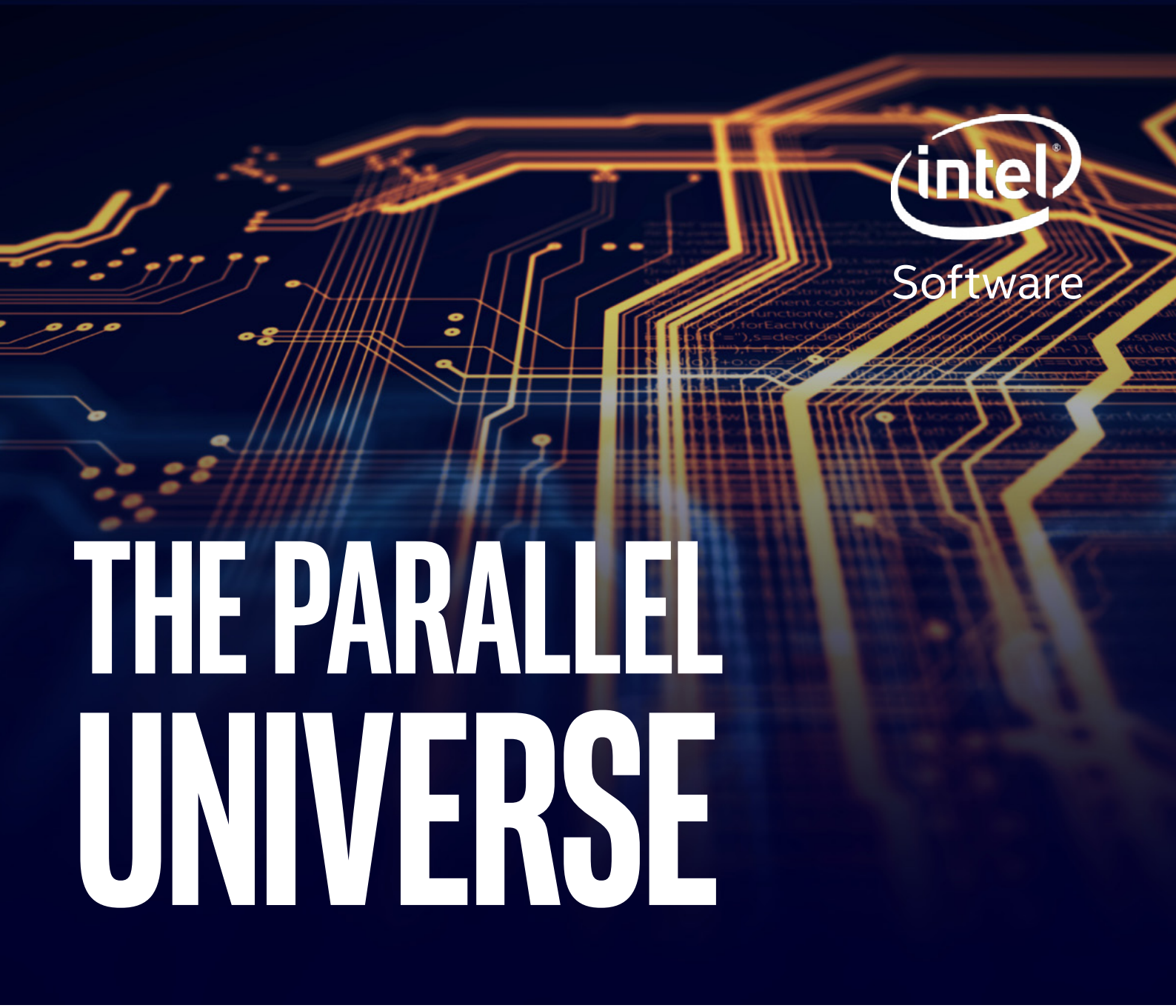


THE PARALLEL UNIVERSE

インテル® Advisor のフローグラフ・アナライザーで
自動運転コードのパフォーマンスを向上

インテル® Parallel Studio XE 最新バージョン 2018
によるコードの現代化

ソフトウェア開発者による FPGA の利用

Issue
30
2017

目次

編集者からのメッセージ

Henry A. Gabb インテル コーポレーション シニア主席エンジニア

HPC と並列コンピューティング分野で長年の経験があり、並列プログラミングに関する記事を多数出版しています。『Developing Multithreaded Applications: A Platform Consistent Approach』の編集者 / 共著者で、インテルと Microsoft* による Universal Parallel Computing Research Centers のプログラム・マネージャーを務めました。

インテル® Parallel Studio XE 2018 のリリース

インテル® Parallel Studio XE 2018 は、インテル® アーキテクチャー上でソフトウェアを現代化するインテルの包括的なツールスイートの最新バージョンです。このリリースを祝って、本号ではインテル® Parallel Studio XE のコンポーネントに関するいくつかの記事を取り上げています。「**コードの現代化によるパフォーマンス、移植性、スケーラビリティーの向上**」では、このツールスイートで提供される多数の新機能を紹介します。(インテル® Parallel Studio XE 2018 のリリースについては、**私のブログ** (英語) もご覧ください。)「**外れ値への対応**」では、**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)** を利用して、クレジットカード取引の実際のデータセットで不正を検出し、ハイパフォーマンスと高精度を達成する方法を紹介します。

インテル® Parallel Studio XE は、常に OpenMP* をサポートしてきました。最新のリリースでは、OpenMP* 4.5 と 5.0 ドラフト仕様の多くの機能をサポートしています。OpenMP* 誕生 20 周年を祝う一連の記事の最後を飾るのは、ニューヨーク州立大学ストーニーブルック校の教授であり、ブルックヘブン国立研究所のコンピューター・サイエンスおよび数学部門ディレクターでもある Barbara Chapman 氏の「**OpenMP* が大人の仲間入り**」です。OpenMP* のこれまでの成功を振り返り、今後も OpenMP* が不可欠な並列プログラミング・モデルとして残るであろう理由を述べます。

間もなく開催される **Intel® HPC Developer Conference** (英語) (11 月 11 日~ 12 日にコロラド州デンバーで開催) と **SC17** (英語) (11 月 12 日~ 17 日にコロラド州デンバーで開催) にちなんで、本号では HPC 関連の 3 つの記事を掲載しています。「**あなたのクラスターは健全ですか ?**」では、**インテル® Cluster Checker** (英語) について取り上げます。インテル® Parallel Studio XE のこのコンポーネントは、多くの最も一般的な手法とシステム診断により、クラスターが効率良く動作するように支援します。いくつかの BIOS オプションは、アプリケーションのパフォーマンスに影響しますが、実稼働中のクラスター環境でそれらのオプションをオンデマンドで変更することは困難です。「**HPC クラスターの最適化**」では、オンデマンドでクラスター設定を変更する手法を習得することができます。そして、「**クラスター全体の効率良い使用**」では、インテル® Parallel Studio XE のいくつかのツールを使用して、HPC アプリケーションをチューニングするケーススタディーを紹介します。

ヘテロジニアス並列コンピューティングの将来

将来は、ヘテロジニアスです。(実際、何年も前から CPU と GPU は同じシステムだけでなく、同じプロセッサー・ダイ上に統合されています—つまり、ヘテロジニアス・コンピューティングの時代はすでに到来しているのです。) マルチコア・プロセッサーにより並列処理が当たり前のものになったように、間もなく CPU、GPU、FPGA、ASIC などが同じシステム内に共存し、ヘテロジニアス並列処理が一般的なものになるでしょう。私はかつてヘテロジニアスの将来に不安を覚えていましたが、新しい並列プログラミング・モデルは、計算処理を最も効率良いプロセッサー・アーキテクチャーへ簡単にマップできるようにします。「**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)** のフローグラフ API はその一例です。

この API については、以前に The Parallel Universe **Special Issue** (英語) の「Heterogeneous Programming with Intel® Threading Building Blocks」で取り上げたため本号では触れませんが、「**インテル® Advisor のフローグラフ・アナライザーで自動運転コードのパフォーマンスを向上**」では、インテル® Parallel Studio XE のプレビュー機能であるフローグラフ・アナライザー (FGA) について詳しく見ていきます。自動運転アプリケーションを例に、フローグラフ計算と解析について説明します。

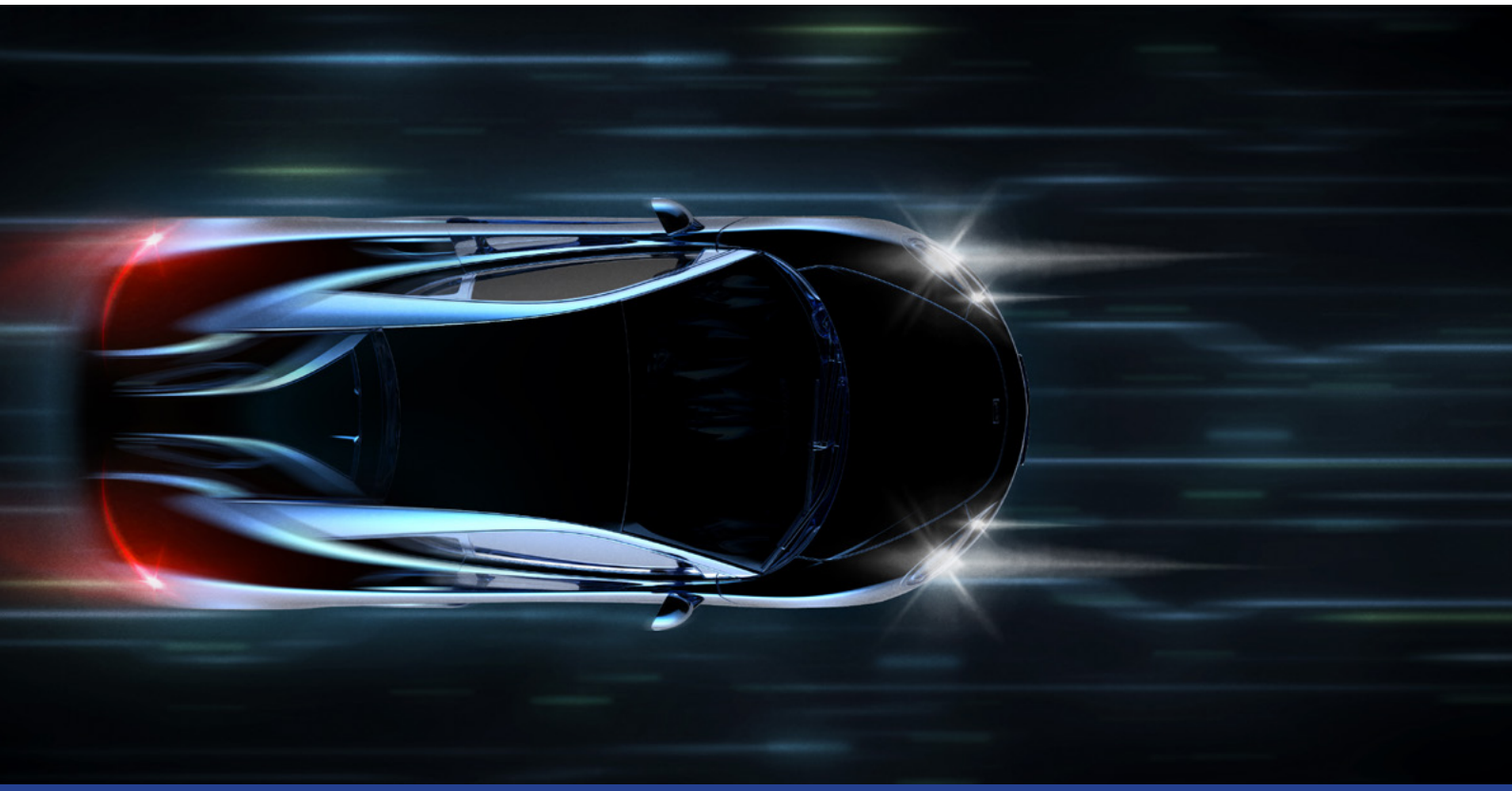
The Parallel Universe の創刊者兼編集者であった James Reinders は、引き続きヘテロジニアスというテーマに取り組んでいます。「**ソフトウェア開発者による FPGA の利用**」では、James と Bernhard Friebe 氏が FPGA プログラミングについて、ハードウェアではなくソフトウェア開発の視点から述べています。FPGA プログラミングについては、次号でも取り上げる予定です。楽しみにしていてください。

最後に、本号を締めくくるのは、インテル® AVX-512 命令セット・アーキテクチャーの新機能について詳しく紹介する「**最新の SIMD 拡張とインテル® 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) による成功のためのチューニング**」です。インテル® AVX-512 向けの新しい SIMD 言語拡張とインテル® コンパイラーの**インテル® Xeon® スケーラブル・プロセッサー**・サポートを利用する、パフォーマンス・チューニングのベスト・プラクティスについて述べます。

今後について

今後の The Parallel Universe では、FPGA プログラミング、Java* パフォーマンス・チューニング、インテル® Parallel Studio XE の新機能など、さまざまなトピックに取り組んでいきます。お見逃しなく。

Henry A. Gabb
2017 年 10 月



インテル® Advisor のフローグラフ・アナライザーで自動運転コードのパフォーマンスを向上

自動運転アプリケーションのパフォーマンスを最適化

Vasanth Tovinkere インテル コーポレーション フローグラフ・アナライザー・アーキテクト
Pablo Reble 同ソフトウェア・エンジニア
Farshad Akhbari 同パーセプチュアル・コンピューティング・テクニカル・リード
Palanivel Guruvareddiar 同パーセプチュアル・コンピューティング・ソフトウェア・アーキテクト

最新の自動車に搭載されている先進運転支援システム (ADAS) と自動運転テクノロジーは、レーダー、ライダー、カメラなどのセンサーによる環境認識に依存しています。この環境認識に、用いられるマシンラーニングやディープラーニングのアルゴリズムは、計算負荷が高く、センサー解像度の向上によりその要件はさらに厳しくなります。例えば、ビデオストリーム内のオブジェクトを検出して分類するには、可能なすべてのアスペクト比とサイズについて、対象オブジェクトの各フレームを調査する必要があります。リアルタイム・オブジェクト検出システムも多くの計算リソースを必要とします。そして、システムの完成度が高まるにつれ、その要件は厳しくなる一方です。

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

これらのシステムには、通常、提案生成プロセス (イメージ中の可能性のある対象領域を提案してから解析する) が含まれます。提案生成は繰り返し行われるため、並列化することでパフォーマンスを向上できます。イメージ全体を 1 つの提案として含めることも、あるいはフレームの各ピクセルを個別の提案として含めることもできるため、表現される並列処理はスケーラブルです。提案処理には依存関係がないため、システムリソースに制限されなければ、(パイプライン方式で) 提案が利用可能になったときにバッチ処理することができます。

この記事では、**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)** のフローグラフ・インターフェイスを使用して並列アプリケーションをサポートするインテルのツールの 1 つである、**インテル® Advisor** のフローグラフ・アナライザー (FGA) でそのようなアプリケーションの並列処理を設計・解析する方法を説明します。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、開発者がマルチコア・アーキテクチャーやヘテロジニアス・システムの利点を活かして並列アプリケーションを簡単に作成できるようにします。フローグラフ・インターフェイスは、依存性グラフとデータフロー・アルゴリズムの効率良い実装を提供することで高レベルで並列化を利用できるようにするため、2011 年にインテル® TBB に追加されました。[**編集者注 : The Parallel Universe Special Edition** (英語) の「Intel® Threading Building Blocks Celebrates 10 Years」に、インテル® TBB の API の進化と今後の方向性に関するいくつかの興味深い記事があります。]

インテル® TBB により並列処理を簡単に表現できたとしても、ADAS と自動運転アプリケーションは複雑で、多くの連携するアルゴリズムと並列処理レイヤーが含まれているため、設計とチューニングが困難です。インテル® Advisor のプレビュー機能である FGA は、このタスクを容易にします。FGA 機能の使用法については、インテル® TBB のフローグラフ API を使用して実装された **OpenCV*** (英語) ベースの自動運転サンプルの解析を通して説明します。

先進運転支援サンプル

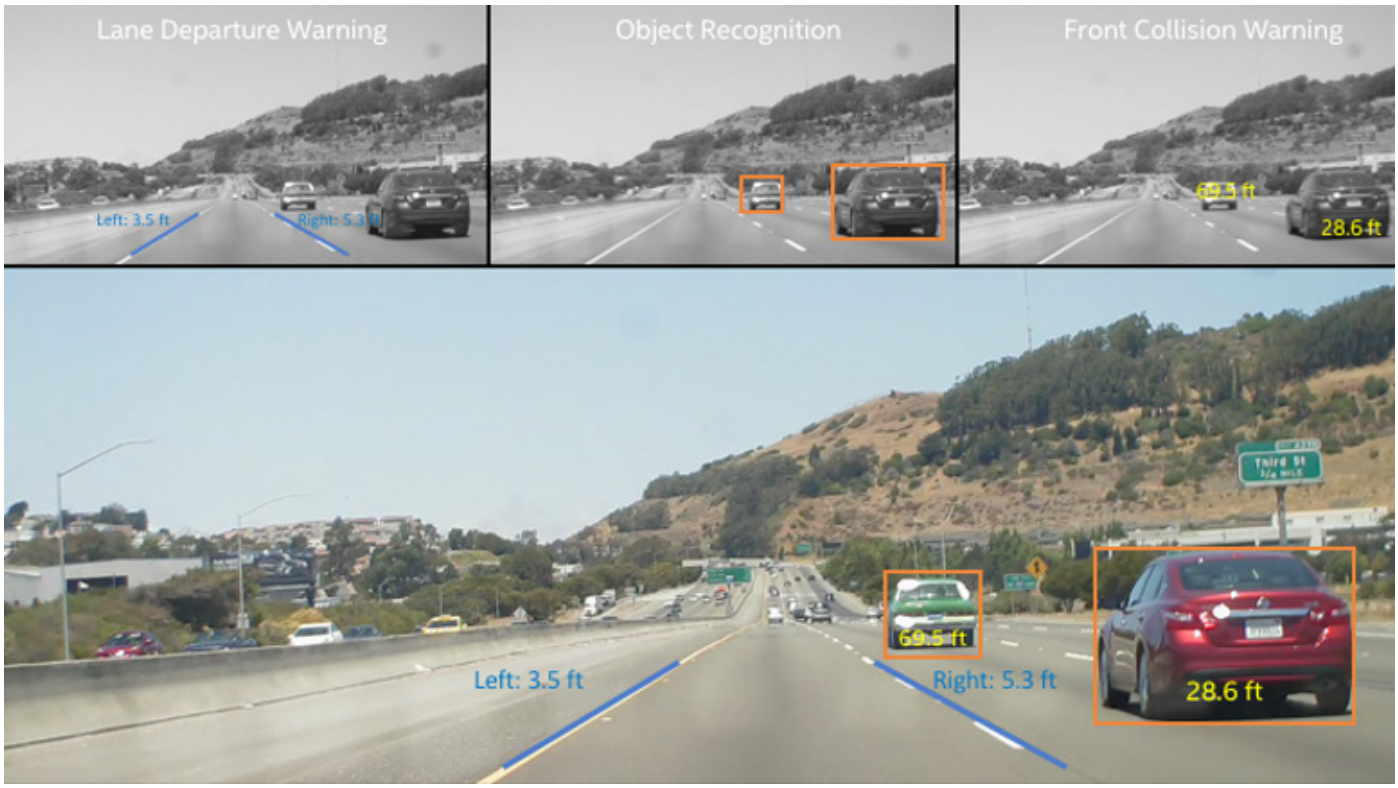
ここでは、実行例として ADAS のデモ・フレームワークを使用します。このサンプル・フレームワークは、2 つの動作モードをサポートします。

1. **オンラインモード** : フレームワークは、バックミラーに搭載されたカメラセンサーからライブ映像を受け取り、車の前方のデータをキャプチャーします。
2. **オフラインモード** : ファイルから録画フレームを読み取り、カメラセンサーの動作をシミュレーションします。

受信した圧縮カメラフレームは、デコードされ、コンピューター・ビジョン・アルゴリズムが処理できるようにフレームの後処理を行うビデオ入力処理モジュールへ送られます。

後処理済みデータは、フレームワーク内の登録されたコンピューター・ビジョン・ユースケース (物体認識、車線逸

脱警報、前方衝突警報など) へブロードキャストされます。これらのユースケースは、同じフレーム上で個別に (並列に) 実行できます。その多くは、内部で提案生成プロセスを使用して、ユースケース・レベルで並列化しています。多くの場合、ユースケースは、すでにループレベルで並列化された OpenCV* アルゴリズムを使用して実装されます。そのため、フレームワークには複数レベルの並列処理が含まれます。フレームワークは拡張可能で、プラグインを介して新しいアルゴリズムを登録することができます。コンピューター・ビジョン・アルゴリズムは、展開済みフレームを処理して検出したオブジェクトや車線に関連するメタデータを生成し、車線逸脱警報や前方衝突警報を出力します。複合モジュールはすべての情報をまとめて、展開済みフレームの上にオーバーレイして、**図 1** の色付きの部分のようにレンダリングします。上段にある小さなグレースケール・イメージが各ユースケースの出力です。



1 ADAS アプリケーションの例

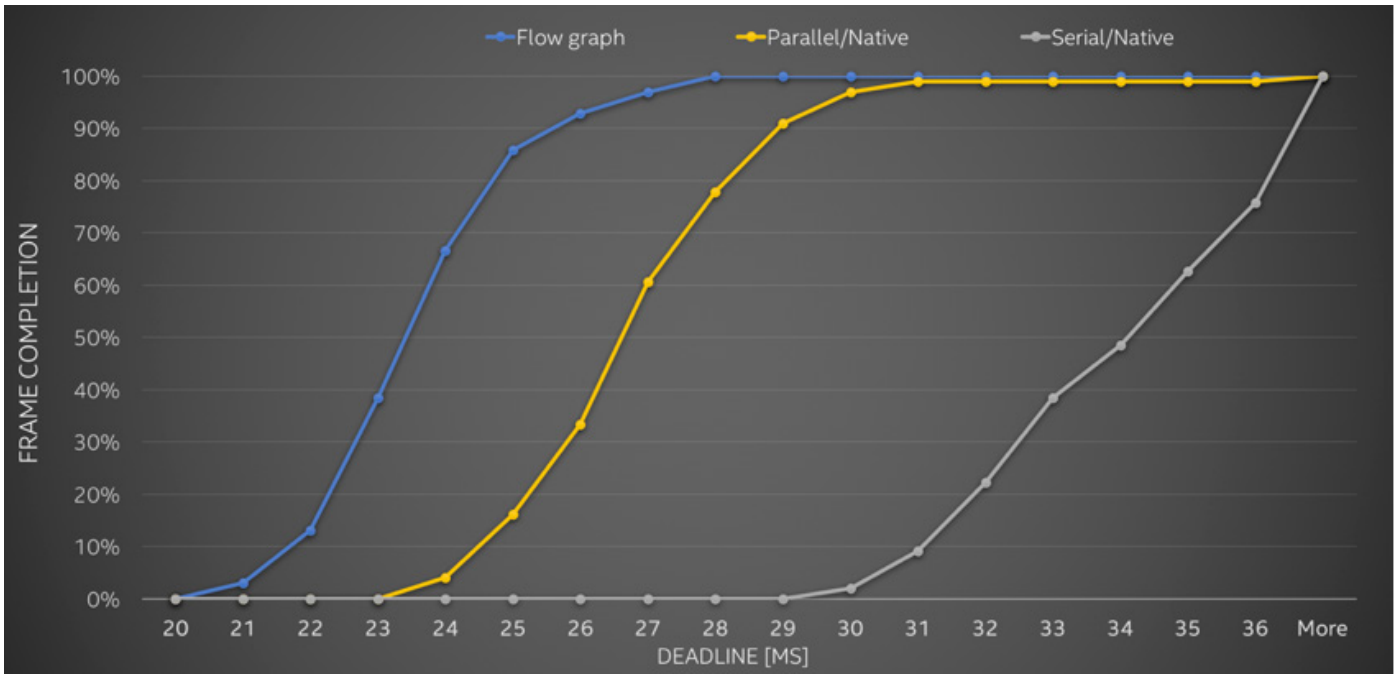
並列フレームワークの進化

一般に、ビデオ処理アプリケーションの主なパフォーマンス指標として、各フレームの処理時間 (単位時間) が決められている場合にどれだけ多くのフレームが完了されたかを示す、フレーム完了率があります。フレーム処理が単位時間を超える場合、そのフレームは廃棄されます。単位時間が 24ms で完了率が 70% の場合、この単位時間では 70% のフレームを完了することができ、30% は廃棄されることを意味します。

このフレームワークの最適化では、次の 3 つの実装を検証しました。

1. オリジナルのシリアル / ネイティブ実装
2. 並列 / ネイティブ実装
3. インテル® TBB のフローグラフ実装

図 2 は、3 つの実装のフレーム完了率です。最適化を適用したほうが単位時間が短くなります。



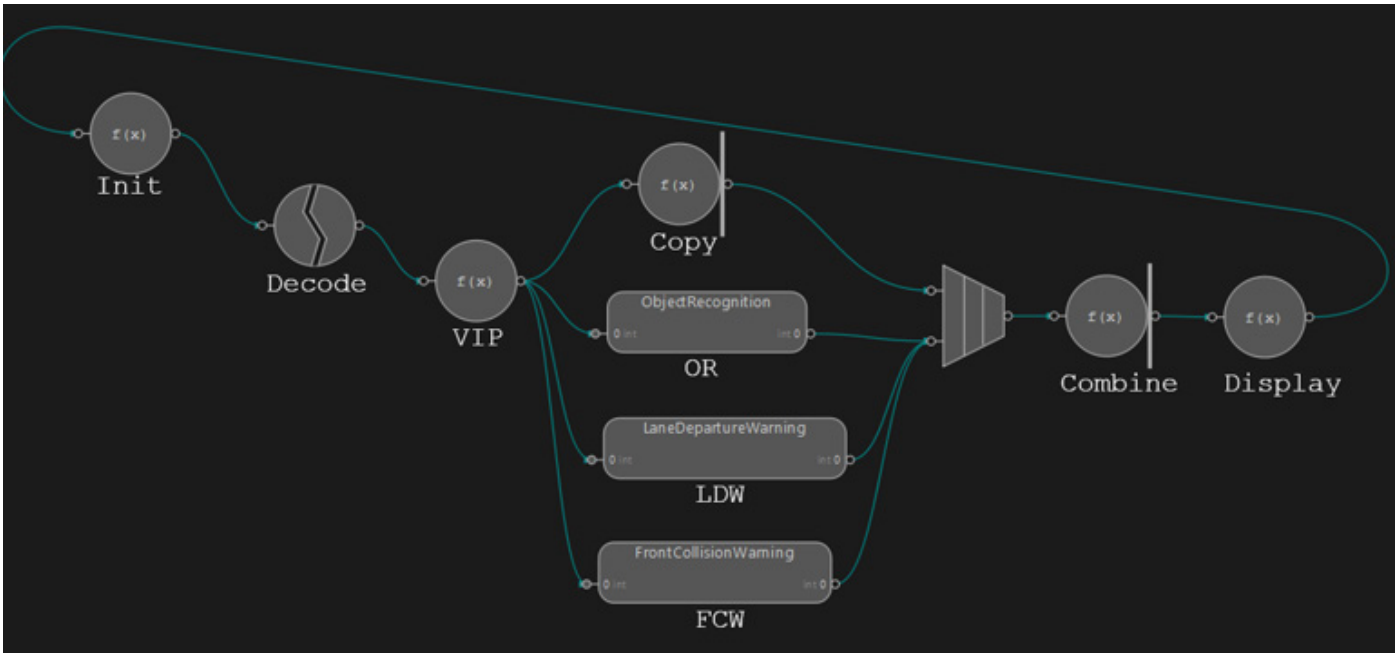
2 フレーム完了率の比較

シリアル / ネイティブ実装は、マルチコア・プロセッサー上でユースケースをシリアルに実行します。ユースケースの多くは OpenCV* で実装されており、ネストされたループレベルで並列化されていますが、このアプローチはシステムのコアを効率良く利用していません。個々のアルゴリズムの並列性は限定的で、アプリケーションのスケーラビリティーに直接影響します。

並列 / ネイティブ実装は、タスクベースで、標準のスレッドを使用してアルゴリズム・モジュールを並列に実行できます。ただし、OpenCV* 内で生成されるインテル® TBB スレッドに加えて、タスク用のスレッドが生成されるため、オーバーサブスクリプションを引き起こします。オーバーサブスクリプションを回避して、柔軟性と拡張性に優れたフレームワークにするため、フローグラフ実装は両方のレベルでインテル® TBB を使用しています。

