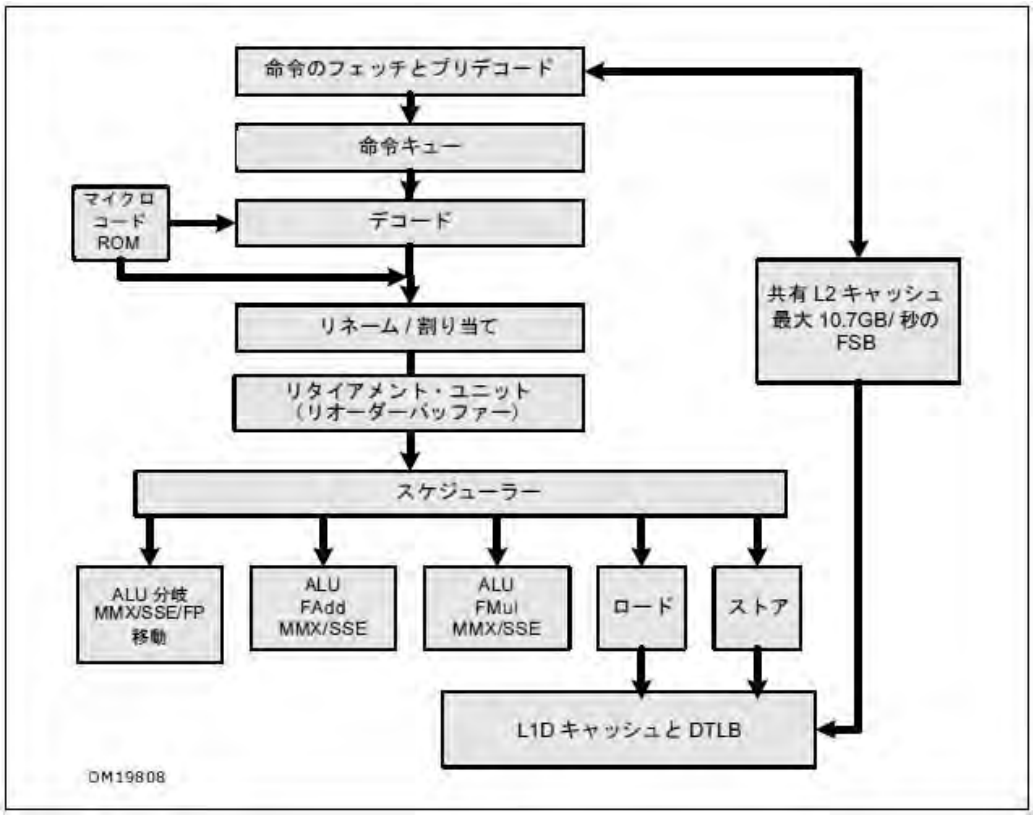


図 E-5 インテル® Core™ マイクロアーキテクチャーのパイプラインの構造

E.3.2 フロントエンド

フロントエンドは、デコードされた命令 (マイクロオペレーション:uop) ストリームの供給を、6 マイクロオペレーション (uop) の発行が可能なアウトオブオーダー・エンジンへ維持する必要があります。表 E-21 に、フロントエンドの構成要素、それぞれの機能、マイクロアーキテクチャー設計上のパフォーマンスの課題を示します。

表 E-21 フロントエンドの構成要素

構成要	機能	パフォーマンスの課題
分岐予測ユニット (BPU)	• 各種の分岐タイプ (条件付き、間接、直接、コール、リターン) を予測することによって、最も実行される可能性の高い命令を命令フェッチユニットがフェッチできるようにします。分岐タイプごとに専用のハードウェアを使用します。	• 投機実行を有効にします。 • パイプラインにフェッチされる「非アーキテクチャー・パス」[1] 内のコード量を削減することによって、投機実行を効率化します。
命令フェッチユニット	• 実行される可能性の高い命令をプリフェッチします。 • 実行頻度の高い命令をキャッシュします。 • 命令をプリデコードおよびバッファリングして、命令ストリームの不規則さにかかわらず一定の帯域幅を維持します。	• 可変長命令フォーマットは、デコード帯域幅のばらつき (バブル) の原因となります。 • 実行される分岐とアライメントされていない分岐先は、フェッチユニットの帯域幅全体に悪影響を及ぼします。
命令キューとデコードユニット	• 最大 4 命令 (マクロフュージョンの場合は最大 5 命令) をデコードできます。 • スタックポインター追跡アルゴリズムによって、プロシージャーの開始/終了処理を効率化します。 • パフォーマンスと効率の向上を図る、マクロフュージョン機能を提供します。 • 命令キューはループキャッシュとしても使用され、一部のループを高帯域幅および低消費電力で実行できます。	• 命令あたりの処理量が変化する場合、不定数のマイクロオペレーション (uop) への展開が必要となります。 • プリフィクス付きの命令はデコードが複雑になります。 • レングス変更プリフィクス (LCP) はフロントエンド・バブルの原因となります。

注意:

1. 実行されるとプロセッサーが判断しましたが、別のパスの方が適切であることが判明したため、予測からそれたコードパスを指します。

E.3.2.1 分岐予測ユニット

分岐予測によって、プロセッサーは、分岐の方向が判明する前に、以前に実行した命令を開始できます。すべての分岐は、分岐予測ユニット (BPU) によって予測されます。BPU には以下の機能があります。

- 16 エントリーのリターン・スタック・バッファー (RSB): BPU は RET 命令を正確に予測できます。
- BPU ルックアップのフロントエンド・キューイング: BPU は、一度に 32 バイト (フェッチエンジンの幅の 2 倍) の分岐予測を行うことができます。これにより、実行される分岐をペナルティーなしで予測できます。
- この BPU のメカニズムでは一般的に、実行される分岐のペナルティーを排除できますが、ソフトウェアは、実行されない分岐よりも実行される分岐の方が多くのリソースを消費するとみなすべきです。

BPU は以下のタイプの予測を行います。

- 直接コールおよびジャンプ: 実行されるか、されないか、いずれの予測にかかわらず分岐先はターゲット配列として読み出されます。
- 間接コールおよびジャンプ: 変化しない分岐先を持つか、プログラム動作に応じて変化する分岐先を持つかの、いずれかとして予測されます。
- 条件分岐: 分岐先と、分岐が実行されるかどうかを予測します。
- ソフトウェアを BPU 向けに最適化する際の詳細については、3.4 節「フロントエンドの最適化」を参照してください。

E.3.2.2 命令フェッチユニット

命令フェッチユニットは、命令トランスレーション・ルックアサイド・バッファー (ITLB)、命令プリフェッチャー、命令キャッシュ、命令キュー (IQ) のプリデコード・ロジックで構成されます。

命令キャッシュと ITLB

命令フェッチは、ITLB 介して命令キャッシュと命令プリフェッチ・バッファーへの 16 バイト・アライメントの参照を行います。命令キャッシュでヒットすると、16 バイト分の命令がプリデコーダーに供給されます。実行されるコードによりますが、通常のプログラムでは命令長は平均 1 命令あたり 4 バイトをわずかに下回ります。ほとんどの命令は、すべてのデコーダーでデコード可能であり、多くの場合、デコーダーは 1 サイクルでフェッチした命令全体をデコードできます。

分岐先がアライメントされていないと、フェッチされた 16 バイトのオフセット分だけ命令バイト数が減少します。また、分岐が実行されると、分岐命令以降のバイトがデコードされないため、デコーダーに供給される命令バイト数が減少します。一般的な整数コードでは、平均 10 命令ごとに分岐が実行されます。つまり、3 ～ 4 サイクルごとに「部分的な (パーシャル)」命令フェッチが行われることになります。

通常、パイプラインのほかの部分がストールすることが多いため、フロントエンド・スタベーションによりパフォーマンスが低下することはありません。例えば、より長い命令 (インテル® SSE2 整数メディアカーネルなど) で構成される極端に高速なコードの場合、分岐先をアライメントすることで命令のスタベーションを防止しメリットが得られる場合があります。

命令プリデコーダー

プリデコード・ユニットは、命令キャッシュやプリフェッチ・バッファーから 16 バイトを受け入れて、以下のタスクを実行します。

- 命令長を判断します。
- 命令を修飾するすべてのプリフィクスをデコードします。
- デコーダーのために命令の属性ごとにマークを付けます (例えば、「分岐」)。

プリデコード・ユニットは、1 サイクルあたり最大 6 命令を命令キューに書き込むことができます。6 個以上の命令がフェッチに含まれていた場合、すべての命令が命令キューに書き込まれるまで、プリデコーダーは 1 サイクルあたり最大 6 命令の割合でプリデコードを続行します。以降のフェッチがプリデコードを開始できるのは、現在のフェッチが完了してからです。

7 命令がフェッチされた場合、プリデコーダーは最初の 6 命令を 1 サイクルでデコードし、次のサイクルで残りの 1 命令をデコードします。この場合、1 サイクルあたり 3.5 命令のデコードに対応することになります。1 サイクルあたりの命令数 (IPC) のレートが十分に最適化されていない場合でも、これはほとんどのアプリケーションのパフォーマンスを上回ります。ソフトウェアは通常、命令スタベーションを防ぐため特別なことを行う必要はありません。

以下の命令プリフィクスを使用すると、命令のデコード中に問題が発生します。これらのプリフィクスは、レングス変更プリフィクス (LCP) と呼ばれ、命令長を動的に変更します。

- オペランド・サイズ・オーバーライド (66H): ワード即値データを持つ命令の先頭に付きます。
- アドレス・サイズ・オーバーライド (67H): リアルモード、16 ビット保護モード、または 32 ビット保護モードで、mod R/M を持つ命令の先頭に付きます。

プリデコーダーは、フェッチラインで LCP に遭遇すると、低速の命令長デコード・アルゴリズムを使用しなければなりません。低速の命令長デコード・アルゴリズムでは、6 サイクルでフェッチがデコードされます (通常は 1 サイクル)。

一般に、プロセッサー・パイプライン内の通常のキューイングでは、LCP のペナルティーを隠蔽できません。

インテル® 64 アーキテクチャー命令セットの REX プリフィクス (4xh) は、2 つの命令 (MOV オフセットと MOV 即値) サイズを変更できます。ただし、LCP のペナルティーは発生しないので、LCP とはみなされません。

E.3.2.3 命令キュー (IQ)

命令キュー (IQ) は、命令プリデコード・ユニットと命令デコーダーの間に存在し、18 命令の深さを持ちます。1 サイクルあたり最大 5 命令を送り出し、1 サイクルあたり 1 つのマクロフュージョンをサポートします。また、18 命令未満のループではループキャッシュとしても機能し、ループキャッシュは、以下のように動作します。

BPU 内にはループストリーム検出器 (LSD) があります。LSD は、IQ からのストリーミングの候補になるループを検出します。該当するループが見つかると、命令バイトがロックダウンされ、ループは予測ミスによって終了するまで IQ からのストリーミングを許可されます。ループが IQ から実行されると、低消費電力で高い帯域幅を提供します (フロントエンド・パイプラインの残りの多くの機能が停止されるため)。

LSD には以下のメリットがあります。

- 実行される分岐による帯域幅の損失がない。
- アライメントされていない命令による帯域幅の損失がない。
- プリデコード・ステージを通過済みなので LCP のペナルティーがない。
- 命令キャッシュ、BPU、プリデコード・ユニットをアイドル状態にできるので、フロントエンドの消費電力が減少する。

ソフトウェアは、状況に応じてループキャッシュ機能を利用すべきです。ループアンロールとその他のコード最適化を適用すると、ループが大きくなりすぎて、LSD に収まらないことがあります。パフォーマンスが高いコードを作成するには、ループキャッシュ機能をオーバーフローしても、通常はループアンロールを行う方がパフォーマンスの観点からは望まれます。

E.3.2.4 命令デコード

インテル® Core™ マイクロアーキテクチャーには、4 の命令デコーダーが備わっています。最初のデコーダー 0 は、最大で 4 マイクロオペレーション (uop) で構成されるインテル® 64 命令と IA-32 命令をデコードできます。残りの 3 つのデコーダーは、単一マイクロオペレーション (uop) の命令を処理します。マイクロシーケンサーは、1 サイクルあたり最大 3 マイクロオペレーション (uop) に対応可能であり、5 マイクロオペレーション (uop) 以上の命令のデコードを支援します。

すべてのデコーダーは、一般的なケースにおける単一マイクロオペレーション (uop) のフロー (マイクロフュージョン、マクロフュージョン、スタックポインター追尾など) をサポートしています。したがって、3 つのシンプルなデコーダーは、単一マイクロオペレーション (uop) の命令のデコードに限定されているわけではありません。命令を 4-1-1-1 のテンプレートにパックする必要はなく、また推奨されません。

マクロフュージョンでは、2 つの命令が単一のマイクロオペレーション (uop) にマージされます。インテル® Core™ マイクロアーキテクチャーは、32 ビット操作 (インテル® 64 アーキテクチャーの互換サブモードを含む) では 1 サイクルあたり 1 つのマクロフュージョンを処理できます。ただし、バイト長が長い命令を頻繁に使用するコードほどマクロフュージョンのハードウェア・サポートを利用する可能性が少なく、64 ビット・モードでは対応していません。

E.3.2.5 スタックポインター追尾

インテル® 64 アーキテクチャーと IA-32 アーキテクチャーには、PUSH、POP、CALL、LEAVE、RET のように、パラメーターの受け渡しやプロシージャーの開始/終了に共通して使用される命令があります。各命令は、スタック・ポインター・レジスター (RSP) を暗黙的に更新し、コントロールとパラメーターが組み合わされたスタック操作をソフトウェアの介入なしに行います。これらは、従来のマイクロアーキテクチャーでは複数のマイクロオペレーション (uop) によって実装されていた命令です。

スタックポインター追尾は、この暗黙的な RSP の更新をすべて、デコーダーに搭載されるロジックで行います。この機能には、以下の利点があります。

- インテル® Core™ マイクロアーキテクチャーでは PUSH、POP、RET は単一マイクロオペレーション (uop) の命令なので、デコード帯域幅が向上します。
- RSP の更新が実行リソースと競合しないので、実行帯域幅の節約になります。
- マイクロオペレーション (uop) 間の暗黙的なシリアル依存関係が排除されるので、アウトオブオーダー実行エンジンの並列処理が向上します。
- 小型の専用ハードウェアで RSP が更新されるので、電力効率が向上します。

E.3.2.6 マイクロフュージョン

マイクロフュージョンでは、同一命令の複数のマイクロオペレーション (uop) が単一の複雑なマイクロオペレーション (uop) に融合されます。複雑なマイクロオペレーション (uop) は、アウトオブオーダー実行コアにディスパッチされます。マイクロフュージョンには、次のようなパフォーマンス上の利点があります。

- デコードからリタイアメントへの命令帯域幅が向上します。
- 複雑なマイクロオペレーション (uop) では、短いフォーマット (ビット密度が低い形式) の処理が増加するので、一定処理量におけるマシン全体の「ビット・トグル」が減少し、アウトオブオーダー実行エンジンのストレージ量が事実上増加します。そのため消費電力が減少します。

多くの命令には、レジスター方式とメモリー方式があります。メモリーオペランドを含む方式は、レジスター方式よりも長いフローのマイクロオペレーション (uop) にデコードされます。マイクロフュージョンを利用すると、デコード帯域幅の損失を心配することなく、ソフトウェアがメモリーとレジスター間の操作によって、実際のプログラム動作を表現することを可能にします。

E.3.3 実行コア

インテル® Core™ マイクロアーキテクチャーの実行コアは、スーパースケーラーであり、命令をアウトオブオーダーで処理できます。依存関係チェーンが原因でプロセッサーがリソース (2 次データ・キャッシュラインなど) を待機している間、実行コアは別の命令を実行します。このため、1 サイクルあたりに実行される命令数 (IPC) の全体的なレートが増加します。

実行コアは、以下の 3 つのコンポーネントを備えています。

- **リネーマー:** マイクロオペレーション (uop) をフロントエンドから実行コアに移動します。アーキテクチャー・レジスターは、多数のセットを備えるマイクロアーキテクチャー・レジスターにリネームされます。リネーミングによって、リードアフターリード・ハザードやライトアフターリード・ハザードと呼ばれる依存関係を排除できます。
- **リオーダーバッファー (ROB):** 各ステージで処理されたマイクロオペレーション (uop) を保持し、完了したマイクロオペレーション (uop) のバッファリング、インオーダーでのアーキテクチャー・ステートの更新、および例外の順序付けを管理します。ROB には、命令をインフライトで処理する 96 個のエントリーがあります。
- **リザベーション・ステーション (RS):** すべてのソースオペランドが準備できるまでマイクロオペレーション (uop) のキューイングを行い、準備ができたマイクロオペレーション (uop) を利用可能な実行ユニットにスケジューリングおよびディスパッチします。RS には 32 個のエントリーがあります。

アウトオブオーダー・コアの初期ステージでは、マイクロオペレーション (uop) がフロントエンドから ROB と RS に移動されます。この過程では、アウトオブオーダー・コアは以下のステップを実行します。

- リソース (例えば、ロードバッファーやストアバッファー) をマイクロオペレーション (uop) に割り当てます。
- マイクロオペレーション (uop) を適切な発行ポートにバインドします。
- マイクロオペレーション (uop) のソースとデスティネーションをリネームして、アウトオブオーダー実行を可能にします。
- データが即値または計算済みレジスター値の場合、データをマイクロオペレーション (uop) に供給します。

以下では、各種の一般的な操作をコアが効率良く実行する方法について説明します。

- **レイテンシーが単一のサイクルであるマイクロオペレーション(uop):** レイテンシーが単一のサイクルであるマイクロオペレーション (uop) のほとんどは、複数の実行ユニットによって実行できるため、複数の依存操作のストリームを迅速に実行できます。
- **レイテンシーが長く、頻繁に使用されるマイクロオペレーション(uop):** この種のマイクロオペレーション (uop) は、実行ユニットがパイプライン化されているので、パイプライン内の異なるステージで複数のマイクロオペレーション (uop) を同時に実行できます。
- **レイテンシーがデータに依存した操作:** 除算など一部の操作では、レイテンシーがデータに依存します。整数の除算では、オペランドを解析し、オペランドの有意部分のみの計算を実行するため、一般的な小さな数の除算が高速化されます。
- **オペランドが特定の制約に一致した場合の、レイテンシーが固定された浮動小数点演算:** この制約に一致しないオペランドは例外的なケースとみなされ、長いレイテンシーと低いスループットで実行されます。低スループットのケースは、一般的なケースのレイテンシーとスループットには影響しません。
- **1 次キャッシュにヒットしてもレイテンシーが可変のメモリーオペランド:** フォワーディングするかしないかが不明なロードは、ストアアドレスが解決されるまで待機してから実行されます。メモリー・オーダー・バッファー (MOB) は、すべてのメモリー操作を受け入れて処理します。MOB の詳細については、2.5.4 節を参照してください。

E.3.3.1 発行ポートと実行ユニット

スケジューラーは、1 サイクルあたり最大 6 つのマイクロオペレーション (uop) を各ポートにディスパッチできます。表 E-22 に、インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャーの発行ポートを示します。前者は CPUID シグネチャーの DisplayFamily 値と DisplayModel 値が 06_0FH によって表され、後者は 06_17H によって表されます。この表には、一般的な整数演算および浮動小数点 (FP) 演算のレイテンシーとスループットのサイクル数が発行ポートごとに記載されています。

表 E-22 インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャーの発行ポート

実行可能な操作	レイテンシー、スループット		説明 [1]
	シグネチャー = 06_0FH	シグネチャー = 06_17H	
整数 ALU 整数 SIMD ALU FP/SIMD/インテル® SSE2 ムーブと論理演算	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	64 ビット・モードの整数 MUL を含む。 発行ポート 0、ライトバック・ポート 0
単精度 FP MUL 倍精度 FP MUL	4, 1 5, 1	4, 1 5, 1	発行ポート 0、ライトバック・ポート 0
FP MUL (X87) FP シャッフル DIV/SQRT	5, 2 1, 1	5, 2 1, 1	発行ポート 0、ライトバック・ポート 0 FP シャッフルは QW シャッフルを処理しない。
整数 ALU 整数 SIMD ALU FP/SIMD/インテル® SSE2 ムーブと論理演算	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	64 ビット・モードの整数 MUL を除く。 発行ポート 1、ライトバック・ポート 1
FP ADD QW シャッフル	3, 1 1, 1[2]	3, 1 1, 1[3]	発行ポート 1、ライトバック・ポート 1
整数ロード FP ロード	3, 1 4, 1	3, 1 4, 1	発行ポート 2、ライトバック・ポート 2
ストアアドレス [4]	3, 1	3, 1	発行ポート 3
ストアデータ [5]			発行ポート 4
整数 ALU 整数 SIMD ALU FP/SIMD/インテル® SSE2 ムーブと論理演算	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	発行ポート 5、ライトバック・ポート 5
QW シャッフル 128 ビットのシャッフル/パック/アンパック	1, 1[2] 2-4, 2-4[6]	1, 1[3] 1-3, 1[7]	発行ポート 5、ライトバック・ポート 5

注意:

1. 同じポートを使用するレイテンシーが異なる操作を混在すると、ライトバック・バスの競合が発生して、全体的なスループットが低下する可能性があります。
2. 128 ビット命令は、長いレイテンシーと低いスループットで実行されます。
3. 128 ビット・シャッフル・ユニットはポート 5 を使用します。
4. ストアされるデータのアドレスを使って、ストア・フォワーディングとストア・リタイアメントのロジックを準備します。
5. ストアされるデータを使って、ストア・フォワーディングとストア・リタイアメントのロジックを準備します。
6. 命令によって異なりますが、128 ビット命令は、QW シャッフルユニットを使って実行されます。
7. 命令によって異なりますが、インテル® Core™ マイクロアーキテクチャーでは、QW シャッフルユニットの代わりに 128 ビット・シャッフル・ユニットが使用されます。

RS は、1 サイクルごとに最大 6 マイクロオペレーション (uop) をディスパッチできます。各サイクルでは、最大で 4 つの結果が RS と ROB にライトバックされ、RS はその結果を次のサイクルですぐに使用できます。この高い実行帯域幅によって、デコードおよびリタイアされるマイクロフュージョン済みマイクロオペレーション (uop) の複雑な処理の実行バーストを維持します。

実行コアは、以下の 3 つの実行スタックを備えています。

- SIMD 整数
- 通常の整数
- x87/SIMD 浮動小数点

実行コアは、メモリークラスターとの接続も備えています。図 E-6 を参照してください。

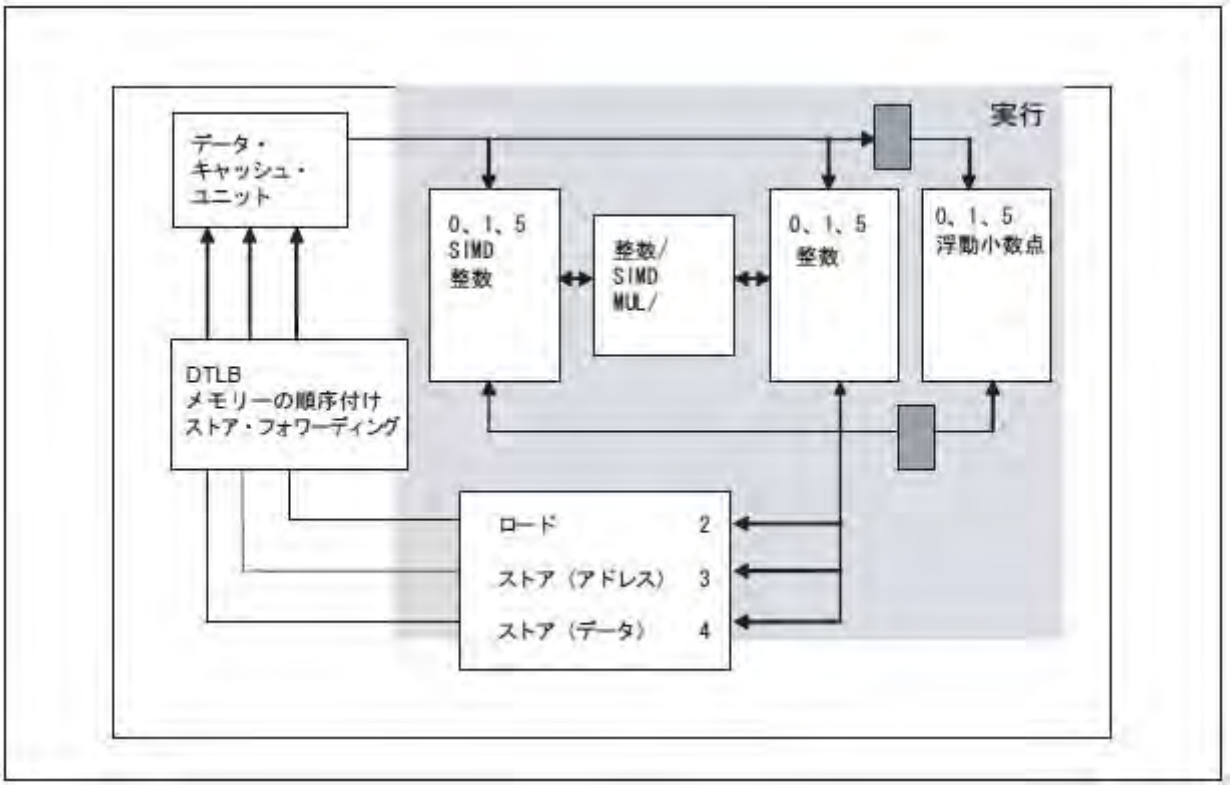


図 E-6 インテル® Core™ マイクロアーキテクチャーの実行コア

実行ブロック（グレー）内の 2 つの濃い色の四角形に注目してください。これらの四角形は、整数スタックと SIMD 整数スタックを浮動小数点スタックに接続するパス上に存在しています。これは、バイパス遅延と呼ばれる余分なサイクルとなって表れます。1 次キャッシュから浮動小数点ユニットへのデータ転送には、1 サイクルのレイテンシーが追加されます。図 E-6 の濃い色の四角形は、この余分なレイテンシー・サイクルの発生を表しています。

E.3.4 インテル® アドバンスト・メモリー・アクセス

インテル® Core™ マイクロアーキテクチャーでは、各コアに 1 次命令キャッシュと 1 次データキャッシュが搭載されています。2 つのコアは 2MB または 4MB の 2 次キャッシュを共有します。すべてのキャッシュはライトバックであり、インクルーシブではありません。各コアは、以下の要素を備えています。

- **データ・キャッシュ・ユニット(DCU)と呼ばれる 1 次データキャッシュ** — DCU は、将来発生すると予測される複数のキャッシュミスを処理して、後続のストアとロードを迅速に処理することができます。DCU はキャッシュ・コヒーレンシーの維持をサポートします。DCU の仕様は以下のとおりです。
  - 32KB のサイズ
  - 8 ウェイ・セット・アソシアティブ
  - 64 バイトのラインサイズ

- **データ・トランスレーション・ルックアサイド・バッファー(DTLB)** — インテル® Core™マイクロアーキテクチャーの DTLB は、2 レベルの階層を実装しています。各レベルの DTLB には複数のエントリーがあり、4KB ページ

またはラージページをサポートします。内側のレベル (DTLB0) のエントリーは、ロードに使用されます。外側のレベル (DTLB1) のエントリーは、ストア操作と、DTLB0 でミスをしたロードをサポートします。すべてのエントリーは、4 ウェイ・アソシアティブです。各 DTLB のエントリーのリストを以下に示します。

- ラージページの DTLB1: 32 エントリー
- 4KB ページの DTLB1: 256 エントリー
- ラージページの DTLB0: 16 エントリー
- 4KB ページの DTLB0: 16 エントリー

ロードが DTLB0 でミスし DTLB1 でヒットした場合、2 サイクルのペナルティーが発生します。一部のディスパッチで DTLB0 が使用される場合のみ、ソフトウェアはこのペナルティーを受けます。DTLB1 および PMH へのミスに関連した遅延は通常、スマート・メモリー・アクセスの設計によりノンブロッキングです。

- **ページ・ミス・ハンドラー(PMH)**
- **メモリー・オーダー・バッファー(MOB)** — 以下の機能を備えています。
  - ロードとストアを投機的にアウトオブオーダー発行することができます。
  - リタイアしたロードとストアがリタイア時に正確なデータを得られます。
  - ロードとストアがインテル® 64 命令と IA-32 アーキテクチャーのメモリーの順序付け規則に従うことを確実にします。

インテル® Core™ マイクロアーキテクチャーのメモリークラスターは、以下を利用してメモリー操作の高速化を図ります。

- 128 ビットのロード操作とストア操作
- 1 次キャッシュからのデータ・プリフェッチ
- 2 次キャッシュへのプリフェッチで利用されるデータ・プリフェッチ・ロジック
- ストア・フォワーディング
- メモリー・ディスアンビゲーション (明確化)
- 8 個のフィル・バッファー・エントリー
- 20 個のストア・バッファー・エントリー
- メモリー操作のアウトオブオーダー実行
- パイプライン化された所有権読み出し (RFO) 操作