図 3 は、インテル® TBB のデータフロー・グラフとして実装された先進運転支援フレームワークです。各フレームに適用される初期セットアップルーチンをカプセル化した標準のフローグラフ関数ノード (Init) を呼び出して実行を開始します。次に、非同期ノード (Decode) を介して外部アクティビティーとの相互作用 (つまり、各フレームのデコーディング) を表現します。ビデオ入力処理 (VIP) ノードは、入力イメージを各ユースケース (物体認識、車線逸脱警報、前方衝突警報など) と多機能ノード (Copy) へブロードキャストします。このノードは、2 つのアクションを実行します。

1. **ストア** : 後でユースケースの出力とマージする変更前の入力イメージを格納します。
2. **生成** : 各ユースケースのトークンを生成します。



3 自動運転のサンプル・フレームワークに対応するグラフ

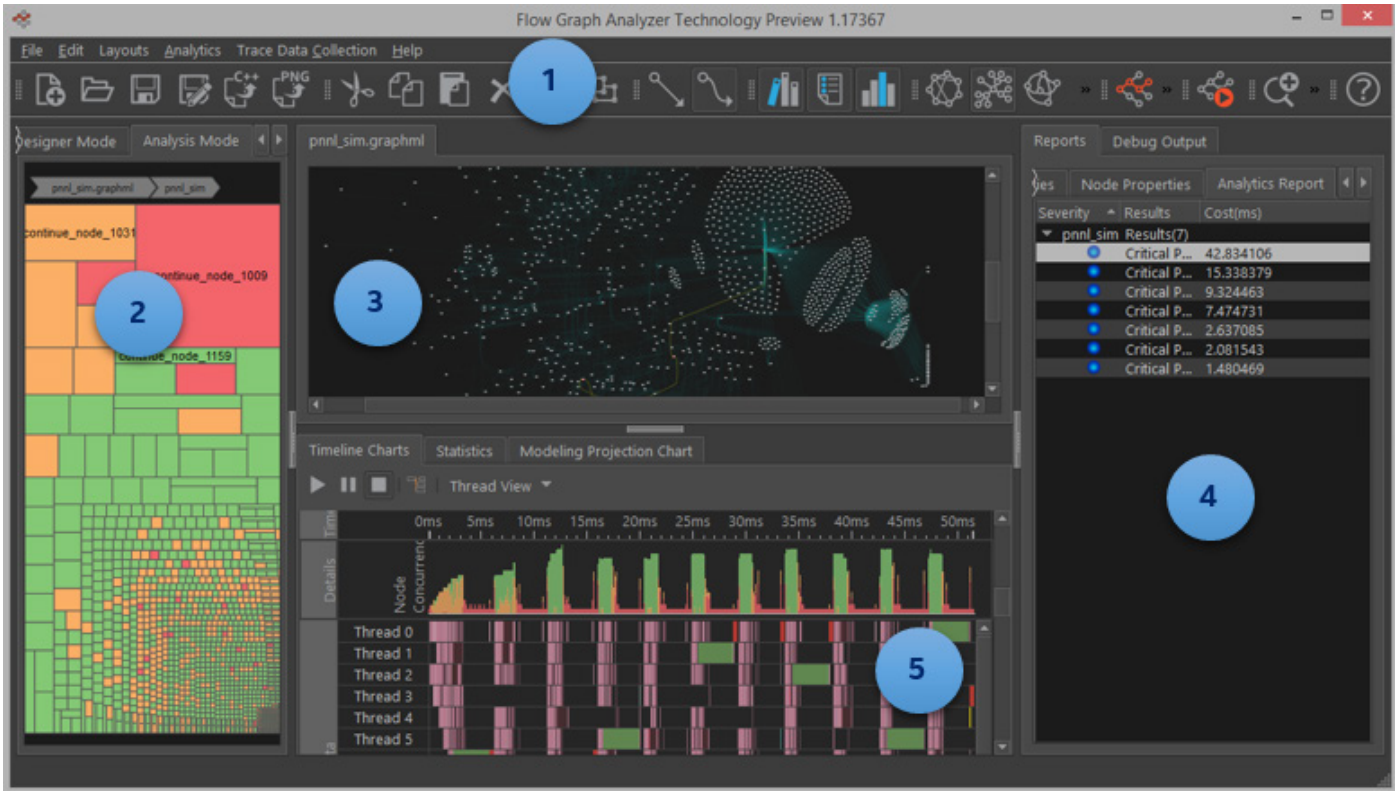
その次のノードは、このトークンと出力バッファーのタプルを生成し、生成されたタプルは多機能 Combine ノードで使用されます。**図 1** の色付きの部分は、**図 3** の中央にあるフローグラフ合成ノードによって表現された異なるユースケースからの出力です。サンプルでは、OpenCV* を使用して物体認識 (OR)、車線逸脱警報 (LDW)、前方衝突警報 (FCW) が実装されています。Display ノードは、出力イメージを表示します。[**編集者注 :** グラフノードのすべての属性については、記事の本題からは外れるためここでは取り上げません。API の詳細は、**インテル® TBB ドキュメントの「フローグラフ」**(英語) を参照してください。]

データフロー設計の主な利点は、より多くの並列処理を表現できることです。これは、パフォーマンスの向上につながります。また、サンプルでは、ノードを 1 つの関数にまとめて表現して、各ノードで process_frame() 関数を呼び出すため、既存のユースケースの変更が不要です。つまり、フローグラフ実装はすべての既存フレームワークのプラグインと下位互換性があります。

次に、FGA の機能を利用して、このフローグラフのパフォーマンスを解析し、動作を詳しく理解します。オリジナルの実装よりもパフォーマンスは向上していますが、さらにパフォーマンスを引き出す余地があります。

FGA の概要

図 4 は、FGA によって生成されたグラフと関連するパフォーマンス・トレース・データです。FGA は 5 つのビューで構成されています。



4 キャプチャーしたグラフと関連トレースデータを示す FGA

1. **メニュー / ツールバー** : グラフの設計、操作、視覚化、解析を行うためのメニューとアイコンです。
2. **インデックス領域** : 3 つのカテゴリーに分類されます。デザイナーモードは、新しいグラフの作成に利用できるノードタイプを表示します。階層ビューは、グラフのトポロジーを階層ツリーとして表示します。そして、解析モード (**図 4**) は、グラフのパフォーマンスをツリーマップに四角で表します。(この記事では、解析モードに注目します。)
3. **キャンバス領域**: グラフの構築や表示に使用するメインのグラフ領域です。ここでは、新しいグラフの設計を支援するため、一般的な編集操作をサポートしています。
4. **レポート領域** : グラフ、ノードのプロパティーなどの基本的なレポートに加えて、セマンティクス・ルール・チェックやクリティカル・パス・アルゴリズムなどの解析によって生成されるレポートが表示されます。
5. **グラフ領域** : 関連するトレースデータを含むグラフのパフォーマンス統計と実行トレース・タイムラインを表示します。このデータは、トレースコレクターによって自動的にキャプチャーされます。トレースコレクターは、GUI からまたはコマンドライン・ユーティリティーとして起動できます。

FGA では、ビュー領域とモードを使用して設計と解析の 2 つのメイン・ワークフローを作成できます。

1. **設計ワークフロー** : ドラッグアンドドロップでインタラクティブにインテル® TBB のフローグラフを作成します。
2. **解析ワークフロー** : インテルのフローグラフ・アプリケーションからパフォーマンス・データをキャプチャーします。

FGA は、パフォーマンス・データを視覚化して活用するための機能を提供します。ここでは、解析ワークフローを使用して、ADAS サンプルで収集されたパフォーマンス・データを検証します。

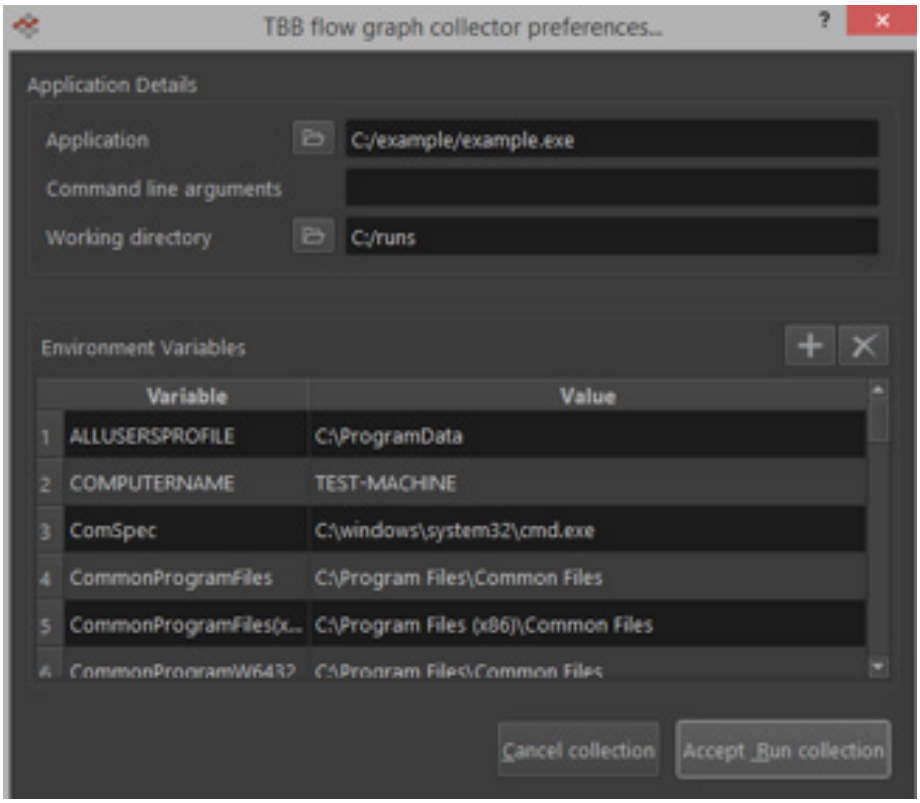
ADAS サンプルのパフォーマンス解析

サンプルのパフォーマンスを最適化するには、最初にボトルネックを特定する必要があります。FGA を利用することで、実行中のインテル® TBB のフローグラフ・アプリケーションからグラフトポロジーとタスク実行トレースをキャプチャーできます。**図 5** は、トレース収集ダイアログウィンドウです。ここで、トレースするアプリケーションを指定します。データ収集が完了すると、2 つのファイルが生成されます。

1. **GraphML ファイル** : 実行したグラフのトポロジーに関する説明です。
2. **TraceML ファイル** : グラフのタスク実行トレースが含まれます。

アプリケーションの最重要部分の特定

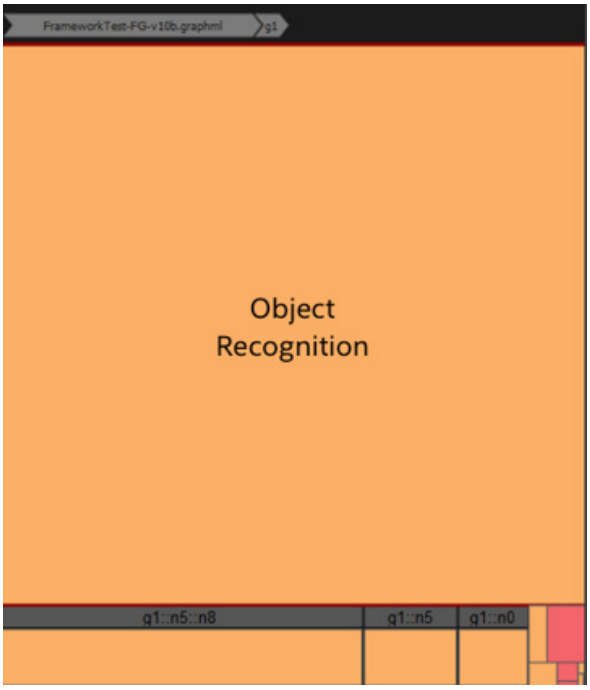
アプリケーション・パフォーマンスを解析する一般的な最初のステップは、アプリケーションで多くの時間が費やされている場所を特定することです。FGA には、このステップに役立つ 2 つの機能があります。



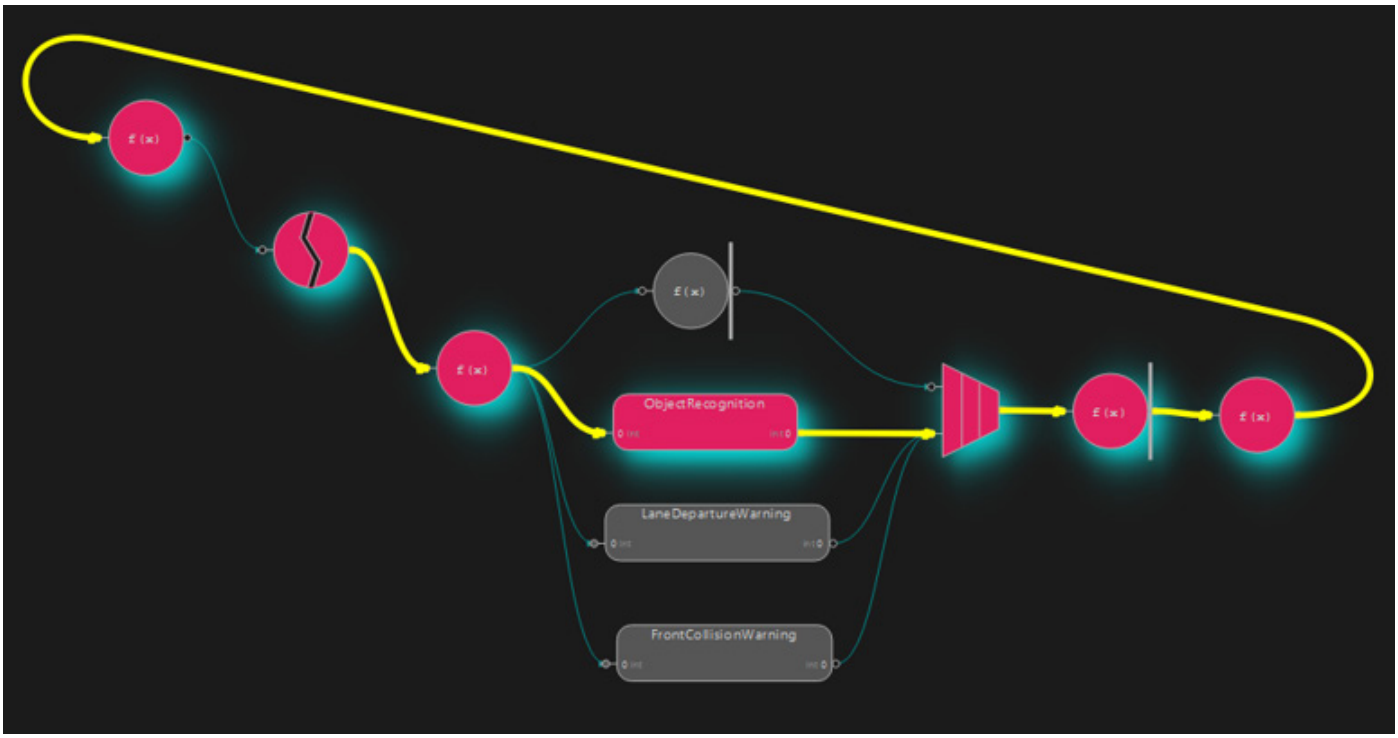
5 FGA トレース収集ダイアログ

最初に、ツリーマップ・ビューは、合計 CPU 時間を大きな長方形で示します。この長方形は、グラフ内のノードを表す小さな長方形で構成されています。各ノードの長方形の大きさは、そのノードの合計 CPU 時間に比例しています。色は、ノードの実行時の並列性を表します (赤色は並列性が低いことを示します)。**図 6** は、ADAS サンプルのツリーマップ・ビューです。この図から、ユースケースごとに実行時間が異なり、フレームワーク全体の CPU 時間の多くが OR に費やされていることが分かります。ユースケースの実行時間が異なることは、各ユースケースが結果を個別にオーバーレイ・ノードへ送るという決定を裏付けるものです。バリアはロード・インバランスを引き起こします。

FGA は、アプリケーションのクリティカル・パスを計算し、フローグラフのトポロジーに投影することもできます (**図 7**)。ADAS サンプルのクリティカル・パスは、最適化を検討すべき主要ノードを示しています。ツリーマップ・ビューとクリティカル・パス・ビューから、OR が最も多くの時間を費やしているだけでなく、アプリケーションの最も重要なパス上にあることが分かります。グラフのパフォーマンスは、OR のパフォーマンスによって制限されるため、OR の最適化を最優先すべきです。場合によっては、厳しいパフォーマンス要件を満たすため、アクセラレーターへオフロードするアルゴリズムを決定する必要があるかもしれません。



6 ツリーマップ・ビュー



7 グラフのクリティカル・パスを示すスクリーンショット

ボトルネックのパフォーマンス特性の解析

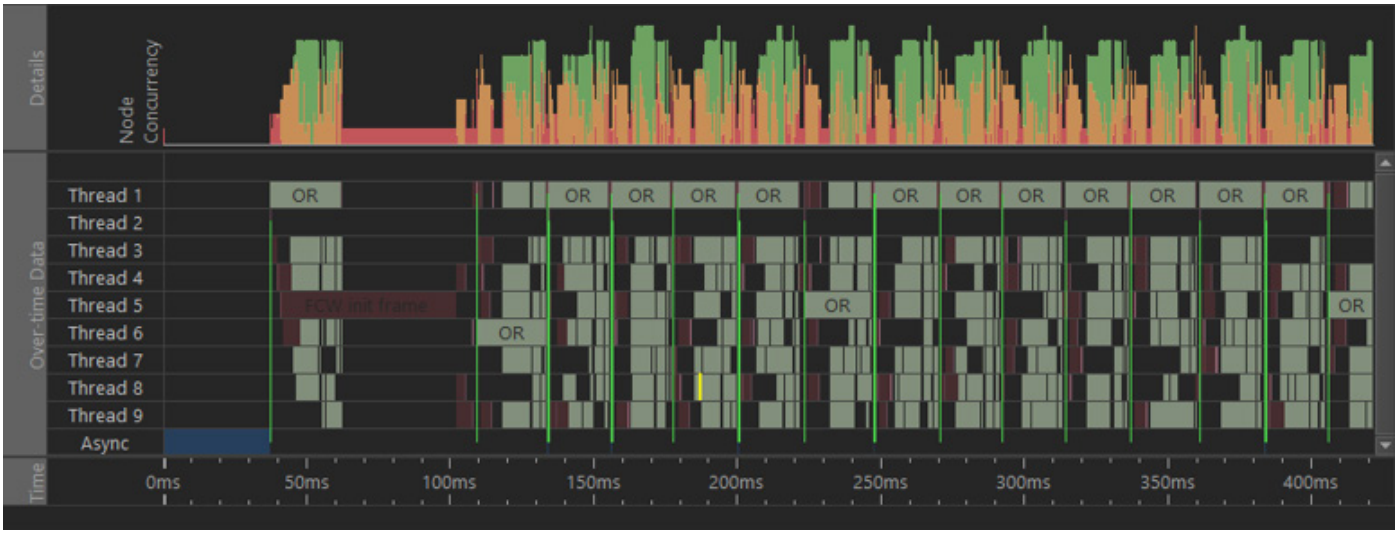
OR が最重要アルゴリズムであることが分かったため、FGA のタイムライン・ビューで OR のパフォーマンスをさらに詳しく調査します。スレッドごとのオーバータイム・タスク・データは、収集したトレースの未加工のビューを提供します。このグラフでは、各スレッドで実行されたタスクとその実行時間を確認できます。**図 8** は、1 フレームのデータ処理を拡大表示したものと色の凡例を示しています。タスクは、その種類に応じて色付けされます。グラフのタスクは、タスクの実行時間に応じて色付けされ、タスクのスケジュール・コストに対して実行時間が短いものは薄い色で示されます。入れ子の並列タスクと非同期タスクは青色で示されています。

8 データの 1 フレームと関連する色の凡例を示すオーバータイム・トレース・グラフ

グラフのタスクの色は、実行時間が長く、パフォーマンス・スケーリングに影響する可能性があるタスク (ノード) を特定できるように、1 μ s から 1ms の範囲で一定の領域ごとに区別して表示されています。タイムラインでタスクを選択すると、キャンバス上のグラフのノードがハイライト表示されます。このグラフを利用することで、スレッドにスポーンするには小さすぎるタスクに注目して、並列性が低い場合、グラフのパフォーマンスをデバッグすることができます。同様の情報は、入れ子の並列アルゴリズムからスポーンされたタスクでも利用できます。

図 8 から、OR の実行中はスレッドがビジー状態になることがありますが、まだ改善の余地があることが分かります。青色のタスクは、インテル® TBB を使用する入れ子の並列処理があることを示し、黒色の領域はコアがアイドル状態であったことを示します。OR は最も多くの時間を費やしているため、対応するタスクを確認してみると良いでしょう。すべてのタスクは、**図 8** にあります。クリティカル・パス上のタスクのみに注目する場合は、選択ベースのハイライト表示を使用してクリティカル・パス上のタスクを確認できます。

図 9 は、クリティカル・パス関連のタスクのみをハイライト表示したタイムライン・ビューです。入れ子のタスクの表示は、現在 FGA の試験的な機能で、手動インストルメンテーションが必要です。この図から、タイムラインの多くのタスクが OR ノードへマップできることが分かります。黒色の領域で示されるコアのアイドル時間は、OR の計算時間を短縮することで減らしたり、並列に実行できる追加のアルゴリズムをスケジュールしてコアをビジーに保つことができます。



9 実行トレースに表示されたクリティカル・パス

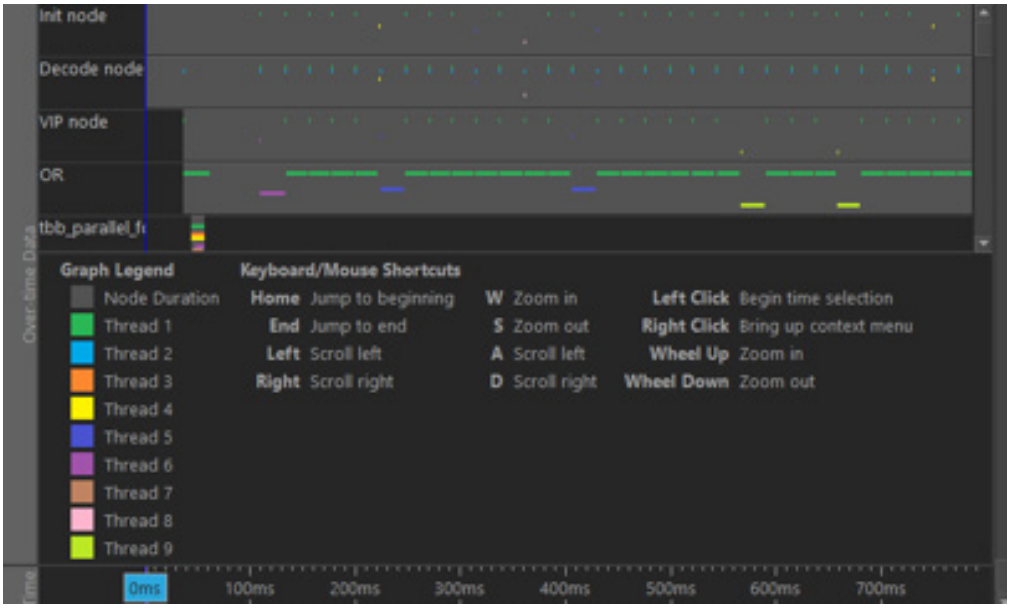
図 9 に表示されているフレームの OR ノードの実行トレースを調査してみましょう。このフレームを詳しく見ると、OR ノードで 4 つの入れ子の並列アルゴリズムが実行されていることが分かりました (**図 10**)。図 10 には、これらのアルゴリズムが効率良く実行されているかどうか判断するのに役立つ追加情報も提示されています。統計ビューでは、各ノードのパフォーマンス統計がグラフレベルでまとめられ、ノードごとのデータがテーブル形式で表示されます。サンプルの OR ノードの解析では、各ノードで実行されるアルゴリズムに注目します。**図 10** は、選択したフレームで実行された 4 つの並列アルゴリズムのアルゴリズムごとのデータです。OR::tbb149 と OR::tbb213 は比較的効率が良く、パフォーマンス・チューニングをそれほど必要としません。一方、OR::tbb261 と OR::tbb284 は効率が悪く、小さな多数のタスクを実行していることが原因として考えられます。これはロード・インバランスを引き起こし、並列処理の効率とスケーラビリティーを低下させる可能性があります。

Graph | Algorithms

Severity	For	Efficiency(%)	Task Count	Duration (ms)	CPU Time(%)	Other Time(%)	Task Size (min)	Task Size(max)	Active Elsewhere(%)
tbb_parallel_for(tbb1474)	Results(7)	83.87	103		83.87	16.13	0.263844	0.819408	1.70
tbb_parallel_for(tbb149)	Results(3)	94.54	46		94.54	5.46	0.317937	0.959297	3.70
	[Worker Thread]	100.00	19	9.485726	100.00	0.00	0.317937	0.830145	11.11
	[Worker Thread]	94.87	14	9.938140	94.87	5.13	0.509535	0.959297	0.00
	[Worker Thread]	88.75	13	9.341695	88.75	11.25	0.436504	0.940199	0.00
tbb_parallel_for(tbb213)	Results(4)	86.21	30		86.21	13.79	0.211302	0.737325	5.41
	[Worker Thread]	100.00	9	2.931293	100.00	0.00	0.211302	0.409358	21.63
	[Worker Thread]	86.06	9	3.085395	86.06	13.94	0.229712	0.485408	0.00
	[Worker Thread]	75.83	7	2.723886	75.83	24.17	0.277538	0.519274	0.00
	[Worker Thread]	82.95	5	2.914032	82.95	17.05	0.481382	0.737325	0.00
tbb_parallel_for(tbb261)	Results(3)	71.74	10		71.74	28.26	0.105216	0.241558	13.11
	[Worker Thread]	100.00	4	0.670851	100.00	0.00	0.105216	0.192787	39.33
	[Worker Thread]	47.35	2	0.497640	47.35	52.65	0.222739	0.232367	0.00
	[Worker Thread]	67.86	4	0.714158	67.86	32.14	0.109674	0.241558	0.00
tbb_parallel_for(tbb284)	Results(4)	48.84	6		48.84	51.16	0.067957	0.119980	12.71
	[Worker Thread]	100.00	2	0.194970	100.00	0.00	0.067957	0.119980	50.83
	[Worker Thread]	22.24	1	0.084998	22.24	77.76	0.084998	0.084998	0.00
	[Worker Thread]	24.74	1	0.094556	24.74	75.26	0.094556	0.094556	0.00
	[Worker Thread]	48.39	2	0.226341	48.39	51.61	0.082364	0.102569	0.00

10 OR::tbb149、OR::tbb213、OR::tbb261、OR::tbb284 のアルゴリズム統計を示すグラフ統計テーブル

インテル® TBB はワークスチールによりタスクがスケジュールされるため、特定のノードで実行するスレッドを確認してみましょう。FGA では、タイムラインをノードごとのビューに切り替えるだけで、簡単に確認できます (**図 11**)。OR ノードは、多くの場合同じスレッド (緑色) によって実行され、一時的に別のスレッドによって実行されていることが分かります。



11 ノードごとにグループ化されたタスク

タイムライン、アルゴリズム統計、グラフトポロジーに加えて、ノードごとのデータを表形式で表すことができます (**図 12**)。これにより、データを並べ替えて、最適化により最も効果が得られる領域を特定できます。行を選択すると、グラフビューで対応するノードがハイライト表示されます。

Graph | Algorithms

Severity	For	Node Name	Instance Count	In-degree	Out-degree	Total Time (ms)	Avg. Task Duration (ms)	Std. Dev.	Average Concurrency
g1	Results(14)								
g1::n5	Results(3)								
g1::n5::n8	Results(3)								
	Node	g1::n5::n8::n10	153	1	0	3.249428	0.021238	0.01	3.60
	Node	processFCWFrame	153	1	1	547.956726	3.581417	0.71	3.81
	Node	initFCWFrame	51	0	1	11.072482	0.217107	0.20	2.20
	Node	FCW init frame	51	0	1	143.948059	2.822511	8.20	2.73
	Node	g1::n5::n7	51	1	0	1.310975	0.025705	0.01	3.13
g1::n0	Results(4)								
	Node	LDW: Init Frame	51	0	2	10.974422	0.215185	0.11	3.86
	Node	LDW: Process Left	51	1	1	56.322594	1.104365	0.72	3.79
	Node	LDW: Process Right	51	1	1	41.114304	0.806163	0.65	3.67
	Node	LDW: Output Frame	51	2	0	0.991448	0.019440	0.01	3.62
	Node	Init node	52	1	1	1.019682	0.019609	0.01	1.00
	Node	Decode node	102	1	1	7.502239	0.073551	0.20	1.82
	Node	VIP node	51	1	4	2.115185	0.041474	0.02	1.30
	Node	g1::n15	51	1	1	1.561082	0.030609	0.03	3.85
	Node	g1::n16	51	1	1	1.055234	0.020691	0.01	3.66
	Node	g1::n17	51	1	1	1.357667	0.026621	0.01	3.90
	Node	g1::n18	51	1	1	1.276426	0.025028	0.02	3.51
	Node	OR	9792	1	1	6320.552246	0.645481	1.78	5.41
	Node	Copy to node	51	1	1	19.666084	0.385610	0.18	3.85
	Node	Display node	51	1	1	48.786430	0.956597	0.27	1.00
	Node	g1::n22	0	4	1	0.000000	0.000000	0.00	0.00
	Node	Draw overlay node	153	1	1	12.802725	0.083678	0.06	3.03

12 ノードごとの統計を示すグラフ統計テーブル

まとめ

FGA は、インテル® TBB のフローグラフを調査、デバッグ、解析するのに必要なツールセットを提供します。ADAS サンプル・アプリケーションへ適用することで、最適化すべき最重要ノードを素早く特定することができました。OR ユースケースでいくつかの入れ子の並列処理が使用されている一方、別の並列処理に利用できるアイドルコアがあることが分かりました。

intel Software

究極のツール

自然界の類まれなアスリートであるハチドリは 1 秒間に 200 回も羽ばたきます。ベクトル化、スレッドセーフなアルゴリズム、業界最先端のコンパイラーとパフォーマンス・チューニング・ツールにより、コードの優れたパフォーマンスを引き出して 187 倍のスピードアップ [1] を達成することができます。アプリケーションの可能性を引き出しましょう。

intel PARALLEL STUDIO XE

評価版 :
isus.jp/intel-parallel-studio-xe/

#PurePerformance

[1] 性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル ® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、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。詳細については、www.intel.com/benchmarks (英語) を参照してください。

詳細は、**software.intel.com/en-us/intel-parallel-studio-xe/details#configurations** (英語) を参照してください。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。

OpenMP* が大人の仲間入り

20 年を経て関連性をさらに強化

Barbara Chapman ニューヨーク州立大学ストーニーブルック校 教授兼ブルックヘブン国立研究所 コンピューター・サイエンスおよび数学部門ディレクター

OpenMP* は 1997 年に誕生しました。当時、

- Fortran がテクニカル・コンピューティングの主力言語でした。
- マルチコアはまだ専門用語として認識されていませんでした。
- コンピューター・ベンダーは、メモリーを共有する 2 つ以上のプロセッサー (SMP) を搭載したデスクトップ・システムの提供を開始しました。

1980 年代に見つかった共有メモリー・プラットフォームの技術上の制限により、SMP は最大 4 プロセッサーの並列システムを維持すると業界関係者は予測しました。しかし、大規模な分散共有メモリー (DSM) マシンが登場し、ソフトウェア DSM の活発な研究が行われ、Clay MTA* システムは異なる方式の大規模マルチスレッディングを実証しました。

Mats Brorsson (当時ルンド大学に在籍) と私は、1999 年に第 1 回 European Workshop on OpenMP* (EWOMP) を開催しました。OpenMP* 仕様の登場から 2 年経っていませんでしたが、多くの精力的な人々が集まりました。参加者の大部分は、すでに大規模 DSM プラットフォーム上で科学研究に OpenMP* を使用し、一部のケースでは驚くほどの結果を得ていたアプリケーション開発者でした。MPI と OpenMP* を組み合わせたハイブリッド・プログラミングの実験は、すでに興味深い話題として関心を集めていました。参加者は、OpenMP* インターフェイスに対する熱意が高く、OpenMP* プログラムが簡単に作成でき、生成される並列コードの振る舞いが理解しやすいことを評価していました。

その後多くのワークショップが開催され、OpenMP* に興味のある研究者や科学アプリケーション開発者が参加しました。業界関係者により、データベースからエンジニアリング製品設計にわたるさまざまなアプリケーション・コードが紹介されました。North American Workshop on OpenMP* Applications and Tools (WOMPAT) は、第 1 回が 2000 年に San Diego Supercomputing Center で開催され、利用経験、拡張提案 (Compaq* 製 NUMA システム上へのデータ配置機能を含む)、実装経験、パフォーマンス・モデリングに関するプレゼンテーションが行われました。OpenMP* をクラスターへ変換するための取り組みを含む OpenMP* に関する積極的な研究と開発が行われていた日本では、Workshop on OpenMP* Experiences and Implementations (WOMPEI) が開催されました。

一般に、これらのワークショップに企業や業界のアプリケーション開発者を呼び込むことは困難でした。ハードウェアの調達に関する決定が大規模 OpenMP* プログラムの要件に左右されるある主要ユーザーは、「問題なく動作していたので、OpenMP* の会議に出席する必要はありませんでした」と述べていました。業界の一部のユーザーは、全体的な計算の一部を並列化することで得られる、控えめまたは中程度のアプリケーション・スピードアップで満足していました。一方、コンピューター・サイエンティストは、共有メモリーの実用に関する課題に直面し、コンパイラーでの並列プログラム表現を提案するとともに、より効率的な変換の開発と実行時の手法の拡張を行いました。さらに、異なる機能により生じるオーバーヘッドを理解し、個々の実装の品質を追跡しやすくするため、ベンチマークを作成しました。

2005 年には、各地域で開催されていたさまざまなワークショップが International Workshop on OpenMP* (IWOMP) に統合され、毎年数日間、研究者と言語委員会のメンバーが一堂に会して、機能、実装、ツールに関する最新のアイデアを検討することになりました。IWOMP の貢献は、成功と失敗の体験レポート、新しい機能と実装手法に関する提案、オープンな課題を克服するための代替手段の模索、新しいハードウェア・テクノロジーの検討、既存の構文のパフォーマンス向上戦略など、OpenMP* 仕様の進化に影響するあらゆるトピックにわたります。

OpenMP* 仕様の初期バージョンは、共有メモリー・ディレクティブの経験を慎重に評価して作成されましたが、今日の要求に対応するためには、より包括的なアプローチが求められます。幸い、科学アプリケーション開発者とコンピューター・サイエンティストは、OpenMP* の誕生以来、その拡張に活発に取り組んでいます。OpenMP* が進化し続けるためには、彼らの取り組みが不可欠です。しかし、当初は、彼らが OpenMP* 仕様の改善に直接貢献することは困難でした。ARB ベンダーは、OpenMP* の利益を最優先に考える者のみが重要な決定にかかわれるように、学術研究者向けの安価なメンバーシップを意図的に提供していなかったのです。これは、広範なコミュニティー参加による仕様策定の欠点が明らかになった High Performance Fortran (HPF) 仕様の商業的失敗の再現を懸念したためです。

既存の ARB メンバーの強い後押しもあり、時間と専門知識の提供に熱心な研究者の参加を可能にするため、私は後に ARB に参加することになる cOMPunity という組織を設立しました。cOMPunity の初期メンバーには、EPCC の Mark Bull 氏、CEPBA の Eduard Ayguade 氏、アーヘン工科大学の Dieter an Mey 氏、筑波大学の佐藤 三久氏、パデュー大学の Rudi Eigenmann 氏が含まれていました。このグループの注目すべき点は、彼らの大部分が依然として何らかの形で関与していることに加えて、彼らの所属機関の地理的多様性です。現在では、研究機関が ARB に直接参加することが非常に簡単になり、継続的な貢献に興味のある方は直接メンバーになることができます。OpenMP* のこれまでの歩みは、研究者も業界標準に継続的な興味を持ち、好影響を与えられることを示しています。

OpenMP* が導入されるコンピューター・システムは、1997 年から大きく変わりました。少数の共有メモリー型プロセッサーから、マルチコア革命とそれに対応するための明示的な並列処理を経て、ヘテロジニアス・メニーコア・コンピューティングの時代に突入しました。現代のコンピューティング・システムの多種多様なコア構成、性能、リソース共有と接続性の性質の違いは、ターゲット・プラットフォームの拡大をもたらしました。コア数は増加を続け、メモリーシステムもベンダーがサイズ、レイテンシー、帯域幅の制約に対応するため大きく変わりつつあります。また、この 20 年間で OpenMP* を使用するアプリケーションの性質も変わりました。並列処理が主に Fortran の科学計算コードで利用されていた時代から、C、C++、Fortran、またはそれらの組み合わせでさまざまなアプリケーションが並列化される時代になりました。OpenMP* はこれらすべての変更に対応してきました。

当初から、OpenMP* 開発者は急速に変化する世界との関連性を維持するという課題に取り組み、OpenMP* 仕様は動的に変化してきました。有能で先見の明のあるリーダーの下、より入念な仕様による C/C++ のバインドの追加から、今日の最も要件が厳しいアプリケーションとシステムをサポートする機能まで、環境に適合するため進化を続けています。言語委員会は、さまざまなソースから、並列アプリケーション開発に関する新しく優れたアイデアを取り上げ、統合してきました。

当初から、OpenMP* 開発者は急速に変化する世界との関連性を維持するという課題に取り組み、OpenMP* 仕様は動的に変化してきました。

アプリケーション開発者にとって、OpenMP* の最大の利点の 1 つは、その移植性の高さです。いったん機能を理解すれば、さまざまなベンダー・プラットフォーム向けにアプリケーションを適合させることができます。OpenMP* には、今日利用できる多種多様な並列コンピューティング・システムにおいて、アプリケーションのニーズに確実に対応するための多大な労力が費やされています。言語委員会は、共有メモリーを使用せずにヘテロジニアス・コアで実行するといった困難な課題を解決するだけでなく、それに応じてミッション・ステートメントも変更しました。SIMD コンポーネントをサポートする機能が追加され、表現力を強化し、広範なコーディングの問題に対応できるようにタスク機能が拡張されました。アプリケーションが新しいメモリーを効率良く利用できるようにコード内のデータの局所性を高めることを支援する機能が求められていますが、言語委員会ではすでにこれに取り組んでいます。

従来、大規模なコンピューターで効率良くコードを実行するため多大な労力を費やしてきた大規模アプリケーション開発者からは、パフォーマンスの移植性について新たな要求が上がっています。その背景として、大規模アプリケーションも、コードを若干変更するか全く変更することなく、多様な並列コンピューティング・システムで効率良く実行できるべきだという期待が高まっています。これは、あるシステムから別のシステムへコードを移行するコストの面から、非常に合理的な要求であると言えます。しかし OpenMP* は、動作の透明性が重要であると考えられていた時代に生まれ、多くのユーザーは引き続きこの性質を重視しています。OpenMP* が規範的な並列プログラミングだけでなく、パフォーマンスを移植可能な並列アプリケーションの作成にも役立つようにするには、大変な労力が必要になります。

OpenMP* が最も優れている点は、20 年経っても関連性を維持していることです。実際、OpenMP* の利用は拡大しています。そして、すべての主要プラットフォームで実装されているため、広範に利用することができます。委員会に出席するユーザー代表も増えています。アプリケーション開発者からは、特定の、時には非常に厳しい要件を求める声もあります。米国エネルギー省のエクサスケール・コンピューティング・プロジェクトなどの主要機関は、OpenMP* が彼らの要求を満たすことができるように、その取り組みをサポートしています。仕様を進化させるための取り組みは、ベンダー代表者に加えて、長期間にわたって貢献している研究機関と大学からの参加者で構成される活発なチームによって推進されています。コンパイラー・ベースのプログラミング・インターフェイスとしては、OpenMP* は規模が大きいと言えます。

しかし、まだ取り組むべき課題はたくさんあります。そして、複雑さにもかかわらず、高品質な実装を開発し、迅速に提供する必要があります。アプリケーション開発者が、ほかのプログラミング・インターフェイスで記述されたコードを OpenMP* へ移行できるようにしなければなりません。大規模でハイパフォーマンスな電力効率の良いコードの開発を支援するツールも必要です。使いやすさと保守の容易さを重視するユーザーも引き続きサポートしなければなりません。範囲が拡大し、複雑さが増すのに伴い、異なるレベルのユーザー向けトレーニング・プログラムを拡張する必要性もあります。OpenMP* アプリケーション開発の高度化に伴い、研究者、開発者、トレーナー、ユーザーから成る活発なコミュニティーのみがこれらの要求を満たすことができます。そして、OpenMP* にはそのようなコミュニティーがあります。

OpenMP*
1997 年から HPC を支援

詳細
(英語)

intel Software

将来へ向けての秋の技術ウェビナー

スキルアップ
エキスパートからの回答
新しい開発分野への参入

10 月 25 日 9 a.m (米国太平洋時間)	**クラウドおよびエッジ・アプリケーション向けの可逆データ圧縮コードの高速化** インテル® IPP の高度に最適化された可逆データ圧縮関数により システム・パフォーマンスを改善する方法を見つけてください。
11 月 1 日 9 a.m (米国太平洋時間)	**並列プログラミング標準の最新情報 : MPI、OpenMP*、インテル® TBB** これらの実績ある並列プログラミング標準について、 それぞれの長所と解決できる問題の種類を含む概要を提供します。
11 月 8 日 9 a.m (米国太平洋時間)	**さらに効率良く高速でスケーラブル: エクサスケールへの行進** 2018 年にインテル ® MPI ライブラリーは次世代の MPICH (アルゴンヌ国立研究所で使用されているエクサスケール・ソース) へアップグレードします。その重要性が分かります。

今すぐ登録 (英語) >

アーカイブを参照 (英語) >

ソフトウェア開発者による FPGA の利用

自動車、ネットワーク、クラウド・コンピューティングの効率とパフォーマンスを向上

Bernhard Friebe インテル コーポレーション FPGA ソフトウェア・ソリューション・マーケティング部門シニア・ディレクター
James Reinders HPC エンスージアスト

今こそ、フィールド・プログラマブル・ゲート・アレイ (FPGA) (**図 1**) について調べる良い時期です。FPGA の新しいプログラマビリティーにより、コンピューティングの新しい時代が幕を開けようとしています。世界中でデータが爆発的に増加しているため、FPGA の柔軟性を備えた、カスタムシリコン設計による電力効率の良いコンピューティングに対するニーズが非常に高まっています。

この記事では、すべての分野のソフトウェア開発者が FPGA を利用するためにすべきだと我々が考えていることを、次の 3 つに注目して述べます。

1. 自動車
2. ネットワーク (例 : 5G)
3. エンドツーエンドのクラウド・コンピューティング (例 : データセンター)

1 インテル® Stratix® 10 FPGA

データセンターでの利用については詳しく取り上げ、アクセラレーション・スタックとオープン・プログラマブル・アクセラレーション・エンジン (OPAE) について述べます。完全なソリューション・スタックが FPGA IP (FPGA を使用するために実装されるライブラリー・ルーチン) のカプセル化によって実現されるマシンラーニングなどの特定の分野については触れません。ただし、インテル主導の新しく使いやすい FPGA プログラミングについて詳しく説明されているいくつかのリンクを提供します。

FPGA とは?

FPGA は、本質的には回路設計が描かれるのを待っている白紙のハードウェアです。必要なデジタル回路を描き、コンパイルして (FPGA 開発者はこれを合成と呼ぶ) 設定ファイル (ビットファイル) を生成し、FPGA にロードします。いったんロードされると、FPGA は設計したデジタル回路どおりに動作します。

FPGA には、プログラムを恒久的に残しておくことができません。電源がオフになると、プログラム (ビットファイル / ビットストリーム) は失われます。ほとんどのシステムでは、電源がオンになったときに、FPGA ボードのファームウェアからまたはホスト・プロセッサーによってプログラムから FPGA がロードされます。FPGA は、いつでも変更したいときにリロードできます。

FPGA の高度な並列アーキテクチャーにより、FPGA で計算を実行することで、ソフトウェアと比較してパフォーマンスとスループットが向上し、電力消費とレイテンシーが軽減されます。関数をハードウェアで実現できるように設定し、プログラムで使用することを想像してみてください。後述しますが、FPGA 業界で最も注目されているトピックは、FPGA プログラミングをソフトウェア開発者が利用しやすいものにすることです。インテル主導でこの実現に向けた取り組みが行われています。

データの爆発的増加

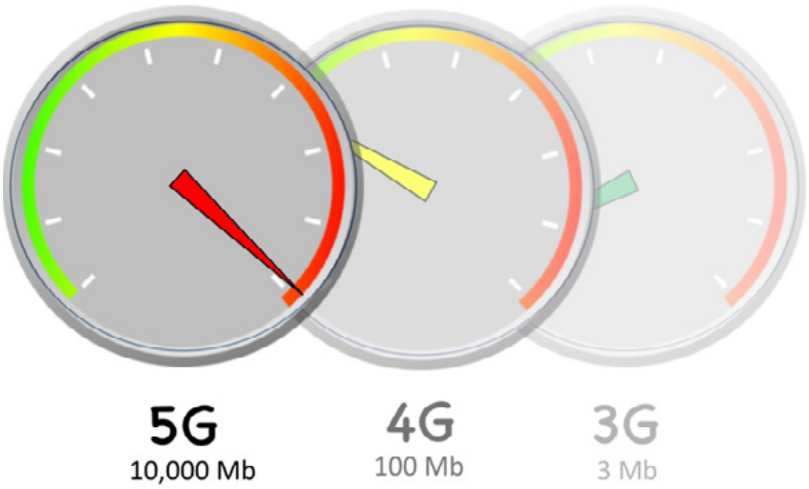
スピードアップ、超低レイテンシー、電力効率、非常に高い柔軟性を提供するためインテル® FPGA が鍵となる 3 つの例は、自動運転と運転支援、高速ネットワーク (5G とソフトウェア定義のネットワーク (SDN) を含む)、クラウド・コンピューティング (ハイパフォーマンス・ディープ・ニューラル・ネットワーク (DNN 推論) を含む) です。3 つの例すべてにおいて、ビッグデータは FPGA の並列処理アーキテクチャーが重要であることを示す理由の一部です。

自動車両

自動運転を含む自動車両は、車両側での計算、ネットワーク、データセンターを組み合わせることで実現可能となります。自動車両は、安全にナビゲーションするため、膨大な量のデータを生成し、使用します。1 秒あたり約 1GB のリアルタイム処理が発生する一方で、安全上の理由からセンサーが感知してから反応するまでの時間は 1 秒未満でなければなりません。例えば、2018 年モデルの **Audi* A8 の自動走行システム** (英語) には、インテルのプロセシング・パワーが採用されています。Audi* の設計では、インテル® FPGA で自動運転車のオブジェクト融合、マップ融合、パーキング、プリクラッシュ、プロセシング、および機能的な安全面を処理しています。

ネットワークと 5G

帯域幅は 1000 倍、デバイスは 100 倍、エンドツーエンドのラウンドトリップは 1ms に、ネットワークは急速に拡大しています。これらの要求に応えるためには FPGA が不可欠です。100 ギガビット・イーサネットをサポートするデバイスの採用が進む中、IEEE 802.3 の 400 ギガビット・イーサネットのスタディーグループ (2013 年に発足) は、今年中に 400 ギガビット・イーサネットが標準化されると予測しています。5G 標準化の取り組みは順調に進んでおり、ワイヤレス・ネットワークで転送されるデータが有線ネットワークを超える日も近いでしょう (**図 2**)。



2 5G への移行

クラウド・コンピューティング

IDC の報告によると、世界中の総データ量は 2013 年には 4.4 ゼタバイト、2015 年には 8.6 ゼタバイト、そして 500 億台のデバイスがインターネットに接続されると予測される 2020 年には 44 ゼタバイトに達すると見られています。クラウド・コンピューティングは、データセンター、エッジ、そしてそれらの間のあらゆるもので必要とされ、スケール、スループット、1 ワットあたりのパフォーマンス効率、柔軟性、低レイテンシーが求められます。これらの要求に応えるためにも、FPGA が不可欠です。

コンピューティングの必然性: さらなる並列化、低消費電力、柔軟性

このような大量のデータを理解するには、意思決定を自動化し、コネクテッド・デバイスに関するリアルタイムの詳細を把握して、ユーザーがインタラクティブかつ直感的にデータを利用できるようにする必要があります。高速計算なしでは、多くのアプリケーション (人工知能など) のスケールアウトは実用的ではなくなります。新しいコンピューティング要件では、さらなる並列化、低消費電力、そしてアクセラレーターではこれまで見られなかった柔軟性が求められます。この要求に応えるため、エッジからクラウドまで広範にわたるハードウェア・プラットフォームは、CPU とアクセラレーターを組み合わせて進化してきました。FPGA は、並列性が高く、電力効率が良く、再プログラム可能なヘテロジニアス・コンピューティング・プラットフォームへのトレンドにおいて重要な役割を果たします。つまり、FPGA はソフトウェアのプログラマビリティーとハードウェアのパフォーマンスの両立を可能にします。ただし、FPGA のプログラミング・モデルは、一般にハードウェア中心です。FPGA がコンピューティング環境の標準コンポーネントになり、ハードウェアがソフトウェア定義されることをユーザーが期待するようになるにつれ、ハードウェア開発者だけでなく、ソフトウェア開発者も FPGA にアクセスできるようにする必要があります。

ソフトウェア向けの FPGA プログラミング

FPGA は、ハードウェア設計の問題を解決するため長年にわたって使用されてきました。FPGA のプログラミングは、ソフトウェア開発のプログラミング言語ではなく、ハードウェア設計者に馴染みのある方法で行われていました。新しい FPGA 設計は、ハードウェアを置き換えるだけでなく、新しいソフトウェア開発ツールと併用することで、ソフトウェア開発をサポートしており、ソフトウェア開発者は FPGA プログラミングを検討する価値があります。

この変化は、GPU プログラミング手法の発展過程に似ていますが、FPGA は特定の用途 (例えば、グラフィック処理) 向けに設計されているわけではないため、GPU よりも柔軟性があり、これまでにないようなコンピューティングの新しい時代へと導いてくれるでしょう。

FPGA 向けの完全なソリューション・スタック

FPGA をプログラマブル・アクセラレーターとして利用してきた先駆者たちは、わずかなツールを頼りに長い道のりを歩んできました。彼らは、Verilog* や VHDL* などのハードウェア記述言語 (HDL) を使用してプログラムすることで、自力でハードウェア設計に取り組んできました。数年前、FPGA 開発に関心のある人を調査したとき、「FPGA 開発者のコミュニティーがあり、[さまざまな FPGA] フォーラムはそれぞれ活発に活動しており、多くの文献もあります。しかし、コミュニティー参加者のほとんどはハードウェア・エンジニアであり、ソフトウェア開発者は質問の回答を得るのに苦労しているでしょう。」と言うコメントがありました。この状況は変わりつつありますが、このコメントは非常に重要です。調査対象者のうち、FPGA に関心を示しつつも、Verilog* や VHDL* でプログラミングすることに抵抗がないと答えたのは、10 人に 1 人未満だったからです。さらに、ソフトウェア開発者の 90% 以上は C、C++、または OpenCL* の知識があり、75% 以上はライブラリーを介して FPGA 機能を使用したいと述べていました。このことから、インテルの次世代 FPGA ツールが C、C++、OpenCL*、そしてライブラリーに注力しているのは理にかなっています。

これまで通り、ソフトウェア・スタックはハードウェア・フローに依存し続けますが、完全なソリューション・スタック (**図 3**) は、効率良くソフトウェア・フレンドリーなインターフェイスを提供するプログラミング・レイヤーを備えています。これらの新しい手法は、従来の FPGA ツールを放棄するものではなく、従来の FPGA ツールは今後も改善され、利用され続けるでしょう。ただし、FPGA の完全なソリューション・スタックにより、FPGA プログラミングは CPU プログラミングと似たようなものになるでしょう。アーキテクチャーの詳細な知識がコンパイラーやライブラリーに組込まれ、ほかの分野 (つまり、コンピューター・アーキテクチャー以外の科学、エンジニアリング分野) の専門知識を持つプログラマーが利用できるようになります。FPGA IP を利用して FPGA を使用するアプリケーション開発者と FPGA IP を開発するアプリケーション開発者では、懸念事項が異なります。完全なソリューション・スタックは、FPGA プログラミングに関するこの新しい考え方を可能にします。

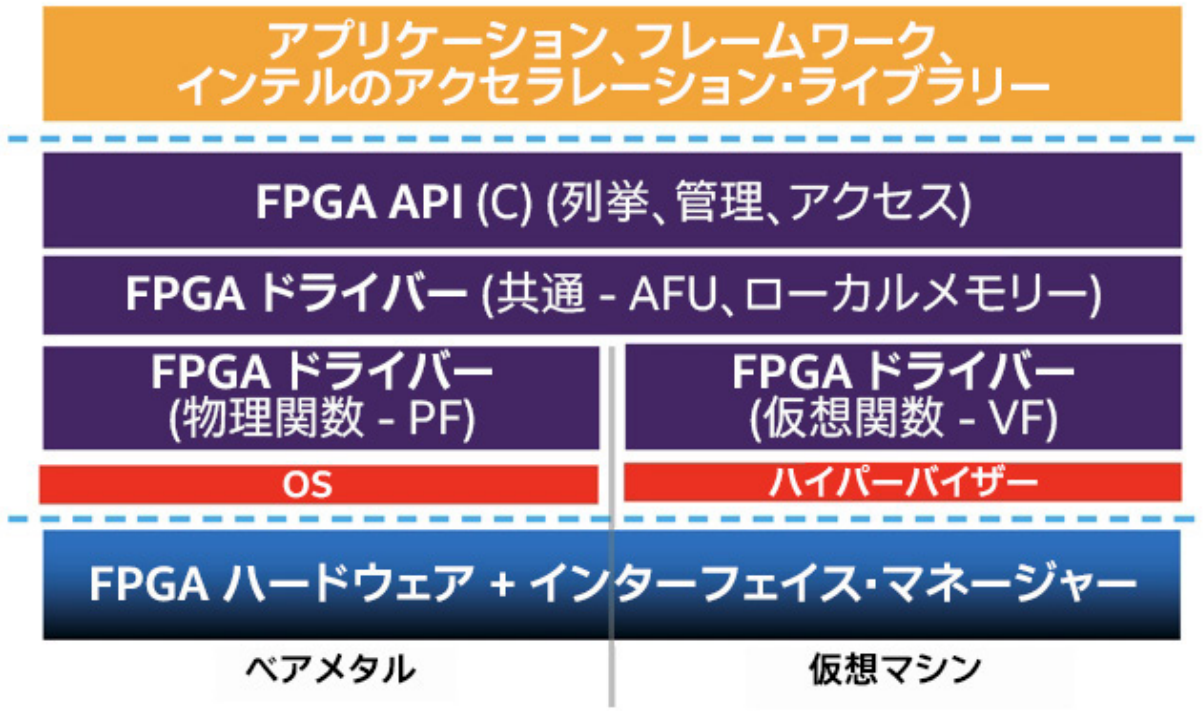
3 完全なソリューション・スタックはソフトウェア開発者とハードウェア開発者の両方に対応

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

FPGA が統合されたインテル® Xeon® プロセッサー向けアクセラレーション・スタック

インテルは、FPGA 向けの完全なソリューション・スタックを実現するため、FPGA の設計、プログラミング・ツール、ライブラリーに投資しています。クラウド・コンピューティング、特にデータセンター分野でのインテルの取り組みについて述べたいと思います。

FPGA が統合されたインテル® Xeon® プロセッサー向けアクセラレーション・スタック (**図 4**) は、インテルが設計および配布するソフトウェア、ファームウェア、ツールの堅牢なコレクションで、データセンターのワークロードを最適化するインテル® FPGA の開発と配備を容易にします。ソフトウェア開発者が独自のソリューションの付加価値に集中できるように、最適化および簡素化されたハードウェア・インターフェイスとソフトウェア API を提供します。インテル® FPGA 用の共通の開発インターフェイスにより、かつてないコードの再利用が可能です。開発期間をさらに短縮するため、一部のドメインではシステムに最適化されたリファレンス・ライブラリーが提供され、FPGA 初心者のアプリケーション開発者でもインテル® FPGA を使用してパフォーマンスを向上できます。



4 FPGA の抽象化と管理

ソフトウェアで FPGA を使用するための洗練されたソリューション

インテルは、通常のライブラリー呼び出しにより FPGA を利用できるようにしようとしています。初期の FPGA ユーザーも同様のアプローチを採用しましたが、移植可能な標準規格がありませんでした。次の標準フレームワーク (**図 5**) について考えてみます。

- **アプリケーション開発者**には、必要な機能を指定する標準フレームワークまたは SDK を提供し、その機能の FPGA により高速化されるバージョンを利用できるようにします。開発者は、引き続き C、C++、Fortran、または C インターフェイスを備えた高水準言語 (Python* など) でプログラミングできます。そのため、オープンソースおよび商用ツール開発者のさまざまなコミュニティーを利用できます。
- **システム**には、配備する FPGA にアクセスし、使用する標準の方法を提供します。例えば、データセンターのオペレーターは、アクセラレーション・スタックのオープン・プログラマブル・アクセラレーション・エンジン (OPAE) を利用して、データセンター環境で FPGA を管理できます。
- **FPGA コード開発者**には、高速化したルーチンをパッケージおよび共有する標準の方法を提供します。FPGA コード開発者は、OpenCL* またはその他の方法 (例えば VHDL* や Verilog* などの HDL/RTL) を使用して FPGA ルーチンを細かくチューニングし、エクスポートします。