ソフトウェアをメモリークラスター向けに最適化する詳細については、3.6 節「メモリーアクセスの最適化」を参照してください。

E.3.4.1 ロードとストア

インテル® Core™ マイクロアーキテクチャーでは、1 サイクルあたり 1 つの 128 ビット・ロードと 1 つの 128 ビット・ストアを、それぞれ異なるメモリー・ロケーションに対して実行できます。他の命令やメモリー操作は、アウトオブオーダーでメモリー操作を実行できます。

ロードは以下を実行できます。

- ロードアドレスとストアアドレスの範囲が競合していないことが明らかな場合は、先行するストアの前にロードを発行することができます。
- 先行する分岐が解決される前に、投機実行できます。
- キャッシュミスにアウトオブオーダーでオーバーラップして対応します。
- 競合するアドレスにストアが行われないことを見込んで、先行するストアの前にロードを発行できます。

ロードは以下を実行できません。

- あらゆるフォルトやトラップに対し投機的に対応すること。
- キャッシュできないメモリータイプに投機的にアクセスすること。

フォルトの発生したロードやキャッシュできないロードが検出されると、リタイアメントまで待機した上で、プログラマーから可視な状態に更新されます。x87 ロードと浮動小数点 SIMD ロードでは、1 クロックのレイテンシーが追加されます。

メモリーへのストアは、2 つのフェーズで実行されます。

- **実行フェーズ** — ストア・フォワーディングに対処するためアドレスとデータを使ってストアバッファーを準備します。ディスパッチ・ポート (ポート 3 および 4) が使用されます。
- **完了フェーズ** — ストアがリタイアし、プログラマーがメモリーを参照できるようになります。実行中のロードとキャッシュバンクが競合する場合があります。ストアのリタイアメントはメモリー・オーダー・バッファーによってバックグラウンド・タスクとして処理され、データはストアバッファーから 1 次キャッシュに移動されます。

E.3.4.2 1 次キャッシュからのデータ・プリフェッチ

インテル® Core™ マイクロアーキテクチャーでは、2 つのハードウェア・プリフェッチャーを提供して、1 次データキャッシュへのプリフェッチを行うことにより、プログラムが高速にデータをアクセスすることを可能にしています。

- **データ・キャッシュ・ユニット (DCU) プリフェッチャー:** このプリフェッチャーはストリーミング・プリフェッチャーとも呼ばれ、最後にロードされたデータへのユニット・ストライド・アクセスによって開始されます。プロセッサーは、このアクセスをストリーミング・アルゴリズムの一部であるとみなし、次のラインを自動的にフェッチします。
- **命令ポインター (IP) ベースのストライド・プリフェッチャー:** このプリフェッチャーは、個々のロード命令を追尾します。ロード命令に規則的なアドレススキップがあることが検出された場合、現在のアドレスとこの間隔の合計である次のアドレスがプリフェッチされます。前方または後方へのプリフェッチが可能であり、最大で 4KB ページの半分または 2K バイトのストライドを識別できます。

データ・プリフェッチは、以下の条件が満たされた場合にロード操作により起動されます。

- ライトバック・メモリーからのロード。
- 4K バイトのページ境界内でのプリフェッチ要求である。
- パイプライン内にフェンスやロックがない。
- その他のロードミスがあまり発生していない。
- バスがそれほどビジー状態ではない。
- 連続的なストアのストリームが発生していない。

DCU プリフェッチは以下の影響を及ぼします。

- 大きな構造体のデータが、プログラムで使用される順序で連続して配置されている場合、パフォーマンスが向上します。
- アクセスパターンに局所性がなくまばらな場合、帯域幅の問題によりわずかにパフォーマンスが低下することがあります。
- まれに、アルゴリズムのワーキングセットがキャッシュの大部分を占めるようにチューニングされ、プログラムから要求されたキャッシュラインが不要なプリフェッチによって排出されている場合、1 次キャッシュの容量が原因でハードウェア・プリフェッチが重大なパフォーマンス低下をもたらすことがあります。

ハードウェアに依存してデータ・トラフィックを予想するハードウェア・プリフェッチャーとは異なり、ソフトウェア・プリフェッチ命令は、プログラマーに依存してキャッシュミスの頻度を予想します。ソフトウェア・プリフェッチは、データを含むキャッシュラインを要求されるレベルのキャッシュ階層に移動するヒントとして機能します。ソフトウェア制御プリフェッチは、コードのプリフェッチではなく、データのプリフェッチを対象とします。

E.3.4.3 データ・プリフェッチ・ロジック

データ・プリフェッチ・ロジック (DPL) は、2 次 (L2) キャッシュへの DCU の過去の要求パターンに基づいて、2 次キャッシュに対しデータのプリフェッチを行います。DPL は、DCU からのアドレスを保持するため 2 つの独立した

配列を備えており、一方はアップストリーム用 (12 エントリー)、他方はダウンストリーム用 (4 エントリー) です。DPL は、エントリーごとに 1 つの 4KB ページへのアクセスを追尾します。アクセスしたページがこの配列のどちらにも存在しない場合、新たに配列のエントリーが割り当てられます。

DPL は、インクリメントする要求シーケンス (ストリームとも呼ばれる) があるかどうか DCU の読み出しを監視します。ストリームの 2 回目のアクセスを検出すると、DPL は次のキャッシュラインをプリフェッチします。例えば、DCU がキャッシュライン A および A + 1 を要求した場合、DPL は、近い将来に DCU によってキャッシュライン A + 2 が要求されると仮定します。次に DCU が A + 2 を読み出すと、DPL はキャッシュライン A + 3 をプリフェッチします。DPL は、「後方参照」ループでも同様に機能します。

DPL は、インテル® Pentium® M プロセッサーで導入されました。インテル® Core™ マイクロアーキテクチャーでは、以下の機能が DPL に追加されています。

- ストリームがキャッシュラインをスキップする場合など、DPL がより複雑なストリームを検出できるようになりました。DPL は、L2 ルックアップごとに 2 つのプリフェッチ要求を発行できます。インテル® Core™ マイクロアーキテクチャーでは、DPL はロード要求に先立ち、最大 8 のラインを処理できます。
- インテル® Core™ マイクロアーキテクチャーの DPL は、バス帯域幅と要求数に応じて動的に調整を行います。DPL プリフェッチャーはバスがビジーでない場合は最大限先までプリフェッチし、バスがビジーな場合はあまり先までプリフェッチしません。
- DPL は、各種のアプリケーションやシステム構成に応じて調整を行います。

マルチコア・プロセッサーのそれぞれのコアのエントリーは、別々に処理されます。

E.3.4.4 ストア・フォワーディング

インテル® Core™ マイクロアーキテクチャーでは、ストアに続くロードがストアによってメモリーに書き込まれたデータを再読み込みする場合、ストア操作からロードにデータを直接転送できます。これはストア-ロード・フォワーディングと呼ばれ、ロードがメモリーを介さずにストア操作から直接データを取得できるので、サイクル数の節約になります。

ストア-ロード・フォワーディングを行うには、以下の規則に従わなければなりません。

- ストアは、ロードに先行する該当アドレスへの最後のストアでなければなりません。
- ストアされるデータは、ロードされるデータとサイズが同じか、それよりも大きくなければなりません。
- ロードは、キャッシュライン境界をまたぐことはできません。
- ロードは、8 バイト境界をまたぐことはできません。ただし、16 バイト・ロードは例外です。
- ロードは、以下の例外を除いて、ストアアドレスの先頭にアライメントされていなければなりません。
  - アライメントされた 64 ビット・ストアでは、その半分である 32 ビットのいずれにも転送できます。
  - アライメントされた 128 ビット・ストアでは、その 4 分の 1 である 32 ビットのいずれにも転送できます。
  - アライメントされた 128 ビット・ストアでは、その半分である 64 ビットのいずれにも転送できます。

ソフトウェアは、最後の規則に対する例外を適用して、サブフィールドを転送する能力を犠牲にせず複雑な構造を移動できます。

拡張版インテル® Core™ マイクロアーキテクチャーでは、ストア・フォワーディングに関するアライメントの制限が緩和されています。拡張版インテル® Core™ マイクロアーキテクチャーは、後続のロードのアライメントが先行するストアに合っていない複数の状況でストア・フォワーディングを実行できます。拡張版インテル® Core™ マイクロアーキテクチャーでは許可されていますが、インテル® Core™ マイクロアーキテクチャーでは許可されていない 6 つのケース (背景がグラデーションのもの) を図 E-7 に示します。背景が網かけのものは、両者で実行可能なストア・フォワーディングです。

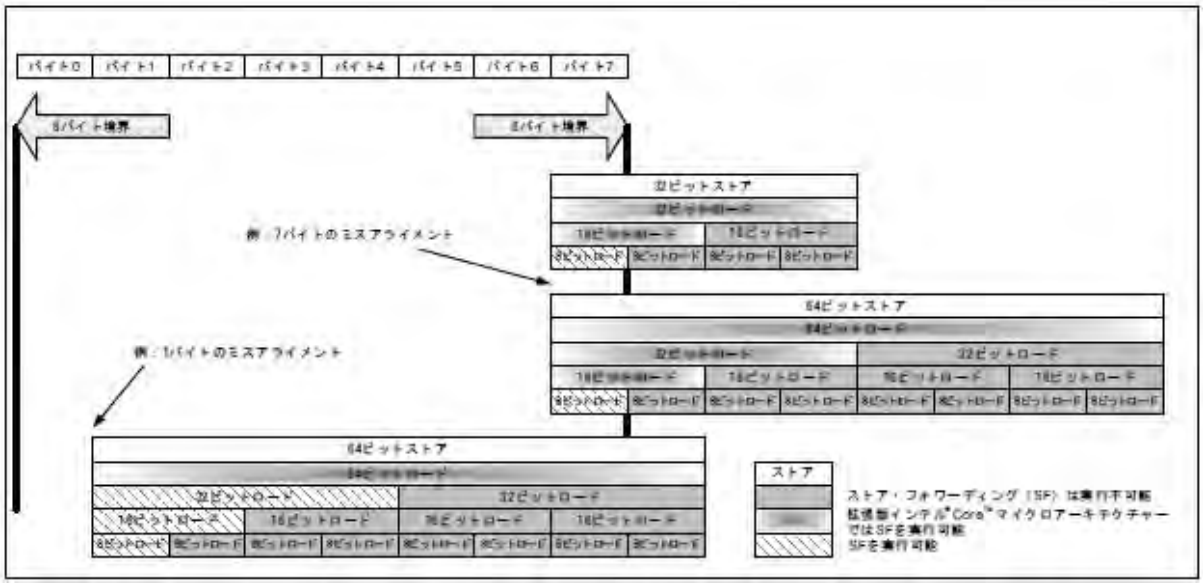


図 E-7 拡張版インテル® Core™ マイクロアーキテクチャーにおけるストア・フォワーディングの強化

E.3.4.5 メモリー・ディスアンビゲーション

E.2.5.2 節「L1D キャッシュ」の「メモリー・ディスアンビゲーション」を参照してください。

E.3.5 インテル® アドバンスト・スマート・キャッシュ

インテル® Core™ マイクロアーキテクチャーでは、多くの機能が単一ダイ上の 2 つのプロセッサー・コアに最適化されています。2 つのコアは、インテル® アドバンスト・スマート・キャッシュと呼ばれる 2 次キャッシュおよびバス・インターフェイス・ユニットを共有します。ここではインテル® アドバンスト・スマート・キャッシュの構成要素について説明します。図 E-8 に、インテル® アドバンスト・スマート・キャッシュのアーキテクチャーを示します。

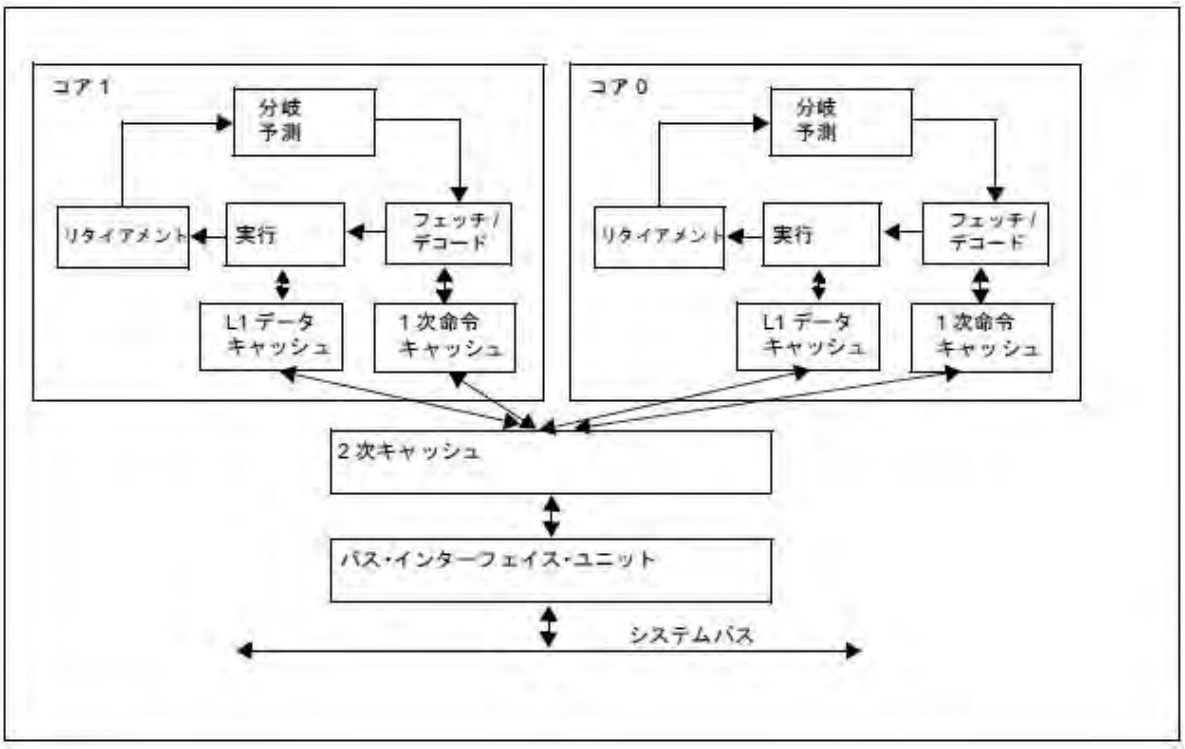


図 E-8 インテル® アドバンスト・スマート・キャッシュのアーキテクチャー

表 E-23 に、インテル® Core™ マイクロアーキテクチャーにおけるキャッシュのパラメーターの詳細を示します。CPUID 命令のキャッシュ・パラメーターを使用してキャッシュ階層 ID を列挙する際の詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』を参照してください。

表 E-23 インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサーのキャッシュ・パラメーター

レベル	容量	アソシアティブ (ウェイ)	ラインサイズ (バイト)	アクセス・レイテンシー (クロック)	アクセス・スループット (クロック)	書き込みアップデート方式
1 次	32 KB	8	64	3	1	ライトバック
命令	32 KB	8	なし	なし	なし	なし
2 次 (共有 L2)[1]	2、4 MB	8 または 16	64	14[2]	2	ライトバック
2 次 (共有 L2)[3]	3、6MB	12 または 24	64	15[2]	2	ライトバック
3 次 [4]	8、12、16 MB	16	64	~110	12	ライトバック

注意:

1. インテル® Core™ マイクロアーキテクチャー (CPUID シグネチャー DisplayFamily = 06H、DisplayModel = 0FH)。
2. ソフトウェアから見えるレイテンシーは、アクセスパターンとその他の要因に依存するため異なります。
3. 拡張版インテル® Core™ マイクロアーキテクチャー (CPUID シグネチャー DisplayFamily = 06H、DisplayModel = 17H または 1DH)。
4. 拡張版インテル® Core™ マイクロアーキテクチャー (CPUID シグネチャー DisplayFamily = 06H、DisplayModel = 1DH)。

E.3.5.1 ロード

ライトバック (WB) 方式のメモリー・ロケーションから命令がデータを読み出す場合、プロセッサーはそのデータが含まれるキャッシュラインを以下の順序でキャッシュとメモリーから検索します。

1. 自コアの DCU
2. 他コアの DCU と 2 次キャッシュ
3. システムメモリー

キャッシュラインは、変更された場合のみ他方のコアの DCU から取得され、キャッシュラインの可用性や 2 次キャッシュの状態は無視されます。

表 E-24 に、局所性が異なる最初の 4 バイトをメモリークラスターからフェッチする際の特性を示します。レイテンシーの列には、アクセス・レイテンシーの概算値が記載されています。ただし、実際のレイテンシーは、キャッシュのロード、メモリー・コンポーネント、そのパラメーターによって異なります。

表 E-24 インテル® Core™ マイクロアーキテクチャーでのロード操作とストア操作の特性

データ局所性	**ロード**		**ストア**	
	レイテンシー	**スループット**	**レイテンシー**	**スループット**
DCU	3	1	2	1
変更状態にある他方のコアの DCU	14 + 5.5 バスサイクル	14 + 5.5 バスサイクル	14 + 5.5 バスサイクル	
2 次キャッシュ	14	3	14	3
メモリー	14 + 5.5 バスサイクル + メモリー	バス読み出しプロトコルによって異なります	14 + 5.5 バスサイクル + メモリー	バス書き込みプロトコルによって異なります

変更されたキャッシュラインを排出して、新しいキャッシュライン用にスペースを確保しなければならないことがあります。変更されたキャッシュラインは新しいデータの取得と並行して排出されるので、レイテンシーは追加されません。ただし、データがメモリーにライトバックされる際は、キャッシュ帯域幅のほか、バス帯域幅も排出に使用される可能性があります。したがって、変更されたキャッシュラインの排出を伴うキャッシュミスが短時間に複数発生した場合、キャッシュ応答時間が全体的に低下します。

E.3.5.2 ストア

ライトバック方式のメモリー・ロケーションに命令がデータを書き込む際、プロセッサーはまず、自身の DCU においてキャッシュラインが排他状態 (Exclusive) または変更状態 (Modified) であることを確認します。プロセッサーは、指定された順序で以下の場所からキャッシュラインを検索します。

1. 自コアの DCU
2. 他コアの DCU と 2 次キャッシュ
3. システムメモリー

キャッシュラインは、変更された場合のみ他方のコアの DCU から取得され、キャッシュラインの可用性や 2 次キャッシュの状態は無視されます。所有権読み出しが完了した後、データが 1 次データキャッシュに書き込まれ、キャッシュラインが変更済みとしてマークされます。

所有権読み出しとデータのストアは、命令のリタイアメントの後に発生し、リタイアメントの順序に従います。したがって、ストア・レイテンシーは、通常、ストア命令自体には影響を与えることはありません。ただし、一部の連続したストアは、レイテンシーを累積してパフォーマンスに影響を与えることがあります。表 E-24 に、キャッシュラインの階層によって異なるストア・レイテンシーを示します。

E.4 Nehalem† マイクロアーキテクチャー

Nehalem† マイクロアーキテクチャーは、インテル® Core™ i7 プロセッサーおよびインテル® Xeon® プロセッサー 3400/5500/7500 番台が持つ数多くの革新的な機能の基盤となっています。このアーキテクチャーは、45nm の拡張版インテル® Core™ マイクロアーキテクチャーの成功を受けて開発され、以下の拡張機能を提供しています。

- **強化されたプロセッサー・コア**
  - 分岐予測と予測ミスからの回復を改善
  - ループ・ストリーミングの強化により、フロントエンドのパフォーマンスを高め、消費電力を軽減
  - アウトオブオーダー・エンジンでのバッファリングを深くすることにより、並列性を向上
  - 実行ユニットの強化により、CRC、文字列/テキスト処理、データ・シャッフリングを高速化

- **インテル® ハイパースレッディング・テクノロジー**
  - 1 コアあたり 2 つのハードウェア・スレッド (論理プロセッサー) を提供
  - 4 マイクロオペレーション (uop) の発行が可能な実行エンジン、大容量 L3 キャッシュ、広いメモリー帯域幅の利点を活用

- **インテル® スマート・メモリー・アクセス**
  - システムメモリーへの低レイテンシー・アクセスとスケーラブルなメモリー帯域幅を実現する統合メモリー・コントローラー
  - インクルーシブ (包括的) な共有 L3 キャッシュを持つ新しいキャッシュ階層構造によって、スヌープ・トラフィックを軽減
  - 2 レベルの TLB と、TLB サイズを拡大
  - アライメントされないメモリーの高速アクセス

- **革新的な専用パワー・マネジメント機能**
  - 最適化された組込みファームウェアと統合マイクロコントローラーによって消費電力を管理
  - 温度、電流、消費電力を測定する組込みリアルタイム・センサー
  - コアごとに消費電力をオン/オフできる統合パワーゲート
  - メモリーやリンク・サブシステムの消費電力を削減する汎用性

Westmere† マイクロアーキテクチャーは、Nehalem† マイクロアーキテクチャーの 32nm バージョンです。後者の機能はすべて前者にも適用されます。

E.4.1 マイクロアーキテクチャー・パイプライン

Nehalem† マイクロアーキテクチャーは、65nm のインテル® Core™ マイクロアーキテクチャーで導入された、4 マイクロオペレーション (uop) の発行が可能なマイクロアーキテクチャー・パイプラインを継承しています。図 E-9 に、インテル® Core™ i7 プロセッサーで実装されている Nehalem† マイクロアーキテクチャーのパイプラインの基本構成要素を示します。ただし、図 E-9 のパイプラインでは、4 つのコアのうち 2 つのみが描かれています。

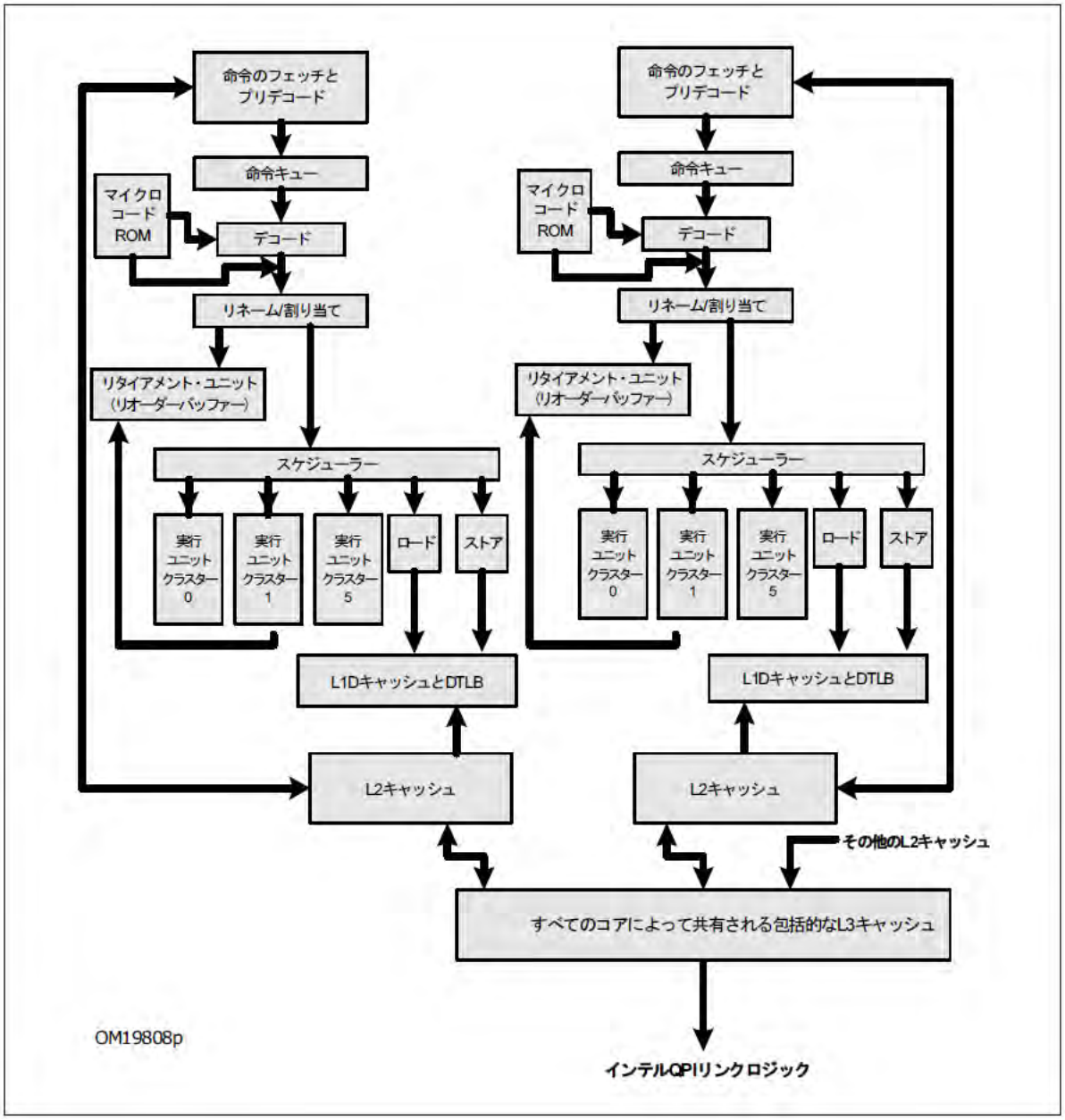


図 E-9 Nehalem† マイクロアーキテクチャーのパイプラインの構造

Nehalem† マイクロアーキテクチャーのパイプライン長は、分岐予測ミスの遅延を測定した場合、45nm のインテル® Core™2 プロセッサー・ファミリーにおける従来のパイプラインよりも 2 サイクルだけ長くなっています。フロントエンドは、1 サイクルあたり最大 4 命令をデコード可能であり、2 つの論理プロセッサーが交互のサイクルで命令ストリームをデコードすることによって、2 つのハードウェア・スレッドをサポートします。フロントエンドには、分岐処理、ループ検出、MSROM スループットなどを扱う拡張機能が備わっています。

スケジューラー (リザベーション・ステーション) は、1 サイクルあたり最大 6 マイクロオペレーション (uop) を 6 つの発行ポートにディスパッチできます (図 E-9 では 5 つの発行ポートのみが表示されています。ストア操作ではストアアドレスとストアデータに個別のポートが必要ですが、図では 1 つとして描かれています)。

アウトオブオーダー・エンジンには多くの実行ユニットがあり、各ユニットは図 E-9 に示す 3 つの実行クラスターに配置されています。アウトオブオーダー・エンジンは、従来と同様に、1 サイクルあたり 4 マイクロオペレーション (uop) をリタイアできます。

E.4.2 フロントエンドの概要

図 E-10 に、このマイクロアーキテクチャーのフロントエンドの主な構成要素を示します。命令フェッチユニット (IFU) は、1 サイクルあたり最大 16 バイトのアライメントされた命令バイトを、命令キャッシュから命令長デコーダー (ILD) にフェッチします。命令キュー (IQ) は、ILD で処理された命令をバッファリングして、1 サイクルあたり最大 4 命令を命令デコーダーに供給できます。

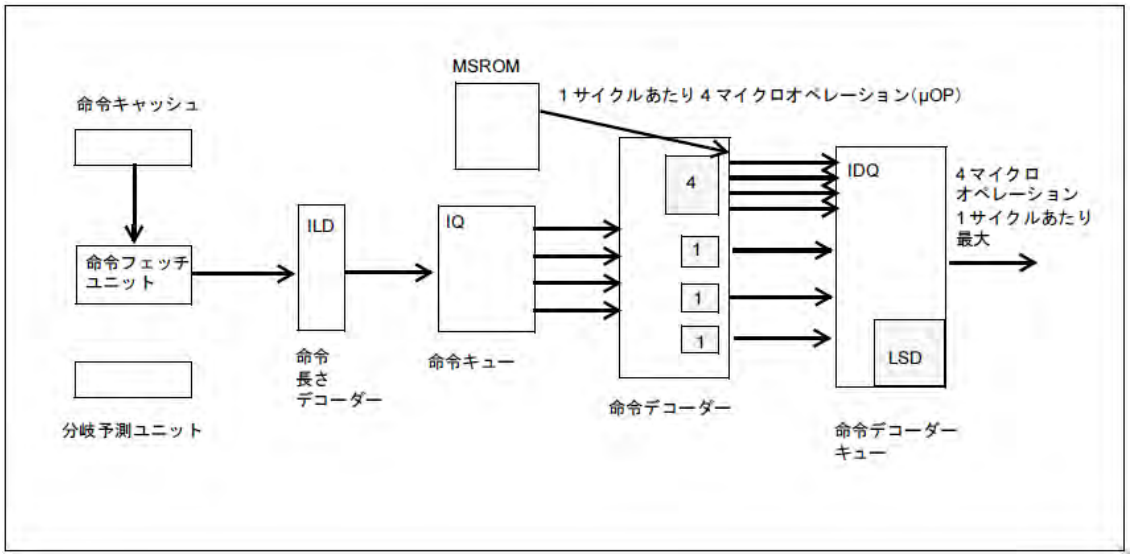


図 E-10 Nehalem† マイクロアーキテクチャーのフロントエンド

命令デコーダーは 、1 サイクルあたり 1 つの単純な命令をデコードできる 3 つのデコーダーユニットを備えています。もう 1 つのデコーダーユニットは、単純な命令でも、複数のマイクロオペレーション (uop) で構成された複雑な命令でも、1 サイクルあたり 1 命令をデコードできます。4 つを超えるマイクロオペレーション (uop) で構成された命令は、MSROM から供給されます。1 サイクルあたり最大 4 マイクロオペレーション (uop) を命令デコーダーキュー (IDQ) に供給できます。