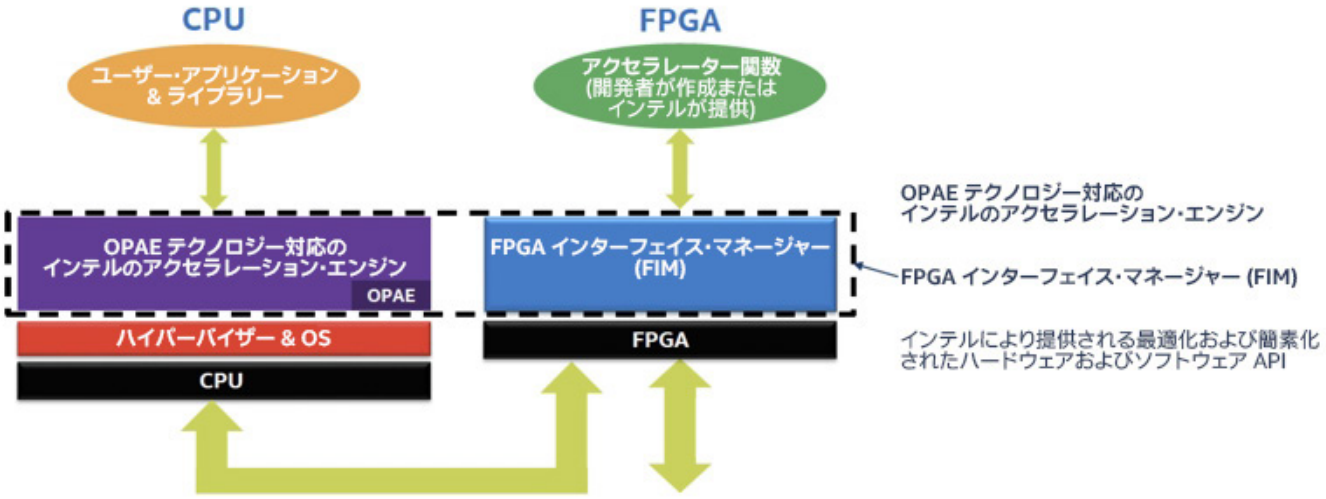
システム所有者は、フレームワークを使用して、アプリケーションが必要な FPGA パッケージをロード / アンロードできるようにします。アプリケーション開発者は、すべてのプログラミングを高水準言語で行う一方、フレームワークを使用して FPGA パッケージを要求し、使用します。FPGA プログラマーは、低水準言語で FPGA モジュールを作成し、アプリケーション開発者向けにパッケージします。

オープン・プログラマブル・アクセラレーション・エンジン (OPAE)

データセンターでこのアクセラレーション・スタックを実現できるように、インテルはオープン・プログラマブル・アクセラレーション・エンジン (OPAE) (**図 6**) の作成に協力しました。OPAE はプロセッサー上で動作し、FPGA の再設定プロセスのすべての詳細を処理します。また、FPGA 用のルーチンの開発に使用可能なライブラリー、ドライバー、サンプルプログラムも提供します。OPAE は、ハードウェア固有の FPGA リソースの詳細を抽象化して、製品の世代間に一貫性をもたらします。ソフトウェア・オーバーヘッドとレイテンシーを最小限に抑えるように設計されており、ライトウェイトのユーザー空間ライブラリー (libfpga) を提供します。OPAE は、仮想マシンとベアメタルの両方のプラットフォームをサポートします。

OPAE の C API は、ソフトウェア・システムに FPGA へのインターフェイスを提供します。API には、デバイス固有またはプラットフォーム固有の拡張があり、ターゲット・アーキテクチャーの特定の機能をモデル化できます。例えば、インテルは、インテル® FPGA IP ライブラリーの一部として組込まれている、FPGA が統合されたインテル® Xeon® プロセッサーのコヒーレント・メモリー・インターコネクトを介して低レイテンシー通知メカニズムを利用可能にするプラットフォーム固有の API 拡張を作成しました。

5 FPGA 向けインテル ® Xeon® プロセッサーのアクセラレーション・スタック



6 アクセラレーション環境

インテル® FPGA の主要能力は、FPGA の残りの部分は動作したまま、一部を動的に再設定できることです。つまり、必要に応じて、実行時に FPGA の一部を再設定して、異なる機能を実装できます。OPAE は専用のビットストリームを介して、部分的な再設定を利用します。これは、コンパイル済み FPGA プログラムに特有のものです。FPGA インターフェイス・マネージャー (FIM、別名ブルー・ビットストリーム) には、PCIe* IP コア、CCI-P ファブリック、オンボード・メモリー・インターフェイス、マネージメント・エンジンを含む FPGA アクセラレーターをサポートするためのロジックが含まれています。アクセラレーターの機能ユニット (AFU、別名グリーン・ビットストリーム) は、カスタム機能のコンパイル済みバージョンです。AFU は、FPGA ロジックで実装されている高速化された計算ルーチンです。パフォーマンスを向上するため、OPAE は、AFU をインテル® FPGA へオフロードします。OPAE は同じ FPGA 上で複数の AFU スロットをサポートします。そのため、AFU は、アプリケーションがロードして使用可能な FPGA により高速化された関数と考えることができ、同時に複数のライブラリー (AFU) を利用できます。

インテルは、AFU シミュレーション環境 (ASE) もサポートしており、インテル® QuickAssist Accelerator Abstraction Layer Software Development Kit でコード開発およびシミュレーション・ツール・スイートを提供しています。この SDK を利用することで、AFU シミュレーションで OPAE 対応ソフトウェア・アプリケーションをテストできます。この SDK は、一貫したトランザクションレベルのハードウェア・インターフェイスとソフトウェア API を提供することで、ユーザーがプロダクション品質の AFU とホスト・アプリケーションを開発およびデバッグして、その後変更なしで実際の FPGA システム上で実行できるようにすることを目的としています。

FPGA IP ライブラリー : 高速化ライブラリー

FPGA のエキスパートは引き続き RTL/HDL を使用して高度に最適化されたコードを記述する機会があると前述しましたが、FPGA の新機能では、そのような専門知識を FPGA エキスパートではないアプリケーション開発者が標準的な方法で利用できるように、ライブラリーとしてパッケージすることができます。そのようなライブラリーを FPGA IP ライブラリーと呼び、その可能性は無限です。FPGA IP ライブラリーは、マシンラーニング、ゲノミクス、データ解析、ネットワーク、自動運転、ビーム形成、または FPGA を使用してハードウェアで行うことで優れた並列性、低レイテンシー、柔軟性、および電力効率が得られるあらゆるものを含む、さまざまな分野のアプリケーションを支援します。驚くべきことではありませんが、インテルは、FPGA を使用するための基本的かつ重要な機能を支援する FPGA IP ライブラリーの作成を主導しています。インテル® FPGA Intellectual Property (IP) ライブラリーは、我々を支援し、FPGA IP ライブラリーの例、そして完全なソリューションに簡単に統合できる例を提供します。

インテル® FPGA IP ライブラリー

インテル® FPGA IP ライブラリーは、計算環境でインテル® FPGA IP リソースに抽象化を提供する、ライトウェイトのユーザー空間ライブラリーです。インテル® FPGA IP デバイスをサポートするドライバースタックをベースに構築されており、ハードウェアとオペレーティング・システム (OS) に固有の詳細を抽象化して、インテル® FPGA IP リソースを、ホスト上で実行するソフトウェア・プログラムからアクセス可能な機能セットとして利用できるようにします。

これらの機能には、デバイスで事前に設定されている高速化ロジックに加えて、デバイスを管理および再設定する関数が含まれています。ライブラリーを使用することで、ユーザー・アプリケーションはインテル® FPGA IP による高速化を透過的かつシームレスに利用できます。

統一された C API を提供することで、インテル® FPGA IP ライブラリーは、1 つ以上のインテル® FPGA IP デバイスを搭載したシングルノード・システムからデータセンターにおける大規模配備まで、さまざまな統合および配備モデルに対応します。例えば、単純なユースケースとして、インテル® FPGA IP PCIe* デバイスを搭載したシステムで動作するアプリケーションの特定のアルゴリズムを、インテル® FPGA IP で簡単に高速化することができます。対照的な例として、データセンターのリソース管理とオーケストレーション・サービスは、この API を利用してインテル® FPGA IP リソースを見つけ、選択して、高速化が必要なワークロードに割り振ることができます。

インテル® FPGA SDK for OpenCL*

インテル® FPGA 上で動作するカスタム・アクセラレーター関数を作成したい FPGA 開発者向けに、アクセラレーション・スタックはインテル® FPGA SDK for OpenCL* を提供しています。OpenCL* (Open Computing Language) は、従来のハードウェア FPGA 開発フローを抽象化し、より迅速な高レベルのソフトウェア開発フローを使用する、業界標準の C ベースのプログラミング言語です。インテル® FPGA SDK for OpenCL* を利用することで、高レベルのソフトウェア・フローを使用して、C で FPGA 設計を開発できます。OpenCL* の C アクセラレーター・コードを、x86 ベースのホストで数秒エミュレートして、特定のアルゴリズムのパイプライン依存関係に関する情報を含む詳細な最適化レポートを取得したり、仮想 FPGA ファブリック上でアクセラレーター・カーネルのプロトタイプを数分で生成できます。

まとめ

インテルは、高度にプログラマブルな FPGA により FPGA 分野を先導しています。今こそ、FPGA のプログラミングについて調査すべきです。ムーアの法則は FPGA 設計者に多くの処理能力を提供し、設計者はトランジスターを使ってソフトウェア・プログラマビリティーを考慮した機能を追加してきました。OpenCL* は、FPGA 開発者とアプリケーション・ニーズを結び付けました。そして、インテルは、システム所有者、アプリケーション開発者、FPGA プログラマーが標準的な方法でやり取りできるフレームワークをリリースしました。これにより、共通の開発インターフェイス、ツール、IP からインテル® FPGA を使用できるため、FPGA の利用とコードの再利用が容易になります。

関連情報

- **インテルの FPGA プログラミングのページ**
- **FPGA 向けインテル® Xeon® プロセッサーのアクセラレーション・スタックに関するインテルの記事** (英語)
- **オープン・プログラマブル・アクセラレーション・エンジン (OPAE)** (英語)
- **ホワイトペーパー: The Open Programmable Acceleration Engine (OPAE)** (英語)
- **インテル® FPGA SDK for OpenCL***
- **インテル® FPGA で実現する人工知能 (AI)**
- **インテル® FPGA 製品概要**
- **ビデオ: Inside Microsoft's FPGA-Based Configurable Cloud by Mark Russinovich, CTO, Azure** (英語)
- **インテルの自動運転に関する報道資料 (多数のリンクあり)** (英語)
- **Forbes Insights: The Coming Data Avalanche – and How We'll Handle It** (英語)
- **Accelerating the Future: The Economic Impact of the Emerging Passenger Economy** (英語)
- **FPGA スタートガイドおよびその他の資料**
- **C++ ベースのインテル® HLS コンパイラーで FPGA 開発を高速化**

BLOG HIGHLIGHTS

インテル® コンピューター・ビジョン SDK—概要

MARTIN K. INTEL CORPORATION

インテル ® コンピューター・ビジョン SDK Beta がリリースされました。この SDK は、コンピューター・ビジョン・アプリケーションを正確かつ高速に作成できるように開発者を支援します。私は Tudor Panu から、インテル® コンピューター・ビジョン SDK とは何か聞き、いくつかの機能のデモを見せてもらい、開発者がコンピューター・ビジョン・プロジェクトを活用する方法を聞く機会がありました。その様子はこちらから (英語) ご覧いただけます。
このブログでは、インテル® コンピューター・ビジョン SDK について私が学んだことと、コンピューター・ビジョン・アプリケーションの開発に役立ついくつかのリソースを皆さんと共有したいと思います。

この記事の続きはこちら (英語) でご覧になれます。>

コードの現代化によるパフォーマンス、移植性、スケーラビリティーの向上

インテル® Parallel Studio XE 2018 の新機能

Jackson Marusarz インテル コーポレーション テクニカル・コンサルティング・エンジニア

HPC クラスター、リモートクラウド、ローカル・ワークステーション、あるいはそれ以外であっても、コンパイラー、ライブラリー、ツールを備えた**インテル® Parallel Studio XE** は、生産性とアプリケーション・パフォーマンスの向上を支援します。インテル® Parallel Studio XE の最新リリース (バージョン 2018) でもインテルは革新を続けています。新しいハードウェアとプログラミング言語標準のサポートに加えて、成長するテクノロジーと新しい環境に対応するため、いくつかの新機能が追加されています。

ハードウェア

インテル® Parallel Studio XE に含まれるツールは、**インテル® Xeon Phi™ プロセッサー**と**インテル® Xeon® スケーラブル・プロセッサー**を含む最新のハードウェアをサポートします。インテル® Xeon Phi™ プロセッサーの 60 を超えるコア (200 を超えるスレッド)、およびこれらのプラットフォームでサポートされる新しい**インテル® AVX-512** ベクトル化命令を利用して、将来も通用するアプリケーション・パフォーマンスを実現できるように支援します。

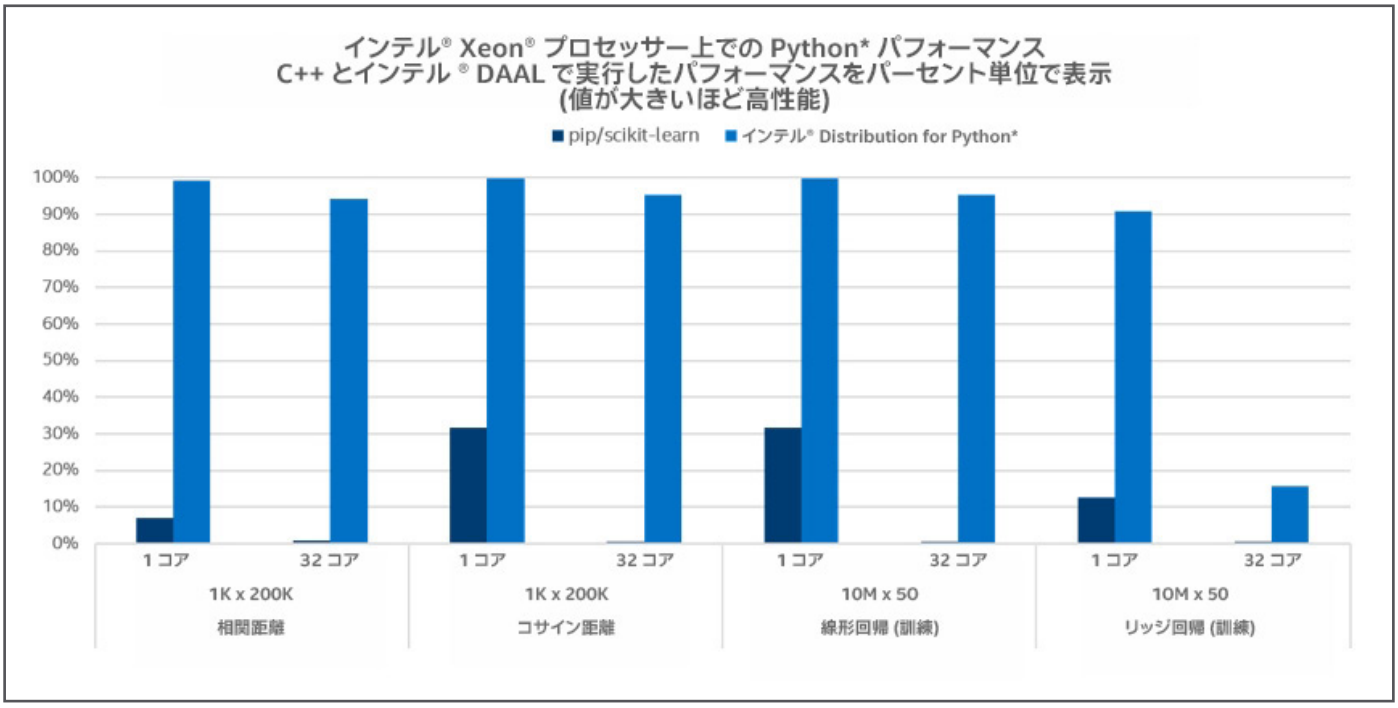
コンパイラー

インテル® C/C++ および **Fortran** コンパイラーがアップデートされ、パフォーマンスを損なうことなく、多くの最新の標準規格に対応しています。インテル® C/C++ コンパイラーは、C11 と C++14 の完全サポートに加えて、C++ 17 の初期サポートを提供しています。インテル® Fortran コンパイラーは、Fortran 2008 の完全サポートと Fortran 2015 の初期サポートを提供しています。増え続けるコア数とベクトルレジスターの幅を活用するため、このリリースでは C++ STL を並列およびベクトル実行する Parallel Standard Template Library (Parallel STL) と、OpenMP* 5.0 ドラフトの初期サポートが追加されました。インテル® コンパイラーは、Makefile のいくつかの変数を変更するか、Microsoft* Visual Studio* や Xcode* などの環境に統合するだけで簡単に使用できます。**無料の評価版**でアプリケーションのパフォーマンスが向上するかお試しください。

ハイパフォーマンスな Python*

インテル® Distribution for Python* の最新リリースは、多くの一般的なライブラリーとアルゴリズム (特にデータ解析) をスピードアップします (**図 1**)。SciPy* ライブラリーと NumPy* パッケージの新しい最適化は、最もよく使用されているマシンラーニング・パッケージの 1 つである scikit-learn をスピードアップします。インテル® Distribution for Python* は、pyDAAL インターフェイスを介して**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)** を利用します。scikit-learn のいくつかのアルゴリズムでは、最大で 140 倍ものスピードアップが達成されています。

このリリースには、**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)** により高速化された **OpenCV*** (英語) パッケージも含まれています。OpenCV* は、さまざまな分野でコンピューター・ビジョンに使用されている人気のライブラリーです。



1 インテル® Xeon® プロセッサー上での Python* パフォーマンス - C++ とインテル® DAAL で実行したパフォーマンスをパーセント単位で表示 (数値が大きいほど高性能)

パフォーマンス・ライブラリー

インテル® Parallel Studio XE には、ハイパフォーマンス・ソフトウェアの作成を容易にする、いくつかのライブラリーが含まれています。この最新リリースでは、全体的なアルゴリズムと使いやすさが向上しています。

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 は、いくつかの最も一般的な数学ルーチンを最新のプロセッサー機能を利用するように高度に最適化およびチューニングしたバージョンを実装します。バッチおよびパックド操作に対応するためいくつかの改善が行われ、線形代数計算の大きなグループを効率良く実行できるようになりました。これには、バッチおよびパックド API を使用します。[**編集者注：**バッチ API については、**インテル® デベロッパー・ゾーン** (英語) の「**バッチ GEMM 操作について**」(英語) を参照してください。パックド API については、**The Parallel Universe Issue 27** の「行列 - 行列乗算のパックのオーバーヘッドを減らす」を参照してください。]
インテル® MKL は、内部でグループ化と並列化を行います。また、このリリースでは 24 個の新しいベクトル演算関数が追加され、高度に最適化されたルーチンの選択肢が増えました。

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 は、画像処理、信号処理、データ処理 (データ圧縮 / 展開 と暗号化) アプリケーション向けのパフォーマンスが最適化された低レベルのビルディング・ブロックを提供します。インテル® IPP の最新バージョンでは、LZO データ圧縮においてインテル® ストリーミング SIMD 拡張命令 4.2 (インテル® SSE4.2) 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) のベクトル命令を利用する関数が改善されるなど、パフォーマンスが重要なアルゴリズムがさらに高速になりました。以前のバージョンにあった暗号化パッケージのメイン・ライブラリーへの依存性が排除され、インテル® IPP の強力な暗号化アルゴリズムを利用しやすくなりました。

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) と Parallel STL

インテル® TBB は、C++ アプリケーションの並列化に取り組む開発者により長年使用されてきました。これまで何度も改善が重ねられ、インテル® TBB の最新リリースでは、インテルの Parallel STL 実装が追加されています。

解析ツール

インテル® Parallel Studio XE Professional Edition および Cluster Edition には、複雑さを増すハードウェアおよびソフトウェア・エコシステムでのアプリケーションの解析、チューニング、デバッグを支援するいくつかのツールが含まれています。

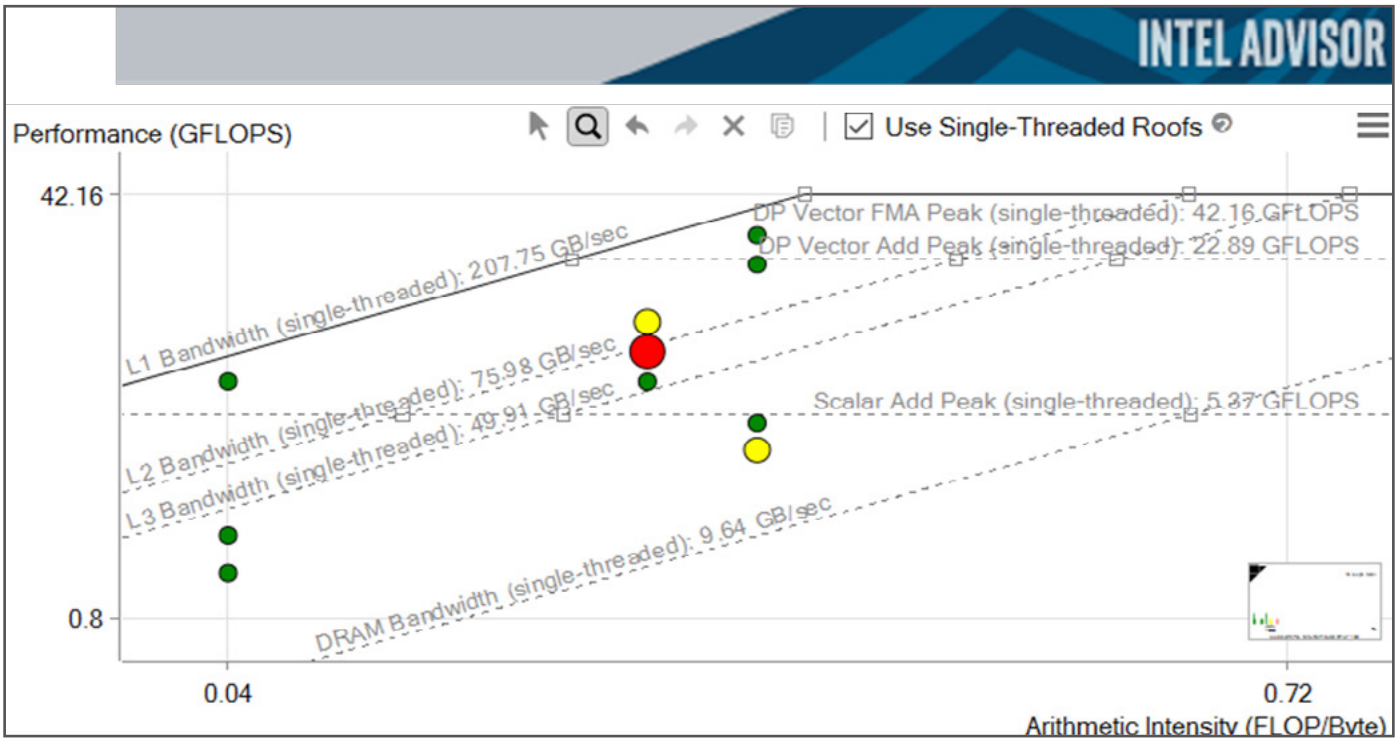
インテル® Advisor

並列化またはベクトル化によるパフォーマンスの向上を検討している場合は、インテル® Advisor が役立ちます。どの関数とループを最適化すべきか？ コンパイラーによるベクトル化を妨げているのは何か？ 推奨される変更を行ったら、パフォーマンスがどれぐらい向上するのか？ インテル® Advisor はこのような疑問に答えるように設計されています。インテル® Advisor 2018 には、メモリーのボトルネック (具体的には、効率的でないベクトル化やメモリーの局所性によりパフォーマンスが制限されているループ) を特定する最先端のルーフライン解析機能が含まれています (**図 2**)。

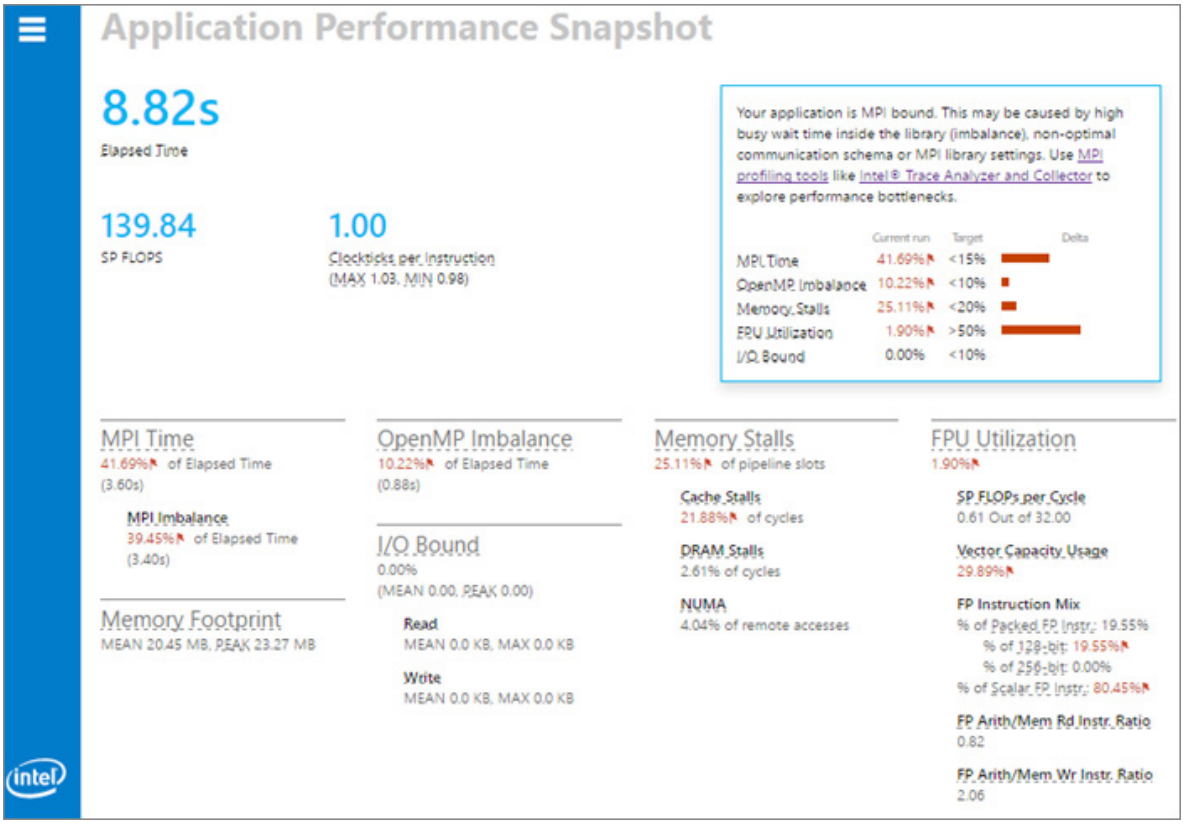
インテル® Advisor を利用することで、インテル® AVX-512 などの最新テクノロジーと高度な並列性を備えたインテル® Xeon® プロセッサーやインテル® Xeon Phi™ プロセッサーにより利点が得られるコード部分とその方法が簡単に分かります。

インテル® VTune™ Amplifier—パフォーマンス・スナップショットを追加

インテル® VTune™ Amplifier は、アプリケーションのパフォーマンスを理解して向上できるように支援する、エキスパートによるガイダンスを備えた強力な解析ツールです。このリリースでは、パフォーマンス・スナップショット (**図 3**) が追加され、素早く簡単に使用できるスクリプトにより、アプリケーションのハードウェア利用状況 (CPU、FPU、およびメモリー) を示す高レベルのビューを提供します。MPI、OpenMP*、または両方を利用して並列化する場合、これらのライブラリー内で費やされた時間や、並列処理の完了の待機に費やされた時間は、パフォーマンスを低下させます。



2 インテル® Advisor のルーフライン解析



3 インテル® VTune™ Amplifier のパフォーマンス・スナップショット

さらに、浮動小数点ユニットをアイドル状態にしたり、メモリーアクセスによる待機で CPU がストールしたりすると、多くのリソースが無駄になります。パフォーマンス・スナップショットを使用することで、これらのサイクルを有効活用できます。

インテル® VTune™ Amplifier は、コンテナーやクラウド・コンピューティングなどの新しいテクノロジーを利用するユーザー向けの機能も提供しています。Docker* および Mesos* コンテナーで実行しているアプリケーションをプロファイルしたり、インテル® VTune™ Amplifier のパフォーマンス解析を実行中の Java* サービスやデーモンにアタッチできるようになりました。

インテル® Inspector

アプリケーションのパフォーマンスを最適化することはこれまで以上に重要ですが、アプリケーションが正しく実行されなければ無意味です。インテル® Inspector は、アプリケーションの実行中にスレッドとメモリーのエラーを自動的にチェックします。これらの診断が困難な問題は、正しくない結果に依存する標準のテストでは検出されない可能性があります。インテル® Inspector のアルゴリズムは、メモリーリークや非決定的な競合状態などの将来発生する可能性がある問題を検出できます。このリリースでは、より高度なロックモデルが追加されました。インテル® Inspector は特別な再コンパイルを必要としないため、数クリックでプロファイルを開始し、コードの潜在的な問題を見つけることができます。

クラスターツール

以前のクラスター・コンピューティングは、大規模で、通常は厳重に管理されたクラスターへのリモートアクセスを必要とする、特定の分野に限定されたものでした。最近では、クラウド・コンピューティング・リソースとインテル® Xeon Phi™ プロセッサーのように多数のコアを備えシングルノード・クラスターとして動作可能なプロセッサーの普及により、クラスター・コンピューティングはさまざまな分野に拡大しています。インテルは、これまで長い間この分野での取り組みを進めてきました。インテル® Parallel Studio XE Cluster Edition は、インテルの経験を集約したツールスイートです。

インテル® MPI ライブラリーの最新バージョンでは、アプリケーションの起動とファイナライズが大幅に最適化され、生産性とスケーラビリティーが拡張されています。

インテル® Trace Analyzer & Collector では、区分化大域アドレス空間 (Partitioned Global Address Space: PGAS) コミュニティーで関心が高まっている OpenSHMEM* のサポートが追加されました。

最近ツールスイートに追加された**インテル® Cluster Checker** (英語) は、ビルトイン・エクスパート・システムを使用して、クラスターの問題を診断し、具体的な対応策を提案します。[**編集者注：**インテル® Cluster Checker については、「**あなたのクラスターは健全ですか？**」で詳しく述べます。]

インテル® Cluster Checker は、最新のインテル製ハードウェアとソフトウェアを含む、クラスターの機能、パフォーマンス、均一性の問題を特定するためのテストを実行します。クラスターの状態をよくチェックするシステム管理者に役立つツールです。また、API を使用することで、開発者がこれらの診断にアクセスし、必要な解析を利用できるようになりました。

ハードウェアとソフトウェアの進化

今日のエコシステムの要求を満たすアプリケーションを作成するだけでも困難ですが、将来のプラットフォームと環境にシームレスに適応し、維持し続けることはさらに困難な課題です。インテル® Parallel Studio XE に含まれているような支援ツールを利用することで、スケーラブルでハイパフォーマンスなソフトウェアを簡単に効率良く設計することができます。無料の評価版をお試しください。そして、将来のプラットフォームへコードを対応させましょう。

BLOG HIGHLIGHTS

新しいインテル® Parallel Studio XE 2018 でハイパフォーマンスかつスケーラブルで移植性の高い並列コードを作成する

HENRY GABB INTEL CORPORATION

インテル® Parallel Studio XE は、インテル® プロセッサー・アーキテクチャー上での HPC、エンタープライズ、クラウド・コンピューティング向けソフトウェアの開発、デバッグ、チューニングを支援するインテルのフラッグシップ製品です。コンパイラー、ハイパフォーマンスな数学ライブラリーから、デバッガー、大規模クラスター・アプリケーション向けのプロファイラーまで、すべてを備えた統合ツールスイートです。これらのツールを利用することで、開発者はインテル® プロセッサーの潜在的なパフォーマンスを最大限に引き出すことができます。インテル® Parallel Studio XE は、ハイパフォーマンスで安定したスケーラブルな並列コードを素早く開発できるように開発者を支援します。
最新リリースのインテル® Parallel Studio XE 2018 には、多くの興味深い新機能が追加されています [1]。最初に、製品名の一部でもある並列性についてみてみましょう。

この記事の続きはこちら (英語) でご覧になれます。>

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

intel Software

効率良い高速なコードのビルド

インテルの強力な実績あるライブラリーでコードを最適化して、開発時間を短縮しましょう。インテル® アーキテクチャー上での革新と素晴らしいパフォーマンスをサポートするというインテルのミッションに基づき、これらのライブラリーは無料でご利用いただけます。

無料のダウンロード (英語) >

外れ値への対応

実際のデータセットで不正取引を見つける方法

Oleg Kremnyov インテル コーポレーション QA エンジニア
Mikhail Averbukh 同ソフトウェア・エンジニア
Ivan Kuzmin 同ソフトウェア・エンジニア・マネージャー

データ解析における非常に重要な課題の 1 つは、外れ値への対応です。一般に、外れ値は、他のデータと矛盾するサンプルまたはイベントと定義されます [1]。外れ値には、通常、データ生成に影響するシステムとエンティティーの異常特性に関する役立つ情報が含まれています [2]。

外れ値検出アルゴリズムの一般的な適用例には、次のものがあります。

- 侵入検知システム
- クレジットカード詐欺
- 興味深いセンサーイベント
- 医療診断

この記事では、外れ値検出の最も一般的な適用例の 1 つである、クレジットカード詐欺に注目します。いくつかの単純な外れ値検出アプローチにより、実際のデータセットにおいて誤検出 1% 未満で 75% ~ 85% の不正取引を見つけることができます。

アプローチの定義

外れ値を検出する基本アプローチは 2 つあります。

1. **教師あり**: 不正な例とそうでない例を使用して、新しい観測のクラスを予測します。
2. **教師なし**: ラベル付けされた例を使用しないで、外れ値または異常として不正な例を検出します。

さらに細かく分類すると、3 つの基本教師ありアプローチがあります。

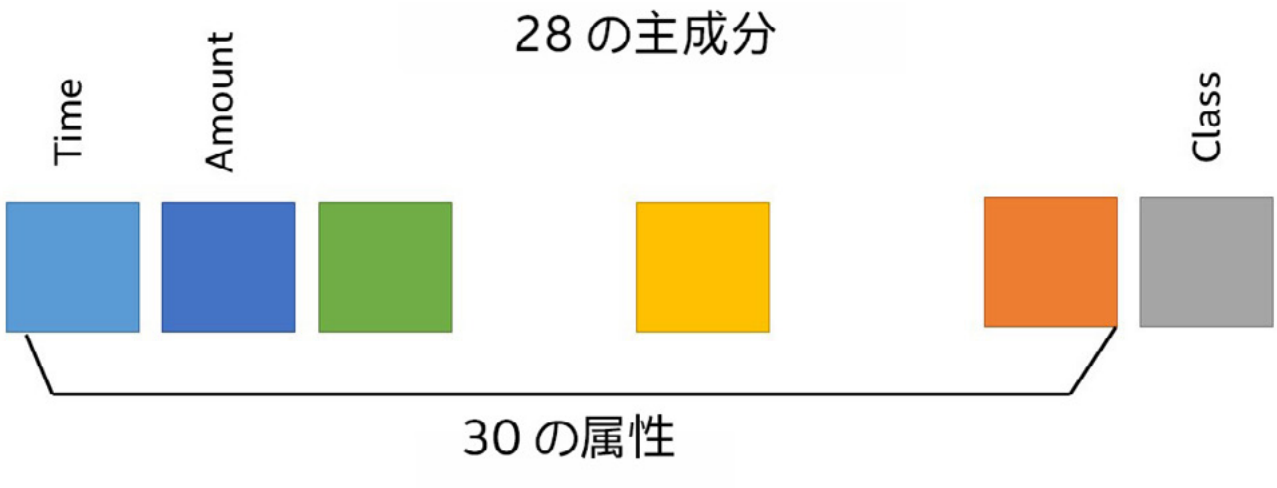
1. ニューラル・ネットワーク
2. SVM
3. ロジスティック回帰

そして、3 つの教師なしアプローチがあります。

1. BACON 外れ値検出などの統計ベースの手法
2. 多変量外れ値検出
3. K 平均法、期待値最大化 (EM)、DBSCAN[5]、外れ値として処理可能なデータ内のノイズを検出するアルゴリズム [6] などのクラスタリングベースの手法 [3, 4]。

この記事では、これらの手法を比較し、特定の設計基準に基づいて各手法を評価します。正確さの指標には、検出率と誤検出率の両方が含まれます [7]。

不正の検出では、不正行為を見つけるだけではなく、できるだけ早く見つけることが求められるため、アルゴリズムのパフォーマンスも重要です [8]。ここでは、**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)** を含むいくつかの一般的なマシンラーニング・ライブラリーを使用して、外れ値のアルゴリズムを実装し、精度とパフォーマンスを比較します。



1 データセットの属性

データセットの説明

ここでは、企業や研究者が投稿したデータに対し、世界中の統計学者とデータマイナーが最良のモデルの生成を競う、予測モデリングおよび解析コンペティションのプラットフォームである Kaggle[10] のデータセット[9] を使用します。ここで使用する Kaggle で最も人気のある実際のクレジットカード詐欺のデータセット[11] には、31 個の数値形式の入力変数が含まれています (**図 1**)。

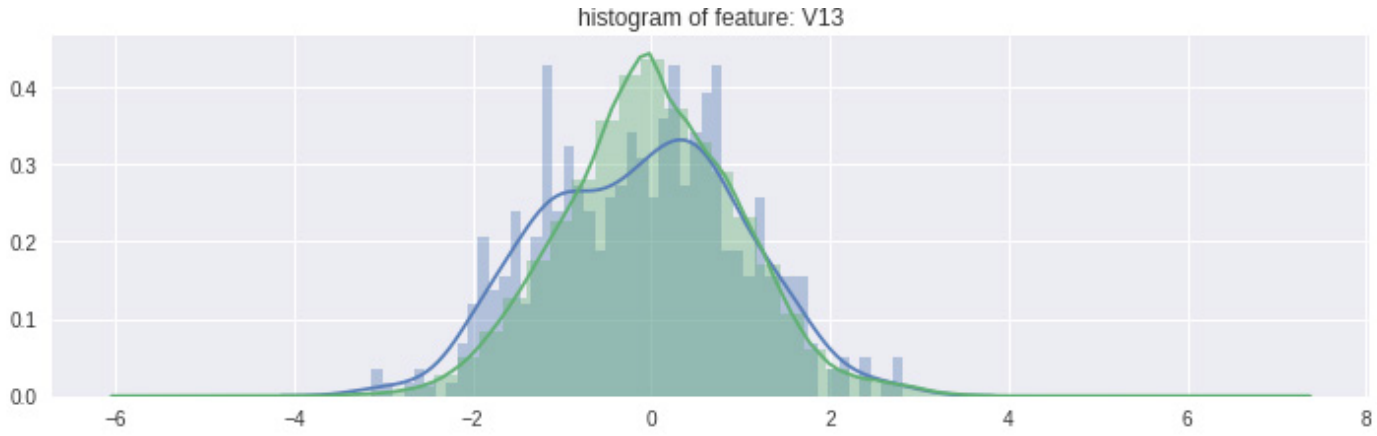
Time は、データセットの最初の取引と各取引の間の経過時間 (秒) を示します。Amount は、取引金額を示します。残りの 28 個の特徴 (属性) は、PCA により取得される主成分です。機密性の問題から、元の特徴値は提供されません。

Class は、応答変数です。不正取引の場合は 1 に、そうでない場合は 0 になります。不正取引は、通常の取引と比べるとはるかに少ないため、データセットは非常に不均衡です。不正取引は、全取引の 0.172% のみです (284,807 件中 492 件)。

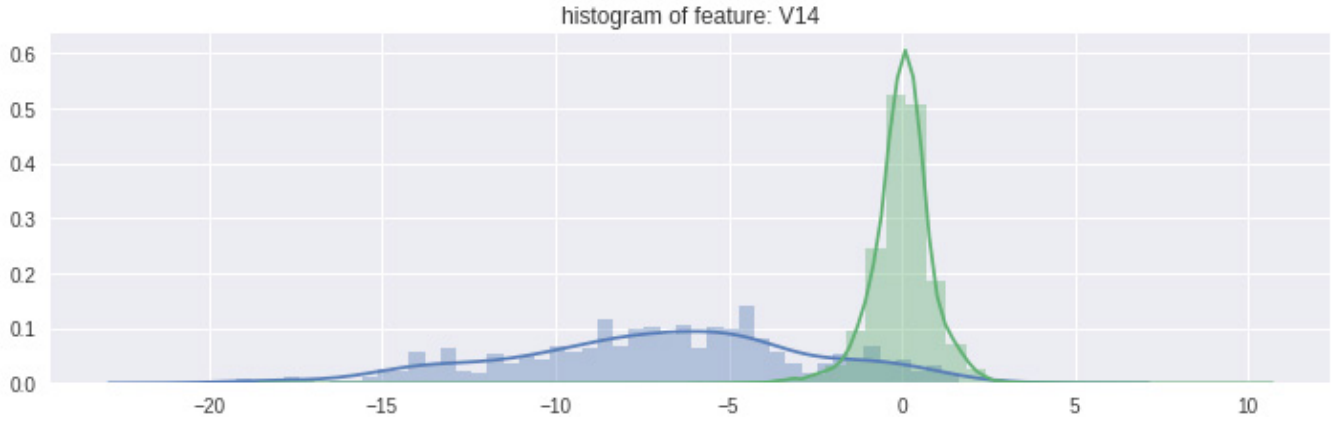
いくつかの主成分の条件付き分布を見てみましょう (**図 2** と**図 3**)。主成分の間には相関性がないため、不正取引と通常取引の例で類似する条件付き分布の主成分を排除することで、より精度の高いモデルを構築することができます。点双列相関係数[12] を使用して、各主成分とターゲット変数の間の関連性を測定します (**表 1**)。この情報は、後で特徴量の設計に使用します。

クレジットカード詐欺の検出アプローチ

異なる外れ値検出手法をデータに適用して、それぞれの精度とパフォーマンスを評価および比較します。クレジットカード詐欺の検出には、教師ありと教師なしの両方の手法を使用します。



2 2 種類の取引で類似する分布を持つ属性



3 2 種類の取引で異なる分布を持つ属性

表 1. 主成分の点双列相関係数

変数	係数	変数	係数	変数	係数	変数	係数
V22	0.000805	V28	0.009536	V2	0.091289	V7	0.187257
V23	0.002685	V27	0.017580	V5	0.094974	V3	0.192961
V25	0.003308	V8	0.019875	V9	0.097733	V16	0.196539
V15	0.004223	V20	0.020090	V1	0.101347	V10	0.216883
V26	0.004455	V19	0.034783	V18	0.111485	V12	0.260593
V13	0.004570	V21	0.040413	V4	0.133447	V14	0.302544
V24	0.007221	V6	0.043643	V11	0.154876	V17	0.326481

教師なしアプローチ

統計アプローチは、外れ値検出において最初に使用されたアルゴリズムです [13]。ここでは、外れ値検出専用アルゴリズムである多変量外れ値検出と BACON 外れ値検出から開始します。これらの手法の最大の利点は、計算効率に優れており、大きなデータセットに簡単に適用できることです [14]。ここでは、点双列相関係数が高い 6 つの主成分 (V11, V12, V14, V16, V17, V18) を特徴とした場合に、最良の結果が得られました。インテル® DAAL と R の robustX パッケージの実装を使用しました。精度は、多変量外れ値検出のほうが BACON 外れ値検出よりも優れていました (**表 2**)。

表 2. 統計ベースの外れ値検出アプローチ

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒)
多変量外れ値検出、インテル® DAAL (しきい値 = 8.0)	83.54	0.20	16.16
多変量外れ値検出、インテル® DAAL (しきい値 = 7.0)	84.55	0.41	17.29
BACON 外れ値検出、インテル® DAAL (alpha = 1e-9, initMethod = baconMahalanobis)	84.55	0.70	82.25
BACON 外れ値検出、R (alpha = 1e-9, initMethod = baconMahalanobis)	84.73	1.80	1,860
BACON 外れ値検出、インテル® DAAL (alpha = 1e-9, initMethod = baconMedian)	84.55	0.70	78.14
BACON 外れ値検出、R (alpha = 1e-9, initMethod = baconMedian)	84.55	1.80	808

クラスタリングと外れ値検出は補完関係にあります。クラスタリングの目標は、ポイントを密集したサブセットに分割することです。外れ値検出の目標は、密集したサブセットに適合しないポイントを見つけることです [2]。DBSCAN などの一部のクラスタリング・アルゴリズムは、外れ値を考慮する機能も備えています。

外れ値検出には、多くのクラスタリングベースのアプローチがあります [2,3,4,15]。ここでは、そのうちのいくつかを適用します。犯人は通常、一連の不正取引を生成するため、クレジット詐欺行為のクラスタリングでは、固有のクラスターが形成されます [16]。

外れ値には、ローカルとグローバルの 2 種類があります。異なるクラスタリングベースのアルゴリズムにより両方の種類を検出できます。グローバルな外れ値の属性値は、大部分のデータポイントの値から外れています。ローカルな外れ値の属性値は、近傍値と比較して極端です [17]。グローバルな外れ値を検出するには、データセットをクラスタリングして、いくつかのクラスターを外れ値として定義します [18] (**表 3**)。K 平均法と EM を利用してクラスタリングを行い、小さなクラスターを外れ値と見なします [19]。K と n_components の値は、範囲 [2, 3, …, 50] のグリッド検索によって取得されます。インテル® DAAL と scikit-learn の K 平均法には、インテル® DAAL の実装を使用して K-Means++ 初期化により取得した同じ初期セントロイドを使用しました。ここでは、点双列相関係数が高い 17 の主成分を特徴とした場合に、最良の結果が得られました。また、これらの手法では、比較的低い誤検出率を達成しました。

表 3. クラスタリングベースのグローバルな外れ値検出手法

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒)
K 平均法、インテル® DAAL (k=20、10 反復)、外れ値のクラスターの検出	76.42	0.02	30.15
K 平均法、scikit-learn (k=20、10 反復)、外れ値のクラスターの検出	76.42	0.02	1,975.49
EM-GMM、scikit-learn (n_components=7)、外れ値のクラスターの検出	81.10	0.09	73,490.88

ローカルな外れ値を検出するクラスタリング手法も適用しました [20]。ナイーブアプローチでは、K 平均法アルゴリズム [21] を使用して取引をクラスタリングし、クラスターの中心から離れているポイントを外れ値と見なしました (**表 4**)。

検出率は良好でしたが、誤検出率が高くなりました。これは、K 平均法アルゴリズムが、クラスター構成に大きく影響する外れ値に対して非常に敏感なため、外れ値とされるべきデータポイントがクラスタリングによってマスクされてしまったことが原因です。そのため、外れ値をうまく処理できるより堅固な K 平均法である、(K,L)-Means[15] (**表 5**) を使用することにしました。K 平均法と同様に、このアルゴリズムには初期セントロイドが必要です。ここでは、インテル® DAAL の異なる初期化アルゴリズム—乱数、K-Means++[22]、および K-Means||[23]—を使用して初期セントロイドを計算し、結果を比較します。

表 4. K 平均法ベースのローカルな外れ値検出手法

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒)
K 平均法、インテル® DAAL (k=7、10 反復)、ローカルな外れ値の検出	80.08	0.94	26.94
K 平均法、scikit-learn (k=7、10 反復)、ローカルな外れ値の検出	80.08	0.94	1,210.25

表 5. (K,L)-Means アルゴリズムの結果

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒)
(K,L)-Means (k=10、l=800、5 反復、乱数初期化手法)	81.71	0.14	165.15
(K,L)-Means (k=10、l=1000、5 反復、乱数初期化手法)	83.13	0.21	167.05
(K,L)-Means (k=10、l=2000、5 反復、乱数初期化手法)	85.16	0.56	169.03
(K,L)-Means (k=10、l=2000、5 反復、K-Means++ 初期化手法)	84.95	0.55	169.46
(K,L)-Means (k=10、l=2000、5 反復、K-Means|| 初期化手法)	81.30	0.55	179.84

このアプローチのほうが、単純な K 平均法よりも精度が高いことが分かります。初期化手法に応じて結果は異なります。この問題では、乱数手法が最良であると証明されました。

また、データ内のノイズを検出できる密度ベースのクラスタリング・アルゴリズムも使用します (**図 6**)。このアルゴリズムは、近傍値と大きく異なるポイントをノイズとして検出します。ノイズデータはローカルな外れ値と仮定するのが自然です。統計手法と同じ 6 つの特徴を使用します。パラメーターの値は、グリッド検索によって取得されます。R パッケージの DBSCAN を使用します。適度なレベルの誤検出で、良好な不正検出率が達成できましたが、前述のアプローチと比較してパフォーマンスが大幅に低下しました。

表 6. DBSCAN による外れ値検出

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒)
DBSCAN、R (eps = 1、minPts = 8)	82.52	0.43	181,214
DBSCAN、R (eps = 0.9、minPts = 8)	84.35	0.64	121,348

教師ありアプローチ

教師あり学習手法には、各クラス (この例では、通常取引と不正取引) の例を含む訓練データが必要です[2]。ここでは、データセットを訓練セットと評価セットに分割して、80% を訓練に、残り 20% をモデルの評価に使用します。

Kaggle カーネルで提案されているニューラル・ネットワーク (**表 7**) から開始します[24]。カーネルでは、3 つの隠れた層を持つ単純な多層パーセプトロンが使用されています。多層パーセプトロン (MLP) は、分類と外れ値検出に広く使用されているニューラル・ネットワークです[25]。

表 7. ニューラル・ネットワーク

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒) (訓練)	パフォーマンス (ミリ秒) (予測)
ニューラル・ネットワーク、TensorFlow*	82.93	0.10	48,005.41	7.62

ここで外れ値検出に使用した別の教師あり学習アルゴリズムは、ロジスティック回帰です[16] (**図 8** と**図 9**)。点双列相関係数が最も高い 17 の属性 (**表 1**) と Z スコア正規化により前処理済みの Amount 特徴を使用します。ヒストグラムから[24]、ロジスティック回帰の使用により、不正と通常の観測が線形に分離可能であることが分かります。データセットは不均衡なので、ランダム・アンダーサンプリングを使用することで、より正確に不正を検出するモデルを作成できる可能性があります。これにより、訓練から多くの通常の例を排除し、ランダムにアンダーサンプルされた訓練データセットで各推定量が訓練された多数決分類器[26] を使用します。(正規化係数 C は、さまざまな値で検証後に 1 に設定されました。) この手法では、誤検出率はほぼ同じままで、検出率が 3% 向上しました。

表 8. ロジスティック回帰

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒) (訓練)	パフォーマンス (ミリ秒) (予測)
ロジスティック回帰、scikit-learn (訓練データ内のサンプルの 1% が不正取引)	80.61	0.02	323.69	1.45
ロジスティック回帰、scikit-learn (訓練データ内のサンプルの 9% が不正取引)	83.67	0.02	20.88	1.51

表 9. ロジスティック回帰モデルの連携

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒) (訓練)	パフォーマンス (ミリ秒) (予測)
ロジスティック回帰、scikit-learn (訓練データ内のサンプルの 9% が不正取引、100 個の分類器を連携)	85.71	0.19	3,530.13	131.72
ロジスティック回帰、scikit-learn (訓練データ内のサンプルの 9% が不正取引、10 個分類器を連携)	86.73	0.18	335.81	12.22

外れ値検出に利用可能な別の教師あり学習アルゴリズムである、サポート・ベクトル・マシン (SVM) が、同じデータでどのように動作するか見てみましょう。100,000 反復、線形カーネル、C=1 で SVM を実行します。(これらのパラメーターは、グリッド検索により取得されたものです。) ロジスティック回帰と同じ特徴を使用します。SVM を連携させることで、検出率が大幅に向上し、誤検出率が低下します (**表 10**)。

異なる異常検出アプローチの評価

不正検出には、さまざまな教師なしアプローチがあります。ここでは、そのうちのいくつかを適用しました。外れ値検出専用アルゴリズムの K,L-Means と DBSCAN は、最高の検出率を達成し、不正の 85% を検出しました。一方、K 平均法による不正取引のクラスターの検出では、誤検出数が最小になりました。不正取引をできるだけ早く検出することで、不正なクレジットカードによる取引を停止し、被害の拡大を防ぎます[16]。そのため、アルゴリズムのパフォーマンスも重要な指標です。不正検出に最適なアプローチは、検出された不正行為とそれを停止するためのコストを含むさまざまな要因に依存します [7]。教師なし手法は、観測のクラスに関する情報が事前にない場合に役立ちます。

表 10. SVM

アルゴリズム	検出率 (%)	誤検出率 (%)	パフォーマンス (ミリ秒) (訓練)	パフォーマンス (ミリ秒) (予測)
SVM、scikit-learn (訓練データ内のサンプルの 9% が不正取引)	83.67	0.04	250.00	174.40
SVM、インテル® DAAL (訓練データ内のサンプルの 9% が不正取引)	76.53	0.01	246.68	12.23
SVM、scikit-learn (訓練データ内のサンプルの 9% が不正取引、10 個の分類器を連携)	91.83	0.04	2,781.41	1,792.12
SVM、インテル® DAAL (訓練データ内のサンプルの 9% が不正取引、10 個の分類器を連携)	91.83	0.04	2,662.94	119.24
SVM、scikit-learn (訓練データ内のサンプルの 9% が不正取引、100 個の分類器を連携)	91.83	0.03	28,508.28	21,806.03
SVM、インテル® DAAL (訓練データ内のサンプルの 9% が不正取引、100 個の分類器を連携)	91.83	0.03	27,718.29	1,191.73

観測クラスに関する情報が事前にある場合は、教師あり学習アルゴリズムを使用できます。ここでは、検出される不正の数を向上するため、アンダーサンプリングなどの不均衡な分類向けの手法を使用しました。SVM 分類器を連携させることで、検出率が最高になり、ランダムフォレストの誤検出率は最低になります。SVM などの教師あり手法は、教師なし手法と比較して精度が非常に優れていますが、使用することで次の問題が生じることがあります。

- **正確にラベル付けされた訓練データ**は、入手コストが非常に高い可能性があり[25]、データ・サイエンティストはモデルの作成に使用される訓練データの正確なクラスについて確信を持っていなければなりません。
- **モデルの訓練時間**は、サンプル数とサンプルあたりの特徴の数に応じて長くなることがあります。
- **教師なしアプローチと異なり**、教師あり学習では、再訓練しないで新しい種類の不正を検出することはできません[16,27]。

ここで示したように、特定の用途に合った適切なマシンラーニング・アルゴリズムの選択は、熟考を必要とする重要な問題です。

参考資料 (英語)

1. Victoria Hodge, “Outlier and Anomaly Detection: A Survey of Outlier and Anomaly Detection Methods.”
2. Charu C. Aggarwal, “Outlier Analysis,” Second Edition.
3. Vaishali, “Fraud Detection in Credit Card by Clustering Approach.”
4. Yogesh Bharat Sonawane, Akshay Suresh Gadgil, Aniket Eknath More, Niranjan Kamalakar Jathar, “Credit Card Fraud Detection Using Clustering Based Approach.”
5. Ester, M., H. P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise” in Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231.1996.
6. Samson Kiware, B.A, “Detection of Outliers in Time Series Data.”
7. Elham Hormozi , Hadi Hormozi, Mohammad Kazem Akbari, Morteza Sargolzaei Javan, “Accuracy Evaluation of a Credit Card Fraud Detection System on Hadoop MapReduce.”
8. Shraddha Ramesh Bhagwat, Vaishali Londhe, “A Review of Various Credit Card Detection Techniques.”
9. Credit card fraud detection dataset.
10. Overview of Kaggle on Wikipedia.
11. Credit card fraud datasets on Kaggle.
12. Point-biserial correlation coefficient on Wikipedia.
13. Victoria J. Hodge and Jim Austin, “A Survey of Outlier Detection Methodologies.”
14. Nedret Billor, Ali S. Hadi, Paul F. Velleman, “BACON: Blocked Adaptive Computationally Efficient Outlier Nominators.”
15. Sanjay Chawla, Aristides Gionis, “k-means: A unified Approach to Clustering and Outlier Detection.”
16. Sanjeev Jha, Montserrat Guillen, and J. Christopher Westland, “Employing Transaction Aggregation Strategy to Detect Credit Card Fraud.”
17. Marie Ernst and Gentiane Haesbroeck, “Detection of Local and Global Outliers in Spatial Data.”
18. Bin-mei Liang, “A Hierarchical Clustering Based Global Outlier Detection Method.”
19. Yuan Wang, Xiaochun Wang, Xia Li Wang, “A Spectral Clustering Based Outlier Detection Technique.”
20. Zengyou He, Xiaofei Xu, Shengchun Deng, “Discovering Cluster-Based Local Outliers.”
21. Stuart P Lloyd, “Least Squares Quantization in PCM,” IEEE Transactions on Information Theory 1982, 28 (2): 1982pp: 129–137.
22. Arthur, D. and Vassilvitskii, S, “**k-means++: The Advantages of Careful Seeding**,” Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007, pp. 1027-1035.
23. B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, “**Scalable K-means++**,” Proceedings of the VLDB Endowment, 2012.