IDQ 内のループストリーム検出器は、短い命令シーケンスからなるループの消費電力を削減してフロンドエンドの効率化を図ります。

命令デコーダーはマイクロフュージョンをサポートすることで、フロンドエンドのスループットを高め、スケジューラーおよびリオーダーバッファー (ROB) 内の有効キューサイズを拡大します。マイクロフュージョンの規則は、インテル® Core™ マイクロアーキテクチャーと同様です。

命令キューはマクロフュージョンもサポートしており、可能な限り、隣接する命令を単一のマイクロオペレーション (uop) に組み合わせます。前世代のインテル® Core™ マイクロアーキテクチャーでは、CMP/Jcc シーケンスでのマクロフュージョンのサポートは CF および ZF フラグに限定されており、64 ビット・モードではマクロフュージョンがサポートされていませんでした。

Nehalem† マイクロアーキテクチャーでは、64 ビット・モードでもマクロフュージョンがサポートされるようになり、以下の命令シーケンスがサポートされています。

- 以下を比較する場合に CMP または TEST をフュージョンできます (変更なし)。
  レジスター-レジスター。次に例を示します: CMP EAX,ECX; JZ label
  レジスター-即値。次に例を示します: CMP EAX,0x80; JZ label
  レジスター-メモリー。次に例を示します: CMP EAX,[ECX]; JZ label
  メモリー-レジスター。次に例を示します: CMP [EAX],ECX; JZ label

- TEST はすべての条件分岐とフュージョンされます (変更なし)。

- CMP は以下の条件分岐とフュージョンできます。これらの条件分岐では、キャリーフラグ (CF) またはゼロフラグ (ZF) をチェックします。以下は、マクロフュージョン可能な条件分岐のリストです (変更なし)。
  JA または JNBE
  JAE または JNB または JNC
  JE または JZ
  JNA または JBE
  JNAE または JC または JB
  JNE または JNZ

- Nehalem† マイクロアーキテクチャーでは、CMP は以下の条件分岐とフュージョンできます (拡張機能)。
  JL または JNGE
  JGE または JNL
  JLE または JNG
  JG または JNLE

ハードウェアは、いくつかの方法によって分岐処理を改善しています。分岐ターゲットバッファーが拡大し、分岐予測の精度が向上しました。リターン・スタック・バッファーでのリネームがサポートされ、コード中のリターン命令の予測ミスが減少しました。さらに、ハードウェアの強化によりリソースの再利用を促進することで、分岐予測ミスの処理が向上しました。これにより、予測ミスしたコードパスの実行にリソースが割り当てられていても、フロントエンドは構成済みのコードパス (リタイアメントに到達すると予測されるコードパス) 中の命令デコードを待機しなくて済みます。構成済みのコードパスの命令をフロントエンドがデコードすると、すぐに新しいマイクロオペレーション (uop) ストリームが順方向で処理を開始できます。

E.4.3 実行エンジン

IDQ (図 E-10) は、マイクロオペレーション (uop) ストリームをパイプラインの割り当て/リネームステージ (図 E-9) に供給します。アウトオブオーダー・エンジンは、インフライト (パイプライ中で同時に進行する) のマイクロオペレーション (uop) を最大 128 個サポートします。各マイクロオペレーション (uop) は、リオーダーバッファー (ROB) 内のエントリー、リザベーション・ステーション (RS) 内のエントリー、およびロード/ストアバッファー (メモリーアクセスが必要な場合) などのリソースとともに割り当てられる必要があります。

アロケーターは、インフライトの各マイクロオペレーション (uop) のレジスター・ファイル・エントリーをリネームします。マイクロオペレーション (uop) に関連する入力データは、通常 ROB またはリタイアしたレジスターファイルから読み出されます。

RS の深さが 36 エントリーに拡大されました (前世代では 32 エントリー)。マイクロオペレーション (uop) の実行準備が整っている場合、RS は 1 サイクルあたり最大 6 uop をディスパッチできます。RS は、発行ポートを通じてマイクロオペレーション (uop) を個々の実行クラスターにディスパッチします。各クラスターは、整数/FP/SIMD 実行ユニットの集合で構成されます。

マイクロオペレーション (uop) を実行した実行ユニットの結果は、レジスターファイルにライトバックされるか、結果を必要とするインフライトの uop にバイパス・ネットワーク経由で転送されます。Nehalem† マイクロアーキテクチャーは、各ポートで 1 サイクルごとに 1 つのレジスターファイルに書き込むライトバック・スループットに対応しています。バイパス・ネットワークは、整数/FP/SIMD の 3 ドメインで構成されます。同じバイパスドメイン内で結果を生産 (producer) するマイクロオペレーション (uop) から消費 (consumer) する uop にハードウェア上遅延なく効率良く転送できます。異なるバイパスドメイン間での結果の転送は、バイパス遅延が増加する可能性があります。バイパス遅延は、個々の実行ユニットのレイテンシーとスループットに加え、ソフトウェアにも表れる場合があります。表 E-25 に、異なるバイパスドメインにおける生産マイクロオペレーション (uop) と消費 uop の間のバイパス遅延を示します。

表 E-25 生産側と消費側の uop 間のバイパス遅延 (サイクル)

	FP	整数	SIMD
FP	0	2	2
整数	2	0	1
SIMD	2	1	0

E.4.3.1 発行ポートと実行ユニット

表 E-26 に、マイクロアーキテクチャーで一般的な操作に関する、発行ポートおよび実行ユニットのレイテンシー/スループットの特性を示します。

表 E-26 Nehalem† マイクロアーキテクチャーの発行ポート

ポート	実行可能な操作	レイテンシー	スループット	ドメイン	説明
ポート 0	整数 ALU	1	1	整数	
	整数シフト	1	1		
ポート 0	整数 SIMD ALU	1	1	SIMD	
	整数 SIMD シャッフル	1	1		
ポート 0	単精度（SP）FP MUL	4	1	FP	
	倍精度 FP MUL	5	1		
	FP MUL（X87）	5	1		
	FP/SIMD/SSE2 ムーブと論理演算	1	1		
	FP シャッフル	1	1		
	DIV/SQRT				
ポート 1	整数 ALU	1	1	整数	
	整数 LEA	1	1		
	整数 MUL	3	1		
ポート 1	整数 SIMD MUL	1	1	SIMD	
	整数 SIMD シフト	1	1		
	PSAD	3	1		
	文字列比較				
ポート 1	FP ADD	3	1	FP	
ポート 2	整数ロード	4	1	整数	

ポート	実行可能な操作	レイテンシー	スループット	ドメイン	説明
ポート 3	ストアアドレス	5	1	整数	
ポート 4	ストアデータ			整数	
ポート 5	整数 ALU 整数シフト ジャンプ	1 1 1	1 1 1	整数	
ポート 5	整数 SIMD ALU 整数 SIMD シャッフル	1 1	1 1	SIMD	
ポート 5	FP/SIMD/SSE2 ムーブと論理演算	1	1	FP	

E.4.4 キャッシュとメモリー・サブシステム

Nehalem† マイクロアーキテクチャーでは、各コアに命令キャッシュ、L1 データキャッシュ、ユニファイド L2 キャッシュが搭載されています (図 E-9 を参照)。それぞれの物理プロセッサーは、複数のプロセッサー・コアと、「アンコア」と呼ばれる共有サブシステムで構成されています。具体的には、インテル® Core™ i7 プロセッサーのアンコアは、物理プロセッサー内のすべてのコアで共有されるユニファイド L3 キャッシュ、インテル® QuickPath インターコネクト・リンク、および関連ロジックを含みます。L1 キャッシュと L2 キャッシュはライトバックで、かつインクルーシブではありません。

共有 L3 キャッシュはライトバックで、かつインクルーシブです。つまり、L1 データキャッシュ、L1 命令キャッシュ、ユニファイド L2 キャッシュのいずれかに存在するキャッシュラインは、L3 キャッシュにも存在します。L3 キャッシュはインクルーシブである特性を利用して、プロセッサー・コア間のスヌープ・トラフィックを最小限に抑えるように設計されています。表 E-27 に、キャッシュ階層の特性を示します。L3 キャッシュアクセスのレイテンシーは、プロセッサーとアンコア・サブシステムとの周波数比に応じて異なります。

表 E-27 インテル® Core™ i7 プロセッサーのキャッシュ・パラメーター

レベル	容量	アソシアティブ (ウェイ)	ライン サイズ (バイト)	アクセス・レイテンシー (クロック)	アクセス・スループット (クロック)	書き込みアップデート方式
L1 データ	32KB	8	64	4	1	ライトバック
命令	32KB	4	なし	なし	なし	なし
L2	256KB	8	64	10[1]	状況により異なる	ライトバック
L3 (共有)[2]	8MB	16	64	35-40+[2]	状況により異なる	ライトバック

注意:

1. ソフトウェアから見えるレイテンシーは、アクセスパターンなどの要因によって異なります。
2. コアとアンコアとの周波数比が 1:1 である場合、L3 キャッシュの最小レイテンシーは 35 サイクルです。

Nehalem† マイクロアーキテクチャーは、2 レベルのトランスレーション・ルックアサイド・バッファー (TLB) を実装しています。第 1 レベルは、データおよびコード向けの個別の TLB を構成します。DTLB0 はデータアクセスのアドレス変換を処理し、4KB ページをサポートする 64 個のエントリーと、ラージページをサポートする 32 個のエントリーを備えています。ITLB は、4KB ページ向けに 1 スレッドあたり 64 個のエントリーと、ラージページ向けに 1 スレッドあたり 7 個のエントリーを備えます。

第 2 レベルの TLB (STLB) は、4KB ページのコードアクセスとデータアクセスの両方を処理します。これは、DTLB0 や ITLB でミスをした 4KB ページの変換操作をサポートします。すべてのエントリーは、4 ウェイ・アソシアティブです。各 DTLB のエントリーのリストを以下に示します。

- 4KB ページの STLB:512 エントリー (データ・ルックアップと命令ルックアップの両方を処理)
- ラージページの DTLB0:32 エントリー
- 4KB ページの DTLB0:64 エントリー

DTLB0 でミスし STLB でヒットした場合、7 サイクルのペナルティーが発生します。一部のディスパッチで DTLB0 が使用される場合のみ、ソフトウェアはこのペナルティーを受けます。STLB および PMH へのミスに関連した遅延は通常大きくノンブロッキングです。

E.4.5 ロード操作とストア操作の強化

Nehalem† マイクロアーキテクチャーのメモリークラスターは、以下の拡張機能によってメモリー操作の高速化を図ります。

- 1 サイクルあたり 1 つの 128 ビット・ロード操作と 1 つの 128 ビット・ストア操作のピーク発行率
- ロード操作とストア操作向けの深いバッファー:48 個のロードバッファー、32 個のストアバッファー、10 個のフィルバッファー
- アライメントされていないメモリーの高速アクセスと、堅牢なメモリー・アライメント・ハザード処理
- ストア・フォワーディングの改善により、アライメントされたシナリオにもアライメントされないシナリオにも対応
- ほとんどのアドレス・アライメントに対応したストア・フォワーディング

E.4.5.1 効率的なアライメント・ハザードの処理

あらゆるワークロードにおいて、キャッシュ・サブシステムとメモリー・サブシステムが大部分の命令を処理しています。アドレス・アライメントのシナリオが異なると、メモリー操作やキャッシュ操作におけるパフォーマンスへの影響も変化します。例えば、L1 キャッシュからの、自然境界にアライメントされたロードには通常、L1 キャッシュでの 1 サイクルのスループット (表 E-28 を参照) が適用されます。しかし、アライメントしないロード命令 (MOVUPS、MOVUPD、MOVDQU など) によって L1 キャッシュのデータにアクセスすると、マイクロアーキテクチャーやアライメントのシナリオに応じて遅延は異なります。

表 E-28 L1 キャッシュからの MOVDQU におけるアドレス・アライメントがパフォーマンスに与える影響

スループット (サイクル)	インテル® Core™ i7 プロセッサー	45nm のインテル® Core™ マイクロアーキテクチャー	65nm のインテル® Core™ マイクロアーキテクチャー
アライメント・シナリオ	06_1AH	06_17H	06_0FH
16 バイト境界にアライメント	1	2	2
16 バイト境界にアライメントされず、キャッシュ分割なし	1	~2	~2
キャッシュライン境界が分割	~4.5	~20	~20

表 E-28 に、各種のアドレス・アライメント・シナリオで MOVDQU 命令を発行し L1 キャッシュからデータをロードした場合のおよそのスループットを示します。16 バイト・ロードがキャッシュライン境界をまたいだ場合、前世代のマイクロアーキテクチャーでは、ソフトウェアから見ても大きな遅延が発生しました。

Nehalem† マイクロアーキテクチャーでは、ハードウェアの強化により、キャッシュライン分割など各種アドレス・アライメント・シナリオを処理する際の遅延が短縮されています。

E.4.5.2 ストア・フォワーディングの強化

マイクロアーキテクチャーでは、ストアに続くロードがストアによってメモリーに書き込まれたデータを再読み込みする場合、ストア操作からロードにデータを直接転送できます。これはストア-ロード・フォワーディングと呼ばれ、ロードがメモリーを介さずにストア操作から直接データを取得できるので、サイクル数の節約になります。

遅延なくストア-ロード・フォワーディングを行うには、以下の規則に従わなければなりません。

- ストアは、ロードに先行する該当アドレスへの最後のストアでなければなりません
- ストアされるデータは、ロードされるデータとサイズが同じか、それよりも大きくなければなりません
- ロードデータは、先行するストア内に完全に含まれていなければなりません

ストア操作とロード操作間のアドレス・アライメントとデータサイズによって、ストア・フォワーディングでデータが転送されるか、キャッシュ/メモリー・サブシステムによる遅延が生じるかが決まります。45nm の拡張版インテル® Core™ マイクロアーキテクチャーでは、アドレス・アライメントとデータサイズの要件が従来のマイクロアーキテクチャーよりも緩和されました。Nehalem† マイクロアーキテクチャーではさらに機能が強化され、さらに多くの状況で迅速にデータ転送できます。

図 E-11 に、16 バイト・ストア操作おけるストア・フォワーディング・シナリオを示します。

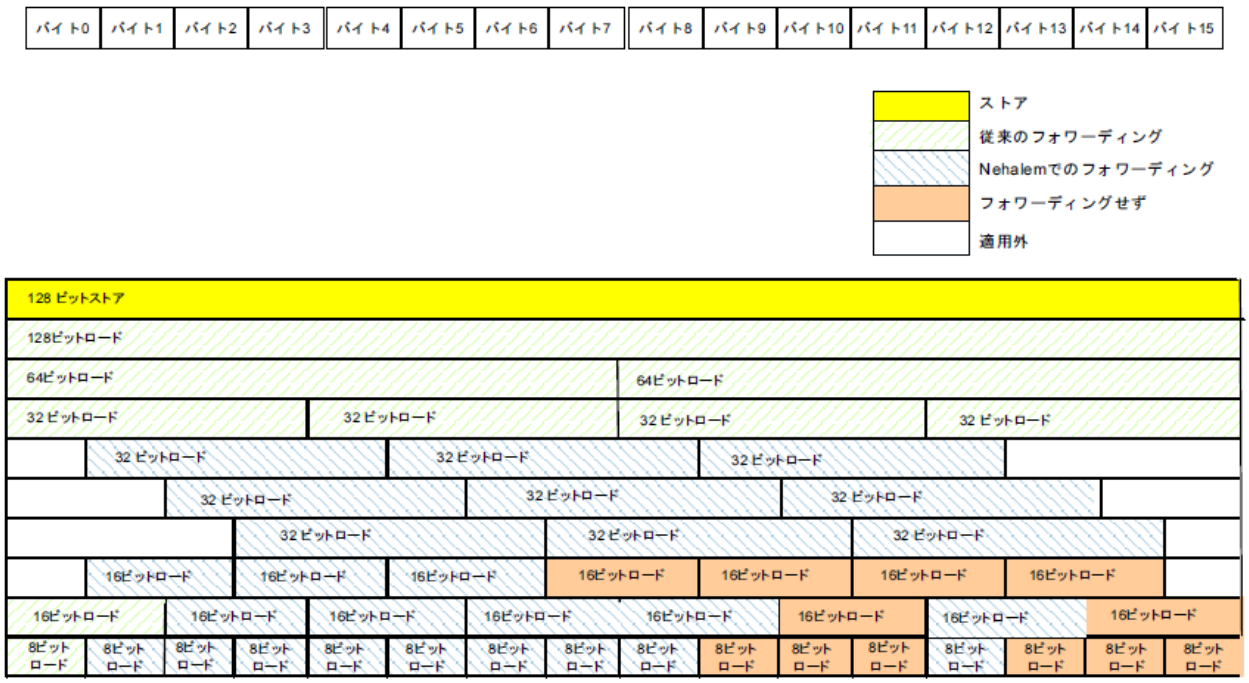


図 E-11 16 バイト・ストア操作のストア・フォワーディング・シナリオ

Nehalem† マイクロアーキテクチャーでは、ストアアドレスのアライメントにかかわらずストア-ロード・フォワーディングを実行できます (図中の空欄は該当ストア-ロードシナリオが適用されません)。図 E-12 に、8 バイト以下のストア操作のシナリオを示します。

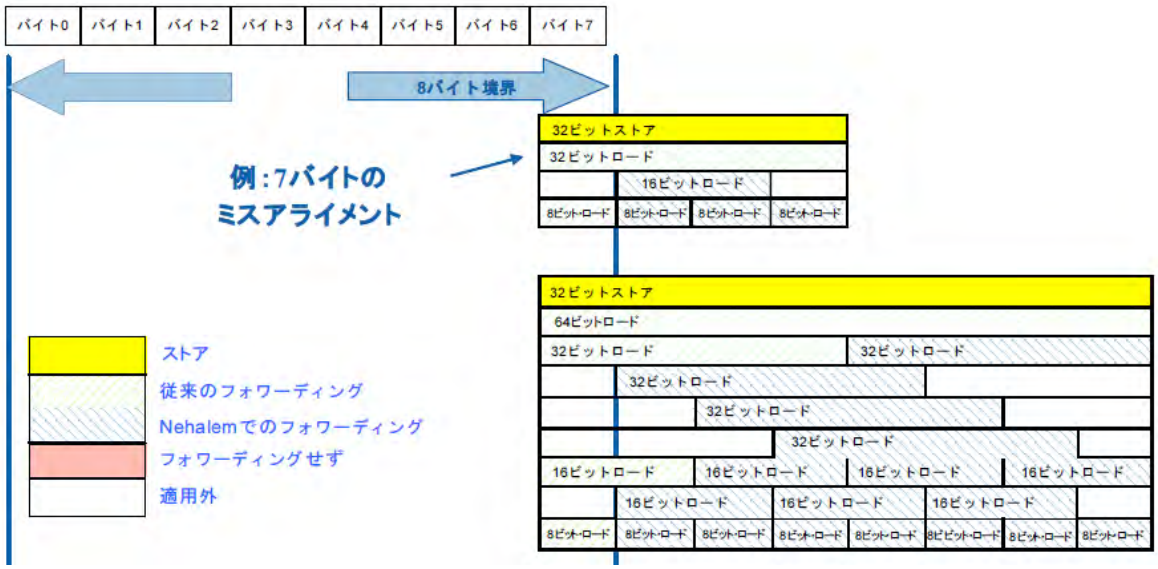


図 E-12 Nehalem† マイクロアーキテクチャーにおけるストア・フォワーディングの強化

E.4.6 REP (リピート) 文字列の強化

memcpy/memset などのライブラリー関数の実装では、REP プリフィクスを MOVS/STOS 命令および ECX のカウント値と組み合わせることが頻繁に行われます。これは、「REP 文字列 (文字列リピート)」命令と呼ばれます。この命令では、反復ごとに一定の値をバイト/ワード/ダブルワード/クワッドワード単位でコピー/書き込みできます。文字列 REP を使用する上でのパフォーマンスは、開始オーバーヘッドとデータ転送スループットという 2 つの要素によって決まります。

REP 文字列命令のパフォーマンスを決める 2 つの要素は、さらに単位、アライメント、カウント値によって異なります。MOVSB は通常、極めて小さなデータチャンクの処理に使用されます。したがって、REP MOVSB を実装したプロセッサーは、ECX < 4 の処理に最適化されており、ECX > 3 の REP MOVSB を使用すると、バイト単位のデータ転送に加えて開始オーバーヘッドの増加により、データ・スループットが低下します。ECX < 4 の場合、MOVSB のレイテンシーは 9 サイクルですが、ECX > 9 の REP MOVSB では、50 サイクルの開始コストが発生します。

さらに大きな単位でデータ転送が行われる REP 文字列の場合、ECX 値の増加に伴い、REP 文字列の開始オーバーヘッドは以下のようにステップ単位で増加します。

- 短い文字列 (ECX <= 12): REP MOVSW/MOVSD/MOVSQ のレイテンシーは約 20 サイクルです。
- 高速文字列 (ECX >= 76: REP MOVSB を除く): プロセッサーは、できるだけ多くのデータを 16 バイト単位で移動することによりハードウェアの最適化を行います。REP 文字列のレイテンシーは、16 バイト・データ転送のいずれかがキャッシュライン境界をまたいでいるかどうかによって異なります。
  - 分割なし: レイテンシーは約 40 サイクルの開始コストからなり、64 バイトのデータごとに 4 サイクルが追加されます。
  - キャッシュが分割: レイテンシーは約 35 サイクルの開始コストからなり、64 バイトのデータごとに 6 サイクルが追加されます。

- 中間の文字列長：REP MOVSW/MOVSD/MOVSQ のレイテンシーは約 15 サイクルの開始コストからなり、ワード/ダブルワード/クワッドワード単位でのデータ移動の反復ごとに 1 サイクルが追加されます。

Nehalem† マイクロアーキテクチャーでは、REP 文字列のパフォーマンスは以下の面で従来のマイクロアーキテクチャーに比べ大幅に向上しています。

- ほとんどのケースで開始オーバーヘッドが従来のマイクロアーキテクチャーに比べて削減されました
- データ転送スループットが前世代よりも向上しました

- 従来のマイクロアーキテクチャーの場合、REP 文字列命令を「高速文字列」モードで動作させるには、アドレス・アライメントが必要でしたが、Nehalem† マイクロアーキテクチャーでは、アドレスが 16 バイト境界にアライメントされていなくても、REP 文字列命令を「高速文字列」モードで動作できます

E.4.7 システム・ソフトウェアの強化

Nehalem† マイクロアーキテクチャーでは、アプリケーション・レベル・ソフトウェアとシステム・レベル・ソフトウェアの両方に利点があるマイクロアーキテクチャーの強化に加えて、主にシステム・ソフトウェアに適した強化も行われています。

Lock プリミティブ: Lock プリフィクス (XCHG、CMPXCHG8B など) を使用して同期プリミティブを実行すると、従来のマイクロアーキテクチャーよりもレイテンシーが大幅に減少します。

VMM のオーバーヘッド改善: 従来のマイクロアーキテクチャーでは、仮想マシン (VM) とそのスーパーバイザー (VMM) との間の VMX 遷移は、一度に数千サイクルを要することがあります。Nehalem† マイクロアーキテクチャー・ベース・プロセッサーでは、VMX 遷移のレイテンシーが軽減されています。

E.4.8 電力消費の効率化

Nehalem† マイクロアーキテクチャーは、さまざまな負荷状況のもとで高いパフォーマンスと電力効率に優れたパフォーマンスを発揮できるように設計されているだけでなく、システムアイドル時の消費電力を軽減する拡張機能を備えています。Nehalem† マイクロアーキテクチャーではプロセッサー固有の C6 ステートがサポートされます。これは、OS が ACPI と OS のパワー・マネジメント機構によって管理できる中でリーク消費電力が最も少ないステートです。

E.4.9 Nehalem† マイクロアーキテクチャーにおけるインテル® ハイパースレッディング・テクノロジーのサポート

Nehalem† マイクロアーキテクチャーは、ハイパースレッディング (HT) テクノロジーをサポートしています。このテクノロジーの実装では、2 つの論理プロセッサーが各コアの実行/キャッシュリソースの大半を共有します。Nehalem† マイクロアーキテクチャーの HT テクノロジーは、Intel NetBurst® マイクロアーキテクチャー・ベースの旧世代の HT テクノロジーと以下の点が異なります。

- Nehalem† マイクロアーキテクチャーでは、4 マイクロオペレーション (uop) 幅の実行エンジンと、演算操作を発行可能な 3 つの発行ポートに組み合わされた実行ユニットが搭載されています。
- Nehalem† マイクロアーキテクチャーでサポートされている統合メモリー・コントローラーにより、インテル® Core™ i7 プロセッサーでは最大 25.6GB/秒のピークメモリー帯域幅を提供します。
- 以下のように、バッファリングが深くなり、リソース共有/分割ポリシーが強化されました。
  - HT テクノロジー操作向けに複製されるリソース: レジスター状態、リネームされたリターン・スタック・バッファー、ラージページ ITLB
  - HT テクノロジー操作向けに分割されるリソース: ロードバッファー、ストアバッファー、リオーダーバッファー、スモールページ ITLB は、2 つの論理プロセッサー間で静的に割り当てられます
  - HT テクノロジー操作中に共有されるリソース: リザベーション・ステーション、キャッシュ階層、フィルバッファー、DTLB0 と STLB の両方
  - HT テクノロジー操作中に交互に実行: フロントエンド操作は通常、公平に 2 つの論理プロセッサー間で交互に実行されます
  - HT テクノロジーで管理されないリソース:実行ユニット

付録 F
旧世代の Intel Atom® マイクロアーキテクチャーとソフトウェアの最適化

F.1 概要

45nm プロセスの Intel Atom® プロセッサーは、Intel Atom® マイクロアーキテクチャーをベースにしています。同じマイクロアーキテクチャーは、32nm プロセスの Intel Atom® プロセッサーでも採用されています。この章では、Intel Atom® マイクロアーキテクチャーの概要を説明し、このマイクロアーキテクチャー・ベースのプロセッサーをターゲットとするソフトウェア向けの固有のコーディング手法を示します。Intel Atom® プロセッサーの主な機能は、低消費電力と以下を含む効率良いパフォーマンスをサポートすることです。

- 拡張版 Intel SpeedStep® テクノロジー：ワークロード実行中にプロセッサーが周波数/電圧レベルを低い状態に遷移できるようにオペレーティング・システム (OS) をプログラムできます。
- プロセッサー内のキャッシュと他のサブシステムへの電力を切断することによって、スタティック消費電力を削減するディープ・パワー・ダウンをサポートします。
- インテル® ハイパースレッディング・テクノロジーは、マルチタスクとマルチスレッド・ワークロード向けに 2 つの論理プロセッサーを提供します。
- 1 つの命令で複数のデータ用を処理する拡張 (インテル® SSE3 とインテル® SSSE3) をサポートします。
- インテル® 64 および IA-32 アーキテクチャーをサポートします。

Intel Atom® マイクロアーキテクチャーは、小型フォームファクターや熱的に制約された環境における消費電力の許容範囲内で、最新のワークロードのパフォーマンス要件を満たすように設計されています。

F.2 Intel Atom® マイクロアーキテクチャー

Intel Atom® マイクロアーキテクチャーは、2 マイクロオペレーション (uop) の発行が可能なハイパースレッディング・テクノロジー対応のインオーダー・パイプラインによって、効率良いパフォーマンスと低消費電力の動作を実現しています。インオーダー・パイプラインは、メモリーオペランドを持つ IA-32 命令を複数のマイクロオペレーション (uop) ではなく単一のパイプライン操作として扱う点が、アウトオブオーダー・パイプラインとは異なります。