参考資料 (英語) (続き)

24. Kaggle kernel, use of neural network for fraud detection.
25. Varun Chandola, Arindam Banner Jee, and Vipin Kumar, “Outlier Detection : A Survey.”
26. Gareth James, “Majority Vote Classifiers: Theory and Applications.”
27. Bolton, R.J. and Hand, D.J., “Unsupervised Profiling Methods for Fraud Detection” in Conference on Credit Scoring and cCedit Control, Edinburgh, 2001.

システム構成

ハードウェア

- プロセッサー : インテル® Xeon® Platinum 8168 プロセッサー @ 2.70GHz
- ソケットごとのコア数 : 24
- ソケット数 : 2
- メモリー : 63GB

ソフトウェア

- インテル® DAAL 2018 Gold
- R 3.4.1
- scikit-learn 0.19.1
- robustX 1.2-2
- DBSCAN 1.1-1

最新の SIMD 拡張とインテル® AVX-512 による成功のためのチューニング

最新アーキテクチャーの機能を利用するためのベスト・プラクティス

Xinmin Tian インテル コーポレーション インテル・コンパイラーおよび言語研究室 シニア主席エンジニア
Hideki Saito 同主席エンジニア
Sergey Kozhukhov 同シニア・スタッフ・エンジニア
Nikolay Panchenko 同スタッフエンジニア

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-512) は、最新の x86 ベクトル命令セットで、最大 2 つの FMA (Fused-Multiply-Add) ユニットとほかの最適化を提供します。次のようなワークロードのパフォーマンスを高速化できます。

- 科学シミュレーション
- 金融解析
- 人工知能 (AI)
- 3D モデリングと解析
- 画像、オーディオ / ビデオ処理
- 暗号化
- データ圧縮 / 展開

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

この記事では、インテル® AVX-512 命令セット・アーキテクチャー (ISA) の概要を紹介し、インテル® コンパイラー 18.0 の**インテル® Xeon® スケーラブル・プロセッサー**向けの新機能と、インテル® AVX-512 サポートに関連したいくつかの新しい **simd** 言語拡張について説明します。さらに、インテル® AVX-512 ISA を利用して最適なパフォーマンスを達成するためのパフォーマンス・チューニングのベスト・プラクティスを皆さんと共有します。

新機能

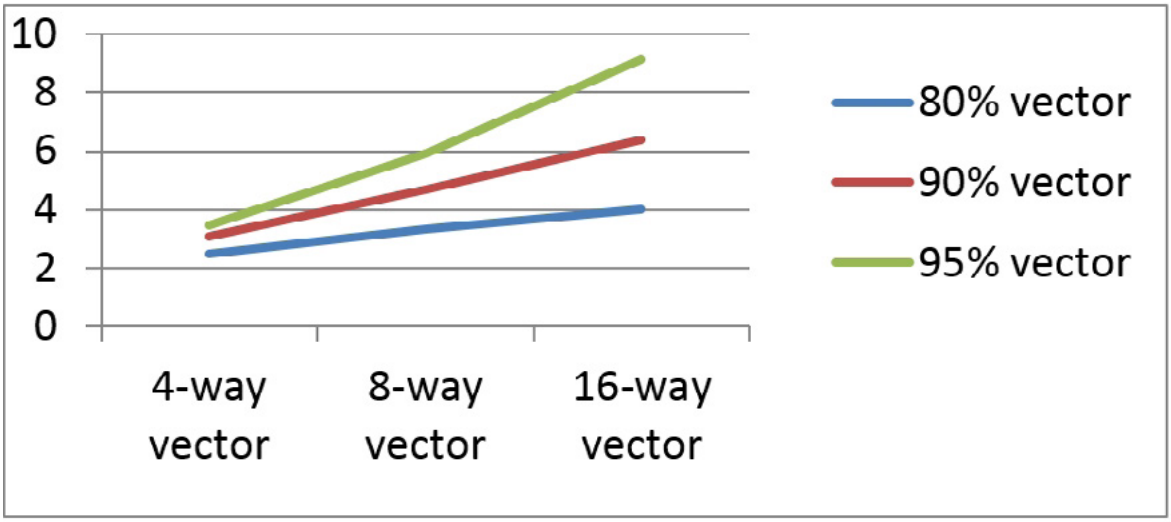
インテル® Xeon® スケーラブル・プロセッサーは、いくつかの新しいインテル® AVX-512 命令をサポートします。前世代のインテル® AVX2 と比較したインテル® Xeon® スケーラブル・プロセッサー・ベースのサーバーにおけるインテル® AVX-512 ISA の主なパフォーマンス機能は、次の通りです。

- インテル® AVX-512 基本命令 :
  - 512 ビット・ベクトル長
  - 32 個の 512 ビット・ベクトル・レジスター
  - データ展開 / 圧縮命令
  - 三項演算子の論理命令
  - 8 つの新しい 64 ビット・マスク・レジスター
  - 2 つのソースのクロスレーンの置換命令
  - 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 ダブルワードおよびクアッドワード命令 (DQ): QWORD サポート
- インテル® AVX-512 バイトおよびワード命令 (BW): バイトおよびワードサポート
- インテル® AVX-512 ベクトル長拡張命令 (VL): ベクトル長の直交性
- インテル® AVX-512 競合検出命令 (CDI): ベクトル競合命令

これらのインテル® AVX-512 機能は、ハードウェアで直接サポートされます。この記事では、インテル® Xeon® スケーラブル・プロセッサーのインテル® AVX-512F、インテル® AVX-512BW、インテル® AVX-512CD、インテル® AVX-512DQ、インテル® AVX-512VL 向けの **simd** 言語拡張、コンパイラー・サポート、チューニングに注目します。

コンパイラーとプログラマーにとってのチューニングの課題

図 1 は、コードの大部分で完璧なベクトル・スピードアップを達成できる理論的なアプリケーションのスピードアップの期待値です。XMM レジスターで 32 ビットの整数 / 単精度浮動小数点数を計算する 4 ウェイベクトルでは、アプリケーションの 80% をベクトル化することで得られるスピードアップは 2.5 倍に過ぎません。コンパイラーやプログラマーが頑張ってコードの 90% あるいは 95% をベクトル化しても、達成できるスピードアップは限定的です。



1 理論的なアプリケーションの理想的なベクトル・スピードアップ

しかし、8 ウェイや 16 ウェイベクトルでは、さらなるベクトル化により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。**図 1** は、コンパイラーやプログラマーが、インテル® Xeon® スケーラブル・プロセッサーの YMM と ZMM ベクトルを利用するようにアプリケーションをベクトル化することで、さらなるパフォーマンス・ゲインが得られることを示しています。ベクトル幅が広くなっただけでなく、インテル® AVX-512 ISA 拡張は、より多くのコードパターンをベクトル化できるようにさまざまなアーキテクチャー拡張を備えています。

インテル® AVX-512F、インテル® AVX-512VL、インテル® AVX-512BW

インテル® AVX-512F、インテル® AVX-512VL、インテル® AVX-512BW には、32 個のレジスターとマスキングをサポートする、インテル® AVX およびインテル® AVX2 の拡張が含まれます。この 3 つの命令セットは、インテル® AVX-512 ファミリーの大部分を占めます。インテル® AVX-512F は、32 ビットおよび 64 ビットの整数と浮動小数点要素データのベクトルレジスターのサイズを 512 ビットに拡張します。また、次の機能を提供します。

- アーキテクチャーのベクトルレジスターの数を 16 個から 32 個に**増やします**。
- 専用のマスクレジスターが**追加されています**。
- ロード/ストア以外の操作でマスキングを**サポートします**。

インテル® AVX-512VL は、32 個のレジスターとマスキングが 128 ビットおよび 256 ビット・ベクトルにも適用できるようにインテル® AVX-512F を拡張します。インテル® AVX-512BW は、32 個のレジスターとマスキングが 8 ビットおよび 16 ビット整数要素データにも適用できるようにインテル® AVX-512F を拡張します。

表 1. 要素のデータ型と命令の機能 (xmm/ymm/zmm、マスクあり/なし、など) の関係

	(v)paddb	(v)paddw	(v)paddw	(v)paddq	(v)addps	(v)addpd
xmm (0-15) マスクなし	インテル® SSE2	インテル® SSE2	インテル® SSE2	インテル® SSE2	インテル® SSE	インテル® SSE2
xmm (16-31) マスクなし	インテル® AVX-512VL + インテル® AVX-512BW	インテル® AVX-512VL + インテル® AVX-512BW	インテル® SSE2	インテル® AVX-512VL	インテル® AVX-512VL	インテル® AVX-512VL
xmm マスクあり	インテル® AVX-512VL + インテル® AVX-512BW	インテル® AVX-512VL + インテル® AVX-512BW	インテル® AVX-512VL	インテル® AVX-512VL	インテル® AVX-512VL	インテル® AVX-512VL
ymm (0-15) マスクなし	インテル® AVX2	インテル® AVX2	インテル® AVX2	インテル® AVX2	インテル® AVX	インテル® AVX
ymm (16-31) マスクなし	インテル® AVX-512VL + インテル® AVX-512BW	インテル® AVX-512VL + インテル® AVX-512BW	インテル® AVX-512VL	インテル® AVX-512VL	インテル® AVX-512VL	インテル® AVX-512VL
ymm マスクあり	インテル® AVX-512VL + インテル® AVX-512BW	インテル® AVX-512VL + インテル® AVX-512BW	インテル® AVX-512VL	インテル® AVX-512VL	インテル® AVX-512VL	インテル® AVX-512VL
zmm (0-31) マスクあり/なし	インテル® AVX-512BW	インテル® AVX-512BW	インテル® AVX-512F	インテル® AVX-512F	インテル® AVX-512F	インテル® AVX-512F

表 1 は、ベクトル加算命令を使用して、要素のデータ型 (b/w/d/q/ps/pd)、命令の機能 (xmm/ymm/zmm、マスクあり/なし、レジスター番号 0-15/16-31)、インテル® 64 プラットフォーム上で必要な最小 ISA 拡張の関係を示したものです。例えば、XMM16 レジスターでバイト型のベクトルを加算 (**vpaddb**) するには、マスクを使用するかどうかに関係なく、インテル® AVX-512VL とインテル® AVX-512BW の両方がサポートされている必要があります。インテル® AVX-512F、インテル® AVX-512VL、インテル® AVX-512BW をすべてサポートすることで、インテル® Xeon® スケーラブル・プロセッサーでは、ほとんどの操作で 32 個のレジスターとマスキング機能を、すべてのレジスターサイズ (XMM、YMM、または ZMM) とすべてのデータ要素型 (バイト、ワード、ダブルワード、クアッドワード、単精度、倍精度) で利用できます。

インテル® AVX-512F には、インテル® AVX とインテル® AVX2 以外にも次の操作が含まれています。

- **データ圧縮 (vcompress) 操作 :** マスクレジスターで 1 に設定されているビットによって指定されたインデックスを使用して入力バッファーから要素を読み取ります。読み取られた要素は、デスティネーション・バッファーへ書き込まれます。要素の数がデスティネーション・レジスターのサイズよりも小さい場合、残りのスペースはゼロになります。
- **データ展開 (vexpand) 操作 :** ソース配列 (レジスター) から要素を読み取り、マスクレジスターの有効なビットに対応するデスティネーション・レジスターの位置に配置します。有効なビットの数がデスティネーション・レジスターのサイズよりも小さい場合、残りの値は無視されます。

インテル® AVX-512CDI

インテル® AVX-512CDI は、インテル® AVX-512F とともに、メモリーに関してベクトル依存関係 (競合) の可能性があるループを効率良くベクトル化できるようにします。最も重要な命令は、単一のベクトルレジスター内の要素の水平比較を実行する **VPCONFLICT** です。具体的には、**VPCONFLICT** はベクトルレジスターの各要素をそのレジスターのそれ以前の要素と比較して、すべての比較の結果を出力します。インテル® AVX-512CDI のほかの命令は、比較結果の効率良い操作を可能にします。**VPCONFLICT** には、ループのベクトル化に役立つ別の使い道もあります。最も単純なものは、指定された **simd** レジスターに重複するインデックスが存在するかどうかチェックすることです。存在しない場合、**simd** 命令を安全に使用してすべての要素を並列に計算できます。存在する場合は、それらの要素をスカラーループで実行します。ループのスカラーバージョンへ分岐することは、インデックスの重複がまれな場合に良い方法です。しかし、ベクトル化されたループの 1 つの反復で 1 つ以上の重複が予想される場合でも、並列処理を最大限に利用できるように、できるだけ **simd** を使用するほうが良いでしょう。

インテル® コンパイラー 18.0 のインテル® AVX-512 向けチューニングに関する新機能

インテル® Xeon® スケーラブル・プロセッサー上で最適なパフォーマンスを達成するためには、プロセッサー固有のオプション **-xCORE-AVX512** (Linux* および macOS*) または **/QxCORE-AVX512** (Windows*) を指定してアプリケーションをコンパイルすべきです。生成される実行ファイルは、インテル以外のプロセッサーまたは古い命令セットのみをサポートするインテル® プロセッサーでは動作しません。インテル® コンパイラー 18.0 では、ZMM コードの生成をチューニングするための新しいオプション **-qopt-zmm-usage=[high|low]** が追加されています。

- **値 low:** エンタープライズ・アプリケーションなどにおいて、インテル® AVX2 ISA から開発コード名 Skylake Server (SKX) ベースのターゲット上のインテル® AVX-512 ISA へ円滑に移行できます。プログラマーは、**#pragma omp simd simdlen()** などの明示的なベクトル構文を利用して ZMM 命令の使用についてチューニングすることが推奨されます。
- **値 high:** 幅広いベクトルを利用して 1 命令あたりの演算数を増やすため、ベクトル演算に依存するアプリケーション (例 : HPC アプリケーション) に推奨されます。

開発コード名 Skylake Server ベースの計算ターゲット (例 : **-xCORE-AVX512**) のデフォルト値は **low** です。インテル® コンパイラーには、複数の命令セットをサポートするファットバイナリーを生成する **-ax** オプションがあります。例えば、**-axCORE-AVX512,CORE-AVX2** を指定してコンパイルすると、インテル® AVX-512 とインテル® AVX2 ターゲット向けの専用コードに加えて、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2) 以上をサポートするインテル® プロセッサーおよび互換プロセッサーで動作するデフォルトのコードパスが生成されます。実行時に、アプリケーションはインテル® プロセッサーで実行しているかどうかを自動検出します。そして、インテル® プロセッサーの場合は、インテル® プロセッサー向けの最適なコードパスを選択し、そうでない場合はデフォルトのコードパスを選択します。

指定するオプションに関係なく、コンパイラーは、検出されたプロセッサーに応じて適切なコードパスをディスパッチする特別なライブラリー・ルーチン (memset/memcpy の最適化されたバージョンなど) の呼び出しを挿入することがあります。

インテル® AVX-512 向けの新しい simd 拡張

インテル® コンパイラーは、C/C++ および Fortran アプリケーション・プログラムにおける明示的なベクトル化に率先して取り組み、OpenMP* 4.0 と 4.5 での標準化に尽力してきました [3, 4, 5, 7, 8]。そして、明示的なベクトル化を拡張するためのインテルの継続的なイノベーションの成果として、将来の OpenMP* 仕様への追加を提案し、OpenMP* 標準化の先駆けとしてインテル® コンパイラー 18.0 でその一部を実装しました [3, 8]。

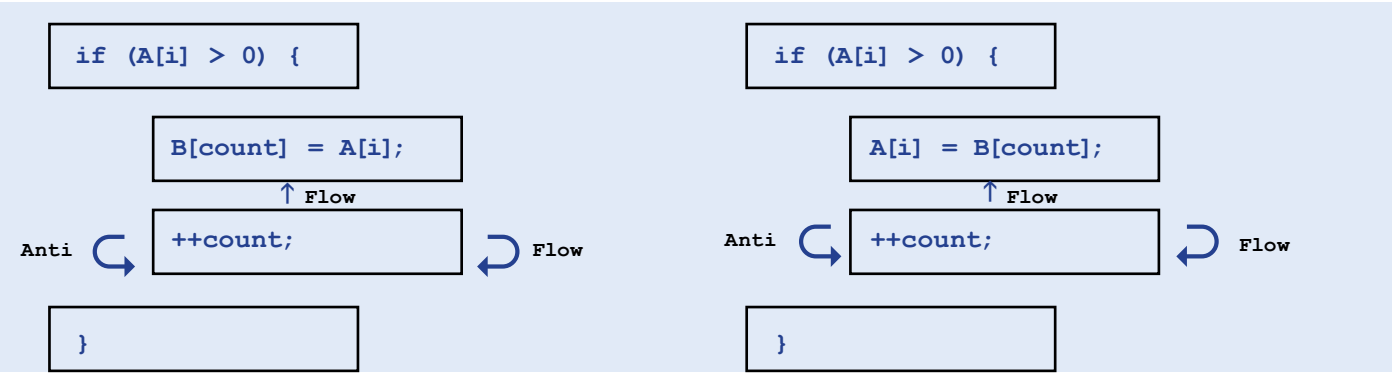
圧縮と展開

図 2 は、圧縮と展開の典型的なコードです。圧縮と展開という用語は、コードのセマンティクスを語源としています。例えば、圧縮パターンは、配列 **A[]** のすべての正である要素を圧縮し、配列 **B[]** へ連続して格納します。どちらのパターンもループ伝播依存があるため (**図 3** と **図 4**)、並列化できません。

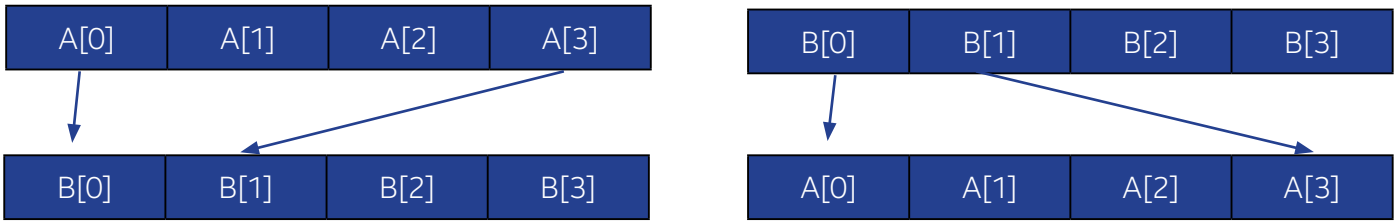
```
// A を B へ圧縮
int count = 0;
for (int i = 0; i < n; ++i) {
    if (A[i] > 0) {
        B[count] = A[i];
        ++count;
    }
}
```

```
// A を B へ展開
int count = 0;
for (int i = 0; i < n; ++i) {
    if (A[i] > 0) {
        B[i] = A[count];
        ++count;
    }
}
```

2 圧縮と展開の典型的なコード



3 圧縮と展開パターンのデータ依存関係グラフ



4 圧縮と展開の実行

しかし、依存関係を特別に処理することで、圧縮と展開コードをベクトル化することが可能です。インテル® AVX-512 ISA の **v[p]compress** と **v[p]expand** 命令は、これらのコードを効率良くベクトル化するのに役立ちます。**図 5** は、新しいインテル® AVX-512 命令 **vcompressps** を使用して生成された圧縮の **simd** コードです。

```
..B1.14:
vmovups (%rdi,%r11,4), %zmm2
vcmpps $6, %zmm0, %zmm2, %k1
kmovw %k1, %edx
testl %edx, %edx
je ..B1.16
..B1.15:
popcnt %edx, %edx
movl $65535, %r10d
vcompressps %zmm2, %zmm1{%k1}
movslq %ebx, %rbx
shlx %edx, %r10d, %r12d
notl %r12d
kmovw %r12d, %k1
vmovups %zmm1, (%rsi,%rbx,4){%k1}
addl %edx, %ebx
..B1.16:
addq $16, %r11
cmpq %r9, %r11
jb ..B1.14
```

5 インテル® AVX-512 による圧縮コードのベクトル化

圧縮と展開コードの自動検出とベクトル化は、単純なケースについてはすでにコンパイラーで実装されていますが、複雑なケースについては保証されていません。複雑なケースでもベクトル化を保証するため、インテルでは OpenMP* 4.5 の **simd** 構文に圧縮と展開を表現するための単純な言語拡張を提案し、実装しました。**図 6** は、OpenMP* の **ordered simd** に追加された新しい **monotonic** 節の使用法を示しています。

```
int j = 0;
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (A[i]>0) {
#pragma omp ordered simd monotonic(j:1)
        {
            B[j++] = A[i];
        }
}
    }
int j = 0;
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (A[i]>0) {
#pragma omp ordered simd monotonic(j:1)
        {
            A[i] = B[j++];
        }
}
    }
```

6 圧縮と展開パターン用のブロックベースの構文

monotonic 節と関連する **ordered simd** コードブロックでは、次のセマンティクスと規則が強制されます。

- **ベクトル反復は構造化コードブロックに到達すると、ステップを評価します。**結果は、**simd** ループに関して不変の整数値またはポインター値でなければなりません。構造化コードブロックの実行に使用される **simd** 実行マスクも計算されます。構造化コードブロックの各 **monotonic** リスト項目のプライベート・コピーが作成され、実行する (**mask==T** の) **simd** 要素 (インデックスの下限から上限の範囲) ごとに **item**、**item + step**、**item + 2 * step**、…、に初期化されます。構造化コードブロックの実行の最後で、項目の統合コピーが **item + popcount(mask) * step** に更新され、リスト項目のプライベート・コピーは未定義になります。
- **関連する ordered simd 構文の範囲外での monotonic リスト項目の使用は許可されません (図 7)。**1 つの **ordered simd** 構文で同じリスト項目を複数回使用することも許可されません。

```
int j = 0;
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (A[i]>0) {
        #pragma omp ordered simd monotonic(j:1)
        {
            ++j;
        }
        B[j] = A[i];
    }
}
```

7 ブロックの範囲外での **monotonic** 項目の使用は許可されない

- **monotonic 節を含む ordered simd 構文**は、**monotonic** 節を含まない **ordered simd** 構文として実装できることがあります。

ヒストグラム

ヒストグラム **図 8**) は、よく知られたイディオムです。ヒストグラム・コードのセマンティクスは、配列 **B[]** の等しい要素をカウントして、結果を配列 **A[]** へ格納します。配列 **B[]** の内容に応じて、このループにはループ伝播依存が存在する可能性があります。

```
for (int i = 0; i < n; ++i) { ++A[B[i]]; }
```

8 単純なヒストグラム・コード

基本的に 3 つの状況が考えられます。

1. **配列 `B[]` の値が固有の場合 :** ループ伝播依存は存在しません。
2. **配列 `B[]` の値がすべて等しい場合 :** 文はスカラー値のリダクションを表しています。
3. **配列 `B[]` でいくつかの値が繰り返される場合 :** ループ伝播依存が存在します。

コンパイル時に、**`B[]`** の値によりデータ依存関係が生じるかどうか証明できない場合、コンパイラーはループ伝播依存があると仮定します。ケース 1 の場合、**`#pragma ivpdep`** を追加することでコンパイラーによるベクトル化を助けることができます。ケース 2 の場合、プログラマーが手動で変換する必要があるかもしれません。ケース 3 の場合、ターゲット・アーキテクチャーに応じて、明示的なベクトル化のアノテーションを追加すると次の方法でベクトル化が実装されます。

- **配列リダクションの実装** (**`simd`** 構文と配列 **`A[]`** のリダクションを使用)
- **シリアル実装** (自動更新文に **`ordered simd`** 構文を使用)
- **ギャザー - 更新 - スキャッター実装**と **`B[]`** の等しい値の繰り返し

ギャザー-更新-スキャッター実装では、インデックスのベクトル内でオーバーラップするインデックスを効率良く計算する、インテル® AVX-512CDI の **`v[p]conflict`** 命令を利用します。ヒストグラム・コードへの対応をプログラマーが行い、インテル® AVX-512CDI を効率良く利用できるようにするため、**図 9** に示すように、インテル® コンパイラーで新しい節 **`overlap`** が提案および実装されました。

```
#pragma omp simd
for (int i = 0; i < n; i++) {
    #pragma omp ordered simd overlap(B[i])
    {
        A[B[i]]++;
    }
}
```

9 ヒストグラム・コードでの **`overlap`** 節の使用例

この新しい節の言語セマンティクスは以下の通りです。

- **実行は `ordered simd` 構文に到達すると**、オーバーラップする `B[i]` の値を rvalue として評価し、結果は整数値またはポインター値になります。`simd` ループに囲まれた 2 つの反復が同じ値を計算する場合、論理反復番号が小さい反復は、もう一方の反復が `ordered simd` コードブロックの実行を開始する前に、そのコードブロックの実行を完了する必要があります。
- **コンパイラーは完全な順序付けを保証しません。** `A[]` へのストアと `A[]` および `B[]` の任意のロードの間で可能なエイリアシングについて、部分的な順序付けで十分かどうか確認するのは、プログラマーの責任です。
- **`ordered simd` ブロックはオーバーラップをチェックする範囲を示します。** そのため、`ordered simd overlap` ブロックの範囲外の文では、オーバーラップのチェックは実行されません。
- **`overlap` 節を含む `ordered simd` 構文**は、`overlap` 節を含まない `ordered simd` 構文として実装できることがあります。

条件付きの `lastprivate` 節

多くのプログラムにおいて、ループは、代入文が実際に実行される (つまり、スカラーへの書き込みが行われる) 最後の反復の値を生成します。スカラーの値は、ループの後の処理で使用されます (**図 10**)。

```
int t = 0;
#pragma omp simd lastprivate(t)
for (int i = 0; i < n; i++) {
    t = A[i];
}
```

```
int t = 0;
for (int i = 0; i < n; i++) {
    if (A[i] > 0) {
        t = A[i];
    }
}
```

10 左のコードは `A[i]` の最後の要素を取得し、右のコードは `A[i]` の最後の正の要素を取得する

図 10 のコード例はどちらもベクトル化できますが、左のコードのみ `lastprivate` 節により OpenMP* で明示的にサポートされます。右のコードでは、セマンティクスが原因で `lastprivate` 節を使用することができません。スカラー `t` はループの最後に実行された反復から値を受け取るため、`A[n-1]>0` が真でない場合、`t` は未定義になります。この問題に対応するため、新しい修飾子 `conditional` が提案され、`lastprivate` 節に追加されました (**図 11**)。

```
int t = 0;
#pragma omp simd lastprivate(conditional:t)
for (int i = 0; i < n; i++) {
    if (A[i] > 0) {  t = A[i]; }
}
```

11 スカラーへの条件付き代入をサポートする **lastprivate** の例

新しい **conditional** 修飾子の言語セマンティクスは以下の通りです。

- **条件付き lastprivate 節のリスト項目**は、**private** 節のセマンティクスに依存します。**simd** 構文の最後で、オリジナルのリスト項目は、ループのスカラー実行中に字句的に最後の条件付き代入が実行されたかのように値を受け取ります。
- 定義前に使用される場合、または定義の範囲外では、**項目を使用すべきではありません。**

早期終了を含むループ

OpenMP* 4.5 の既存の **simd** ベクトル化は、複数の出口を持つループをサポートしません。例えば、配列のある要素のインデックスを特定するような一般的な使用法 (**図 12**) は、既存の **simd** 構文では適切に明示的にベクトル化できません。

```
for (int i = 0; i < n; ++i) {
    if (A[i] == B[i]) {
        j = i; break;
    }
}
// j の使用
```

```
int j = 0;
#pragma omp simd early _ exit lastprivate(conditional:j)
for (int i = 0; i < n; i++) {
    if (A[i] == B[i]) { j = i; break; }
}
// j の使用
```

12 配列 **A[]** と **B[]** の最初の等しい要素を探す、2 つの出口があるループ

この問題に対応するため、インテル® コンパイラーに以下のセマンティクスを持つ新しいループレベルの節 **early_exit** (**図 12** の右側) が追加されました。

- ループの最後の字句的な早期終了前の各操作は、simd 範囲内で早期終了がトリガーされなかったかのように実行されることがあります。
- ループの最後の字句的な早期終了後、すべての操作は、ループの最後の反復が見つかったかのように実行されます。
- スカラー実行の **`linear`** および **`conditional lastprivate`** の最終値は保持されます。
- リダクションの最終値は、最後の **`simd`** 範囲の最後の反復がループ終了時に実行されたかのように計算されます。
- スカラー実行の共有メモリーの状態は保持されない可能性があります。
- 例外は許可されていません。
- 最内ループに **`simd`** 構文の範囲外への出口がある場合、最内ループの実行は完全にアンロールした場合と同様になります。

パフォーマンス結果

表 2 から **表 4** は、**`monotonic`** 節、**`overlap`** 節、**`early_exit`** 節を指定した場合のマイクロカーネル・プログラムのパフォーマンス結果です。パフォーマンスの測定には、リリース前のインテル® Xeon® スケーラブル・プロセッサー 2.10GHz、2 ソケット、2666MHz DIMM (24 枚) を搭載したシステムを使用しました。実際の結果はハードウェアやソフトウェアの構成によって異なります。前述のセクションで紹介したループの例を使用し、ループのトリップカウント **`n`** は 109、データ型はすべて 32 ビット整数、ベクトル長は 16 として、**`icc -xCORE-AVX512`** コマンドでコンパイルしました。ベースラインは、ループの同じセットをスカラー実行した結果です。

表 2 は、ベクトル化された圧縮および展開コードの正規化したスピードアップです。「すべて False」列では、すべての圧縮/展開条件が False です。圧縮ストアや展開ロードは完全にスキップされます。条件のベクトル評価とベクトル化による反復数の減少により、さらに 1.17 倍～ 1.19 倍のスピードアップを達成できます。「すべて True」列では、圧縮/展開条件が True のため、16 要素すべてがストア/ロードされます。

表 2. スカラー実行と比較した圧縮/展開の正規化したスピードアップ - 各列の True 条件の数は異なる

	すべて False	2 要素	4 要素	8 要素	15 要素	すべて True
圧縮	1.20 倍	1.21 倍	1.25 倍	1.32 倍	1.81 倍	4.09 倍
展開	1.20 倍	1.18 倍	1.18 倍	1.13 倍	1.35 倍	3.98 倍

間の列では、16 ウェイベクトルの 2、4、8、または 15 要素が圧縮ストア/展開ロードされます。この表から、実際に圧縮/展開された要素数に関係なく、ベクトル化されたバージョンのほうがスカラー実行よりもパフォーマンスが優れていることが分かります。同じテストを、アライメントされた/されていない配列の圧縮/展開で実施したところ、アライメントされたケースのほうがキャッシュラインに効率良くアクセスできる「すべて True」の場合を除いて、予想通りに同様の傾向がみられました。

表 3 は、ベクトル化されたヒストグラムの例の正規化したスピードアップです。左の「オーバーラップなし」のケースでは、スカラー実行よりも 23% も遅いことが分かります。オーバーラップなしをアサートして、オーバーラップ拡張を使用しないで済む別の方法を見つけることを検討してください。場合によっては、短いベクトル長を選択するだけで済むこともあります。ベクトルを適切に実行するため競合の解決が必要な場合、ベクトル化によってループのほかの部分に大きな利点がもたらされなければなりません。興味深い点は、繰り返し 8 回と完全なオーバーラップ (繰り返し 16 回) のケースのほうが、繰り返し回数が少ないケースよりもスピードアップしていることです。これは、ベクトル実行のレジスターベースの繰り返しが、スカラー実行のオンメモリー依存のスピードを上回っているためだと考えられます。競合パフォーマンスに関する詳細は、この記事の最後にある参考資料 9 を参照してください。

表 3. スカラー実行と比較したヒストグラムの正規化したスピードアップ - 各列の 1 ベクトル反復内の競合の数は異なる

	オーバーラップなし	繰り返し 2 回	繰り返し 4 回	繰り返し 8 回	完全なオーバーラップ
ヒストグラム	0.77 倍	0.65 倍	0.66 倍	0.82 倍	1.3 倍

表 4 は、ベクトル化された検索ループの例の正規化したスピードアップです。一致が見つかったループ・インデックス値の範囲は 0 から 5,000 です。わずかなベクトル反復回数でも、検索ループのベクトル化によるスピードアップが見られます。

表 4. スカラー実行と比較した検索ループの正規化したスピードアップ

一致したループ・インデックス	i=0	i=5	i=10	i=50	i=100	i=500	i=1,000	i=5,000
ベクトル反復回数	1	1	1	4	9	32	63	313
検索ループ	1.0 倍	0.99 倍	1.02 倍	1.31 倍	1.54 倍	3.04 倍	4.12 倍	4.46 倍

ここで述べた明示的なベクトル化手法の標準化は、まだ進行中であることに留意してください。構文、セマンティクス、パフォーマンス特性は、将来の実装では変更される可能性があります。また、一部の機能は、インテル固有の拡張となる可能性があります。

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

インテル® AVX-512 向けチューニングのベスト・プラクティス

ベースラインの設定

すべてのチューニング作業は、適切なパフォーマンス・ベースラインを設定することから始まります。これは、インテル® Xeon® スケーラブル・プロセッサーでも同じです。インテル® コンパイラー 17.0 の **-xCORE-AVX2** オプションを使用して、すでにインテル® Xeon® プロセッサー向けにある程度チューニングされたアプリケーションがあると仮定します。最初に、インテル® コンパイラー 18.0 で、次の 3 つのバイナリーを生成することを推奨します。

1. **-xCORE-AVX2 -mtune=skylake-avx512** (Linux* および macOS*) または **/QxCORE-AVX2 /tune=skylake-avx512** (Windows*) でビルドしたバイナリー (以降、バイナリー A と呼ぶ)
2. **-xCORE-AVX512** (Linux* および macOS*) または **/QxCORE-AVX512** (Windows*) でビルドしたバイナリー (以降、バイナリー B と呼ぶ)
3. **-xCORE-AVX512 -qopt-zmm-usage=high** (Linux* および macOS*) または **/QxCORE-AVX512 /Qopt-zmm-usage=high** (Windows*) でビルドしたバイナリー (以降、バイナリー C と呼ぶ)

バイナリー B と C の全体のパフォーマンスは、バイナリー A と同程度か上回ることが予想されます。

最適化レポートから hotspot を見つける

-qopt-report (Linux* および macOS*) または **/Qopt-report** (Windows*) オプションを指定してコンパイルすると、インテル® コンパイラーは最適化レポートを生成します。

コンパイルの設定を変更する前に、コンパイラーが何を行い、何が判明しているのかを理解したほうが良いでしょう。この記事の最後にある参考資料 1 では、最適化レポートの使用方法を紹介しています。[**編集者注: The Parallel Universe Issue 27** の「インテル® AVX-512 で向上したベクトル化のパフォーマンス」でも、コンパイラー・レポートの最新の概要を説明しています。]

hotspot のパフォーマンスの比較

hotspot に応じて、各種コンパイラー・オプションの組み合わせの影響は異なります。ある hotspot はバイナリー B で高速に動作し、別の hotspot はバイナリー C で高速に動作する場合、それぞれのケースでコンパイラーが何を行ったかを理解することは、両方の hotspot で最適なパフォーマンスを達成するのに非常に役立ちます。

ZMM の使用の調整

バイナリー B と C では、OpenMP* の **simd** 構文の **simdlen** 節を使用して、ベクトル長を明示的に制御できます。いくつかの hotspot でベクトル長を長くまたは短くして、アプリケーション・パフォーマンスへの影響を確認してみてください。複数のチューニング手法を組み合わせる必要があるかもしれません。

アライメント

インテル® AVX-512 のベクトルロード/ストア命令を使用する場合、最適なパフォーマンスを得るため、データを 64 バイトでアライメントすることを推奨します[6]。そうしないと、64 バイトのキャッシュラインでは、64 バイトのインテル® AVX-512 のアライメントされていないロード/ストアを実行するたびに、キャッシュライン分割が発生します。これは、32 バイト・レジスターを使用するインテル® AVX2 と比較して 2 倍の頻度です。メモリー依存のコードでキャッシュライン分割が頻繁に発生すると、パフォーマンスが 20% ~ 30% 低下する可能性があります。次の構造体について考えてみましょう。

```
struct RGB_SOA {
__declspec(align(64)) float Red[16];
__declspec(align(64)) float Green[16];
__declspec(align(64)) float Blue[16];
}
```

この構造体を次のように使用すると、64 バイトでアライメントされたメモリーが割り当てられます。

```
RGB_SOA rgb;
```

しかし、次のように動的メモリー割り当てを使用すると、**__declspec** アノテーションは無視され、メモリーが 64 バイトでアライメントされる保証はありません。

```
RGB_SOA* rgbPtr = new RGB_SOA();
```

このケースでは、アライメントを行う動的メモリー割り当ての使用と **new** 演算子の再定義のいずれかまたは両方を行うべきです。インテル® AVX-512 では、可能な場合は、次のアプローチを使用してデータを 64 バイトにアライメントします。

- インテル® コンパイラーの **_mm_malloc** 組込み関数を使用するか、Microsoft* コンパイラーの **_aligned_malloc** を使用して動的にデータをアライメントします。
  例: `DataBuf = (float *)_mm_malloc (1024 * 1024 * sizeof(float), 64);`
- **__declspec(align(64))** を使用して静的にデータをアライメントします。
  例: `__declspec(align(64)) float DataBuf[1024*1024];`

データの割り当てと使用は、異なるサブルーチン/ファイルで行われる可能性があります。そのため、コンパイラーはデータの使用位置では、データの割り当て位置でアライメントが最適化されているかどうか分かりません。**__assume_aligned()/ASSUME_ALIGNED** は一般的なアサーションで、データの使用位置に挿入することでデータがアライメントされていることを示します。(詳細は、この記事の最後にある参考資料 2 と『インテル® コンパイラー・デベロッパー・ガイドおよびリファレンス』(**こちらのページ**からインテル® Parallel Studio XE の評価版をお申し込みいただくことで入手可能) を参照してください。)

ピールすべきかどうか

ベクトライザーは、多くの場合、1 つのループに対して次の 3 つのバージョンを生成します。

1. ループの最初の処理 (ピールループ)
2. メインのベクトルループ
3. ループの最後の処理 (リマインダー・ループ)

ピールループは、アライメントの最適化のために生成されますが、この最適化は諸刃の剣と言えます。ループのトリップカウントが小さい場合、メインのベクトルループで処理されるデータ要素の数が少なくなることがあります。また、コンパイラーが不適切な配列をピールすると、ループ実行のデータ・アライメントが悪化する可能性があります。**-qopt-dynamic-align=F** (Linux* および macOS*) または **/Qopt-dynamic-align=F** (Windows*) コンパイラー・オプションを使用することで、ループピーリングの最適化を抑止できます。**#pragma vector unaligned** はループ単位で使用できます。ループの入り口ですべてのデータがアライメントされている場合は **#pragma vector aligned** を使用します。

ピール / リマインダーとトリップカウントが小さいループのベクトル化

インテル® AVX-512 のマスキングは、ベクトル化されたピールおよびリマインダー・ループの生成とトリップ数の小さいループのベクトル化を可能にします。しかし、これらの特殊なループのベクトル化バージョンでは、ほとんどの命令をマスクしなければならないため、必ずしもパフォーマンスが向上するとは限りません。パフォーマンスの問題の主な原因として、ストアフォワードの制限、およびある操作が別の操作に依存する場合にマスク付きのマージ操作でストールする可能性が挙げられます。コンパイラーは、データフローの詳細な解析により、できるだけマスキングを回避することで、ストアフォワードの問題に対応できます。ストアフォワードの問題が避けられない場合は、ループのマスクなしバージョン (例: スカラーまたは短いベクトル長でベクトル化されたループ) を使用したほうが良いでしょう。マスク付きのマージ操作でのストールは、ゼロマスキング操作を使用することで回避できる可能性があります。次のプラグマは、コンパイラーの動作を変更するヒントを提供します。

- リマインダー・ループ :

  #pragma vector novecremainder - リマインダー・ループをベクトル化しません。
  #pragma vector vecremainder - コンパイラーのコストモデルに応じてリマインダー・ループをベクトル化します。
  #pragma vector always vecremainder - 常にリマインダー・ループをベクトル化します。

- トリップカウントが小さいループ :

  #pragma vector always - ベクトル化を強制します。
  #pragma novector - ベクトル化を無効にします。

ギャザーおよびスキャッターの最適化

ギャザーおよびスキャッター命令を利用することで、より多くのループをベクトル化できます。しかし、ベクトル化されたコードが必ずしもパフォーマンスを向上させるとは限りません。ギャザー/スキャッター・コードを含むループのパフォーマンスは、計算の数とデータのロード/ストアの数の比率と、ギャザー/スキャッター命令のレイテンシーを軽減する最適なベクトル長の選択に依存します。ベクトル長が短くなると、ギャザー/スキャッター・コードのレイテンシーも短くなります。通常、ソースコード・レベルでいくつかの単純な最適化を行うことで、ギャザーおよびスキャッター命令の数を軽減できます。

以下に、2 つの単純な手動によるギャザー/スキャッターの最適化を示します。コンパイラーも自動でこれらの最適化を試みます。1 つ目の最適化は、2 つのギャザー/スキャッターの結果をブレンドする (**図 13** の左のコード) 際のギャザー/スキャッターの数を減らします。このケースでは、インデックスをブレンドすることで、2 つのギャザーの代わりに 1 つのギャザーで処理を行うことができます (右のコード)。同様の方法でスキャッターも最適化できます。

2 つ目のギャザー/スキャッターの最適化は、逆の変換を行います (**図 14**)。このケースでは、ユニットストライドの線形インデックスのブレンドから得たインデックスを使用してギャザーを実行します (左のコード)。パフォーマンスを向上するには、2 つのユニットストライドのロードを実行してから、ブレンド処理を行います (右のコード)。

```
// ループ中に 2 つのギャザーがある
float *a, sum = 0; int *b, *c;
… …
for (int i; i < n; i++) {
  if ( pred(x[i]))
      sum += a[b[i]]; // ギャザー
  else
     sum += a[c[i]]; // ギャザー
}
```

```
// ループ中のギャザーを 1 つにする最適化
for (int i; i < n; i++) {
  int t;
  if ( pred(x[i]))
     t = b[i];
  else
      t = c[i];
  sum += a[t]; // 1 つのギャザーで処理
}
```

13 ループ中のギャザーを減らす

```
// ループ中の 1 つのギャザー
float *a, sum = 0; int a;
for (int i; i < n; i++) {
  int t;
  if (pred(x[i])) {
      t = i + b;
  }
  else {
    t = i;
  }
  sum += a[t]; // ギャザー
}
```

```
// ギャザーをユニットストライド・ロード + ブレンドに置換
for (int i; i < n; i++) {
  float s;
  if ( pred(x[i])) {
      s = a[i + b];  // ユニットストライドのベクトルロード
  }
  else {
      s = a[i];      // ユニットストライドのベクトルロード
  }
  sum += s;
}
```

14 ループ中のギャザーを 2 つのユニットストライドのロード + ブレンドに置換

実行レイテンシーの改善

ループのベクトル化では、コンパイル時にループのトリップカウントが不明であるという、コンパイラーにとって既知の問題があります。しかし、プログラマーは多くの場合、トリップカウントを予測して、**`#pragma loop count`** でコンパイラーにヒントを提供することができます。一部のケースでは、**`#pragma unroll`** の情報を基にループのおおよそのトリップカウントを予測することが可能です。ループの本体が、ループの反復数を軽減し、反復の独立性を高められる大きさの場合、アウトオブオーダーのインテル® アーキテクチャーで並列に実行して、実行レイテンシーを隠蔽できるように **`unroll`** プラグマを使用することを推奨します。ループアンロールは、計算依存の (つまり、メモリーアクセスよりも計算に時間がかかる) ループに対して非常に有効です。例えば、**図 15** のループは、アンロール係数 (UF) が 8 の場合、UF が 0 の場合よりも 45% 高速に動作します (n=1,000 で測定)。

```
float *a,*b, *c;
#pragma unroll(8)
#pragma omp simd
for (i= 0; i < n; i++) {
  if (a[i] > c[i]) sum += b[i] * c[i];
}
```

15 **`simd + unroll(8)`** を使用してレイテンシーを軽減

まとめと今後の課題

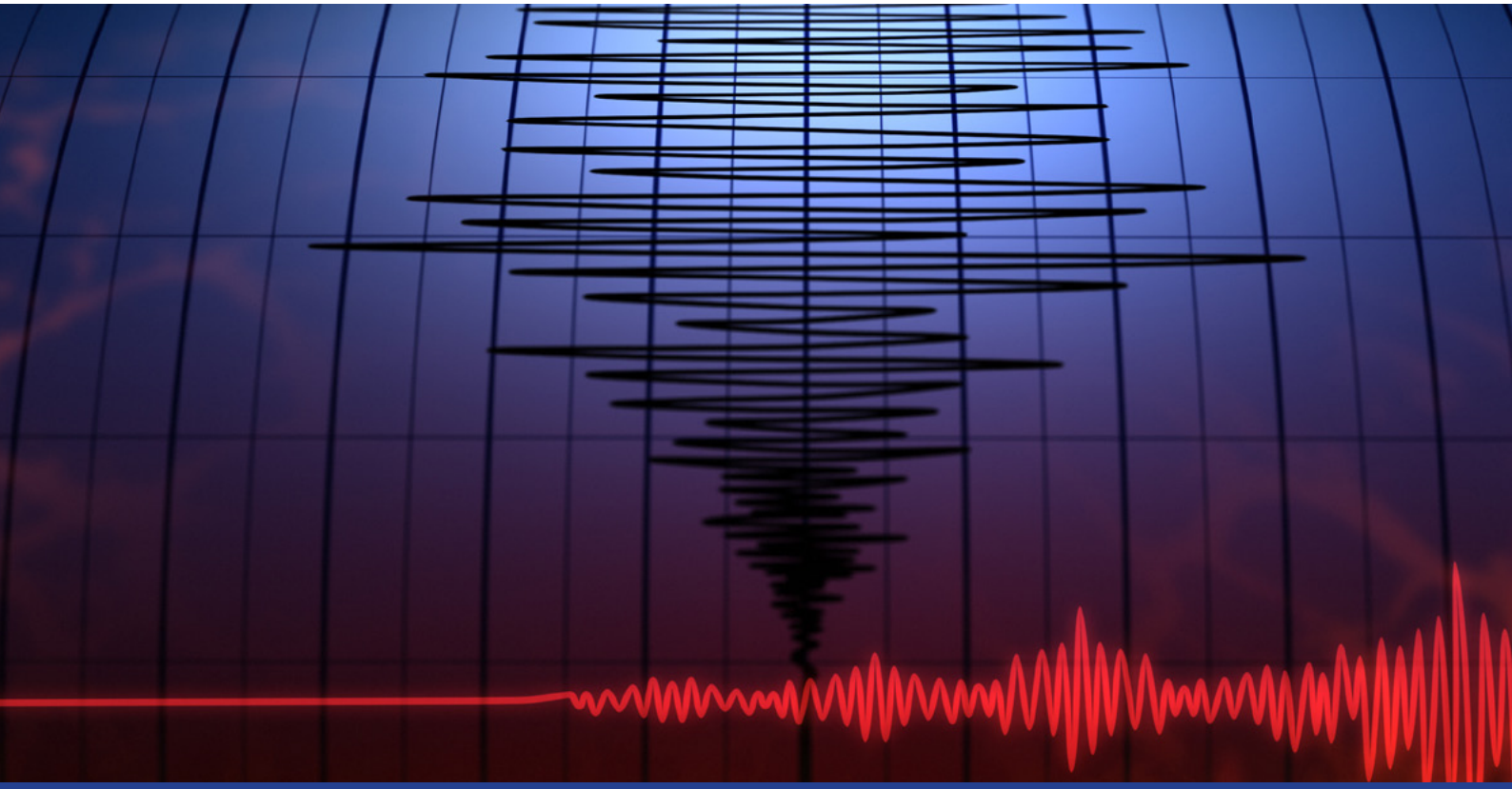
この記事では、インテル® コンパイラー 18.0 のインテル® Xeon® スケーラブル・プロセッサー向けインテル®AVX-512 サポートに含まれるいくつかの新しい **simd** 言語拡張について説明しました。インテル® AVX-512 で最適なパフォーマンスを達成する、パフォーマンスの最適化とチューニング手法も紹介しました。新しい **simd** 拡張とチューニング手法の効果を示すため、マイクロカーネルのパフォーマンス結果を提供しました。今後に向けて、OpenMP* および C++ Parallel STL (包括的スキャン、排他的スキャン、範囲ベース C++ ループ、ラムダ式のサポートを含む) を利用してインテル® AVX-512 を最大限に利用する C/C++ 向けのいくつかの新しい **simd** 拡張が検討されています。

関連資料

- **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® Xeon® スケーラブル・プロセッサー >**

参考資料

1. M. Corden. “**Getting the Most out of your Intel® Compiler with the New Optimization Reports**,” Intel Developer Zone, 2014. 日本語訳:「**新しい最適化レポートを使用してインテル® コンパイラーをさらに活用する**」
2. R. Krishnaiyer. “**Data Alignment to Assist Vectorization**,” Intel Developer Zone, 2015. 日本語訳:「**ベクトル化の可能性を高めるデータ・アライメント**」
3. H. Saito, S. Preis, N. Panchenko, and X. Tian. “Reducing the Functionality Gap between Auto-Vectorization and Explicit Vectorization.” In Proceedings of the International Workshop on OpenMP (IWOMP), LNCS9903, pp. 173-186, Springer, 2016.
4. H. Saito, S. Preis, A. Cherkasov, and X. Tian. “**Obtaining the Last Values of Conditionally Assigned Privates**,” OpenMPCon Developer Conference, 2016.
5. H. Saito, “**Extending LoopVectorizer: OpenMP4.5 SIMD and Outer Loop Auto-Vectorization**,” presentation at LLVM Developer Conference, 2016.
6. X. Tian. H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A. Duran. “Putting Vector Programming to Work with OpenMP* SIMD,” **The Parallel Universe magazine, Issue 22**, September 2015. 日本語訳:「**OpenMP* SIMD によるベクトル・プログラミング**」
7. X. Tian, R. Geva, B. Valentine. “Unleash the Power of AVX-512 through Architecture, Compiler and Code Modernization,” ACM Parallel Architecture and Compiler Technology, September 11-15, 2016, Haifa, Israel.
8. X. Tian, H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A. G. Cherkasov, C. Nelson, N. Panchenko, R. Geva. Compiling C/C++ SIMD Extensions for Function and Loop Vectorizaion on Multicore-SIMD Processors. IEEE IPDPS Workshops 2012: 2349-23
9. Intel Corporation. **“Vectorization with Conflict Detection” In Section 15.16.1 of Intel® 64 and IA-32 Architectures Optimization Reference Manual**, October 2017. 日本語訳:『**インテル® 64 アーキテクチャーおよび IA-32 アーキテクチャー最適化リファレンス・マニュアル**』の 15.16.1 節「競合検出とベクトル化」



クラスター全体の効率良い使用

インテル® アーキテクチャー上での SPECFEM3D_GLOBE パフォーマンスの最適化

Rama Kishan Malladi インテル コーポレーション テクニカル・マーケティング・エンジニア

インテルは、最新の**インテル® Xeon® プロセッサー**や**インテル® Xeon Phi ™ プロセッサー**などのデータセンター向けハードウェアとソフトウェアの両方におけるリーディング・プロバイダーです。しかし、多くのハイパフォーマンス・コンピューティング (HPC) アプリケーションは、プロセッサーの高度な機能を最大限に利用しているわけではありません。この記事では、3 次元の世界的および地域的な地震波伝播をシミュレートして、全波形イメージング (FWI: Full Waveform Imaging) またはスペクトル要素法 (SEM) に基づく随伴断層撮影を実行するソフトウェア・パッケージ、**SPECFEM3D_GLOBE** (英語) のパフォーマンスを向上するステップバイステップの手法を提供します。すべての SPECFEM3D_GLOBE ソフトウェアは、完全な移植性を考慮して Fortran 2003 で記述されており、Fortran 2003 標準に厳密に準拠しています。分散メモリー並列処理は、メッセージ・パッシング・インターフェイス (MPI) を使用して記述されています。最近、ソルバーのソースコードに OpenMP* の共有メモリー並列構文が追加されました。

次の手順に従って、このコードをビルドして実行します。

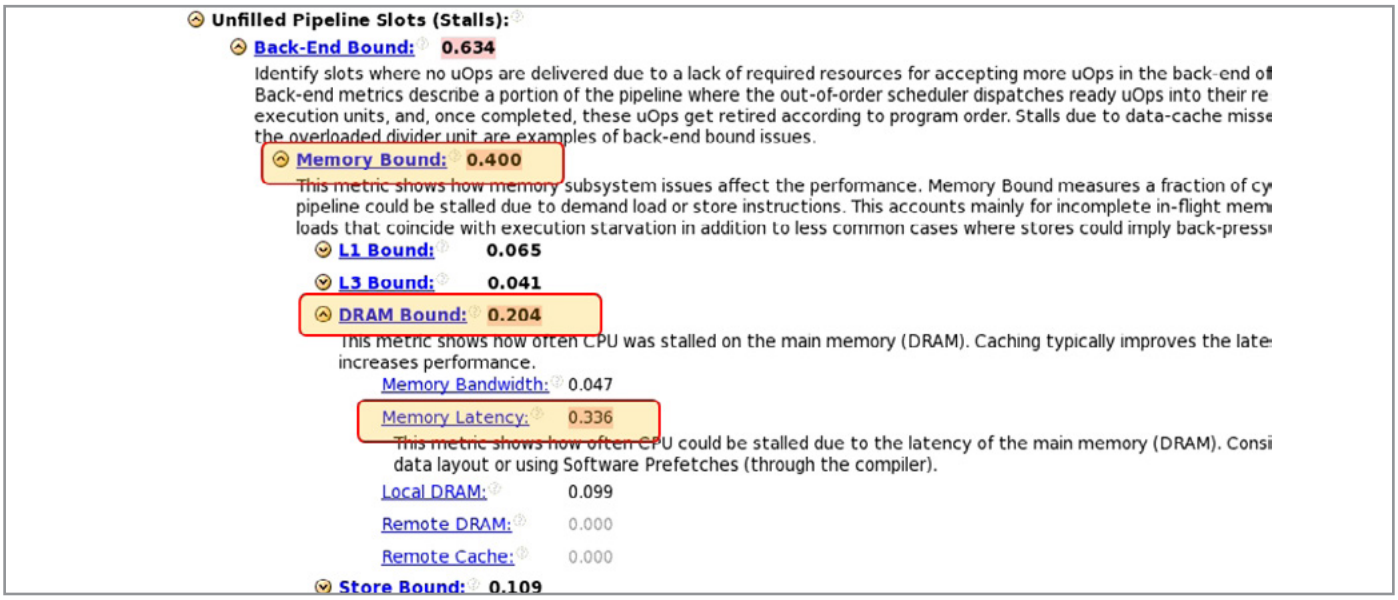
```
$ cd specfem3d_globe
$ ./configure FC=ifort MPIFC=mpiifort CC=icc CXX=icpc FCFLAGS="-O3 -xMIC-AVX512
-qopenmp"
$ cp EXAMPLES/small_benchmark_run_to_test_more_complex_Earth/Par_file DATA
$ make -j 8 xcreate_header_file xmeshfem3D xspecfem3D
$ cd EXAMPLES/small_benchmark_run_to_test_more_complex_Earth
$ ./run_this_example.sh
```