図 F-1 に、Intel Atom® マイクロアーキテクチャーのパイプラインの基本ブロック図を示します。

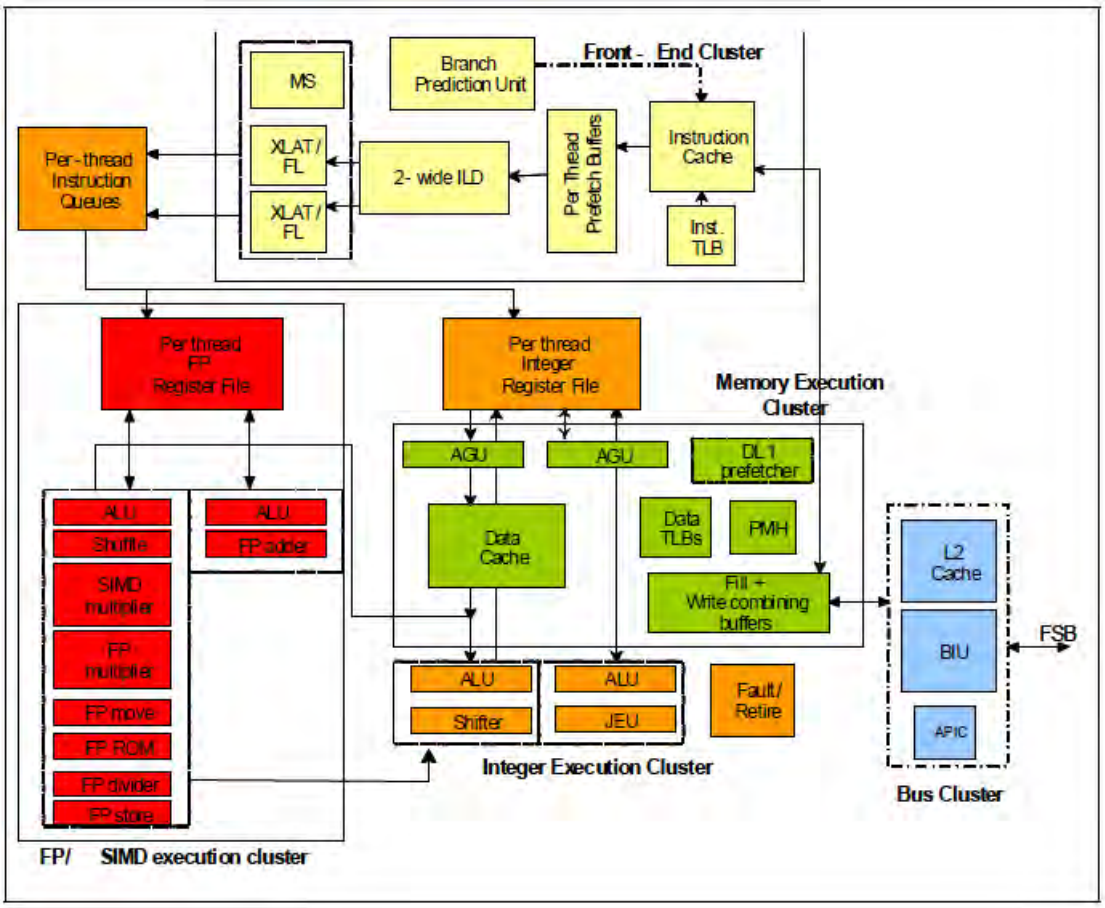


図 F-1 Intel Atom® マイクロアーキテクチャーのパイプライン

フロントエンドは、以下で構成される省電力パイプラインを備えています。

- 32KB の 8 ウェイ・セット・アソシアティブ L1 命令キャッシュ
- 分岐予測ユニットと ITLB
- 2 つの命令デコーダー (それぞれ 1 サイクルあたり最大 1 命令をデコード可能)

フロントエンドでは、1 サイクルあたり最大 2 命令を命令キューに格納して、スケジューリングすることができます。スケジューラーは、2 つの発行ポートを通じて 1 サイクルあたり最大 2 命令を整数実行クラスターまたは SIMD/FP 実行クラスターに発行できます。

2 つの発行ポートはそれぞれ、1 サイクルあたり 1 命令を、実行する整数クラスターまたは SIMD/FP クラスターにディスパッチできます。整数クラスターと SIMD/FP クラスターのポート・バインディングは、以下の機能を備えています。

- 整数実行クラスター:
  - ポート 0: ALU0、シフト/ローテートユニット、ロード/ストア
  - ポート 1: ALU1、ビット処理ユニット、分岐ユニットと LEA
  - 0 サイクルの実効「ロード後利用」レイテンシー

- SIMD/FP 実行クラスター:
  - ポート 0: SIMD ALU、シャッフルユニット、SIMD/FP 乗算ユニット、除算ユニット (IMUL、IDIV をサポート)
  - ポート 1: SIMD ALU、FP 加算器
  - SIMD/FP クラスター内の 2 つの SIMD ALU とシャッフルユニットは 128 ビット幅ですが、64 ビット SIMD 整数演算はポート 0 のみで実行されます
  - FP 加算器は、128 ビット・データパスで ADDPS/SUBPS を実行できます。ほかの FP 加算のデータパスは 64 ビット幅です

  - FP/SIMD 実行の安全な命令認識アルゴリズムにより、例外を生じる可能性がある古い FP/SIMD 命令によってブロックされることなく、レイテンシーの短い新しい整数命令を実行できます
  - FP 乗算パイプはメモリーロードもサポートしています
  - メモリーロードを伴う FP ADD 命令は、両方のポートをディスパッチに使用できます

インテル® 64 アーキテクチャーでは、メモリー実行サブシステム (MEU) は、32 ビットまたは 36 ビットの物理アドレスモードで 48 ビットのリニアアドレスをサポートします。MEU は以下を備えています。

- 24KB 第 1 レベルキャッシュ
- L1 データキャッシュへのハードウェア・プリフェッチ
- 4KB およびそれよりも大きなページ構造に対応した 2 レベルの DTLB
- DTLB と ITLB のミスを処理するハードウェア・ページウォーカー
- 2 つのアドレス生成ユニット (ポート 0 はロードとストアをサポート。ポート 1 は LEA とスタック操作をサポート)
- 整数演算でのストア・フォワーディングのサポート
- 8 つのライト・コンバイン・バッファー

バス論理サブシステムは以下の機能を備えています。

- 512KB の 8 ウェイ・セット・アソシアティブ・ユニファイド L2 キャッシュ
- L2 へのハードウェア・プリフェッチとフロントサイド・バスへのインターフェイス・ロジック

F.2.1 Intel Atom® マイクロアーキテクチャーにおけるハイパースレッディング・テクノロジーのサポート

命令キューは、2 つのスレッドからの命令実行をスケジューリングできるように静的に分割されています。スケジューラーは、いずれかのスレッドから命令を 1 つ選択し、実行に向けてポート 0 またはポート 1 にディスパッチすることができます。ハードウェアは、公平な基準と、各スレッドでの順方向進行の対応状況に基づいて、2 つのスレッド間で命令のフェッチ/デコード/ディスパッチを選択します。

F.3 Intel Atom® マイクロアーキテクチャー向けのコーディングの推奨事項

アウトオブオーダー・マイクロアーキテクチャーに適用される命令スケジューリングとコーディング手法が、インオーダー・マイクロアーキテクチャーで最適なパフォーマンスを実現できるとは限りません。同様に、Intel Atom® マイクロアーキテクチャーのようなインオーダー・パイプライン向けの命令スケジューリングとコーディング手法が、アウトオブオーダー・マイクロアーキテクチャーで最適なパフォーマンスを実現できるとは限りません。ここでは、主な導入対象が Intel Atom® マイクロアーキテクチャー・ベースのプロセッサーであるソフトウェアに固有のコーディング推奨事項について説明します。

F.3.1 Intel Atom® マイクロアーキテクチャーのフロントエンドの最適化

Intel Atom® マイクロアーキテクチャーのフロントエンドにある 2 つのデコーダーは、インテル® 64 アーキテクチャーと IA-32 アーキテクチャーの命令の大半を処理できます。複雑な演算を処理する一部の命令では、フロントエンドの MSROM を使用する必要があります。2 つのデコーダーを通過する命令はほとんどの場合、フロントエンドのデコーダーユニットのいずれかによってデコードされます。MSROM が必要な命令や、フロントエンドでデコーダー割り当ての再調整が行われる場合、フロントエンドで遅延が発生します。

ソフトウェアは、固有のパフォーマンス監視イベントを使用して、フロントエンドでデコーダー割り当ての再調整が行われる命令シーケンスや条件を検出できます。

アセンブリー/コンパイラー・コーディング規則 1 (影響 MH、一般性 ML): Intel Atom® プロセッサーでは、MSROM を必要とする複雑な命令を最小限に抑えることで、2 つのデコードユニットの最適なデコード帯域幅を活用できます。

パフォーマンス監視イベントの「MACRO_INSTS.NON_CISC_DECODED」と「MACRO_INSTS.CISC_DECODED」を使用して、ワークロード中で MSROM を必要とした命令の割合を評価できます。

アセンブリー/コンパイラー・コーディング規則 2 (影響 M、一般性 H): Intel Atom®プロセッサーでは、命令ワーキングセットの要素を小さくすると、2 つのデコードユニットによって提供される最適なデコード帯域幅をフロントエンドで活用できます。

アセンブリー/コンパイラー・コーディング規則 3 (影響 MH、一般性 ML): Intel Atom®プロセッサーでは、連続する x87 命令の使用を避けると、2 つのデコードユニットによって提供される最適なデコード帯域幅をフロントエンドで活用できます。

パフォーマンス監視イベント「DECODE_RESTRICTION」を使用すると、ワークロード中でデコード・スループットの低下をもたらす遅延が発生した回数をカウントできます。

一般に、リタイアした「命令あたりのサイクル数」が 1 未満になるまで、フロントエンドの制約はパフォーマンスの制限要素にはなりません (理論上の最大リタイアメント・スループットは CPI 値 0.5 に相当)。CPI を 1 未満にするには、フロントエンドを通過する際に命令ペアが 2 つのデコーダーによって並列にデコードされる命令シーケンスを作成することが重要です。フロントエンドの通過後は、スケジューラーや実行ハードウェアがデコードペアを同じ順序でポート 0 とポート 1 にディスパッチする必要はありません。

デコーダーはジャンプ命令より先をデコードできないため、ジャンプはデコーダーに最適化されたペア内の 2 番目の命令として組み合わせる必要があります。フロントエンドでは x87 命令を 1 サイクルあたり 1 個のみ処理可能であり、デコーダーユニット 0 だけが MSROM への転送を要求できます。命令が 8 バイトを超えていたり、命令のプリフィクスが 3 つを超えていると、MSROM の転送が行われ、フロントエンドで 2 サイクルの遅延が発生します。

命令の長さとアライメントは、デコードのスループットに影響を与えることがあります。フロントエンドがバッファーをプリフェッチする場合、7 サイクル内に 48 バイトを超えるデコードを行うとバッファー待機によってフロントエンドで遅延が生じます。これはスループットを制限します。また、命令ペアが 16 バイト境界をまたぐたびに、フロントエンド・バッファーを少なくとも 1 サイクル余分に保持する必要が生じます。したがって、16 バイト境界をまたぐ命令アライメントは、大きな問題となります。無視できるプリフィクスと命令を組み合わせて使用することで命令アライメントを改善できます。

例 F-1 Intel Atom® マイクロアーキテクチャー上でデコードを最適化するための命令ペアとアライメント

アドレス	命令バイト	逆アセンブル
7FFFFDF0	0F594301	mulps xmm0, [ebx+ 01h]
7FFFFDF4	8341FFFF	add dword ptr [ecx-01h], -1
7FFFFDF8	83C2FF	add edx, , -1
7FFFFDFB	64	; FS プリフィクスのオーバーライドは無視され、コードのアライメントを改善
7FFFFDFC	F20f58E4	add xmm4, xmm4
7FFFFE00	0F594B11	mulps xmm1, [ebx+ 11h]
7FFFFE04	8369EFFF	sub dword ptr [ecx- 11h], -1
7FFFFE08	83EAFF	sub edx, -1
7FFFFE0B	64	; FS プリフィクスのオーバーライドは無視され、コードのアライメントを改善
7FFFFE0C	F20F58ED	addsd xmm5, xmm5
7FFFFE10	0F595301	mulps xmm2, [ebx +1]
7FFFFE14	8341DFFF	add dword ptr [ecx-21H], -1
7FFFFE18	83C2FF	add edx, -1
7FFFFE1B	64	; FS プリフィクスのオーバーライドは無視され、コードのアライメントを改善
7FFFFE1C	F20F58F6	addssd xmm6, xmm6
7FFFFE20	0F595B11	mulps xmm3, [ebx+ 11h]
7FFFFE24	8369CFFF	sub dword ptr [ecx- 31h], -1
7FFFFE28	83EAFF	sub edx, -1

小さなループ中にレイテンシーの長い操作が含まれている場合、ループアンロールは、レイテンシーの長い命令とペアにできる隣接した命令を見つけて、その隣接した命令の順方向進行を可能にするための手法です。ただし、コードサイズの増加や分岐ターゲットバッファーへの負荷など、ループアンロールの影響も評価しなければなりません。

パフォーマンス監視イベント「BACLEARS」は、ループアンロールがフロントエンドのパフォーマンスにとってプラスであるかマイナスであるかを評価する手段となります。「ICACHE_MISSES」イベントも、ループアンロールによって命令要素が増加するかどうかを評価できます。

Intel Atom® プロセッサーの分岐予測機構は、異なる分岐タイプを区別しません。異なる分岐タイプが混在すると、分岐予測ハードウェアで混乱が生じることがあります。

パフォーマンス監視イベント「BR_MISSP_TYPE_RETIRED」は、分岐タイプに起因する分岐予測上の問題を評価する手段となります。

F.3.2 実行コアの最適化

ここでは、2 マイクロオペレーション (uop) の発行が可能な実行コアを利用して、ソフトウェアが 2 つの命令による順方向進行をより頻繁に行うためのいくつかの事項について説明します。

F.3.2.1 整数命令の選択

インオーダー・マシンでは、命令の選択と組み合わせが、データの実行準備が整った命令の命令レベルの並列性を検出するマシンの能力に影響を与えることがあります。以下にその例をいくつか示します。

- **EFLAG**: EFLAG フラグビットを参照する命令は、EFLAG レジスターを更新する命令と同じサイクル内では発行できません。例えば、ADD はキャリービットを変更する可能性がある、更新命令です。JC (または ADC) はキャリービットを読み出すため参照命令となります。
  - 条件ジャンプは、参照命令の後に続くサイクルで発行できます。
  - その他の EFLAG ビットの参照命令を更新命令の後に発行する場合、1 サイクル待機しなければなりません (2 サイクル遅延)。

アセンブリー/コンパイラー・コーディング規則 4(影響 M、一般性 H): Intel Atom® プロセッサーでは、2 サイクルの遅延が生じるフラグ更新命令とフラグ参照命令の間に MOV 命令を配置します。これにより、部分的なフラグ依存関係を防止できます。

- **長いレイテンシーの整数命令**: これらの命令は、同じスレッド内のレイテンシーの短い命令の発行をブロックします (プログラムの順序により必要)。また、ライトバック・リソースを解決する際は、両方のスレッド内のレイテンシーの短い命令も 1 サイクルの間ブロックします。
- **共通のデスティネーション**: 結果を同じデスティネーションに格納する 2 つの命令は、同じサイクル内では発行できません。
- **コストがかかる命令**: 一部の命令には特別な要件があり、実行中長期間にわたりハードウェア・リソースを消費するため多くのコストがかかります。命令キュー内でその命令が最も古くなるまで、実行ユニットで遅延することがあります。また、ほかの新しい命令の発行を遅らせることもあります。このような例としては、FDIV や両方のポートから実行ユニットを必要とする命令などが挙げられます。

F.3.2.2 アドレス生成

ハードウェアは、実行準備が整った命令が利用するデータは、準備されている必要がある、という一般要件の最適化を行います。また、アドレス生成は、データの準備よりも先に行われます。ほかの命令が生成するデータを必要とする依存関係がアドレス生成で生じた場合、3 サイクルの遅延が発生します。

2 マイクロオペレーション (uop) の発行が可能なマシンの実行スループットに影響を与える 3 つの状況では、アドレス生成ユニット (AGU) を直接使用できます。この状況を以下に示します。

- **暗黙的な ESP 更新**: ESP レジスターが命令のデスティネーション (明示的な ESP 更新)として使用されない場合、PUSH、POP、CALL、RETURN などの命令では暗黙的の ESP 更新が行われます。明示的な ESP 更新と暗黙的な ESP 更新を混在させても、アドレス生成とデータ実行との依存関係が発生します。
- **LEA**: LEA 命令は、ALU ではなく AGU を使用します。LEA のソースレジスターの 1 つが実行ユニットから提供される場合、依存関係により 3 サイクルの遅延が発生します。したがって、2 つの値を加算して結果を第 3 のレジスターに格納する手法では、LEA を使用してはなりません。LEA は、アドレス計算に使用すべきです。
- **整数から FP/SIMD への転送**: 整数データをマシンの FP/SIMD 側に転送する命令でも、AGU が使用されます。このような命令の例として、MOVD や PINSRW が挙げられます。これらの命令のソースレジスターの 1 つが実行ユニットの結果に依存している場合、この依存関係でも 3 サイクルの遅延が発生します。

例 F-2 AGU と実行ユニットとの依存関係を回避する代替手法

```
a) 三項演算で LEA を使用した場合の 3 サイクルの遅延
     mov eax, 0x01
     lea eax, 0x8000[eax+ebp]; eax の値は直線の命令で生成される
     ; lea と実行の依存性により、3 サイクルの遅延
b) 実行中の依存関係を処理し、AGU と実行との依存関係を回避する
     mov eax, 0x01
     add eax, 0x8000
     add eax, ebp
```

アセンブリー/コンパイラー・コーディング規則 5 (影響 MH、一般性 H): Intel Atom® プロセッサーでは、アドレス操作に LEA を使用すべきですが、ソフトウェアは、ALU から AGU に対する依存関係 (アドレス操作または ESP 更新に LEA の代わりに ALU 命令を使用する場合、または三項加算や、アドレス生成への供給を行わない非破壊的書き込みに LEA を使用する場合) を回避しなければなりません。または、AGU を参照する命令の前に更新命令を配置する際、3 サイクル以上の間隔を空けます。

F.3.2.3 整数乗算

整数乗算命令の実行には、数サイクルかかります。整数乗算命令はレイテンシーの長い別の命令とともに、実行フェーズで順方向に進行できるようにパイプライン化されます。ただし整数乗算命令は、プログラム順序の要件により、単一サイクルのほかの整数命令の発行をブロックします。

アセンブリー/コンパイラー・コーディング規則 6 (影響 M、一般性 M): Intel Atom® プロセッサーでは、整数乗算命令の後に独立した FP 乗算または整数乗算を配置すると、IMUL をパイプライン実行できます。

例 F-3 整数計算における命令のパイプライン実行

```
a) 複数サイクル IMUL 命令が 1 サイクル整数命令をブロック
     imul eax, eax
     add ecx, ecx ; 1 サイクル整数命令は IMUL で 4 サイクルブロックされる
     imul ebx, ebx ; インオーダー発行による命令ブロック
独立して連続し発行される IMUL はパイプライン化される
     imul eax, eax
     imul ebx, ebx ; 2 番目の IMUL は 1 サイクル後に発行できる
     add ecx, ecx ; 1 サイクル整数命令は IMUL でブロックされる
```

F.3.2.4 整数シフト命令

シフトカウントを即値バイトでエンコードする整数シフト命令では、1 サイクルのレイテンシーが発生します。一方、ECX レジスターでシフトカウントを使用するシフト命令は、レジスターカウントが更新されるのを待機しなければならないことがあります。そのため、レジスターカウントを使用するシフト命令では、3 サイクルのレイテンシーが発生します。

アセンブリー/コンパイラー・コーディング規則 7 (影響 M、一般性 M): Intel Atom® プロセッサーでは、整数シフト命令の暗黙的なレジスターカウントの更新命令は、シフト命令の 2 サイクル以上前に配置します。

F.3.2.5 パーシャル・レジスター・アクセス

パーシャル・レジスター・アクセスによって追加の遅延は生じませんが、インオーダー・ハードウェアはレジスター全体への依存関係を追跡します。したがって、AL や AH などの 8 ビット・レジスターは、独立したレジスターとして扱われません。

また、LEA、単純なロード、POP など一部の命令では、入力値が 4 バイト未満であると速度が低下します。

アセンブリー/コンパイラー・コーディング規則 8 (影響 M、一般性 MH): Intel Atom® プロセッサーでは、LEA、単純なロード、POP の入力値が 4 バイト未満であると速度が低下します。

F.3.2.6 FP/SIMD 命令の選択

表 F-1 に、ソフトウェアで最も頻繁に使用される Intel Atom® マイクロアーキテクチャーの各種実行ユニットの特性を示します。

表 F-1 Intel Atom® マイクロアーキテクチャーの命令レイテンシー/スループットのまとめ

命令カテゴリー	レイテンシー (サイクル)	スループット	実行ユニット数
SIMD 整数 ALU			
128 ビット ALU/論理/移動	1	1	2
64 ビット ALU/論理/移動	1	1	2
SIMD 整数 シフト			
128 ビット	1	1	1
64 ビット	1	1	1
SIMD シャッフル			
128 ビット	1	1	1
64 ビット	1	1	1
SIMD 整数乗算			
128 ビット	5	2	1
64 ビット	4	1	1
FP 加算			
X87 op (FADD)	5	1	1
スカラー SIMD (addsd, addss)	5	1	1
パックド単精度 (addps)	5	1	1
パックド倍精度 (addpd)	6		1
FP 乗算			
X87 Ops (FMUL)	5	5	1
スカラー単精度 (mulss)	4		1
スカラー倍精度 (mulsd)	5	2	1
パックド単精度 (mulps)	5	1	1
パックド倍精度 (mulpd)	9	2	1
IMUL		2	
IMUL r32, r/m32	5	9	1
IMUL r12, r/m16	6		1
		1	
		1	

SIMD/FP 命令の選択では通常、最初にレイテンシーの短いものを選び、その次には可能な限りスループットの高いものを選ぶ必要があります。パックド倍精度命令はパイプライン化されない点に注意してください。代わりに 2 つのスカラー倍精度命令を使用すると、実行クラスターのパフォーマンスを高めることができます。

アセンブリー/コンパイラー・コーディング規則 9 (影響 MH、一般性 H): Intel Atom® プロセッサーでは、FP スタックを使用する x87 命令よりも、XMM レジスターで動作する SIMD 命令を優先します。そして、可能な限り、パックド単精度命令を使用します。パックド倍精度命令はスカラー倍精度命令に置き換えます。

アセンブリー/コンパイラー・コーディング規則 10 (影響 M、一般性 ML): Intel Atom® プロセッサーでは、超越関数のような高度な演算を実行するライブラリー・ソフトウェアは、ネイティブの x87 命令ではなく、XMM レジスターで動作する SIMD 命令を使用すべきです。

アセンブリー/コンパイラー・コーディング規則 11(影響 M、一般性 M): Intel Atom® プロセッサーでは、可能な限り DAZ と FTZ を有効にします。

いくつかのパフォーマンス監視イベントは、SIMD/FP 命令選択のチューニングに有効です。例えば「SIMD_INST_RETIRED.{PACKED_SINGLE, SCALAR_SINGLE, PACKED_DOUBLE, SCALAR_DOUBLE}」は、プログラムでの命令選択の判断に使用できます。「FP_ASSIST」や「SIR」を使用すると、浮動小数点例外 (または不正なアラーム) がプログラムのパフォーマンスに影響を与えているかどうか確認できます。

除算命令のレイテンシーとスループットは、入力値やデータサイズによって異なります。Intel Atom® マイクロアーキテクチャーでは、基数 2 の除算器ユニットを実装しています。したがって、divide/sqrt のレイテンシーは、ほかの

FP 演算よりも大幅に長くなります。また、divide/sqrt の発行スループット速度も同様に低下します。除算器ユニットは 2 つの論理プロセッサー間で共有されるため、ソフトウェアは除算命令の代替手段を検討すべきです。

アセンブリー/コンパイラー・コーディング規則 12 (影響 H、一般性 L): Intel Atom® プロセッサーでは、どうしても必要なときのみ除算命令を使用し、データサイズが最小のオペランドを使用するように考慮します。

パフォーマンス監視イベント「DIV」と「CYCLES_DIV_BUSY」を使用すると、除算がプログラム中のボトルネックになっているかどうかを確認できます。

一般に、FP 演算は整数命令よりもレイテンシーが長くなります。FP 演算からの結果のライトバックは通常、整数パイプラインよりも後のパイプライン・ステージで発生します。したがって、命令が FP 演算の結果に依存関係を持っている場合、2 サイクルの遅延が発生します。この種の命令の例として、FP から整数への変換である CVTxx2xx や、XMM から汎用レジスターへの MOVD があります。

連続する 4 つの単精度データ要素を用いた演算を必要とする場合、MOVUPS よりも PALIGNR + MOVAPS を優先します。MOVUPS xmm1, _pArray[k] (メモリーアドレス _pArray は 16 バイト境界にアライメント) のように、制約なしに配列インデックス k によって 4 つのデータ要素をロードすると、定期的にキャッシュライン分割が発生し 14 サイクルの遅延が生じます。

最適なアプローチでは、4 の倍数でない k ごとに、j = 4*(k/4) で k を 4 の倍数に丸め、MOVAPS と MOVAPS xmm1, _pArray[j] と MOVAPS xmm1, _pArray[j+4] を実行してから、PALIGNR を使用して演算に必要な 4 つのデータ要素を結合します。

アセンブリー/コンパイラー・コーディング規則 13 (影響 MH、一般性 M): Intel Atom® プロセッサーでは、MOVUPS よりも MOVAPS + PALIGN のシーケンスを優先します。同様に、MOVDQU よりも MOVDQA + PALIGNR を優先します。

F.3.3 メモリーアクセスの最適化

ここでは、ソフトウェアによってメモリー・サブシステムのパフォーマンスを最適化できるいくつかの事項について説明します。

特定のハザードに遭遇するキャッシュアクセスからのシステムメモリーに対するメモリーアクセスを行うと、そのメモリーアクセスは、データの実行準備が整っているときでもレイテンシーの短い命令の発行をブロックするため、コストのかかる操作になります。

パフォーマンス監視イベント「REISSUE」を使用して、プログラムにおける再発行されたメモリー命令の影響を評価できます。

F.3.3.1 ストア・フォワーディング

一部の限られた状況では、Intel Atom® マイクロアーキテクチャーは、先行するストア操作から後続のロード命令にデータを転送できます。この状況を以下に示します。

- ストア・フォワーディングは、整数パイプラインでのみサポートされており、FP データや SIMD データには適用されません。また、以下の条件を満たす必要があります。

  a. ストア操作とロード操作は、同じサイズであり、かつ同じアドレスに対するものでなければなりません。
  b. データサイズが 8 バイトを超えると、ストア操作からの転送は行われません。

- データ転送が行われる際、データはアドレスの最下位 12 ビットに基づいて転送されます。そのためソフトウェアは、あるアドレスにストアした後に、その最下位 12 ビットをエイリアシングする別のアドレスからロードするような、アドレス・エイリアシングの状況を回避しなければなりません。

F.3.3.2 第 1 レベルキャッシュ

Intel Atom® マイクロアーキテクチャーは、16 個の 4 バイト・チャンクで構成される L1 データキャッシュの各 64 バイト・キャッシュラインを処理します。この実装特性は、データ・アライメントと一部のデータ・アクセス・パターンにパフォーマンス上の影響を与えます。

アセンブリー/コンパイラー・コーディング規則 14 (影響 MH、一般性 H) : Intel Atom® プロセッサーでは、データがメモリー内で自然サイズにアライメントされるように配置します。例えば、4 バイトのデータは、4 バイト境界にアライメントする必要があります。また、チャンク内の小規模なアクセス (4 バイト未満) では、異なるバイトにアクセスすると、遅延が生じることがあります。

F.3.3.3 セグメントベース

Intel Atom® マイクロアーキテクチャーでは、アドレス生成ユニットは、セグメントベースがデフォルトで 0 になると想定されます。そのためセグメントベースが 0 以外であると、ロード操作やストア操作で遅延が発生します。

- セグメントベースがキャッシュライン境界にアライメントされていない場合、メモリー操作の最大スループットは 9 サイクルあたり 1 操作に減少します。

セグメントベースが 0 以外でも、キャッシュラインにアライメントされている場合、ペナルティーはセグメントベースによって異なります。

- DS では、最大スループットが 2 サイクルあたり 1 操作になります。
- FS と GS では、最大スループットが 2 サイクルあたり 1 操作になります。ただし、FS と GS は 0 以外のベースでのみ使用されることが想定されているため、セグメントベースが 0 であっても、最大スループットは 2 サイクルあたり 1 操作のままです。
- ES:
  - 文字列操作のデスティネーションが暗黙的なセグメントベースとして使用される場合、0 以外であるがキャッシュラインにアライメントされたベースについては、最大スループットが 2 サイクルあたり 1 操作になります。
  - その他の場合は、9 サイクルあたり 1 操作しか実行されません。

- CS と SS では、セグメントベースが 0 以外であるがキャッシュラインにアライメントされている場合は常に、最大スループットが 9 サイクルあたり 1 操作となります。

アセンブリー/コンパイラー・コーディング規則 15 (影響 H、一般性 ML): Intel Atom® プロセッサーでは、可能な限りベースが 0 にセットされたセグメントを使用し、キャッシュライン境界にアライメントされていない 0 以外のセグメント・ベース・アドレスはできるだけ回避します。

アセンブリー/コンパイラー・コーディング規則 16 (影響 H、一般性 L): Intel Atom® プロセッサーでは、0 以外のセグメントベースを使うときは、DS、FS、GS を使用します。さらに文字列操作では、暗黙的な ES を使用すべきです。

アセンブリー/コンパイラー・コーディング規則 17 (影響 M、一般性 ML): Intel Atom® プロセッサーでは、セグメントベースが 0 の FS、GS よりも ES、DS、SS の使用を優先します。

F.3.3.4 文字列移動

Intel Atom® プロセッサー上で MOVS/STOS 命令と REP プリフィクスを使用する場合は、以下の事項について認識する必要があります。

- カウント値が小さい場合、REP プリフィクスを使用すると、REP プリフィクスを使用しないときよりも効率が低下します。これは、ハードウェアによって小さな REP カウントの最適化が行われるためです。

- カウント値が大きい場合、REP プリフィクスを使用すると、16 バイト SIMD 命令を使用するときよりも効率が低下します。
- ループ反復でアドレスをインクリメントする場合は、明示的な ADD 命令よりも LEA 命令を優先すべきです。
- メモリー操作が L2 にアクセスするようなデータ要素の場合、ソフトウェア・プリフェッチを使用してデータを L1 に格納すると、メモリー操作が再発行されることを回避できます。
- 文字列/メモリー操作がシステムメモリーにアクセスする場合は、ストリーミング・ストア命令の非テンポラルなヒントを使用すると、キャッシュの汚染を回避できます。

例 F-4 64 バイトのメモリーコピー

```
T1: prefetcht0 [eax+edx+0x80] ; 先行する 2 つのループ分をプリフェッチ
    movdqa xmm0, [eax+ edx] ; ソースからデータをロード (プリフェッチにより L1 から)
    movdqa xmm1, [eax+ edx+0x10]
    movdqa xmm2, [eax+ edx+0x20]
    movdqa xmm3, [eax+ edx+0x30]
    movdqa [ebx+ edx], xmm0; デスティネーションへストア
    movdqa [ebx+ edx+0x10], xmm1
    movdqa [ebx+ edx+0x30], xmm2
    movdqa [ebx+ edx+0x30], xmm3
    lea edx, 0x40 ; 次の反復のオフセットを合わせるため LEA を利用
    dec ecx
    jnz T1
```

F.3.3.5 引数渡し

Intel Atom® マイクロアーキテクチャーにおけるロード-ストア・フォワーディングのサポートは限定的であるため、スタックを介したパラメーターの受け渡しは、呼び出し先関数による参照を制限します。例えば、「bool」データと「char」データは通常、32 ビット・データとしてスタック上にプッシュされますが、呼び出し先関数がスタックから「bool」データまたは「char」データを読み出すと、ストア・フォワーディングの遅延が発生し、メモリー操作が再発行されます。

コンパイラーはこの制約を認識してプロローグを生成し、呼び出し先関数がサイズの小さなデータではなく 32 ビット・データを読み出せるようにすべきです。

アセンブリー/コンパイラー・コーディング規則 18 (影響 MH、一般性 M): Intel Atom® プロセッサーでは、「bool」値と「char」値は、32 ビット・データとしてスタックに渡したり、スタックから読み出すべきです。

F.3.3.6 関数呼び出し

Intel Atom®マイクロアーキテクチャーでは、PUSH/POP 命令を使用してスタック領域や、関数呼び出し/戻り間のアドレス調整を管理するほうが、ENTER/LEAVE 命令を使用するよりも適しています。

これは、PUSH/POP が MSROM を必要とせず、スタック・ポインター・アドレスの更新が AGU で行われるためです。

呼び出し先関数は、呼び出し元に戻る必要がある場合、POP 命令を発行して、データをリストアしたり、EBP からスタックポインターをリストアできます。

アセンブリー/コンパイラー・コーディング規則 19 (影響 MH、一般性 M): Intel Atom® プロセッサーでは、PUSH/POP のレジスター形式を優先し、LEAVE の使用は避けます。また、ESP の調整には、ADD/SUB ではなく LEA を使用します。

F.3.3.7 乗算/加算の依存関係チェーンの最適化

依存関係がある乗算や加算は、Intel Atom® マイクロアーキテクチャーのフロントエンドとインオーダー実行パイプラインを最適化するための良いコーディング手法例です。

例 F-5a に、アウトオブオーダー・マイクロアーキテクチャーで使用可能なコードシーケンスを示します。このシーケンスは、Intel Atom®マイクロアーキテクチャーで最適なものとは大きく異なります。乗算や加算のレイテンシーが最大になり、2 マイクロオペレーション (uop) 発行が可能なパイプラインを活用するにはあまり適していません。

例 F-5b には、2 マイクロオペレーション (uop) 発行が可能な Intel Atom® マイクロアーキテクチャーのインオーダー・パイプラインを活用する改善されたコードシーケンスを示します。乗算と加算との間に依存関係が存在するため、レイテンシーの発生は部分的にのみ隠匿されます。

例 F-5 依存関係のある乗算と加算の例

a) ストールが発生する命令シーケンス

```
; アキュムレーター xmm2 は初期化されています
Top: movaps xmm0, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     movaps xmm1, [edi] ; 16 バイト境界でメモリーにストアされたベクトル
     mulps xmm0, xmm1
     addps xmm2, xmm0 ; 依存性と分岐は mul と add の遅延を招く
     add esi, 16 ;
     add edi, 16
     sub ecx, 1
     jnz top
```

b) 実行スループットを高めるように改善された命令シーケンス

```
; アキュムレーター xmm4 は初期化されています
Top: movaps xmm0, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm0, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm4, xmm0 ; 独立した命令により、部分的な遅延がカバーされる
     dec ecx ;
     jnz top
```

c) アンロールとインターリーブによってさらに改善された命令シーケンス

```
; アキュムレーター xmm0, xmm1, xmm2, xmm3 は初期化されています
Top: movaps xmm0, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm0, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm5, xmm1 ; アンロールとインターリーブによって依存する乗算は引き離される
     movaps xmm1, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm1, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm6, xmm2 ; アンロールとインターリーブによって依存する乗算は引き離される
     movaps xmm2, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm2, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm7, xmm3 ; アンロールとインターリーブによって依存する乗算は引き離される
     movaps xmm3, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm3, [edi] ;
```

```
    lea esi, [esi+16] ; mulps と同時にスケジュール可能
    addps xmm4, xmm0 ; アンロールとインターリーブによって依存する乗算は引き離される
    sub ecx, 4;
    jnz top
    ; ループ内の依存性を減らすため、アキュムレーターは xmm0, xmm1, xmm2, xmm3 を集計する
```

例 F-5c は、命令レベルの並列性を高め、乗算と加算におけるレイテンシーの発生をさらに抑える手法を示しています。アンロールを 4 回行うことにより、各 ADDPS 命令を依存関係がある更新命令の MULPS から離して配置できます。また、インターリーブ手法を使用することにより、依存関係がない ADDPS と MULPS を近くに配置できます。MULPS と ADDPS を実行するハードウェアはパイプライン化されているため、この手法では、例 F-5b と比べてはるかに効果的にレイテンシーを隠匿できます。

F.3.3.8 位置に依存しないコード

位置に依存しないコードは多くの場合、命令ポインターの値を取得する必要があります。例 F-6a は、一致する RET がない CALL を発行して IP の値を ECX レジスターに格納する手法を示しています。例 F-6b は、一致する CALL/RET ペアを使用して IP の値を ECX レジスターに格納する代替手法を示します。

例 F-6 命令ポインターの照会手法

```
a) RET がない CALL を使用して IP を取得
   call _label; プッシュされた return アドレスは次の命令の IP
_label:
   pop ECX; この命令の IP は、ECX に代入されます
```

```
b) call/ret が一致したペアを使用
   call _lblcx;
   ...; ECX はこの命令の IP を含む
   ...
_lblcx
   mov ecx, [esp];
   ret
```

F.4 命令レイテンシー

ここでは、Intel Atom®マイクロアーキテクチャーのポート･バインディングとレイテンシーに関する情報を提供します。各命令のポート･バインディング情報は、以下の 3 つの状態のいずれかを示しています。

- 1 桁の数字 - 発行される特定のポート番号
- (0, 1) - ポート 0 またはポート 1
- 'B' - 両方のポートが必要

「命令」列では以下のような情報を示します。

- 同じ命令の異なるオペランド構文のポート･バインディングおよびレイテンシーが同じ場合は、オペランド構文を省略しています。
- 同じ命令の異なるオペランド構文のポート･バインディングやレイテンシーが異なる場合は、オペランド構文を記載しています。ただし、オペランドサイズが異なる命令構文については、脚注を使って簡潔に記載している場合があります。

MSROM によるデコーダーの支援が必要な命令には、「説明カラム」にその旨が記されています (よりデコード効率の高い代替手段があれば、そのような命令の使用は最小限に抑えるべきです)。

表 F-2 Intel Atom® マイクロアーキテクチャーの命令レイテンシー

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
ADD/AND/CMP/OR/SUB/XOR/TEST[1] (E)AX/AL, imm;	(0, 1)	1	0.5
ADD/AND/CMP/OR/SUB/XOR[2] mem, lmm8; ADD/AND/CMP/OR/SUB/XOR/TEST[4] mem, imm; TEST m8, imm8	0	1	1
ADD/AND/CMP/OR/SUB/XOR/TEST[2] mem, reg; ADD/AND/CMP/OR/SUB/XOR[2] reg, mem;	0	1	1
ADD/AND/CMP/OR/SUB/XOR[2] reg, lmm8; ADD/AND/CMP/OR/SUB/XOR[4] reg, imm	(0, 1)	1	0.5
ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, mem	B	7	6
ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, xmm	B	6	5
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, mem	B	5	1
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, xmm	1	5	1
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, mem	0	1	1
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, xmm	(0, 1)	1	1
BSF/BSR r16, m16	B	17	16
BSF/BSR[3] reg, mem	B	16	15
BSF/BSR[4] reg, reg	B	16	15
BT m16, imm8; BT[3] mem, imm8	(0, 1)	2; 1	1
BT m16, r16; BT[3] mem, reg	B	10, 9	8
BT[4] reg, imm8; BT[4] reg, reg	1	1	1
BTC m16, imm8; BTC[3] mem, imm8	B	3; 2	2
BTC/BTR/BTS m16; r16	B	12	11
BTC/BTR/BTS[3] mem, reg	B	11	10
BTC/BTR/BTS[4] reg, imm8; BTC/BTR/BTS[4] reg, reg	1	1	1
CALL mem	(0, 1)	2	2
CALL reg; CALL rel16; CALL rel32	B	1	1
CMOV[4] reg, mem; MOV[1] (E)AX/AL, MOFFS; MOV[2] mem, imm	0	1	1
CMOV[4] reg, reg; MOV[2] reg, imm; MOV[2] reg, reg; ; SETcc r8	(0, 1)	1	0.5
CMPPD/CMPPS xmm, mem, imm; CVTTPS2DQ xmm, mem	B	7	6
CMPPD/CMPPS xmm, xmm, imm; CVTTPS2DQ xmm, xmm	B	6	5
CMPSD/CMPSS xmm, mem, imm	B	5	1
CMPSD/CMPSS xmm, xmm, imm	1	5	1
(U)COMISD/(U)COMISS xmm, mem;	B	10	9
(U)COMISD/(U)COMISS xmm, xmm;	B	9	8
CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, mem	B	8	7
CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, xmm	B	7	6
CVTDQ2PS/CVTSD2SS/CVTSI2SS/CVTSS2SD xmm, mem	B	7	6

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H
CVTDQ2PS/CVTSD2SS/CVTSS2SD xmm, xmm	B	6	5
CVT(T)PD2PI mm, mem; CVTPI2PD xmm, mem	B	8	7
CVT(T)PD2PI mm, xmm; CVTPI2PD xmm, mm	B	7	6
CVTPI2PS/CVTSI2SD xmm, mem;	B	5	4
CVTPI2PS xmm, mm;	1	5	1
CVTPS2DQ xmm, mem;	B	7	6
CVTPS2DQ xmm, xmm;	B	6	5
CVT(T)PS2PI mm, mem;	B	5	5
CVT(T)PS2PI mm, xmm;	1	5	1
CVT(T)SD2SI[3] reg, mem; CVT(T)SS2SI r32, mem	B	9	8
CVT(T)SD2SI[3] reg, xmm; CVT(T)SS2SI r32, xmm	B	8	7
CVTSI2SD xmm, r32; CVTSI2SS xmm, r32	B	7; 6	5
CVTSI2SD xmm, r64; CVTSI2SS xmm, r64	B	6; 7	5
CVT(T)SS2SI r64, mem; RCPPS xmm, mem	B	10	9
CVT(T)SS2SI r64, xmm; RCPPS xmm, xmm	B	9	8
CVTTPD2DQ xmm, mem	B	8	7
CVTTPD2DQ xmm, xmm	B	7	6
DEC/INC[2] mem; MASKMOVQ; MOVAPD/MOVAPS mem, xmm	0	1	1
DEC/INC[2] reg; FLD ST; FST/FSTP ST; MOVDQ2Q mm, xmm	(0, 1)	1	0.5
DIVPD; DIVPS	B	125; 70	124; 69
DIVSD; DIVSS	B	62; 34	61; 33
EMMS; LDMXCSR	B	5	4
FABS/FCHS/FXCH; MOVQ2DQ xmm, mm; MOVSX/MOVZX r16, r16	(0, 1)	1	0.5
FADD/FSUB/FSUBR[3] mem	B	5	4
FADD/FADDP/FSUB/FSUBP/FSUBR/FSUBRP ST;	1	5	1
FCMOV	B	6	5
FCOM/FCOMP[3] mem	B	1	1
FCOM/FCOMP/FCOMPP/FUCOM/FUCOMP ST; FTST	1	1	1
FCOMI/FCOMIP/FUCOMI/FUCOMIP ST	B	9	8
FDIV/FSQRT[3] mem; FDIV/FSQRT ST	0	25-65	24-64
FIADD/FIMUL[5] mem	B	11	10
FICOM/FICOMP mem	B	7	6
FILD[4] mem	B	5	4
FLD[3] mem; FXAM; MOVAPD/MOVAPS/MOVD xmm, mem	0	1	1
FLDCW	B	5	4
FMUL/FMULP ST; FMUL[3] mem	0	5	1

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
FNSTSW AX; FNSTSW m16	B	10; 14	9; 13
FST/FSTP[3] mem	B	2	1
HADDPD/HADDPS/HSUBPD/HSUBPS xmm, mem	B	9	8
HADDPD/HADDPS/HSUBPD/HSUBPS xmm, xmm	B	8	7
IDIV r/m8; IDIV r/m16; IDIV r/m32; IDIV r/m64;	B	33;42;57;197	32;41;56;196
IMUL/MUL[6] EAX/AL, mem; IMUL/MUL AX, m16	B	7; 8	6; 7
IMUL/MUL[7] AX/AL, reg; IMUL/MUL EAX, r32	B	7; 6	6; 5
IMUL m16, imm8/imm16; IMUL r16, m16	B	7;	6
IMUL r/m32, imm8/imm32; IMUL r32, r/m32	0	5	1
IMUL r/m64, imm8/imm32;	B	14	13
IMUL r16, r16; IMUL r16, imm8/imm16	B	6	5
IMUL r64, r/m64; IMUL/MUL RAX, r/m64	B	11; 12	10; 11
JCC[1]; JMP[4] reg; JMP[1]	1	1	1
JCXZ; JECXZ; JRCXZ	B	4	1
JMP mem[4];	B	2	1
LDDQU; MOVDQU/MOVUPD/MOVUPS xmm, mem;	B	3	2
LEA r16, mem; MASKMOVDQU; SETcc m8	(0, 1)	2	1
LEA, reg, mem	1	1	1
LEAVE;	B	2;	2
MAXSD/MAXSS/MINSD/MINSS xmm, mem	B	5	1
MAXSD/MAXSS/MINSD/MINSS xmm, xmm	1	5	1
MOV[2] MOFFS, (E)AX/AL; MOV[2] reg, mem; MOV[2] mem, reg	0	1	1
MOVD mem[3], mm; MOVD xmm, reg[3]; MOVD mm, mem[3]	0	1	1
MOVD reg[3], mm; MOVD reg[3], xmm; PMOVMSK reg[3], mm	0	3	1
MOVDQA/MOVQ xmm, mem; MOVDQA/MOVD mem, xmm;	0	1	1
MOVDQA/MOVDQU/MOVUPD xmm, xmm; MOVQ mm, mm	(0, 1)	1	0.5
MOVDQU/MOVUPD/MOVUPS mem, xmm;	B	2	2
MOVHLPS;MOVLHPS;MOVHPD/MOVHPS/MOVLPD/MOVLPS	0	1	1
MOVMSKPD/MOVSKPS/PMOVMSKB reg[3], xmm	0	3	1
MOVNTI[3] mem, reg; MOVNTPD/MOVNTPS; MOVNTQ	0	1	1
MOVQ mem, mm; MOVQ mm, mem; MOVDDUP	0	1	1
MOVSD/MOVSS xmm, xmm; MOVSXD[5] reg, reg	(0, 1)	1	0.5
MOVSD/MOVSS xmm, mem; PALIGNR	0	1	1
MOVSD/MOVSS mem, xmm; PINSRW	0	1	1
MOVSHDUP/MOVSLDUP xmm, mem	0	1	1

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H
MOVSHDUP/MOVSLDUP/MOVUPS xmm, xmm	(0, 1)	1	0.5
MOVSX/MOVZX r16, m8; MOVSX/MOVZX r16, r8	0	3; 2	1
MOVSX/MOVZX reg[3], r/m8; MOVSX/MOVZX reg[3], r/m16	0	1	1
MOVSXD[5] reg, mem; MOVSXD r64, r/m32	0	1	1
MULPS/MULSD xmm, mem; MULSS xmm, mem;	0	5; 4	2
MULPS/MULSD xmm, xmm; MULSS xmm, xmm	0	5; 4	2
MULPD	B	5; 4	2
NEG/NOT[2] mem; PREFETCHNTA; PREFETCHTx	0	10	9
NEG/NOT[2] reg; NOP	(0, 1)	1	0.5
PABSB/D/W mm, mem; PABSB/D/W xmm, mem	0	1	1
PABSB/D/W mm, mm; PABSB/D/W xmm, xmm	(0, 1)	1	0.5
PACKSSDW/WB mm, mem; PACKSSDW/WB xmm, mem	0	1	1
PACKSSDW/WB mm, mm; PACKSSDW/WB xmm, xmm	0	1	1
PACKUSWB mm, mem; PACKUSWB xmm, mem	0	1	1
PACKUSWB mm, mm; PACKUSWB xmm, xmm	0	1	1
PADDB/D/W/Q mm, mem; PADDB/D/W/Q xmm, mem	0	1	1
PADDB/D/W/Q mm, mm; PADDB/D/W/Q xmm, xmm	(0, 1)	1	0.5
PADDSB/W mm, mem; PADDSB/W xmm, mem	0	1	1
PADDSB/W mm, mm; PADDSB/W xmm, xmm	(0, 1)	1	0.5
PADDUSB/W mm, mem; PADDUSB/W xmm, mem	0	1	1
PADDUSB/W mm, mm; PADDUSB/W xmm, xmm	(0, 1)	1	0.5
PAND/PANDN/POR/PXOR mm, mem; PAND/PANDN/POR/PXOR xmm, mem	0	1	1
PAND/PANDN/POR/PXOR mm, mm; PAND/PANDN/POR/PXOR xmm, xmm	(0, 1)	1	0.5
PAVGB/W mm, mem; PAVGB/W xmm, mem	0	1	1
PAVGB/W mm, mm; PAVGB/W xmm, xmm	(0, 1)	1	0.5
PCMPEQB/D/W mm, mem; PCMPEQB/D/W xmm, mem	0	1	1
PCMPEQB/D/W mm, mm; PCMPEQB/D/W xmm, xmm	(0, 1)	1	0.5
PCMPGTB/D/W mm, mem; PCMPGTB/D/W xmm, mem	0	1	1
PCMPGTB/D/W mm, mm; PCMPGTB/D/W xmm, xmm	(0, 1)	1	0.5
PEXTRW;	B	4	1
PHADDD/PHSUBD mm, mem; PHADDD/PHSUBD xmm, mem	B	4	3
PHADDD/PHSUBD mm, mm; PHADDD/PHSUBD xmm, xmm	B	3	2
PHADDW/PHADDSW mm, mem; PHADDW/PHADDSW xmm, mem	B	6; 8	5;7
PHADDW/PHADDSW mm, mm; PHADDW/PHADDSW xmm, xmm	B	5; 7	M
PHSUBW/PHSUBSW mm, mem; PHSUBW/PHSUBSW xmm, mem	B	6; 8	M
PHSUBW/PHSUBSW mm, mm; PHSUBW/PHSUBSW xmm, xmm	B	5; 7	M

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mem	0	4	1
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, xmm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, mem	0	5	1
PMAXSW/UB mm, mem; PMAXSW/UB xmm, mem	0	1	1
PMAXSW/UB mm, mm; PMAXSW/UB xmm, xmm	(0, 1)	1	0.5
PMINSW/UB mm, mem; PMINSW/UB xmm, mem	0	1	1
PMINSW/UB mm, mm; PMINSW/UB xmm, xmm	(0, 1)	1	0.5
PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mm; PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mem	0	4	1
PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, xmm; PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, mem	0	5	1
POP mem[5]; PSLLD/Q/W mm, mem; PSLLD/Q/W xmm, mem	B	3	2
POP r16; PUSH mem[4]; PSLLD/Q/W mm, mm; PSLLD/Q/W xmm, xmm	B	2	1
POP reg[3]; PUSH reg[4]; PUSH imm	B	1	1
POPA ; POPAD	B	9	8
PSHUFB mm, mem; PSHUFD; PSHUFHW; PSHUFLW; PSHUFW	0	1	1
PSHUFB mm, mm; PSLLD/Q/W mm, imm; PSLLD/Q/W xmm, imm	0	1	1
PSHUFB xmm, mem	B	5	4
PSHUFB xmm, xmm	B	4	3
PSIGNB/D/W mm, mem; PSIGNB/D/W xmm, mem	0	1	1
PSIGNB/D/W mm, mm; PSIGNB/D/W xmm, xmm	(0, 1)	1	0.5
PSRAD/W mm, imm; PSRAD/W xmm, imm;	0	1	1
PSRLD/Q/W mm, mem; PSRLD/Q/W xmm, mem	B	3	2
PSRLD/Q/W mm, mm; PSRLD/Q/W xmm, xmm	B	2	1
PSRLD/Q/W mm, imm; PSRLD/Q/W xmm, imm;	0	1	1
PSLLDQ/PSRLDQ xmm, imm; SHUFPD/SHUFPS	0	1	1
PSUBB/D/W/Q mm, mem; PSUBB/D/W/Q xmm, mem	0	1	1
PSUBB/D/W/Q mm, mm; PSUBB/D/W/Q xmm, xmm	(0, 1)	1	0.5
PSUBSB/W mm, mem; PSUBSB/W xmm, mem	0	1	1
PSUBSB/W mm, mm; PSUBSB/W xmm, xmm	(0, 1)	1	0.5
PSUBUSB/W mm, mem; PSUBUSB/W xmm, mem	0	1	1
PSUBUSB/W mm, mm; PSUBUSB/W xmm, xmm	(0, 1)	1	0.5
PUNPCKHBW/DQ/WD; PUNPCKLBW/DQ/WD	0	1	1
PUNPCKHQDQ; PUNPCKLQDQ	0	1	1
PUSHA ; PUSHAD	B	8	7
RCL mem[2], 1; RCL reg[2], 1	0	1	1

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H	06_1CH, 06_26H, 06_27H
RCL m8, CL; RCL m16, CL; RCL mem[3], CL;	B	18;16; 14	17;15;13
RCL m8, imm; RCL m16, imm; RCL mem[3], imm;	B	18; 17; 14	17;16;13
RCL r8, CL; RCL r16, CL; RCL reg[3], CL;	B	17; 16; 14	16;15;14
RCL r8, imm; RCL r16, imm; RCL reg[3], imm;	B	18;16; 14	17;15;13
RCPSS	0	4	1
RCR mem[2], 1; RCR reg[2], 1	B	7; 5	6;4
RCR m8, CL; RCR m16, CL; RCR mem[3], CL;	B	15; 13; 12	14;12;11
RCR m8, imm; RCR m16, imm; RCR mem[3], imm;	B	16,;14; 12	15;13;11
RCR r8, CL; RCR r16, CL; RCR reg[3], CL;	B	14; 13; 12	13;12;11
RCR r8, imm; RCR r16, imm; RCR reg[3], imm;	B	15, 14, 12	14;13;11
RET imm16	B	1	1
RET (far)	B	79	
ROL; ROR; SAL; SAR; SHL; SHR	0	1	1
SETcc		1	1
SHLD[8] mem, reg, imm; SHLD r64, r64, imm; SHLD m64, r64, CL	B	11	10
SHLD m32, r32; SHLD r32, r32	B	4; 2	3; 1
SHLD m16, r16, CL; SHLD r16, r16, imm; SHLD r64, r64, CL	B	10	9
SHLD r16, r16, CL; SHRD m64, r64; SHRD r64, r64, imm	B	9	8
SHRD m32, r32; SHRD r32, r32	B	4; 2	3; 1
SHRD m16, r16; SHRD r16, r16	B	6	5
SHRD r64, r64, CL	B	8	7
STMXCSR	B	15	14
TEST[2] reg, reg; TEST[4] reg, imm	(0, 1)	1	0.5
UNPCKHPD; UNPCKHPS; UNPCKLPD, UNPCKLPS	0	1	1

オペランドサイズ (osize) とアドレスサイズ (asize) に関する注意:

1. osize = 8, 16, 32 または asize = 8, 16, 32
2. osize = 8, 16, 32, 64
3. osize = 32, 64
4. osize = 16, 32, 64 または asize = 16, 32, 64
5. osize = 16, 32
6. osize = 8, 32
7. osize = 8, 16
8. osize = 16, 64

F.5 Silvermont† マイクロアーキテクチャー

Intel Atom® プロセッサー E3000 と C2000 シリーズは、Silvermont† マイクロアーキテクチャーをベースにしています。Silvermont† マイクロアーキテクチャーは、タブレット、携帯電話、そして PC からマイクロサーバーまで、幅広いコンピューター・デバイスで利用できます。インテル® 64 アーキテクチャーと IA-32 アーキテクチャーのサポートに加えて、Silvermont† マイクロアーキテクチャーでは主に次の点が拡張されています。

- 整数命令のアウトオブオーダー実行、および非整数命令とメモリー命令間の実行順序を分離しています。対照的に、45nm と 32nm の Intel Atom® マイクロアーキテクチャー (付録 D を参照) では、インオーダー実行が厳守され、命令レベルの並列性が制限されていました。
- 非ブロッキング命令における複数の未処理ミスの許容 (8 回まで)。前世代のプロセッサーでは、1 つのメモリー命令で問題が発生すると (例えば、キャッシュミスなど)、その問題が解決されるまで後続のすべての命令がストールしましたが、新しいマイクロアーキテクチャーでは最大 8 つの未処理参照が許容されます。
- 2 コアのモジュラーシステム設計。フロントサイド・バスの代わりにポイントツーポイントのインターフェイスを使って、新しい内蔵メモリー・コントローラーに接続された L2 キャッシュを共有します。
- インテル® SSE4.1、インテル® SSE4.2、インテル® AES New Instructions (インテル® AES-NI)、PCLMULQDQ が追加されています。

図 F-2 に Silvermont† マイクロアーキテクチャーの基本パイプライン機能を示します。シングルスレッドのパフォーマンスを向上するため、メモリークラスターと実行クラスターの設計が大幅に見直されている一方、これまでと同様に、小さなフォームファクターで低消費電力を実現する取り組みが行われています。各パイプラインには、リザベーション・ステーション (RSV) と呼ばれる専用のスケジューリング・キューがあります。浮動小数点命令とメモリー命令はそれぞれのキューからプログラム順にスケジュールされ、整数命令はそれぞれのキューからアウトオブオーダーでスケジュールされます。

これは、整数命令がインオーダー実行であった前世代とは対照的です。アウトオブオーダー・スケジューリングにより、これらの命令ではソースやリソースが利用できない場合に発生するストールを許容することができます。メモリー命令は、アドレスの生成 (AGEN) をインオーダーで行い、スケジューリング・キューからインオーダーでスケジュールしなければいけませんが、実行はアウトオブオーダーで行うことができます。

(SIMD 整数、SIMD 浮動小数点、x87 浮動小数点を含む) 非整数命令も、それぞれのスケジューリング・キューからプログラム順にスケジュールされますが、これらは個別のスケジューリング・キューなので、ほかのスケジューリング・キューにある命令とは切り離して実行することができます。

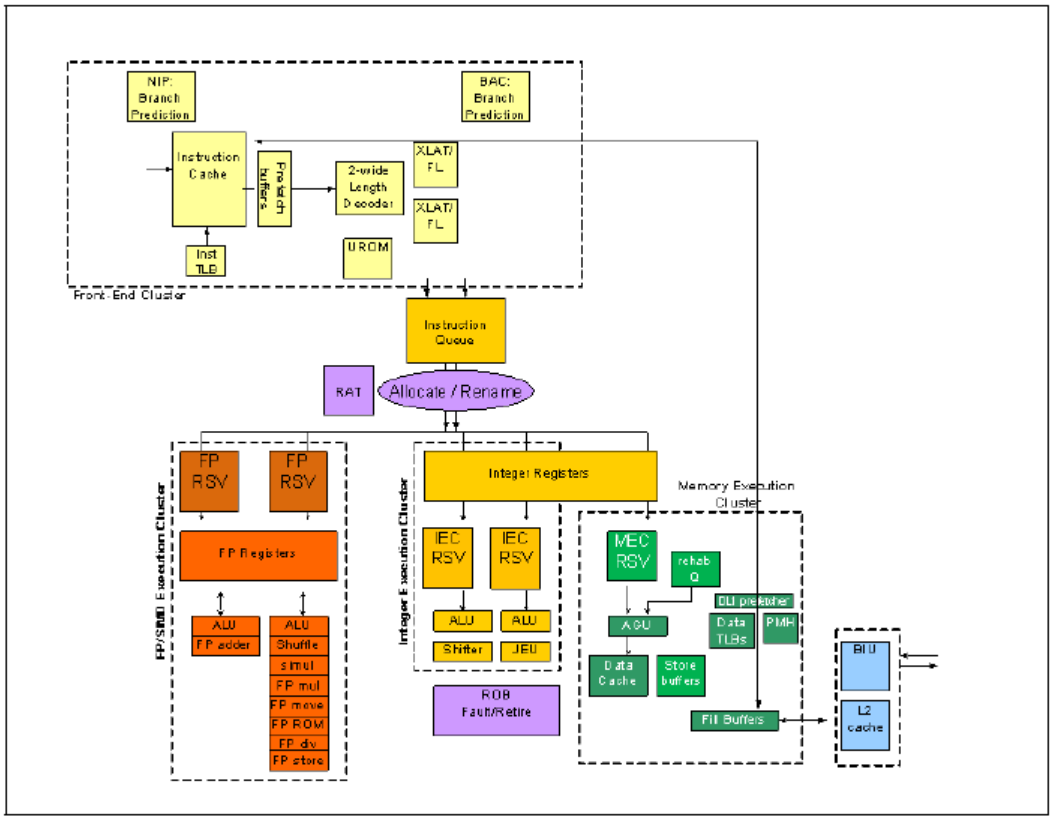


図 F-2 Silvermont† マイクロアーキテクチャーのパイプライン

Silvermont† マイクロアーキテクチャーは、アウトオブオーダー・スケジューリングにより、多様なフォームファクターの (例えば、携帯電話、タブレットからマイクロサーバーにわたる) プラットフォーム・パフォーマンスを最大限に引き出し、消費電力と面積コストを最小限に抑えるように設計されています (つまり、パフォーマンス/電力/コスト効率を最大化しています)。共有 L2 キャッシュを装備したマルチコア・アーキテクチャーを採用しているため、インテル® ハイパースレッディング・テクノロジーはサポートされません。クラスターレベルの機能については、この節の後半で説明します。

図 F-2 に薄黄色で示されているフロントエンド・クラスター (FEC) は、同時に 2 命令を処理できるデコード・パイプラインであり、消費電力が最適化されています。FEC はメモリーから命令をフェッチしデコードを行います。このとき、命令キャッシュからのプリデコード情報を利用することで、コストのかかる命令長の検出をデコード時に行わないようにしています。フロントエンドには分岐ターゲットバッファー (BTB) と高度な分岐予測ハードウェアがあります。

フロントエンドは、アロケーション、リネーミング、およびリタイアメント (ARR) クラスターを介して OOO 実行エンジンに接続されています (図 F-2 の紫色)。ARR は、FEC からマイクロオペレーション (uop) を受け取り、リソースチェックを行います。レジスター・エイリアス・テーブル (RAT) は、論理レジスターから物理レジスターへのリネームを行います。リオーダーバッファー (ROB) は、プログラム順に操作を並べ替えて実行 (リタイア) します。また、割り込み、例外、アシスト時には実行を停止して、マイクロコードに対するプログラム制御を実行します。

Silvermont† マイクロアーキテクチャーは分散スケジューリングを採用しているため、リネーム処理後にマイクロオペレーション (uop) はさまざまなクラスター (IEC: 整数実行クラスター、MEC: メモリー実行クラスター、FPC: 浮動小数点クラスター) に送られ、スケジューリングされます (図 F-2 では FP RSV、IEC RSV、MEC RSV として示されています)。

FPC RSV と IEC RSV は 2 セット (各ポートに 1 つずつ) あり、MEC RSV は 1 セットあります。各 RSV は、ARR クラスターからサイクルごとに最大 2 マイクロオペレーション (uop) を受け取り、実行準備が整ったものから実行ユニットへディスパッチします。

分散型リザベーション・ステーションの概念をサポートするため、整数実行を要求する load-op (ロード-実行) 型や load-op-store (ロード-実行-ストア) 型のマクロ命令は、MEC RSV に送られるメモリー操作と、IEC RSV に送られる整数実行操作に分割する必要があります。IEC スケジューラーは、各 IEC RSV から実行準備が整っている最も古い命令を選択します。一方、MEC スケジューラーと FPC スケジューラーは、それぞれの RSV から最も古い命令を選択します。MEC クラスターと FPC クラスターはインオーダー・スケジューラーを採用していますが、FPC RSV の新しい命令は、別の FPC RSV や IEC RSV、MEC RSV にあるより古い命令よりも前に実行できます。

各実行ポートには固有の機能ユニットがあります。表 F-3 の Silvermont† マイクロアーキテクチャーの機能ユニットのポートへの割り当を示します。図 F-2 では IEC がオレンジ、MEC が緑、FPC が赤で示されています。前世代の Intel Atom® マイクロアーキテクチャーと比べると、Silvermont† マイクロアーキテクチャーでは IEC に整数乗算ユニット (IMUL) が追加されています。

表 F-3 Silvermont† マイクロアーキテクチャーの機能ユニットの割り当て

	Port 0	Port 1
IEC	ALU0, Shift/Rotate Unit, LEA with no index	ALU1, Bit processing unit, Jump unit, IMUL, POPCNT, CRC32, LEA[1]
FPC	SIMD ALU, SIMD shift/Shuffle unit, SIMD FP mul/div/cvt unit, STTNI/AESNI/PCLMULQDQ unit, RCP/RSQRT unit, F2I convert unit	SIMD ALU, SIMD FPadd unit, F2I convert unit
MEC	Load/Store	

注意:

1. 有効なインデックスとディスプレースメントを持つ LEA は複数のマイクロオペレーション (uop) に分けられ、両方のポートを使用します。有効なインデックスを持つ LEA はポート 1 で実行されます。

メモリー実行クラスター (MEC) (図 F-2 に緑色で示されている) は、32 ビットと 36 ビットの物理アドレスモードをサポートします。Silvermont† マイクロアーキテクチャーは、2 つのレベルからなるデータ TLB を実装しており、スモールページとラージページ (2MB または 4MB) をサポートしています。第 1 レベルのマイクロ TLB (μTLB) は小さく、より大きな第 2 レベルの TLB (DTLB) にバックアップされます。命令 TLB ミスとデータ TLB ミスはどちらもハードウェア・ページ・ウォーカーによって処理されます。

MEC には、すべてのロードとストアのスケジューリングを行う MEC RSV もあります。ロード命令とストア命令は、後のパイプラインでメモリーを並べ替えなくても済むように、プログラム順にアドレス生成処理が行われます。そのため、不明なアドレスによって新しいメモリー命令がストールします。(μTLB ミスやリソースが利用できないなどの) 問題が発生したメモリー操作は RehabQ (修復キュー) と呼ばれる別のキューに配置されるため、後続の命令をすべてストールする代わりに、(問題が発生していない) より新しい命令の実行を継続できます。問題が発生した命令は、問題が解決した後に RehabQ から再発行されます。Silvermont† マイクロアーキテクチャーでは、データ・キャッシュ・ミスは 8 回までブロックされないため、ロードミスはそれほど問題と見なされません。

バスクラスター (BIU) の L2 キャッシュは、プロセッサー・コア外部とのすべての通信を処理します。この L2 キャッシュは最大 1MB で、前世代の Intel Atom® マイクロアーキテクチャーと比べるとレイテンシーが最適化されています。前世代の Intel Atom® プロセッサーのフロントサイド・バスに代わり、最適化された新しいメモリー・コントローラーに接続するイントラダイ・インターコネクト (IDI) ファブリックが採用されています。BIU には L2 データ・プリフェッチャーも装備されます。

新しいコアレベルのマルチプロセシング (CMP) システム構成では、2 つのプロセッサー・コアが 1 つの BIU に要求を送り、コア間の多重化は BIU によって処理されます。この基本 CMP モジュールを複製してクアッドコア構成を作成したり、1 コアのみにしてシングルコア構成を作成できます。

F.5.1 整数パイプライン

ロードのパイプライン・ステージがほかの整数パイプラインとインライン化されなくなったため、ロードを伴わない操作の実行を高速化し、分岐予測のペナルティーが前世代の Intel Atom® プロセッサーよりも 3 サイクル少なくなっています。フロントエンドのパイプライン・ステージは前世代の Intel Atom® プロセッサーと同じで、フェッチに 3 サイクル、デコードに 3 サイクルかかります。ARR パイプステージは、アウトオブオーダー・アロケーションとレジスターのリネームを行い、必要に応じてマイクロオペレーション (uop) を分割し、各リザベーション・ステーションへ送ります。RSV ステージでは、各リザベーション・ステーションがそれぞれのスケジューリングを行います。実行パイプラインは前世代の Intel Atom® プロセッサーによく似ています。マイクロオペレーション (uop) のすべての部分の操作が完了すると、ROB がインオーダーで最終処理を行います。

F.5.2 浮動小数点パイプライン

INT パイプラインよりも FP パイプラインのほうが長く、命令に応じて 1 ~ 5 の実行ステージがあります。ほかのインテル® マイクロアーキテクチャーと同様に、Silvermont† マイクロアーキテクチャーでもハイパフォーマンスを達成するため、FP アシスト (特定の浮動小数点操作を実行パイプラインでネイティブに処理できず、マイクロコードで実行しなければならない場合) の数を最小限に抑える必要があります。そのため、可能な場合は例外をマスクし、DAZ (デノーマルをゼロとして扱う) フラグと FTZ (ゼロフラッシュ) フラグを設定して実行します。

前述のように、各 FPC RSV において命令はインオーダーでスケジュールされますが、RSV 間でアウトオブオーダーになってもかまいません。

F.6 Goldmont† マイクロアーキテクチャー

Goldmont† マイクロアーキテクチャーは、Silvermont† マイクロアーキテクチャー (F.5 節を参照) の成功を基に、次の拡張を提供します。

- 3 ワイド・スーパースカラー・パイプラインのアウトオブオーダー実行エンジン。
  - デコーダーはサイクルごとに 3 命令をデコード可能。
  - マイクロコード・シーケンサーは、アロケーションのためサイクルごとに 3 つの uop をリザベーション・ステーションへ送出可能。
  - リタイアメントは、サイクルあたり 3 ピークレートをサポート。

- 命令デコーダーからフェッチ・パイプラインを分離することで、分岐予測を強化。
- 大きなアウトオブオーダー実行ウィンドウとバッファーにより、整数、FP/SIMD、およびメモリー命令タイプに渡り、より深いアウトオブオーダー実行が可能となります。
- 完全なアウトオブオーダー・メモリー実行とディスアンビゲーション。Goldmont† マイクロアーキテクチャーは、サイクルごとに 1 つのロードと 1 つのストアを実行できます (つまり 2 つの操作)。Silvermont† マイクロアーキテクチャーでは、サイクルごとに 1 つのロードまたは 1 つのストアを実行できました。メモリー実行パイプラインはまた、4KB ページで 512 エントリーに拡張された第 2 レベルの TLB を含みます。
- Goldmont† マイクロアーキテクチャーの整数実行クラスターは、3 つのパイプラインを提供し、サイクルごとに最大 3 つの簡単な ALU 操作を実行できます。
- SIMD 整数と浮動小数点命令は、128 ビット幅のエンジンで実行されます。多くの命令のスループットとレイテンシーが改善されています。例えば、PSHUFB 命令は 1 サイクルのスループット (Silvermont† マイクロアーキテクチャーでは 5 サイクルでした) で、その他多くの SIMD 命令は倍のスループットを提供します。詳細は表 16-17 を参照してください。
- Goldmont† マイクロアーキテクチャーでは、暗号化/復号 (AES) とキャリーなしの乗算 (PCLMULQDQ) を加速する命令のスループットとレイテンシーが、かなり改善されました。
- Goldmont† マイクロアーキテクチャーは、ハードウェアによって加速された安全なハッシュ・アルゴリズムをサポートする新しい命令 SHA1 と SHA256 を提供します。
- Goldmont† マイクロアーキテクチャーはまた、NIST SP800-90C 標準に準拠する乱数生成のため RDSEED 命令をサポートします。
- 電力効率を高めるため、PAUSE 命令のレイテンシーが最適化されました。

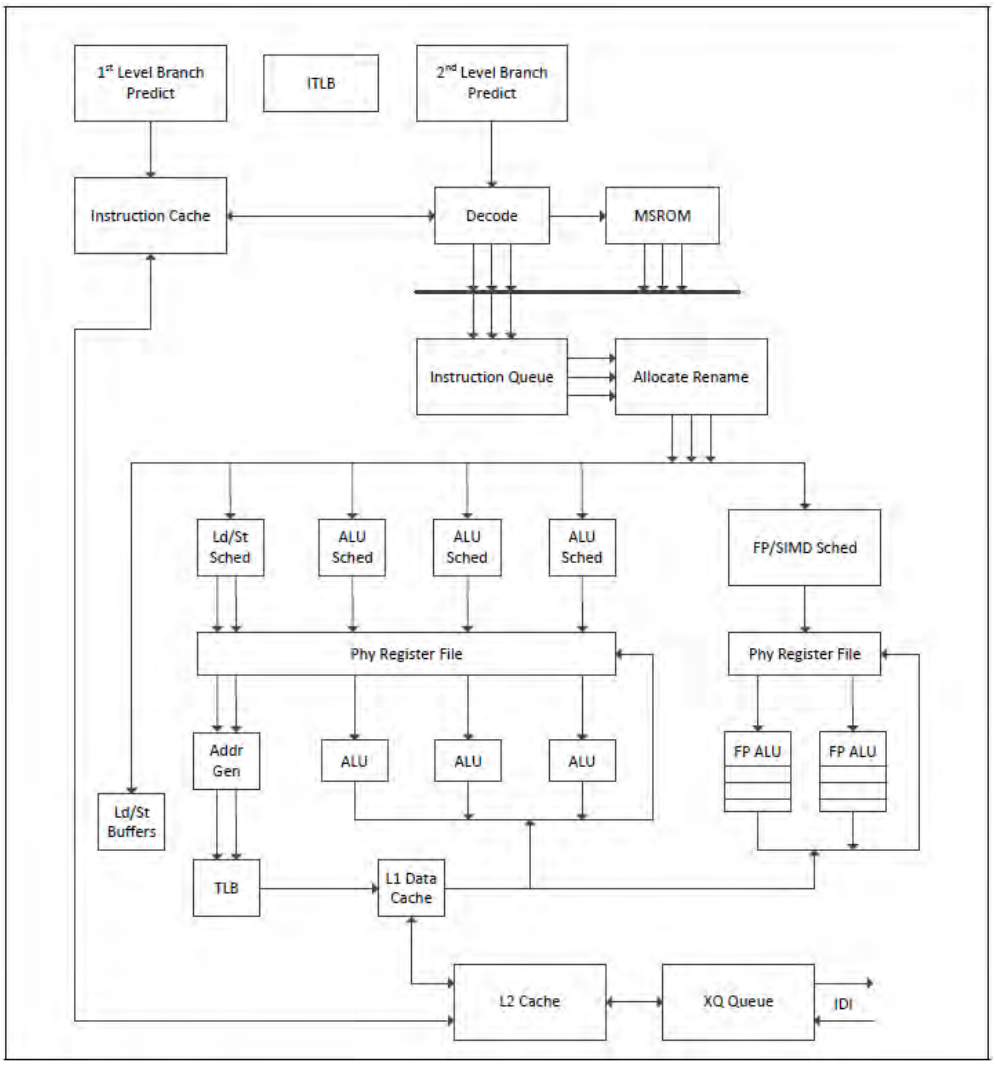


図 F-3 Goldmont† マイクロアーキテクチャーの CPU コア・パイプラインの機能

Goldmont† マイクロアーキテクチャーのフロントエンド・クラスター (FEC) では、Silvermont† マイクロアーキテクチャーの FEC に対し多くの拡張が行われています。表 F-4 にこれらの拡張をまとめています。

表 F-4フロントエンド・クラスター機能の比較

Feature	Goldmont Microarchitecture	Silvermont Microarchitecture
Number of Decoders	3	2
Max Throughput of Decoders	20 Bytes per cycle	16 Bytes per cycle
Fetch and Icache Pipeline	Decoupled	Coupled
ITLB	48 entries, large page support	48 entries
Branch Mispredict Penalty	12 cycles	10 cycles
L2 Predecode Cache	16K	NA

FEC は、アロケーション、リネーミングおよびリタイアメント (ARR) クラスターを介して OOO 実行エンジンに接続されています。uop のスケジューリングは、異なるクラスター (IEC、FPC、MEC) にまたがって分散されたリザベーション・ステーションによって扱われます。それぞれのクラスターは、ARR から複数の uop を受け取るため固有のリザベーション・ステーションを保持しています。表 F-5 は、Goldmont† マイクロアーキテクチャーと Silvermont† マイクロアーキテクチャーのアウトオブオーダーの特徴を比較したものです。

表 F-5 uop をスケジューリングする際の分散リザベーション・ステーションの比較

Cluster	Goldmont Microarchitecture	Silvermont Microarchitecture
IEC Reservation	3x distributed for each port	2x distributed for each port
	Out-of-order within each IEC RSV and between IEC, across FPC, MEC	Out-of-order within each IEC RSV and between IEC, across FPC, MEC
FPC Reservation	1x unified to ports 0, 1	2x distributed for each port
	Out-of-order within FPC RSV and across IEC, MEC	In order within each FPC RSV; out-of-order between FPC, across IEC, MEC
MEC Reservation	1x unified to ports 0, 1	1x to port 0
	Out-of-order within MEC RSV and across IEC, FPC	In order within each MEC RSV; out-of-order across IEC, FPC

メモリーを参照して整数/FP リソースを必要とする命令は、メモリー uop が MEC クラスターへ送られ、整数/FP uop が IEC/FPC クラスターへ送られます。そしてリソースが利用可能になると、表 F-5 で示すヒューリスティックに従ってアウトオブオーダー実行を開始できます。表 F-6 は、それぞれのクラスターに対するポートと実行ユニットのマッピングを示します。

表 F-6 Goldmont† マイクロアーキテクチャーの機能ユニットの割り当て

Cluster	Port 0	Port 1	Port 2
IEC	ALU0, Shift/Rotate, LEA with no index, F2I, converts/cmp, store_data	ALU1, Bit processing, JEU, IMUL, IDIV,POPCNT, CRC32, LEA, I2F, store_data	ALU2, LEA[1], I2F, flag_merge
FPC	SIMD ALU, SIMD shift/Shuffle, SIMD mul, STTNI/AESNI/PCLMULQDQ/SHA ; FP_mul, Converts, F2I convert	SIMD ALU, SIMD shuffle, FP_add, F2I compare	
MEC	Load_addr	Store_addr	

注意:

1. インデックスなしの LEA は、ポート 0、1 もしくは 2 で実行できます。有効なインデックスとディスプレースメントを持つ LEA は複数のマイクロオペレーション (uop) に分けられ、ポート 1 と 2 の両方を使用します。有効なインデックスを持つ LEA はポート 1 で実行されます。

MEC は固有の MEC RSV を保持しており、ポート 0 と 1 を経由するすべてのロードとストアのスケジューリングを行います。ロードとストア命令は、インオーダーまたはアウトオブオーダーでアドレス生成フェーズを通過します。アウトオブオーダーでアドレス生成のスケジューリングが可能である場合、メモリー実行パイプラインはロードバッファーとストアバッファーを使用してアドレス生成パイプラインから分離されます。

アウトオブオーダー実行では、ロードが未知のストアに先行することができるため、未知のストアとロードに依存関係があると、メモリー順序の問題とパイプライン・フラッシュを引き起こす可能性があります。Goldmont Plus† マイクロアーキテクチャーでサポートされるメモリー・ディスアンビゲーション (一義化) は、ロード実行の潜在的な問題を追跡して最小化します。問題となったメモリー操作 (μTLB ミスや利用できないリソースなど) は、再実行のためロードやストアバッファーに戻されます。後続の命令をすべてストールする代わりに、(問題が発生していない) より新しい命令の実行を継続できます。問題が解決されると、問題を引き起こした命令はロード/ストアバッファーから再発行 (状況によっては、リタイアメントで再発行) されます。ロードミスが、データキャッシュが非ブロッキングであることによる問題と考えられる場合、ライトコンバイン・バッファー (WCB) を使用して複数の未処理のミスを吸収することができます。

表 F-7 MEC リソースの比較

MEC Resource	Goldmont Microarchitecture	Silvermont Microarchitecture
L1 Data Cache	24KB	24 KB
uTLB	32 entries	32 entries
DTLB (4KB page)	512 entries	128 entries
DTLB (2M/4M page)	32 entries	16 entries
Load-use Latency	3 cycles	3 cycles
Pipeline	1x load + 1x store	1x share by load/store
AGEN	Out-of-order	In order
WCBs	8	8
Addressing	39-bit physical, 48-bit linear	36-bit physical, 48-bit linear

F.7 Goldmont Plus† マイクロアーキテクチャー

Goldmont Plus† マイクロアーキテクチャーは、Goldmont† マイクロアーキテクチャー (F.6 節を参照) の成功を基に、次の拡張を提供します。

- 前世代の Intel Atom® プロセッサーに比べ、バックエンド・パイプラインが 4 ワイド割り当てと 4 ワイドリタイアに拡張されていますが、3 ワイドフェッチとデコード・パイプラインを維持します。
- 拡張分岐予測ユニット
- 64KB の共有第 2 レベル事前デコードキャッシュ (Goldmont† マイクロアーキテクチャーでは 16KB)
- 広いアウトオブオーダー・ウィンドウをサポートする、大きなリザベーション・ステーションと ROB エントリー
- さらに広い整数実行ユニット高速な分岐のリダイレクションをサポートする新しい専用 JEU ポート
- 高速なスカラー/パックド単精度、倍精度、および拡張精度の浮動小数点除算を実現する Radix-1024 浮動小数点除算器
- インテル® AES-NI 命令のレイテンシーとスループットを改善
- 大きなロードとストア・バッファー・レジスターからデータをストアするストア・ロード・フォワーディングのレイテンシーを改善
- 命令およびデータ向けの共有第 2 レベル TLB ページングキャッシュの拡張 (PxE/ePxE キャッシュ)
- 最大 4MB L2 キャッシュを 4 コアで共有するモジュラーシステム設計
- 新たにプロセッサー ID 読みとり (RDP) 命令をサポート

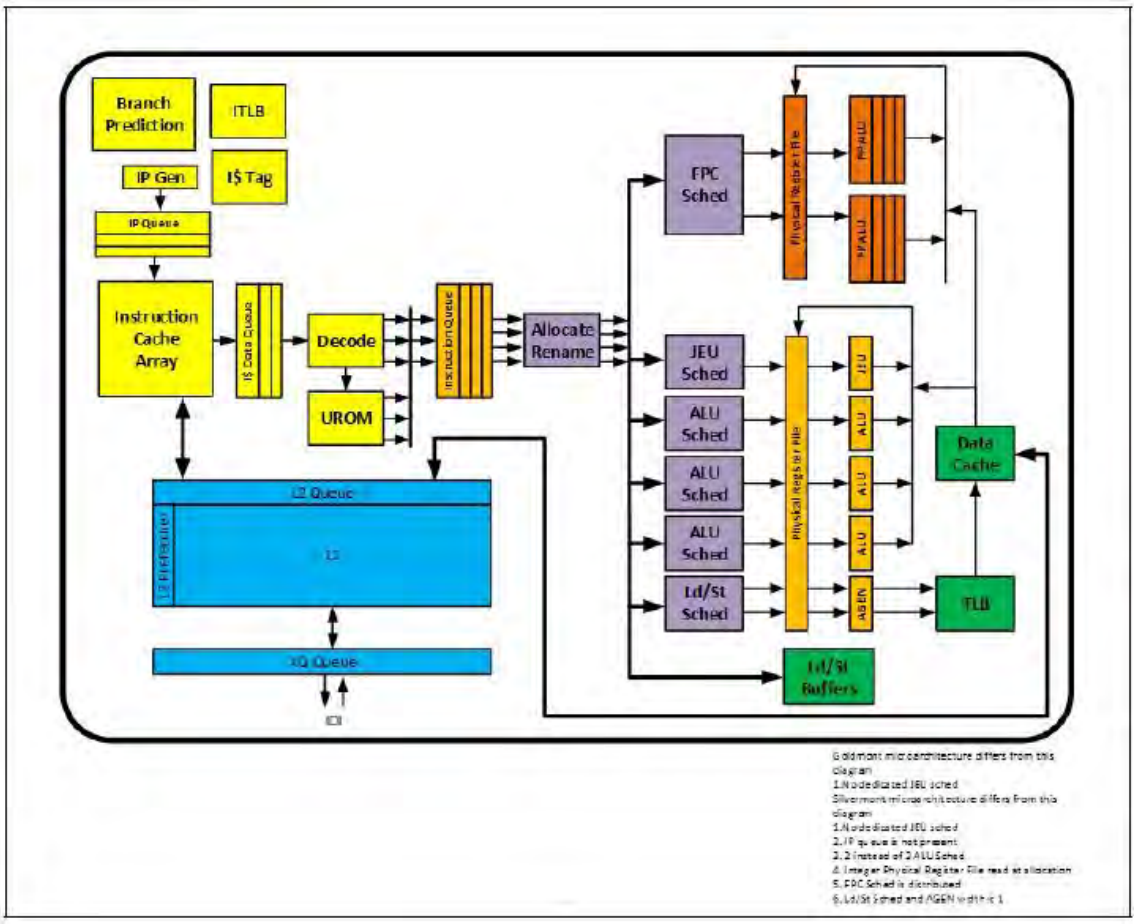


図 F-4 Goldmont Plus† マイクロアーキテクチャーの CPU コア・パイプラインの機能

Goldmont Plus† マイクロアーキテクチャーのフロントエンド・クラスター (FEC) では、Goldmont† マイクロアーキテクチャーの FEC に対し多くの拡張が行われています。表 F-8 にそれらの拡張をまとめています。

表 F-8 フロントエンド・クラスター機能の比較

機能	**Goldmont Plus† マイクロアーキテクチャー**	**Goldmont† マイクロアーキテクチャー**
デコーダー数	3	3
デコーダーの最大スループット	サイクルごとに 20 バイト	サイクルごとに 20 バイト
フェッチと命令キャッシュ・パイプライン	分離	分離
命令 TLB (ITLB)	48 エントリー、ラージページをサポート	48 エントリー、ラージページをサポート
2 次 ITLB キャッシュ	DTLB と共有	
分岐予測ミスのペナルティー	13 サイクル (特定の Jcc では 12 サイクル)	12 サイクル
L2 事前デコードキャッシュ	64K	16K

FEC は、アロケーション、リネーミングおよびリタイアメント (ARR) クラスターを介して OOO 実行エンジンに接続されています。uop のスケジューリングは、異なるクラスター (IEC、FPC、MEC) にまたがって分散されたリザベーション・ステーションによって扱われます。それぞれのクラスターは、ARR から複数の uop を受け取るため固有のリザベーション・ステーションを保持しています。

表 F-9 は、Goldmont Plus† マイクロアーキテクチャーと Goldmont† マイクロアーキテクチャーのアウトオブオーダーの特徴を比較したものです。

表 F-9 uop をスケジューリングする際の分散リザベーション・ステーションの比較

クラスター	Goldmont Plus† マイクロアーキテクチャー	Goldmont† マイクロアーキテクチャー
IEC リザベーション	4x 各ポートに分散	3x 各ポートに分散
	それぞれの IEC RSV 内と IEC と FPC、MEC をまたがるアウトオブオーダー	それぞれの IEC RSV 内と IEC と FPC、MEC をまたがるアウトオブオーダー
FPC リザベーション	1x ポート 0 と 1 へ統合	1x ポート 0 と 1 へ統合
	FPC RSV 内と IEC と MEC をまたがるアウトオブオーダー	FPC RSV 内と IEC と MEC をまたがるアウトオブオーダー
MEC リザベーション	1x ポート 0 と 1 へ統合	1x ポート 0 と 1 へ統合
	MEC RSV 内と IEC と FPC をまたがるアウトオブオーダー	MEC RSV 内と IEC と FPC をまたがるアウトオブオーダー

メモリーを参照して整数/FP リソースを必要とする命令は、メモリー uop が MEC クラスターへ送られ、整数/FP uop が IEC/FPC クラスターへ送られます。そしてリソースが利用可能になると、表 F-9 で示すヒューリスティックに従ってアウトオブオーダー実行を開始できます。表 F-10 は、それぞれのクラスターに対するポートと実行ユニットのマッピングを示します。

表 F-10 Goldmont Plus† マイクロアーキテクチャーの機能ユニットの割り当て

クラスター	ポート 0	ポート 1	ポート 2	ポート 3
IEC	ALU0, シフト/ローテート、LEA インデックスなし、F2I、変換/cmp、ストアデータ	ALU1、ビット処理、IMUL、IDIV、POPCNT、CRC32、LEA、I2F、ストアデータ	ALU2、LEA[1]、I2F、フラグマージ	JEU
FPC	SIMD ALU, SIMD シフト/シャッフル、SIMD 乗算、STTNI/AESNI/ PCLMULQDQ/SHA ; FP_乗算、変換、F2I 変換	SIMD ALU、SIMD シャッフル、FP_加算、F2I 比較		
MEC RS	ロードアドレス	ストアアドレス		

注意:

1. インデックスなしの LEA は、ポート 0、1 もしくは 2 で実行できます。有効なインデックスとディスプレースメントを持つ LEA は複数のマイクロオペレーション (uop) に分けられ、ポート 1 と 2 の両方を使用します。有効なインデックスを持つ LEA はポート 1 で実行されます。

MEC は固有の MEC RSV を保持しており、ポート 0 と 1 を経由するすべてのロードとストアのスケジューリングを行います。ロードとストア命令は、インオーダーまたはアウトオブオーダーでアドレス生成フェーズを通過します。アウトオブオーダーでアドレス生成のスケジューリングが可能である場合、メモリー実行パイプラインはロードバッファーとストアバッファーを使用してアドレス生成パイプラインから分離されます。

アウトオブオーダー実行では、ロードが未知のストアに先行することができるため、未知のストアとロードに依存関係があると、メモリー順序の問題とパイプライン・フラッシュを引き起こす可能性があります。Goldmont Plus† マイクロアーキテクチャーでサポートされるメモリー・ディスアンビゲーション (一義化) は、ロード実行の潜在的な問題を追跡して最小化します。

問題となったメモリー操作 (μTLB ミスや利用できないリソースなど) は、再実行のためロードやストアバッファーに戻されます。後続の命令をすべてストールする代わりに、(問題が発生していない) より新しい命令の実行を継続できます。問題が解決されると、問題を引き起こした命令はロード/ストアバッファーから再発行 (状況によっては、リタイアメントで再発行) されます。ロードミスが、データキャッシュが非ブロッキングであることによる問題と考えられる場合、ライトコンバイン・バッファー (WCB) を使用して複数の未処理のミスを吸収することができます。

Goldmont Plus† マイクロアーキテクチャーは、データと命令の両方の変換をサポートする第 2 レベル TLB を搭載しています (Goldmont† マイクロアーキテクチャーでは、第 2 レベル TLB はデータのみをサポート)。

F.8 コーディングの推奨事項

第 3 章「一般的な最適化ガイドライン」で説明されている一般的なコーディングの推奨事項は、Intel Atom® マイクロアーキテクチャーにも適用できます。この章の残りでは、一般的な推奨事項の補足と Intel Atom® マイクロアーキテクチャー固有の手法について説明します。

F.8.1 フロントエンドの最適化

F.8.1.1 命令デコーダー

一部の IA 命令は、複雑なタスクを実行するため複数のマイクロオペレーション (uop) にデコードされるマイクロコード・シーケンサー ROM (MSROM) のルックアップが必要になります。MSROM ルックアップが必要な命令については、F.9 節のレイテンシー/スループットの表を参照してください。

Silvermont† マイクロアーキテクチャーでは、前の世代よりも MSROM のルックアップが大幅に改善されましたが、Goldmont Plus† と Goldmont† マイクロアーキテクチャーでは、MSROM を必要とする命令の数は、Silvermont† マイクロアーキテクチャーに比べ非常に少なくなっています。マイクロコード・フローは、できるだけ回避することが推奨されます。表 F-11 に、MSROM からデコードされる命令を置き換えることができる非 MSROM 命令のシーケンスを示します。

表 F-11 MSROM 命令の代替

Instruction from MSROM	Recommendation for Silvermont	Recommendation for Goldmont Plus and Goldmont
CALL m16/m32/m64	Load + CALL reg	Load + CALL reg
PUSH m16/m32/m64	Load + PUSH reg	Use as is (non MSROM)
LEAVE	No recommended replacement	Use as is (non MSROM)
FLD/FST/FSTP m80fp	No recommended replacement	Use as is (non MSROM)
FCOM+FNSTSW	FCOMI	FCOMI
(I)MUL r/m16 (Result DX:AX)	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32
(I)MUL r/m32 (Result EDX:EAX)	Use (I)MUL r32, r/m32 if extended precision not required, or (I)MUL r64, r/m64	Use as is (non MSROM)
(I)MUL r/m64 (Result RDX:RAX)	Use (I)MUL r64, r/m64 if extended precision not required	Use as is (non MSROM)
PEXTRB/D/Q	No recommended replacement	Use as is (non MSROM)
PMULLD	No recommended replacement	Use as is (non MSROM)

チューニングの推奨 1: perfmon カウンター MS_DECODED.MS_ENTRY を使用して、MSROM が必要な命令の数を特定します (すべてのアシストとフォルトが含まれる)。

アセンブリー/コンパイラー・コーディング規則 1 (影響 M、一般性 M): 命令長をできるだけ短くすることで、プリデコード・ビットを効率良く再利用できます。

プリデコード・ビットが正しくないと、デコードのスループットが 3 サイクルごとに 1 命令に減少するため、命令キャシュのエイリアシングとスラッシングを避けます。

チューニングの推奨 2: perfmon カウンター DECODE_RESTRICTION.PREDECODE_WRONG を使用して、プリデコード・ビットが正しくないことによるデコードの制限によって命令デコードのスループットが低下した回数を調査します。

F.8.1.2 フロントエンドの IPC が高い場合の考慮事項

一般に、サイクルあたりの命令数 (IPC) が高く (>1 に) なるまで、フロントエンドがパフォーマンスを制限することはありません。

デコーダーでサイクルあたり 2 命令を処理するには、次のデコードの規則に従う必要があります。

- MSROM 命令はできるだけ回避します。典型的な例は CALL near の間接メモリー形式です。メモリーバージョンの PUSH と CALL の代わりに、レジスターへロードし、レジスターバージョンの PUSH と CALL を実行します。
- サイクルごとにデコードできる命令バイト長は、マイクロアーキテクチャーによって異なります。
  - Silvermont† マイクロアーキテクチャーでは、一緒にデコードされる命令ペアの長さの合計は 16 バイト未満に、最初の命令の長さは 8 バイト以下にします。例えば、命令が 8 バイトを超えるとデコーダー 0 では、サイクルあたり 1 命令しかデコードできません。
  - Goldmont† マイクロアーキテクチャー以降では、アライメントに依存してサイクルあたり最大 20 バイトです (例えば、3 つの連続した命令の最初の命令が 4 バイト境界でアライメントされ、3 つの命令シーケンスがデコードの制限を満たす場合)。命令長が 8 バイトを超える場合、デコーダー 0 やサイクルごとに 1 命令に限定されません。

- 複数のプリフィクスを持つ命令は、デコーダーのスループットを制限します。プリフィクスとエスケープの合計バイト数が制限に当てはまります。命令のプリフィクス + エスケープは、以下のマイクロアーキテクチャーの制限を超えないようにします。
  - Silvermont† マイクロアーキテクチャー: 3 バイトを超えるとペナルティーが発生します。
  - Goldmont† マイクロアーキテクチャー以降: 4 バイトを超えるとペナルティーが発生します。したがって、上位 8 つのレジスターをアクセスするインテル® SSE4 や AES 命令には、ペナルティーが科せられません。
  - Silvermont† と Goldmont† マイクロアーキテクチャーでは、デコーダー 0 のみがプリフィクス/エスケープ・バイトの制限を超えた命令をデコードできます。

- 各サイクルでデコード可能な分岐の最大数は、Silvermont† マイクロアーキテクチャーでは 1、Goldmont† マイクロアーキテクチャーでは 2 です。条件分岐を避けることでリステアを防ぎます。

前世代と異なり、Silvermont† マイクロアーキテクチャーでは、同じサイクルで 2 つの x87 命令をデコードしても 2 サイクルのペナルティーは発生しません。分岐デコーダーの制限も緩和されています。前世代の Intel Atom® プロセッサーでは、デコーダー 0 で条件分岐または間接分岐の次の命令のデコードに 2 サイクルのペナルティーが発生しました。

Silvermont† マイクロアーキテクチャーでは、デコーダー 0 で条件分岐または間接分岐の次の命令をペナルティーなしでデコードできます。ただし、(デコーダー 1 にある) 次の命令も分岐である場合、その分岐命令で 3 サイクルのペナルティーが発生します。

Goldmont† マイクロアーキテクチャー以降では、不成立 (not taken) として予測された分岐をデコーダー 0 またはデコーダー 1 でデコードでき、さらに 3 サイクルのリステア・ペナルティーなしで他の分岐をデコーダー 2 でデコードできます。しかし、不成立として予測された分岐がデコーダー 0 と 1 に 2 つある場合、デコーダー 1 にある 2 番目の分岐は 3 サイクルのペナルティーを被ります。

すべての世代の Intel Atom® プロセッサーにおいて、分岐ターゲットが成立すると予測された条件分岐や無条件分岐である場合、1 サイクルのバブルを挟んでデコードされます。

アセンブリー/コンパイラー・コーディング規則 2 (影響 MH、一般性 H): サイクルあたり 2 命令のスループットを達成するため、次の命令の使用はできるだけ控えます: (i) MSROM を使用する命令、(ii) プリフィクス + エスケープが制限を超える命令、(iii) 長さが 8 バイトを超える命令、または (iv) 連続する分岐命令。

例えば、通常 Silvermont† と Goldmont† マイクロアーキテクチャー以降では、3 バイトのプリフィクスとエスケープを持つ下位の 8 つのレジスターをアクセスする命令をデコードできます。次に例を示します。

PCLMULQDQ 66 0F 3A 44 C7 01 pclmulqdq xmm0, xmm7, 0x1

XMM レジスターの上位いずれか (XMM8-15) が参照される場合は、追加の REX プリフィクスも必要になります。その結果、Goldmont† マイクロアーキテクチャー以降では通常どおりにデコードされますが、Silvermont† マイクロアーキテクチャーではデコードのペナルティーが科せられます。次に例を示します。

PCLMULQDQ 66 41 0F 3A 44 C0 01 pclmulqdq xmm0, xmm8, 0x1

(66 と 0F 3A の間に REX バイト 41 が追加されていることが分かります)。

この 4 つ目のプリフィクスにより、デコードで 3 サイクルのペナルティーが生じます。さらに、このプリフィクスは命令をデコーダー 0 でデコードすることを強制します。命令がデコーダー 1 で開始された場合、デコーダー 0 へ切り替えるのに 3 サイクルかかり、ペナルティーはさらに大きくなります (デコーダーのペナルティーは合計 6 サイクルになります)。そのため、ハイパフォーマンスなアセンブリーを記述するには、これらを考慮することを推奨します。これらのケースが頻繁に発生しなければ、成立分岐ターゲットや MS エントリーポイントによってあらかじめデコーダー 0 へアライメントしたほうが良いでしょう。NOP 命令は、パイプラインのほかのリソースを消費するため、NOP の挿入は最終手段として行うべきです。MS エントリーポイントも、デコーダー 1 で開始した場合 3 サイクルのペナルティーが発生するため、同様のアライメントが必要です。プリフィクス/エスケープ長とリステアの制限に関連するペナルティーは、Silvermont† と Goldmont† マイクロアーキテクチャー以降に適用されます。

表 F-12 は、Silvermont†、Goldmont† および Goldmont Plus† マイクロアーキテクチャーのデコーダーの能力の違いを示しています。

表 F-12 デコーダーの能力の比較

	Goldmont Plus and Goldmont Microarchitecture	Silvermont Microarchitecture
Width	3	2
Max Throughput	20 bytes per cycle (1st instr. aligned to 4B boundary and decoder 1 and 2 restrictions)	16 bytes per cycle (1st instr. <= 8 bytes))
Prefix/Escape Limit	4 bytes	3 bytes
Branch	2	1

F.8.1.3 4GB 境界を超える分岐

フロントエンドにおけるもう 1 つの重要なパフォーマンスの考慮事項は分岐予測です。64 ビット・アプリケーションでは、分岐ターゲットが 4GB 以上離れている場合、分岐予測のパフォーマンスに悪影響を与えます。これは、アプリケーションが共有ライブラリーと分離されている場合に発生する可能性があります。新しい glibc のバージョン (2.23 以降) では、この問題を避けるため共有ライブラリーを初めの 2GB に配置できます。環境変数 LD_PREFER_MAP_32BIT_EXEC に 1 を設定します。プログラマーは、コードの局所性を改善するため静的にビルドすることもできます。LTO によるビルドでは、パフォーマンスさらに向上させなければなりません。

F.8.1.4 ループアンロールおよびループストリーム検出器

Silvermont† と Goldmont† マイクロアーキテクチャー以降は、バックエンドにデコード済みのマイクロオペレーション (uop) を提供するループストリーム検出器 (LSD) を備えています。これは、パフォーマンスと消費電力において利点をもたらします。LSD を利用することで、プリフィクス + エスケープのバイト数や命令の長さなどのフロントエンドの制限が排除されます。

ループのオーバーヘッドを減らし、独立したループ反復の作業量を増やす 1 つの方法として、ソフトウェアによるループアンロールが利用できます。ただし、ループアンロールは利点をもたらす一方、パフォーマンスを低下させる恐れもあるため、慎重に使用しなければなりません。パフォーマンスの低下は、コードサイズが大きくなったり、BTB およびレジスターの負荷が増えることで生じます。また、ループアンロールにより、ループサイズが LSD の上限を超える可能性があるため、ループが LSD に収まるようにループサイズを、Goldmont† マイクロアーキテクチャー以降の 3 ワイドのデコーダーでは 27 命令未満に、Silvermont† マイクロアーキテクチャーでは 28 命令未満に抑える対策が必要です。ループサイズが LSD サイズ以下となるように注意する必要があります。

ユーザー/ソース・コーディング規則 1 (影響 M、一般性 M): 反復数の多いショートループでループアンロールを利用する場合は、反復あたりの命令数を 28 未満に抑えます。

チューニングの推奨 3: perfmon カウンター BACLEARS.ANY を使用して、ループアンロールにより負荷が大きくなりすぎていないかを確認します。また、perfmon カウンター ICACHE.MISSES で、ループアンロールにより命令フットプリントに大きな悪影響が生じていないかを確認できます。

F.8.1.5 コードとデータの混在

Intel Atom® プロセッサーは、コードとデータが異なるページにある場合に最適に動作します。ソフトウェアは、フォルス SMC 条件の発生を避けるため同じページ内でコードとデータを共有しないようにしなければなりません。この推奨事項はすべてのページサイズに適用されます。

F.8.2 実行コアの最適化

F.8.2.1 スケジューリング

Silvermont† マイクロアーキテクチャーでは、整数命令でアウトオブオーダー実行が導入されているため、前世代と比べると命令の実行順序が変動する可能性があります。FP 命令には専用のリザベーション・ステーションが 2 つありますが、互いにインオーダーで実行されます。メモリー命令もインオーダーで発行されますが、修復キュー (Rehab Queue) が追加されているため、アウトオブオーダーで完了することができ、メモリーシステムの遅延によって実行が妨げられることはありません。

Goldmont† マイクロアーキテクチャー以降は、IEC、FPC、そして ME パイプライン全体での完全なアウトオブオーダー実行を特徴としており、これは 3 ポートの IEC、128 ビットの FPC データパス、専用のロードアドレスおよびストアアドレス・パイプラインなど広範囲な強化により達成されます。

チューニングの推奨 4: perfmon カウンター uop_NOT_DELIVERED.ANY (Silvermont† マイクロアーキテクチャーでは NO_ALLOC_CYCLE.ANY) を使用すると、バックエンドのパフォーマンス・ボトルネックが分かります。このカウンター値には、メモリーシステムの遅延や実行の遅延などが含まれます。

F.8.2.2 アドレス生成

前世代の Intel Atom® マイクロアーキテクチャーのアドレス生成の制限は、Silvermont† マイクロアーキテクチャーでは解決されています。そのため、Silvermont† マイクロアーキテクチャー以降では、LEA 命令と ADD 命令のどちらを使ってアドレスを生成してもその効果は同じです。

経験則上、SCALE を使用するか、有効なインデックスやディスプレースメントを持つ LEA を非破壊デスティネーション (特にスタックオフセット) に使用します。そうでない場合は ADD を使用すると良いでしょう。

F.8.2.3 FP 乗算-加算-ストアの実行

Goldmont† マイクロアーキテクチャー以降は、統合された FPC リザベーション・ステーションにより、Silvermon† マイクロアーキテクチャーで FPC uop のインオーダー・スケジューリングのポート内の依存性によるパフォーマンスの問題を排除します。次の段落と例 F-7 はこの問題を示します。

Silvermont† マイクロアーキテクチャーでは、異なるポートで実行する FP 算術命令は互いにアウトオブオーダーで実行できます。そのため、アンロールされたループで乗算結果を加算命令に供給し、その結果をストアする場合、ループの最後にストア命令をまとめることでパフォーマンスが向上します。この方法では、乗算命令と加算命令の実行をオーバーラップさせることができます。例 F-7 について考えてみます。

例 F-7 乗算-ストアポートの競合によってアンロールされたループはインオーダーで実行

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
mulps, xmm1, xmm1	EX1	EX2	EX3	EX4	EX5													
addps xmm1, xmm1						EX1	EX2	EX3										
movaps mem, xmm1									EX1									
mulps, xmm2, xmm2										EX1	EX2	EX3	EX4	EX5				
addps xmm2, xmm2															EX1	EX2	EX3	
movaps mem, xmm2																		EX1

データの依存性により、加算命令は、対応する乗算命令が実行されるまで実行を開始できません。乗算命令とストア命令は、同じポートを使用するため、プログラム順に実行しなければなりません。つまり、2 つ目の乗算命令は 1 つ目の乗算命令および加算命令と依存性がないにもかかわらず、実行を開始できません。次のように、ループの最後にストア命令をグループ化することで、2 つ目の乗算命令を 1 つ目の乗算命令と並列に実行できます (乗算命令をオーバーラップさせると 1 サイクルのバブルが発生する)。

例 F-8 ストア命令をグループ化することでバブルを排除し IPC を向上

Instruction	1	2	3	4	5	6	7	8	9	10	11
mulps, xmm1, xmm1	EX1	EX2	EX3	EX4	EX5						
addps xmm1, xmm1						EX1	EX2	EX3			
mulps, xmm2, xmm2		bubble	EX1	EX2	EX3	EX4	EX5				
addps xmm2, xmm2								EX1	EX2	EX3	
movaps mem, xmm1									EX1		
movaps mem, xmm2											EX1

F.8.2.4 整数乗算の実行

Silvermont† マイクロアーキテクチャー以降には専用の整数乗算器が備わっており、一般的に使用される形式の整数乗算のフローを加速します。表 F-13 に MSROM を使用しない各種 MUL/IMUL 命令形式のレイテンシーとマイクロオペレーション (uop) の数を示します。マイクロコードを利用する乗算形式は回避すべきです。

表 F-13 整数乗算命令のレイテンシー

Integer Multiply Operations	Output	Goldmont Plus and Goldmont Latency	Silvermont Latency
imul/mul r/m8	16	4u	5u
imul/mul r/m16	32	4u	5u
imul/mul r/32	64	3	4u
imul/mul r/m64	128	5	7u
imul/mul r16, r/m16; r16, r/m16, imm	16	4u	4u
imul/mul r32, r/m32; r32, r/m32, imm	32	3	3
imul/mul r64, r/m64; r64, r/m64, imm8	64	5	5
u: ucode flow from MSROM			

F.8.2.5 ゼロイディオム

XOR/PXOR/XORPS/XORPD 命令は、ソースレジスターとデスティネーションに同じレジスターを指定して (例: XOR eax, eax)、レジスター値をゼロに設定するのによく使用されます。

同等の命令として MOV eax, 0x0 命令がありますが、MOV エンコードのほうが XOR よりもコードバイトが大きくなるため、コンパイラーにとっては MOV よりもこれらの命令のほうが好まれます。

Silvermont† マイクロアーキテクチャー以降には、これらのケースを認識し、アーキテクチャーのレジスターファイルでどちらのソースも有効としてマークする特別なハードウェア・サポートが備わっています。どのような値であってもそれ自身と XOR することでゼロに設定できるため、これにより XOR を高速に実行できます。

このロジックは、PXOR、XORPS、XORPD でもサポートされます。

Silvermont† マイクロアーキテクチャーでは、REX.W を使用する 64 ビット汎用オペランドのゼロイディオムに遅延が生じます。ゼロイディオムは、XMM8 - XMM15 または、REX.W なしの上位 8 つの汎用レジスターでサポートされます。そのため、r8 をゼロに設定するには、XOR r8, r8 ではなく XOR r8d, r8d と指定します。

Goldmont† マイクロアーキテクチャー以降では、64 ビット・オペランドのゼロイディオムをサポートします。

F.8.2.6 慣用的な NOP

NOP 命令は、パディングやアライメントの目的で使用されることがあります。Goldmont† マイクロアーキテクチャー以降は、NOP をリザベーション・ステーションへ割り当てることなく完了できるハードウェアのサポートを備えています。これは、実行リソースと帯域幅を節約します。しかし、リタイアメントのリソースはまだ必要です。

F.8.2.7 ムーブの排除 (Move Elimination) と ESP の折りたたみ (Folding)

ムーブの排除は、Goldmont† マイクロアーキテクチャー以降でサポートされます。ムーブの排除が有効である場合、それらの命令は高いスループットに加え、0 サイクルのレイテンシーで実行できます。特に、32 ビットと 64 ビットの

オペランドサイズを持つ MOV と、XMM を使用する MOVAPS/MOVAPD/MOVDQA/MOVDQU/MOVUPS/MOVUPD 命令は、ムーブの排除が有効である場合 0.33 サイクルのスループットを持ちます。MOVSX と MOVZX は、ムーブの排除をサポートしていません。

PUSH/POP/CALL/RET を使用するスタック操作は、Goldmont† マイクロアーキテクチャー以降では Silvermont† マイクロアーキテクチャーよりも効率的です。Goldmont† マイクロアーキテクチャー以降では、スタック・ポインター・アドレスの計算にアロケーションと実行リソースを消費しません。さらに、PUSH/POP のスループットは、サイクルあたり 1 から 3 クロックに増加しています。

F.8.2.8 スタック操作命令

間接メモリー形式の CALL m16/m32/m64 は、MSROM からの uop フローにデコードされます。レジスターで指定されるターゲットを持つ間接 CALL は、遅延を回避できます。そのため、ターゲットアドレスのレジスターへのロードに続いて、レジスターオペランドを介して間接 CALL を行うことが推奨されます。

Goldmont† マイクロアーキテクチャー以降では、PUSH m16/m32/m64 のデコードに MSROM を必要としません。これは、LEAVE 命令にも当てはまります。Silvermont† マイクロアーキテクチャーでは、PUSH m16/m32/m64 と LEAVE は デコードに MSROM を必要とします。

F.8.2.9 フラグの利用

多くの命令には、フラグレジスターに格納される暗黙のデータがあります。これらのデータは、条件移動 (CMOVS)、分岐、さまざまな論理/算術演算 (RCL など) といった幅広い命令で利用されます。分岐条件としてよく使用される命令に比較命令 (CMP) があります。CMP 命令に依存する分岐は次のサイクルで実行できます。ADD 命令や SUB 命令に依存する分岐でも同様のことが言えます。

INC 命令と DEC 命令は一部のフラグのみ設定するため、フラグをマージする追加のマイクロオペレーション (uop) が必要になります。そのため、INC 命令や DEC 命令に依存する分岐には 1 サイクルのペナルティーが伴います。このペナルティーは、INC 命令や DEC 命令に直接依存する分岐にのみ適用されます。

アセンブリー/コンパイラー・コーディング規則 3 (影響 M、一般性 M): 分岐条件には、INC/DEC 命令ではなく可能な限り CMP/ADD/SUB 命令を使用します。

F.8.2.10 SIMD 浮動小数点と x87 命令

Silvermont† マイクロアーキテクチャーでは、SIMD FP 実行ユニットのサブセットのみが、128 ビット幅のデータパスで実装されています。Goldmont† マイクロアーキテクチャー以降では、完全な SIMD FP ユニットが 128 ビット・データパスで実装されています。Goldmont† マイクロアーキテクチャー以降を Silvermont† と比較すると、一般にパックド SIMD 命令のレイテンシーは 1 サイクル少なく、スループットは倍で実行されます。

特に、MULPD 命令のレイテンシーは 7 サイクルから 4 サイクルに短縮され、スループットは 4 サイクルごとから 1 サイクルごとで 4 倍を達成します。

Goldmont† マイクロアーキテクチャーではまた、x87 拡張精度のロードとストア、FLD m80fp と FST/FSTP m80fp のレイテンシーとスループットが改善されています。詳細は表 F-19 を参照してください。

Goldmont Plus† マイクロアーキテクチャーでは、浮動小数点除算器が Radix-1024 ベースを使用した設計に改良され、浮動小数点除算と平方根のレイテンシーと帯域幅が大幅に改善されました。詳細は表 F-19 を参照してください。

F.8.2.11 SIMD 整数命令

Silvermont† マイクロアーキテクチャーでは、比較的小さな SIMD 整数命令のサブセットは、サイクルあたり 2 命令のスループットで実行されます。Goldmont† マイクロアーキテクチャー以降の場合、より多くの SIMD 整数命令をサイクルあたり 2 命令のレートで実行を完了できます。

Goldmont† マイクロアーキテクチャー以降におけるレイテンシーと/または、スループットの改善には、1 つのポートのみで実行されるその他の SIMD 整数命令も含まれます。例えば、PMULLD は Silvermont† マイクロアーキテクチャーでは、11 サイクルのレイテンシーと 11 サイクルごとに 1 つのスループットを提供していました。Goldmont† マイクロアーキテクチャー以降では、5 サイクルのレイテンシーと 2 サイクルごとに 1 つのスループットに改善されています。

一般に、SIMD 整数乗算器のハードウェアは、Silvermont† マイクロアーキテクチャーよりもかなり高速化 (4 サイクルのレイテンシー) され、高いスループット (1 サイクル) です。さらに、PADDQ/PSUBQ 命令は、Silvermont† マイクロアーキテクチャーでは 4 サイクルのレイテンシーと 4 サイクルごとのスループットであったのに対し、Goldmont† マイクロアーキテクチャー以降では、2 サイクルのレイテンシーと 2 サイクルごとのスループットを提供します。また、PSHUFB 命令の Goldmont† マイクロアーキテクチャー以降でのレイテンシーとスループットは 1 サイクルですが、Silvermont† では 5 サイクルのレイテンシーと 5 サイクルごとのスループットです。詳細は表 19-17 を参照してください。

F.8.2.12 ベクトル化の注意事項

Silvermont† マイクロアーキテクチャーでは、高いスループットを実装する SIMD 実行ユニットの可用性と、MSROM から長い uop フローへのデコードが要求される SIMD 命令によって、有益なベクトル化の機会が制限される可能性があります。

Goldmont† マイクロアーキテクチャー以降は、直接的なプログラミングと同様にコンパイラーが、各種 SIMD 命令のレイテンシーとスループットの改善によるベクトル化の利益を得ることを可能にします。

アセンブリー/コンパイラー・コーディング規則 4 (影響 M、一般性 M): コードのベクトル化には MSROM 命令の使用を回避します。

F.8.2.13 その他の SIMD 命令

Silvermont† マイクロアーキテクチャーは、ブロック暗号化/復号向けの AES や AES-GCM などの各種暗号化アルゴリズムのパフォーマンスを加速するインテル® AES-NI と PCLMULQDQ 命令をサポートします。

Goldmont† マイクロアーキテクチャー以降では、実行ハードウェアの実行レイテンシー、デコードのスループットが改善されました。例えば、PCLMULQDQ 命令は、Silvermont† マイクロアーキテクチャーでは 10 サイクルのレイテンシーと 10 サイクルごとのスループットであったのに対し、Goldmont† マイクロアーキテクチャー以降では、6 サイクルのレイテンシーと 4 サイクルごとのスループットを提供します。

さらに、Goldmont† マイクロアーキテクチャー以降は、SHA1 と SHA256 などの安全なハッシュ・アルゴリズムのパフォーマンスを加速する SHANI 命令をサポートします。安全なハッシュ・アルゴリズムと SHANI の詳細については次のサイトを参照してください。

https://software.intel.com/en-us/articles/intel-sha-extensions (英語)

インテル® SHA 拡張を使用した実装の例と参考資料は、以下のサイトでご覧いただけます。

https://software.intel.com/en-us/articles/intel-sha-extensions-implementations (英語)

F.8.2.14 命令の選択

表 F-14 に、Silvermont† マイクロアーキテクチャーの浮動小数点操作と SIMD 整数操作のレイテンシーを示します。「スループット」は、利用可能な実行ユニットで実行が完了できる命令ごとのサイクル数を表しています (例えば、4 は 4 サイクルごとに同じ命令を開始できることを示し、0.33 は 3 つの同じ命令が各サイクルで実行を完了できることを示します)。

表 F-14 浮動小数点と SIMD 整数のレイテンシー

	Goldmont Plus		Goldmont		Silvermont	
	Latency	Through put	Latency	Through put	Latency	Throughp ut
SIMD integer ALU						
128-bit ALU/logical/move	1	0.5	1	0.5	1	0.5
64-bit ALU/logical/move	1	0.5	1	0.5	1	0.5
SIMD integer shift						
128-bit	1	0.5	1	0.5	1	1
64-bit	1	0.5	1	0.5	1	1
SIMD shuffle						
128-bit	1	0.5	1	0.5	1	1
64-bit	1	0.5	1	0.5	1	1
SIMD integer multiplier						
128-bit	4	1	4	1	5	2
64-bit	4	1	4	1	4	1

	Goldmont Plus		Goldmont		Silvermont	
	Latency	Throughput	Latency	Throughput	Latency	Throughput
FP Adder						
x87 (fadd)	3	1	3	1	3	1
scalar (addsd, addss)	3	1	3	1	3	1
packed (addpd, addps)	3	1	3	1	4	2
FP Multiplier						
x87 (fmul)	5	2	5	2	5	2
scalar single-precision (mulss)	4	1	4	1	4	1
scalar double-precision (mulsd)	4	1	4	1	5	2
packed single-precision (mulps)	4	1	4	1	5	2
packed double-precision (mulpd)	4	1	4	1	7	4
Converts						
CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTTPD2DQ, CVTPD2PI, CVTPS2DQ	4	1	4	1	5	2
CVTPI2PS, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI	4	1	4	1	4	1
FP Divider						
x87 fdiv (extended-precision)	15	11	39	39	39	39
x87 fdiv (double-precision)	14	10	34	34	34	34
x87 fdiv (single-precision)	11	7	19	19	19	19
scalar single-precision (divss)	11	7	19	18	19	17
scalar double-precision (divsd)	14	10	34	33	34	32
packed single-precision (divps)	16	12	36	35	39	39
packed double-precision (divpd)	22	18	66	65	69	69

インテル® SSE のスカラー単精度命令は、ほとんどの FP 命令よりも 1 サイクル高速であることを覚えておいてください。この表を調べることで、インテル® SSE のパックド倍精度命令はスカラーバージョンと比べて長いレイテンシーと低いスループットであることが分かります。

アセンブリー/コンパイラー・コーディング規則 5 (影響 M、一般性 M): x87 浮動小数点命令よりもインテル® SSE 浮動小数点命令を利用したほうが良いでしょう。

アセンブリー/コンパイラー・コーディング規則 6 (影響 MH、一般性 M): (可能な限り) 例外をマスクして、DAZ フラグと FTZ フラグを設定して実行します。

チューニングの推奨 5: perfmon カウンター MACHINE_CLEARS.FP_ASSIST を使用して、浮動小数点例外がプログラムのパフォーマンスに影響しているかどうかを確認できます。

F.8.2.15 整数除算

Silvermont† マイクロアーキテクチャーでは、整数除算には比較的長く低速なマイクロコード・フローを必要とします。レイテンシーは、入力値とデータサイズによって異なります。Goldmont† マイクロアーキテクチャー以降では、MSROM を使用しない DIV/IDIV の短精度形式向けのハードウェア強化が行われています。高い精度を必要とする DIV/IDIV 形式は MSROM を使用しますが、これもハードウェアの強化により加速されます。表 F-15 と表 F-16 は、除算命令のレイテンシー範囲と MSROM を必要とする命令 ('u' が記されている) を示しています。

表 F-15 符号なし整数除算操作のレイテンシー

	Dividend	Divisor	Quotient	Remainder	Silvermont[u]	Goldmont
DIV r8	AX	r8	AL	AH	25	11-12
DIV r16	DX:AX	r16	AX	DX	26-30	12-17[u]
DIV r32	EDX:EAX	r32	EAX	EDX	26-38	12-25[u]
DIV r64	RDX:RAX	r64	RAX	RDX	38-123	12-41[u]

表 F-16 符号付き整数除算操作のレイテンシー

	Dividend	Divisor	Quotient	Remainder	Silvermont[u]	Goldmont
IDIV r8	AX	r8	AL	AH	34	11-12
IDIV r16	DX:AX	r16	AX	DX	35-40	12-17[u]
IDIV r32	EDX:EAX	r32	EAX	EDX	35-47	12-25[u]
IDIV r64	RDX:RAX	r64	RAX	RDX	49-135	12-41[u]

ユーザー/ソース・コーディング規則 2 (影響 M、一般性 L): 除算は真に必要な場合のみ利用し、最も効率良く実行できるように正しいデータサイズと符号を使用します。

チューニングの推奨 6: perfmon カウンター CYCLES_DIV_BUSY.ANY を使用して、除算がプログラムのボトルネックになっているかどうかを確認できます。

アライメントされている配列からアライメントされていないパックド単精度のグループを取得する場合、MOVUPS よりも PALIGNR が推奨されます。例えば、load A[x+y+3:x+y] について考えてみます。ここで x と y はループ変数です。この場合、(x+y で MOVUPS を使用するよりも) x+y を計算して 4 の倍数に切り下げ、MOVAPS と PALIGNR で要素を取得するほうが適切です。この方法は時間がかかるように見えますが、整数操作は FP 操作と並列に実行できます。また、約 6 サイクルのコストを伴う MOVUPS によるライン分割を回避することもできます。

ユーザー/ソース・コーディング規則 3 (影響 M、一般性 M): パックド単精度要素の取得には PALIGNR を使用します。

F.8.2.16 整数シフト

レジスター (例えば CL) にシフトカウントを設定する整数シフト命令を使用する場合、パイプラインで実行される先行する命令によってカウントレジスターが生成されていると、スケジューリングに 1 サイクルのバブルが発生します。そのため、ソフトカウントを生成する命令は可能な限り手前で実行すべきです。

また、倍精度シフト命令 (SHLD/SHRD) が 64 ビット入力データを処理する場合、長い MSROM フローが必要となります。Silvermont† マイクロアーキテクチャーでは、32 ビットのデスティネーション・レジスターと即値のシフトカウントを持つ SHRD も MSROM からデコードされます。SHLD は MSROM を必要としません。Goldmont† マイクロアーキテクチャー以降では、32 ビットのデスティネーション・レジスターと即値のシフトカウントを持つ SHLD/SHRD は、MSROM からデコードする必要はありません。32 ビットのデスティネーション・メモリー・オペランドや CL シフトカウントを持つ SHLD/SHRD は、Silvermont† と Goldmont† の両方で MSROM からのデコードを必要とします。

F.8.2.17 ポーズ命令

Goldmont† マイクロアーキテクチャー以降では、PAUSE 命令のレイテンシーは Silvermont† マイクロアーキテクチャーと同じですが、スレッド同期プリミティブによって高い節電能力を達成します。

F.8.3 メモリーアクセスの最適化

F.8.3.1 PALIGNR でアライメントされていないメモリーアクセスを軽減

単精度 FP や dword データ配列を使用する場合、配列が 16 バイトにアライメントされていないと 4 つの連続する要素のロードごとにメモリーアクセスが発生する可能性があります。例えば、2 つのインデックス 'i'、'j' を使用する配列 A[i + j] が入れ子になったループにある場合、内部ループで 1 つインクリメントされる実効インデックス "i+j" を使用する 16 バイトのメモリーロードは、4 回のうち 3 回はアライメントされていないアクセスとなります。

これらのアライメントされていないメモリーアクセスは回避できます。配列のベースアドレスが 16 バイトにアライメントされていると仮定すると、オリジナルの "i+j" の残り 4* から得られる imm8 定数 (PALIGNR によって XMM にすでにロードされている 2 つの連続した 16 バイト・チャンクに続く) の 4 倍数である実効インデックスでロードされるべきです。

アセンブリー/コンパイラー・コーディング規則 7 (影響 M、一般性 M): パックド単精度 FP または dword 要素の取得には PALIGNR を使用します。

F.8.3.2 メモリー実行の問題を最小化する

Goldmont† マイクロアーキテクチャー以降では、MEC は完全にアウトオブオーダー実行であり、ロードはアドレス解決されていないストアを先行することができます。ロードが古いストアに依存している場合、ハードウェアがその状況を検出して、ロードとそれ以降の命令を再実行する必要があります。このような再実行が行われると、プログラマーはパフォーマンス・カウンター・イベントを使用して、メモリー実行の問題の原因を評価してその場所を特定できます。

Silvermont† マイクロアーキテクチャーでは、RehabQ には MEC で対処しなければならないいくつかの実行問題があります。この問題には、ロードブロック、ロード/ストア分割、ロック、TLB ミス、不明なアドレス、および過度のストアなどが含まれます。Silvermont† マイクロアーキテクチャーの perfmon カウンター REHABQ は、Silvermont† マイクロアーキテクチャー固有の問題を評価するのに使用できます。

チューニングの推奨 7: perfmon カウンター MACHINE_CLEAR.DISAMBIGUATION を使用して、Goldmont† マイクロアーキテクチャーと前の世代において、古い未知のストアをロードがパスする場合のアプリケーション・パフォーマンスへの影響を評価できます。

F.8.3.3 ストア・フォワーディング

Silvermont† マイクロアーキテクチャー以降では、前世代と比べてストア・フォワーディングが大幅に改善されています。次の条件を満たす場合、先行するストア操作命令から後続のロード命令にデータを転送できます。

- ストア操作とロード操作の開始アドレスが同じである。
- ロード操作の幅がストア操作の幅以下である。
- ストア操作またはロード操作でキャッシュラインの分割が発生しない。

表 F-17 と表 F-18 は、先行するストアのフォワードが成功した場合と、フォワードできない場合の状況を示します。

表 F-17 ストア・フォワーディング条件 (1 バイト・ストアおよび 2 バイト・ストア)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
2	1	F	N														
	2	F	N														

表 F-18 ストア・フォワーディング条件 (4-16 バイト・ストア)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	N	F	N												
	2	F	N	F	N												
	4	F	N	N	N												
8	1	F	N	N	N	N	N	N	N								
	2	F	N	N	N	N	N	N	N								
	4	F	N	N	N	N	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	2	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	4	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	8	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

これらの条件のいずれかが満たされない場合、ロードはブロックされ、再発行のために RehabQ に追加されます。

以下のガイドラインに従って、ストア・フォワーディングの問題を排除/回避できます (推奨順に示します)。

- メモリーの代わりにレジスターを使用する。
- できるだけ早くストア操作を実行する (ストアはロードよりも後のパイプライン・ステージで実行されるため、ロードよりもかなり先行して実行する必要がある)。

正しくストアフォワードされるコストは、マイクロアーキテクチャーによって異なります。Silvermont† マイクロアーキテクチャーでは、ストア・フォワーディングに 3 サイクル追加されます (つまり、ストアが n サイクルで実行されると、ロードは n + 3 サイクルで実行されます)。Goldmont† マイクロアーキテクチャーでのコストは 4 サイクルです。Goldmont Plus† マイクロアーキテクチャーでは、レジスター操作からのストアデータを最適化しており、ロードフォワードへのストアのレイテンシーを 3 サイクルに軽減しています。

F.8.3.4 PrefetchW 命令

Silvermont† マイクロアーキテクチャー以降は、PrefetchW 命令 (0f 0d /1) をサポートしています。この命令は、RFO (read-for-ownership) 要求で指定したラインをキャッシュにプリフェッチするようにハードウェアにヒントを示します。この命令を使用すると、後続のストアはラインがプリフェッチされていない場合や、別の命令でプリフェッチされた場合よりも、そのラインへの操作を速く完了できます。すべてのプリフェッチ命令は、正しく使用しないとパフォーマンスの低下につながる可能性があるため、PrefetchW を含め、プリフェッチ命令を使用する場合は実際にパフォーマンスが向上するように慎重に利用すべきです。命令オペコード 0f0d /0 は引き続き NOP として処理され、指定されたラインはプリフェッチされません。

F.8.3.5 キャッシュラインの分割とアライメント

キャッシュラインの分割は、ロード命令とストア命令の帯域幅を減少させます。そのため、できるだけ回避すべきです。

チューニングの推奨 8: perfmon カウンター REHABQ.ST_SPLIT と REHABQ.LD_SPLIT を使用すると、複数のキャッシュラインにまたがる操作とその数が分かります。

アライメントされたアクセスが推奨されますが、Silvermont† マイクロアーキテクチャーには、アライメントされていないアクセスに対するハードウェア・サポートが備わっています。そのため、前世代の Intel Atom® プロセッサーとは対照的に、MOVUPS/MOVUPD/MOVDQU 命令はすべて単一のマイクロオペレーション (uop) 命令となっています。

F.8.3.6 セグメントベース

簡略化のため、Silvermont† マイクロアーキテクチャーの AGU は、セグメントベースが 0 であると想定しています。ほとんどの場合は問題ありませんが、ゼロ以外のセグメントベース (NZB) を使用しなければならないこともあります。NZB を使用する場合、可能な限りセグメントベースをキャッシュライン (0x40) 境界にアライメントします。Silvermont† マイクロアーキテクチャーでは、NZB アドレスの生成には 1 サイクルのペナルティーが伴います。Goldmont† マイクロアーキテクチャー以降では、サイクルごとに 1 つの NZB アドレス生成が可能です。

F.8.3.7 コピーと文字列のコピー

通常、memcpy/memset ルーチンを含むライブラリーがコンパイラーによって提供されます。これらのライブラリーは優れたパフォーマンスをもたらし、コードサイズとアライメントの問題にも対応しています。

memcpy/memset 操作は、最適なバイト/ダブルワード単位のアライメントされた操作に分割した REP MOVS/STOS 命令で対応できます。これは、ほとんどの場合に汎用メモリーコピー/セットのソリューションとして利用できます。REP MOVS/STOS 命令には固有のオーバーヘッドがあります。REP STOS は複数のキャッシュラインにまたがる長い文字列に対応できますが、REP MOVS はできません。これは、ソースとデスティネーション間のアライメントの一致が複雑であるためです。

特定のメモリーコピー/セットにおいて SIMD 命令を使用するマクロコード・シーケンスは、アライメント、バッファー長、バッファー内のキャッシュの有無に応じて、ある程度のパフォーマンスの向上をもたらします (約 12 サイクル)。ただし、複数のキャッシュラインにまたがる大きなメモリーのコピーは例外です。よく考慮されたマクロコードはキャッシュラインの分割を回避し、REP MOV のパフォーマンスを大幅に向上させます。

Silvermont† マイクロアーキテクチャー・ベースのプロセッサーは、REP MOVSB と STOSB の拡張操作をサポートしています。MOVSB と STOSB を使用する REP 文字列操作は、メモリーのコピー/セット操作などのよくある状況において、最小コードサイズで柔軟かつハイパフォーマンスな REP 文字列操作を提供します。拡張 MOVSB/STOSB 操作をサポートするプロセッサーは、次のように CPUID 機能フラグで検出できます。

CPUID:(EAX=7H, ECX=0H):EBX.[ビット 9] = 1

汎用性のある実装 (将来の実装を含む) で動作する単純なデフォルトの文字列コピー/セットルーチンが必要な場合は、REP MOVSB と STOSB の拡張をサポートする実装でも REP MOVSB または REP STOSB の使用を検討すべきです。これらの命令は、特定の実装では専用のコピー/セットルーチンよりも遅くなることがあり、専用のコピー/セットルーチンは将来のプロセッサーで同じように動作せず、また将来の拡張を利用できない恐れがあります。REP MOVSB と REP STOSB は、将来のプロセッサーでもある程度のパフォーマンスが期待できます。

F.9 命令レイテンシーとスループット

この節では、最新のマイクロアーキテクチャーによる Intel Atom® プロセッサー世代のスループットとレイテンシーの一覧を示します。MSROM によるデコーダーの支援が必要な命令には、「MSROM」列に「Y」マークが記されています (よりデコード効率の高い代替手段があれば、使用は最小限に抑えるべきです)。それぞれの命令のスループットとレイテンシー値は、CPUID DisplayFamily_DisplayModel による対応するマイクロアーキテクチャーごとにグループ化されています。同じ DisplayFamily にタイミング特性が同じ多数の DisplayModels がある場合、DisplayFamily は一度だけ記載されることがあります。

この節でカバーされるマイクロアーキテクチャーと対応する DisplayFamily_DisplayModel のシグネチャーを以下に示します。

- Goldmont Plus† マイクロアーキテクチャー: 06_7AH。Goldmont Plus† マイクロアーキテクチャーの値が Goldmont† マイクロアーキテクチャーと異なる場合、表中に「GLP」を明記しています。
- Goldmont† マイクロアーキテクチャー: 06_5CH, 06_5FH.
- Silvermont† または Airmont† マイクロアーキテクチャー: 06_37H 06_4AH 06_4CH 06_4DH 06_5AH 06_5DH

表 F-19 Intel Atom® プロセッサーの最新のマイクロアーキテクチャーにおける命令レイテンシーとスループット

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
ADC/SBB r32, imm8	1	2	2	2	N	N
ADC/SBB r32, r32	1	2	2	2	N	N
ADC/SBB r64, r64	1	2	2	2	N	N
ADD/AND/CMP/OR/SUB/XOR/TEST r32, r32	0.33	0.5	1	1	N	N
ADD/AND/CMP/OR/SUB/XOR/TEST r64, r64	0.33	0.5	1	1	N	N
ADDPD/ADDSUBPD/MAXPD/MINPD/SUBPD xmm, xmm	1	2	3	4	N	N
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/ SUBSS	1	1	3	3	N	N
MAXPS/MAXSD/MAXSS/MINPS/MINSD/MINSS xmm, xmm	1	1	3	3	N	N
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XO RPD/XORPS	0.5	0.5	1	1	N	N
AESDEC/AESDECLAST/AESENC/AESENCLAST	2 1 (GLP)	5	6 4 (GLP)	8	N	Y
AESIMC/AESKEYGEN	2 1 (GLP)	5	5 4 (GLP)	8	N	Y
BLENDPD/BLENDPS xmm, xmm, imm8	0.5	1	1	1	N	N
BLENDVPD/BLENDVPS xmm, xmm	4	4	4	4	Y	Y
BSF/BSR r32, r32	8	10	10	10	Y	Y
BSWAP r32	1	1	1	1	N	N
BT/BTC/BTR/BTS r32, r32	1	1	1	1	N	N
CBW	4	4	4	4	Y	Y
CDQ/CLC/CMC	1	1	1	1	N	N
CMOVxx r32; r32	1	1	2	2	N	N
CMPPD xmm, xmm, imm	1	2	3	4	N	N
CMPSD/CMPPS/CMPSS xmm, xmm, imm	1	1	3	3	N	N
CMPXCHG r32, r32	5	6	5	6	Y	Y
CMPXCHG r64, r64	5	6	5	6	Y	Y
(U)COMISD/(U)COMISS xmm, xmm;	1	1	4	4	N	N
CPUID	58	60	58	60	Y	Y
CRC32 r32, r32	1	1	3	3	N	N

表 F-19 Intel Atom® プロセッサーの最新のマイクロアーキテクチャーにおける命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
CRC32 r64, r64	1	1	3	3	N	N
CVTDQ2PD/CVTDQ2PS/CVTPD2DQ/CVTPD2PS xmm, xmm	1	2	4	5	N	N
CVT(T)PD2PI/CVT(T)PI2PD	1	2	4	5	N	N
CVT(T)PS2DQ/CVTPS2PD xmm, xmm;	1	2	4	5	N	N
CVT(T)SD2SS/CVTSS2SD xmm, xmm	1	1	4	4	N	N
CVTSI2SD/SS xmm, r32	1	1	7	6	N	N
CVTSD2SI/SS2SI r32, xmm	1	1	4	4	N	N
DEC/INC r32	1	1	1	1	N	N
DIV r8	11-12	25	11-12	25	N	Y
DIV r16	12-17	26-30	12-17	26-30	Y	Y
DIV r32	12-25	26-38	12-25	26-38	Y	Y
DIV r64	12-41	38-123	12-41	38-123	Y	Y
DIVPD[1]	12, 65 18 (GLP)	27-69	13, 66 22 (GLP)	27-69	N	Y
DIVPS[1]	12,35 12 (GLP)	27-39	13, 36 16 (GLP)	27-39	N	Y
DIVSD[1]	12,33 10 (GLP)	11-32	13,34 14 (GLP)	13-34	N	N
DIVSS[1]	12,18 7 (GLP)	11-17	13,19 11 (GLP)	13-19	N	N
DPPD xmm, xmm, imm	5	8	8	12	Y	Y
DPPS xmm, xmm, imm	11	12	14	15	Y	Y
EMMS	23	10	23	10	Y	Y
EXTRACTPS	1	4	4	5	N	Y
F2XM1	87	88	87	88	Y	Y
FABS/FCHS	0.5	1	1	1	N	N
FCOM	1	1	4	4	N	N
FADD/FSUB	1	1	3	3	N	N
FCOS	154	168	154	168	Y	Y
FDECSTP/FINCSTP	0.5	0.5	1	1	N	N
FDIV	39 11 (EP GLP)	39	39 15 (EP GLP)	39	N	N
FLDZ	280	277	280	277	Y	Y
FMUL	2	2	5	5	N	N
FPATAN/FYL2X/FYL2XP1	303	296	303	296	Y	Y
FPTAN/FSINCOS	287	281	287	281	Y	Y

表 F-19 Intel Atom® プロセッサーの最新のマイクロアーキテクチャーにおける命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
FRNDINT	41	25	41	25	Y	Y
FSCALE	32	74	32	74	Y	Y
FSIN	140	150	140	150	Y	Y
FSQRT	40	40	40	40	N	N
HADDPD/HSUBPD xmm, xmm	5	5	5	6	Y	Y
HADDPS/HSUBPS xmm, xmm	6	6	6	6	Y	Y
IDIV r8	11-12	34	11-12	34	N	Y
IDIV r16	12-17	35-40	12-17	35-40	Y	Y
IDIV r32	12-25	35-47	12-25	35-47	Y	Y
IDIV r64	12-41	49-135	12-41	49-135	Y	Y
IMUL r32, r32 (single dest)	1	1	3	3	N	N
IMUL r32 (dual dest)	2	5	3 (4, EDX)	4	N	Y
IMUL r64, r64 (single dest)	2	2	5	5	N	N
IMUL r64 (dual dest)	2	4	5 (6,RDX)	5 (7,RDX)	N	Y
INSERTPS	0.5	1	1	1	N	N
MASKMOVDQU	4	5	4	5	Y	Y
MOVAPD/MOVAPS/MOVDQA/MOVDQU/MOVUPD/MOVUPS xmm, xmm;	0.33[2]/0.5	0.5	0/1	1	N	N
MOVD r32, xmm; MOVQ r64, xmm	1	1	4	4	N	N
MOVD xmm, r32 ; MOVQ xmm, r64	1	1	4	3	N	N
MOVDDUP/MOVHLPS/MOVLHPS/MOVSHDUP/MOVSLDUP	0.5	1	1	1	N	N
MOVDQ2Q/MOVQ/MOVQ2DQ	0.5	0.5	1	1	N	N
MOVSD/MOVSS xmm, xmm;	0.5	0.5	1	1	N	N
MPSADBW	4	5	5	7	Y	Y
MULPD	1	4	4	7	N	N
MULPS; MULSD	1	2	4	5	N	N
MULSS	1	1	4	4	N	N
NEG/NOT r32	0.33	0.5	1	1	N	N
PACKSSDW/WB xmm, xmm; PACKUSWB xmm, xmm	0.5	1	1	1	N	N
PABSB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PADDB/D/W xmm, xmm; PSUBB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PADDQ/PSUBQ/PCMPEQQ xmm, xmm	1	4	2	4	N	Y
PADDSB/W; PADDUSB/W; PSUBSB/W; PSUBUSB/W	0.5	0.5	1	1	N	N
PALIGNR xmm, xmm	0.5	1	1	1	N	N
PAND/PANDN/POR/PXOR xmm, xmm	0.5	0.5	1	1	N	N

表 F-19 Intel Atom® プロセッサーの最新のマイクロアーキテクチャーにおける
命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5AH,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4CH,4DH,5AH,5DH
PAVGB/W xmm, xmm	0.5	0.5	1	1	N	N
PBLENDW xmm, xmm, imm	0.5	0.5	1	1	N	N
PBLENDVB xmm, xmm	4	4	4	4	Y	Y
PCLMULQDQ xmm, xmm, imm	4	10	6	10	Y	Y
PCMPEQB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PCMPESTRI xmm, xmm, imm	13	21	19(C)/ 26(F)[3]	21(C)/ 28(F)	Y	Y
PCMPESTRM xmm, xmm, imm	14	17	15(X)/ 25(F)[1]	17(X)/ 24(F)	Y	Y
PCMPGTB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PCMPGTQ/PHMINPOSUW xmm, xmm	2	2	5	5	N	N
PCMPISTRI xmm, xmm, imm	8	17	14(C)/ 21(F)[1]	17(C)/ 24(F)	Y	Y
PCMPISTRM xmm, xmm, imm	7	13	10(X)/ 20(F)[1]	13(X)/ 20(F)	Y	Y
PEXTRB/WD r32, xmm, imm	1	4	4	5	N	Y
PINSRB/WD xmm, r32, imm	1	1	4	3	N	N
PHADDD/PHSUBD xmm, xmm	4	6	4	6	Y	Y
PHADDW/PHADDSW xmm, xmm	6	9	6	9	Y	Y
PHSUBW/PHSUBSW xmm, xmm	6	9	6	9	Y	Y
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, xmm	1	2	4	5	N	N
PMAXSB/W/D xmm, xmm; PMAXUB/W/D xmm, xmm	0.5	0.5	1	1	N	N
PMINSB/W/D xmm, xmm; PMINUB/W/D xmm, xmm	0.5	0.5	1	1	N	N
PMOVMSKB r32, xmm	1	1	4	4	N	N
PMOVSXBW/BD/BQ/WD/WQ/DQ xmm, xmm	0.5	1	1	1	N	N
PMOVZXBW/BD/BQ/WD/WQ/DQ xmm, xmm	0.5	1	1	1	N	N
PMULDQ/PMULUDQ xmm, xmm	1	2	4	5	N	N
PMULHUW/PMULHW/PMULLW xmm, xmm	1	2	4	5	N	N
PMULLD xmm, xmm	2	11	5	11	N	Y
POPCNT r32, r32	1	1	3	3	N	N
POPCNT r64, r64	1	1	3	3	N	N
PSHUFB xmm, xmm	1	5	1	5	N	Y
PSHUFD xmm, mem, imm	0.5	1	1	1	N	N
PSHUFHW; PSHUFLW; PSHUFW	0.5	1	1	1	N	N
PSIGNB/D/W xmm, xmm	0.5	1	1	1	N	N

表 F-19 Intel Atom® プロセッサーの最新のマイクロアーキテクチャーにおける命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5AH,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4CH,4DH,5AH,5DH
PSLLDQ/PSRLDQ xmm, imm; SHUFPD/SHUFPS	0.5	1	1	1	N	N
PSLLD/Q/W xmm, xmm	1	2	2	2	N	N
PSRAD/W xmm, imm;	0.5	1	1	1	N	N
PSRAD/W xmm, xmm;	1	2	2	2	N	N
PSRLD/Q/W xmm, imm;	0.5	1	1	1	N	N
PSRLD/Q/W xmm, xmm	1	2	2	2	N	N
PTEST xmm, xmm	1	1	4	4	N	N
PUNPCKHBW/DQ/WD; PUNPCKLBW/DQ/WD	0.5	1	1	1	N	N
PUNPCKHQDQ; PUNPCKLQDQ	0.5	1	1	1	N	N
RCPPS/RSQRTPS	6	8	9	9	Y	Y
RCPSS/RSQRTSS	1	1	4	4	N	N
RDTSC	20	30	20	30	Y	Y
ROUNDPD/PS	1	2	4	5	N	N
ROUNDSD/SS	1	1	4	4	N	N
ROL; ROR; SAL; SAR; SHL; SHR (count in CL)	1	1	1 (2 for CL source)	1 (2 for CL source)	N	N
ROL; ROR; SAL; SAR; SHL; SHR (count in imm8)	1	1	1	1	N	N
SAHF	1	1	1	1	N	N
SHLD r32, r32, imm	2	2	2	2	N	N
SHRD r32, r32, imm	2	4	2	4	N	Y
SHLD/SHRD r64, r64, imm	12	10	12	10	Y	Y
SHLD/SHRD r64, r64, CL	14	10	14	10	Y	Y
SHLD/SHRD r32, r32, CL	4	4	4	4	Y	Y
SHUFPD/SHUFPS xmm, xmm, imm	0.5	1	1	1	N	N
SQRTPD	67 26 (GLP)	70	68 30 (GLP)	71	N	Y
SQRTPS	37 14 (GLP)	40	38 18 (GLP)	41	N	Y
SQRTSD	34 14 (GLP)	35	35 18 (GLP)	35	N	Y
SQRTSS	19 8 (GLP)	20	20 12 (GLP)	20	N	Y
TEST r32, r32	0.33	0.5	1	1	N	N
UNPCKHPD; UNPCKHPS; UNPCKLPD, UNPCKLPS	0.5	1	1	1	N	N
XADD r32, r32	2	5	4	5	Y	Y
XCHG r32, r32	2	5	4	5	Y	Y

表 F-19 Intel Atom® プロセッサーの最新のマイクロアーキテクチャーにおける
命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
XCHG r64, r64	2	5	4	5	Y	Y
SHA1MSG1/SHA1MSG2/SHA1NEXTE	1	NA	3	NA	N	NA
SHA1RNDS4 xmm, xmm, imm	2	NA	5	NA	N	NA
SHA256MSG1/SHA256MSG2	1	NA	3	NA	N	NA
SHA256RNDS2	4	NA	7	NA	N	NA

注意:

1. DIVPD/DIVPS/DIVSD/DIVSS では、最初に最速値、次に一般的なケースの値を示しています。最速値は QNAN などの特殊な入力値の場合に適用されます。一般的なケースは、通常の数値の場合に適用されます。
2. ムーブの排除が適用される場合のスループットは 0.33 サイクルですが、それ以外は 0.5 サイクルです。
3. ECX/EFLAGS/XMM0 のレイテンシー値は依存性によるものです: (C/F/X)

索引

N

O

P

Q

S

W

X

あ

い

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

に

は

ひ

ふ

へ

ほ

ま

む

め

も

よ

ら

り

る

れ

ろ

わ