注

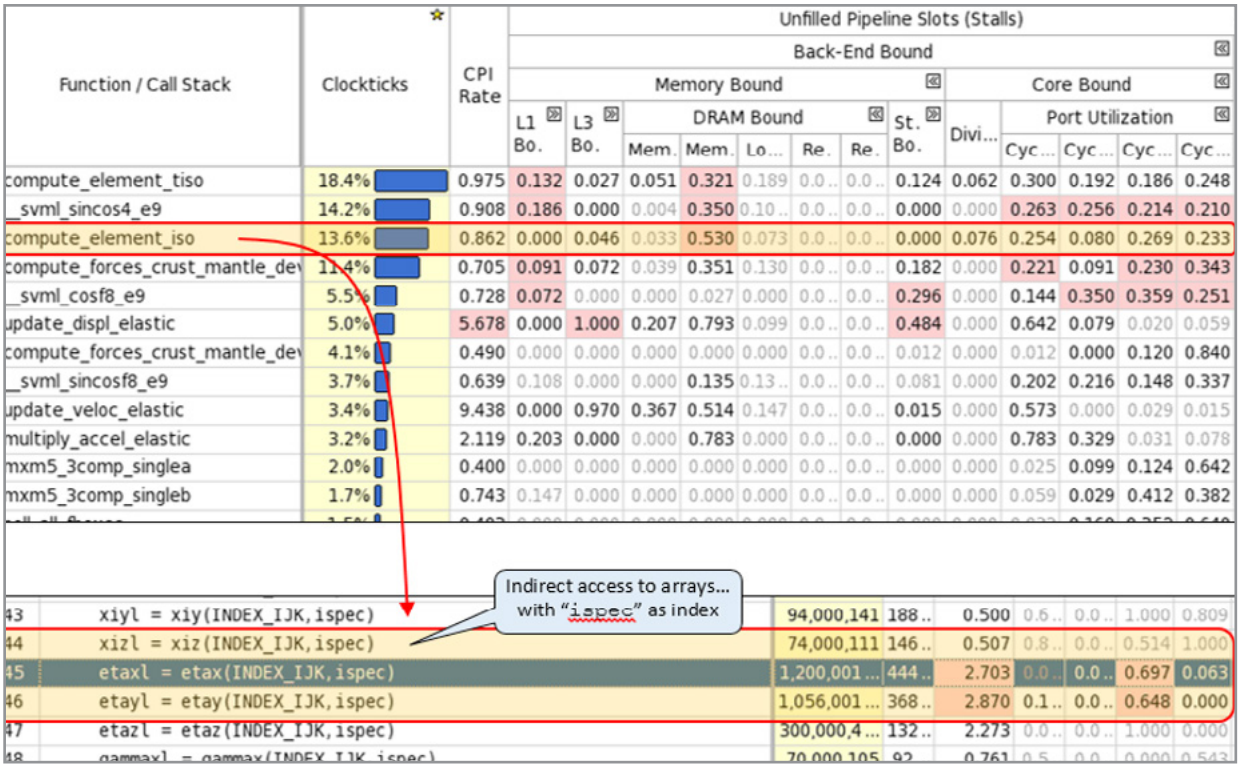
- SPECFEM3D_GLOBE は、**https://geodynamics.org/cig/software/specfem3d_globe/** (英語) からダウンロードできます。
- 実行するテスト・ベンチマークは、EXAMPLES ディレクトリー以下の small_benchmark_run_to_test_more_complex_Earth です。
- ビルドには、インテル® コンパイラーとインテル® MPI ライブラリー、および 'icc'、'icpc'、'ifort'、'mpiifort' の設定オプションを使用します。
- ビルド / 実行には、インテル® Xeon Phi™ プロセッサー (開発コード名 : Knights Landing) ベースのシステムを使用するため、'-xMIC-AVX512' コンパイラー・オプションを使用します。
- '-qopenmp' コンパイラー・オプションは、OpenMP* スレッドが有効なパッケージのビルドで使用します。
- ソルバーコードのパフォーマンスは、指定されたメッシュボリュームについて 'n"' タイムステップのシミュレーションの実行時間で測定します。
- モデルの解像度、パラメーター、MPI プロセス数の編集手順については、SPECFEM3D_GLOBE のドキュメントを参照してください。

ソルバーが正常に動作すると仮定した場合、効率良く実行され、利用可能なリソースを活用できているかどうか知る必要があります。ランタイム実行のプロファイルから、ソルバーコードの各種サブルーチンで費やされた時間のプレビューが得られます。複数のクラスターノードにわたってプロファイルを収集することで、MPI の動作と各ノードの実行統計を監視できます。アプリケーション・パフォーマンス・スナップショットや**インテル® Trace Analyzer & Collector** などのツールは、MPI 実行プロファイルの収集機能を備えています。さらに、シングルノードのプロファイルは、**インテル® VTune™ Amplifier** や**インテル® Advisor** で収集できます。SPECFEM3D_GLOBE は、非同期 MPI 通信 / 計算オーバーラップにより優れた MPI スケーラビリティーを提供するため、ここではノードごとのプロファイルと最適化に注目します。

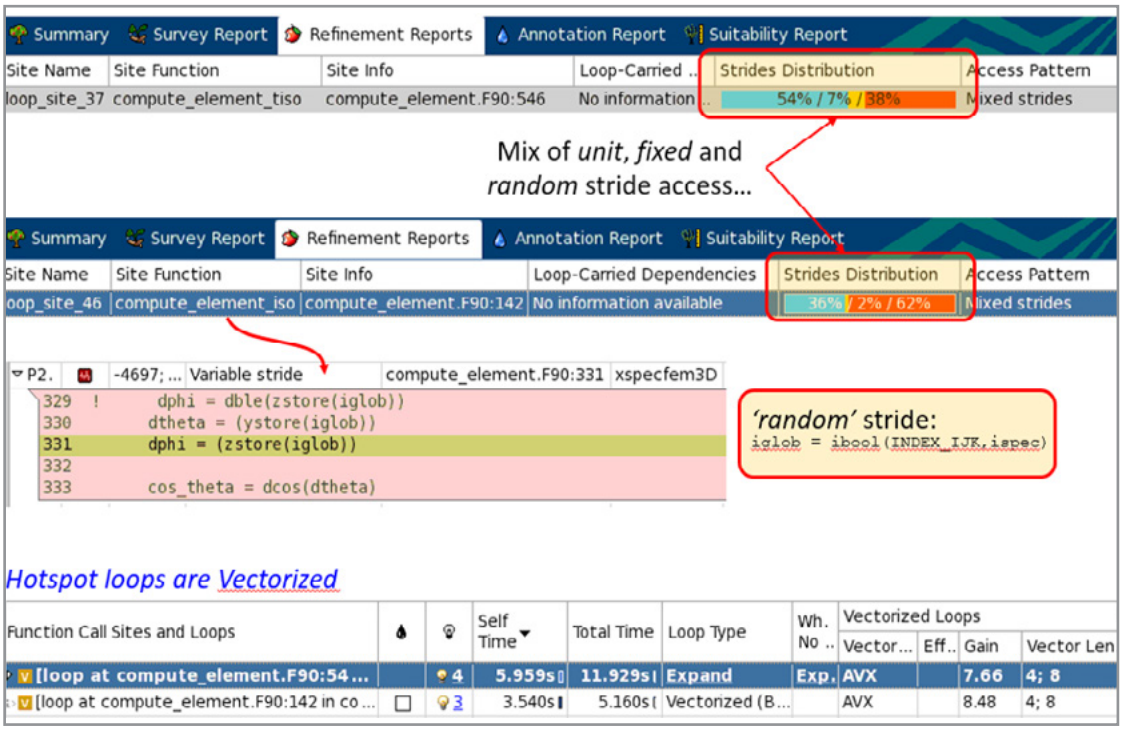
インテル® VTune™ Amplifier により収集される全般解析プロファイルは、このコードがバックエンド依存であることを示しており、さらに詳しく見るとメモリー (DRAM) レイテンシーの問題があることが分かります。**図 1** は、全般解析の [Summary (サマリー)] ビューです。hotspot、そしてソースコードへドリルダウンすると、**図 2** のプロファイルが表示されます。**compute_element_iso** サブルーチンの中でアクセスされる **etax** 配列の次元は 125 x N です。**INDEX_IJK** は 1 から 125 までインクリメントします。インデックス **ispec** は任意で、間接アクセスです。インテル® Advisor などのツールは、このアクセスのランダム性や (コンパイラーにより生成される) コードのベクトル化に関する詳細を提供します。**図 3** のようなプロファイルが得られます。



1 インテル® VTune™ Amplifier の全般解析



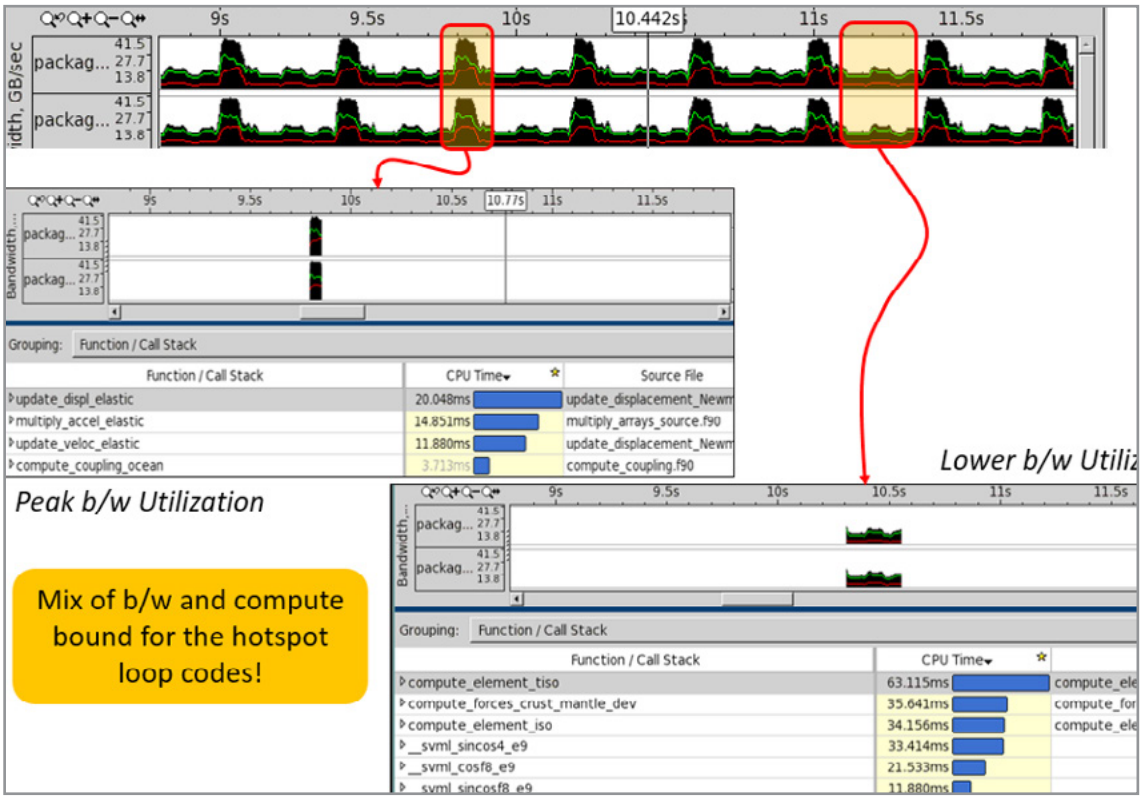
2 インテル® VTune™ Amplifier で hotspot、ソースコードへドリルダウン



3 インテル® Advisor: ユニット / ランダムストライドの分布、ソースへのドリルダウン、ベクトル強度 / ベクトル化されたコード

インテル® Advisor は、ストライドの種類 (ユニット、ランダム、または固定) と分布を含む、メモリー・アクセス・ストライドに関する情報を表示します。また、コンパイラーによるコード生成とループの実行に使用されるベクトル長、命令セット、ベクトル化によるゲインに関する情報も示します。このプロファイルは、アプリケーションが、ベクトル機能やプロセッサーの新しい命令セットによる恩恵を受けられるか判断するのに役立ちます。場合によっては、プロセッサーのベクトル幅をすべて利用するため (インテル® AVX-512)、コンパイラーによってループがベクトル化されても、ループの実行がスピードアップしないことがあります。その 1 つの理由として、メモリーアクセスが帯域幅による制限を受けているものと思われます。インテル® VTune™ Amplifier は、そのような状況のプロファイルを行うことができます (**図 4**)。ループのベクトル・パフォーマンスを向上する最適化では、帯域幅に依存するため、必ずしも良い結果が得られるわけではないことを理解することが重要です。帯域幅の使用率が低いループとコード領域は、ベクトル命令以外を実行していたり、メモリー・アクセス・レイテンシーの問題がある可能性があります。これは修正する必要があります。

SPECFEM3D_GLOBE の実行プロファイルを解析した後、インテル® プロセッサー上でのパフォーマンスが向上するよう、いくつかのコード変更を試してみました。



4 インテル® VTune™ Amplifier のメモリー帯域の解析

メモリー・アクセス・レイテンシーの問題の軽減

間接 (ランダム) アクセスをユニットストライド・アクセスに変換しました。SPECFEM3D_GLOBE ソルバーのほとんどのメッシュデータは、ソルバーのステップを通して不変です。そのため、データをコピーしてリニアアクセスにすることは、有効な変換と言えます。

オリジナルのコード

```
subroutine compute _ element _ tiso(ispec,
...
    c13 = 0.125 _ CUSTOM _ REAL*cosphisq*(rhovphsq + six _ eta _ aniso*rhovphsq \
...
```

初期化/スタートアップ・コード

```
prepare _ timerun (...)
...
    do ispec _ p = 1,num _ elements
      ele _ num = ele _ num + 1
...
          ia _ c1store(idx6+3,tiso _ ele _ num) = c13

----------------------------------------------------------------
```

最適化されたコード

```
subroutine compute _ element _ tiso(ispec, ele _ num, &
...
    c13 = ia _ c1store(idx7+3,tiso _ ele _ num)
...
```

このコード変更は、9 つの配列 —' **xix**'、'**xiy ⋯ gammzl**' — をすべて 1 つの配列 '**ia _ arr**' へコピーします。これにより、(アクセスしなければならない配列が 9 から 1 になるため) 配列アクセスによる帯域幅の負荷が緩和されますが、(各配列の要素が連続していないため) ベクトル化が困難になります。帯域幅の負荷があまり懸念されないケースでは、オリジナルのコードに対応する 9 つの配列を作成して、ベクトル化の恩恵を受けることができます。最適なオプションは、試行錯誤によって決定する必要があります。

コンパイラーによるベクトル化 / ループ・フィッション

計算ループ '**iso**' と '**tiso**' は非常に大きく、コンパイラーはベクトル化することができません。そのため、手動でループ・フィッションを行いました。ループ分散 / フィッション向けにインテル® コンパイラーでサポートされる '**!DIR$ DISTRIBUTE POINT**' 構文を使用することで、同様の効果が得られます。

データ・アライメント / パディング

計算ループ '**iso**' と '**tiso**' は、メッシュの各要素で MPI またはスレッド領域から呼び出されます。これらのループのトリップカウントは 125 です。ループに関連付けられた配列の次元は 125 x N であるため、2n 境界でアライメントする最適化を適用しました。3 要素のパディングを行い 128 x N 配列になるようにしました。

オリジナルのコード

```
subroutine compute _ element _ tiso(ispec,
...
    c13 = 0.125 _ CUSTOM _ REAL*cosphisq*(rhovphsq + six _ eta _ aniso*rhovphsq \
...
```

```
prepare _ timerun (...)
...
    do ispec _ p = 1,num _ elements
      ele _ num = ele _ num + 1
...
          ia _ c1store(idx6+3,tiso _ ele _ num) = c13

----------------------------------------------------------------
```

最適化されたコード

```
subroutine compute _ element _ tiso(ispec, ele _ num, &
...
    c13 = ia _ c1store(idx7+3,tiso _ ele _ num)
...
```

ルックアップ・テーブルによる冗長な計算の置換

‘**tiso**’ ループには、超越関数を呼び出すいくつかの計算が含まれています。これらの計算は、ソルバーの実行中のステップで不変です。そのため、ルックアップ・テーブルを利用して置換できます。

IVDEP または SIMD ディレクティブ

SPECFEM3D_GLOBE ソルバーのいくつかの hotspot は、トリップカウントが 5 x 5 と 5 x 25 の入れ子のループです。これらは、‘m x m’ 行列 - 行列乗算です。コンパイラーの最適化レポート (**-qopt-report** オプションを使用) から、これらのループのうちベクトル化されていないものがあることが分かります。IVDEP または SIMD ディレクティブを使用して、コンパイラーがこれらのループのベクトル化されたコードを生成できるようになりました。

要約すると、いくつかの単純なコード変更 (とデータ変換) によって、インテル® Xeon Phi™ プロセッサー・ベースのシステム上で SPECFEM3D_GLOBE ソルバーのパフォーマンスが約 2.1 倍になりました。このコードには、さらなる最適化の余地があり、現在その可能性を調査中です。

システム構成

- インテル® Parallel Studio XE 2017
- インテル® Xeon Phi™ プロセッサー 7250、96GB DDR メモリー搭載のセルフブート・システム
- インテル® Xeon Phi™ プロセッサー上の MCDRAM はフラットモードに設定、メッシュ・インターコネクトは 4 分割 (クワッド) モードで利用
- CentOS* 7.3.1611 (カーネル 3.10.0-514.10.2.el7.x86_64、glibc 2.17-157.el7_3.1.x86_64)

参考資料

- # SPECFEM3D_GLOBE: Komatitsch and Tromp 1999; Komatitsch and Vilotte (1998): **https://geodynamics.org/cig/software/specfem3d_globe/**
- インテル® ソフトウェア・ツールのマニュアル : **https://software.intel.com/en-us/intel-parallel-studio-xe** 日本語版 : **https://www.xlsoft.com/jp/products/intel/tech/documents.html?tab=1&d=#doc-psxe**
- Intel® 64 and IA-32 Architectures Software Developer Manuals: **https://software.intel.com/en-us/articles/intel-sdm**
- Ahmad Yasin. “A Top-Down Method for Performance Analysis and Counters Architecture.” IEEE Xplore: 26 June 2014. Electronic ISBN: 978-1-4799-3606-9.

あなたのクラスターは健全ですか？

インテル® Cluster Checker によるクラスター診断の重要性

Brock A. Taylor インテル コーポレーション HPC ソリューション・アーキテクト

インテル® Cluster Checker (英語) は、ハイパフォーマンス・コンピューティング (HPC) クラスターの問題を素早く特定して、解決するための強力なツールです。システムのわずかな、そして場合によっては単純な問題が、クラスター・パフォーマンスに影響して、アプリケーションのきめ細かなチューニングや並列化の作業を遅延させます。多くの場合、アプリケーションの実行が遅くなったり、停止してしまう、というのがシステムの問題の最初の兆候です。インテル® Cluster Checker は、アプリケーションで発生している問題の原因が実際にクラスターに関連する問題かどうかを素早く判断する、系統的な方法を提供します。

クラスターシステムの詳細を提供するツール

インテル® Cluster Checker は、クラスターに関する最も一般的な手法とシステム診断を 1 つにまとめたツールです。**インテル® Cluster Ready Program** (英語) の一部として 2007 年のリリース以来、クラスターを設計、配備、管理するさまざまな人たちを支援することを目的としてきました。

HPC クラスターの構築と管理は、単一のシステムの管理よりもはるかに複雑です。その性質上、世界で最も強力な **TOP500** (英語) システムは独自に構築され、専用のスタッフにより運用されています。これらの環境の多くは、長年にわたり有機的に成長し、この分野の専門知識を備えたソリューション・アーキテクトによって維持されてきました。

問題は、大規模な HPC データセンターのアプローチが、小さなシステムにスケールダウンできるとは限らないことです。高度な専門知識が必要とされるため、中小規模のビジネスにとってはハードルが高いと言えます。HPC クラスターへの移行を検討している大規模な企業であっても、そのための時間と労力を検討する必要があります。HPC を導入することで大きな費用対効果が得られますが、[**編集者注 :** Hyperion Research 社の調査によると、HPC への投資 $1 あたり $551 の増収、$52 の利益が見込まれます。(出典 : **IDC Economic Models Linking HPC and ROI** (英語) および **Hyperion (IDC) Paints a Bullish Picture of HPC Future** (英語))] その学習曲線は登るのには急すぎる山のようです。インテル® Cluster Checker は、HPC 機能の導入を検討しているユーザーに必要な専門知識を 1 つにまとめたツールを提供することで、導入のハードルを引き下げます。

インテル® Cluster Checker は、臨床システムのように動作します。問題の兆候を探し、見つかった兆候を総合的に検査して診断し、可能な場合は対応策を提案します。データ・プロバイダーは一般的な診断ツールと関数をカプセル化し、ツールはこれらのプロバイダーを使用してクラスターに関する情報を収集します。そして、ルールベースのエキスパート・システムで情報を解析して、問題の兆候を生成します。異なる兆候を組み合わせることで診断が可能となり、多くの場合は問題の対応策を提案できます。この方法により、インテル® Cluster Checker はクラスター機能の専門的な解析をモデル化して、問題を素早く解決できるようにします。

何をチェックするのか ?

インテル® Cluster Checker には、システム障害やパフォーマンス低下を招く一般的な問題の広範なデータ・プロバイダーとルールが含まれます。高レベルで個々のノードの要素と基本機能を調査し、クラスター全体の機能を強化します。以下は、インテル® Cluster Checker がチェックする項目のいくつかの例です。

- ツールを実行中のユーザーの SSH キーが、メッセージ・パッシング・インターフェイス (MPI) 並列アプリケーションの実行向けに適切に設定されているか**チェック**します。
- アドイン・ネットワーク・カードのファームウェア・バージョンがシステムの各ノードで同じであることを**確認**し、クラスター全体のライブラリー・バージョンとソフトウェアの一貫性を調べます。
- プロセッサーのステッピング、メモリー、ハードウェア・コンポーネントの違いを**検出**します。
- ハードウェアおよびソフトウェア・コンポーネントの構成の違いを**明らかに**します。

また、いくつかの一般的なベンチマークを使用して、実際のパフォーマンスと予想されるパフォーマンスの比較を試みます。これらの関数は、システムの配備時にシステムを稼働可能かどうか判断するのに役立ちます。さらに、システム全体の健全性の維持にも役立ちます。現在、クラスターの数百の項目をチェックしていますが、ツールのアップデートがリリースされるたびに、チェック項目のリストは増えています。

クラスターの運用中に、交換部品、ノードの拡張、ソフトウェア / ハードウェアの再構成によってわずかな変更が生じる可能性があります。例えば、新しいネットワーク・カードが以前とは異なる PCIe* スロットに取り付けられるかもしれません。また、クラスターに追加される新しいノードが、ほかのノードとは異なるインテル ® Xeon® プロセッサーを搭載しているかもしれません。さらに、ノードの 1 つで新しい BIOS 設定をアップデートし忘れることがあるかもしれません。

クラスターの運用中に、交換部品、ノードの拡張、ソフトウェア / ハードウェアの再構成によってわずかな変更が生じる可能性があります。

インテル® Cluster Checker は、これらの問題を検出するのに役立ちます。検出される問題は、特定のシステムやアプリケーションでは実際には問題にならないかもしれませんが、調査すべき項目としてハイライトされます。インテル® Cluster Checker を使用することで、クラスターの管理や運用に関する専門知識がないユーザーでもクラスターを容易に運用でき、専門知識があるユーザーはツールセットを補充することができます。

クラスターの健全性に加えて、インテル® Cluster Checker は、クラスターが、**インテル® スケーラブル・システム・フレームワーク (インテル® SSF) リファレンス・アーキテクチャー**で提示されているアプリケーションの互換性を提供しているか確認します。インテル® SSF リファレンス・アーキテクチャーのシステム要件は、最低限のシステム特性を定義するものです。これらの特性には、Linux* ベースのクラスターのシステム・ソフトウェア要素に加えて、システム・ハードウェアの最小要件が含まれます。仕様に準拠するクラスターは、アプリケーション開発者に共通のプラットフォーム・インターフェイスを提供します。この共通レイヤーで開発されたアプリケーションは、インテル® SSF リファレンス・アーキテクチャーに準拠する任意のシステムで実行できます。このアプリケーションとシステムの組み合わせにより、相互運用性が保証され、HPC クラスターの使用が容易になります。

機能の拡張と組込み

クラスターを構成するテクノロジーとコンポーネントは常に進化しており、それに伴って新しい問題が生じる可能性も高くなります。ユーザーが直面する問題の範囲に対応するため、拡張性はインテル® Cluster Checker の重要な機能です。特定のタイプの問題が見つかると、その問題を検出して解決するためのメカニズムをキャプチャーし、通常のチェック項目に追加します。同様の方法でユーザーが独自のデータ・プロバイダーとチェックを作成して含めることもできます。インテル® Cluster Checker 2018 では、データ収集と解析関数をフレームワークにまとめることができます。このフレームワークは、ツールの柔軟な利用を可能にし、機能を拡張するため新しいチェックを簡単に追加する方法を提供します。

アプリケーション開発者は、API を使用してインテル® Cluster Checker の機能をアプリケーションに直接組込み、データ収集と解析を制御できます。これにより、インテル® Cluster Checker をコマンドラインから実行する場合と同様に、さまざまなオプションを利用することができます。アプリケーションで一般的な健全性やインテル® SSF アーキテクチャーへの準拠をチェックしたり、アプリケーション固有の項目をチェックするカスタムルールを追加することができます。アプリケーションの観点から、プログラムでシステムのチェックとデバッグを実行するメカニズムを提供します。アプリケーションは、クラスターの問題を検出して、潜在的な問題をユーザーに通知できます。API の使用例は、ツールのオンライン・ドキュメントを参照してください。

生産性に注目

多種多様な構成、ハードウェアとソフトウェアの組み合わせ、システムの状態、これらはすべて HPC アプリケーションの問題となる可能性があります。インテル® Cluster Checker は、既知のシステム状態の特定を支援し、ユーザーに優れたアプリケーション・エクスペリエンスをもたらします。問題が存在する場合、インテル® Cluster Checker は可能性のある解決策を直ちに提示します。インテル® Cluster Checker は、HPC クラスターの運用に必要な専門知識のハードルを下げ、より多くのユーザーがクラスター・アプリケーションを実行して、大規模で優れた結果を得られるようにします。

インテル® Cluster Checker は、**インテル® Parallel Studio XE** Cluster Edition の一部または**インテル® HPC オーケストレーター**を使用するシステムで利用できます。また、典型的な HPC クラスター向けのインテル® SSF リファレンス・アーキテクチャー準拠のソリューションに含まれている可能性もあります。

インテル® Cluster Checker の詳細 (英語) >

HPC クラスターの最適化

HPC クラスターでのオンデマンドの BIOS 設定変更

Michael Hebenstreit インテル コーポレーション データセンター・エンジニア

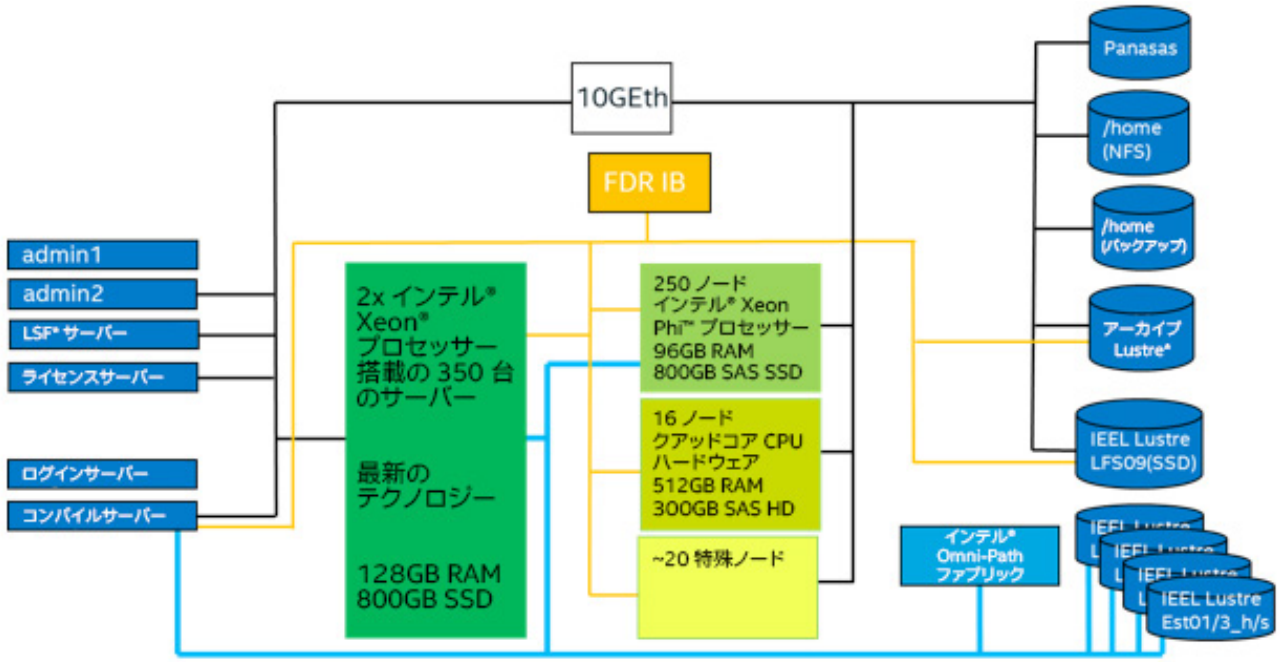
2000 年頃から、ほとんどのハイパフォーマンス・コンピューティング (HPC) システムは、汎用 x86 ハードウェア・ベースのクラスターとしてセットアップされています。これらのクラスターは、1 ソケットまたは 2 ソケットのサーバー群で構成され、実際の計算処理の実行に加えて、ストレージシステムと管理ノードとしての役割を果たします。

最近の**インテル® Xeon® プロセッサー**・ベースのシステムと Linux* カーネルは、特定のアプリケーション向けにハードウェアとオペレーティング・システム (OS) の両方を最適化するさまざまな方法を提供しています。特定のワークフローのみに使用されるクラスターは簡単に最適化できますが、用途が複雑な場合、ほとんどのクラスター・マネージャーは対応できません。

ジョブ単位で複雑な最適化を行う方法があります。インテルでは、約 500 の計算ノードで構成されるベンチマーク・データセンターでこのアプローチをテストしてみました。この Endeavor クラスターは最新のハードウェアで

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

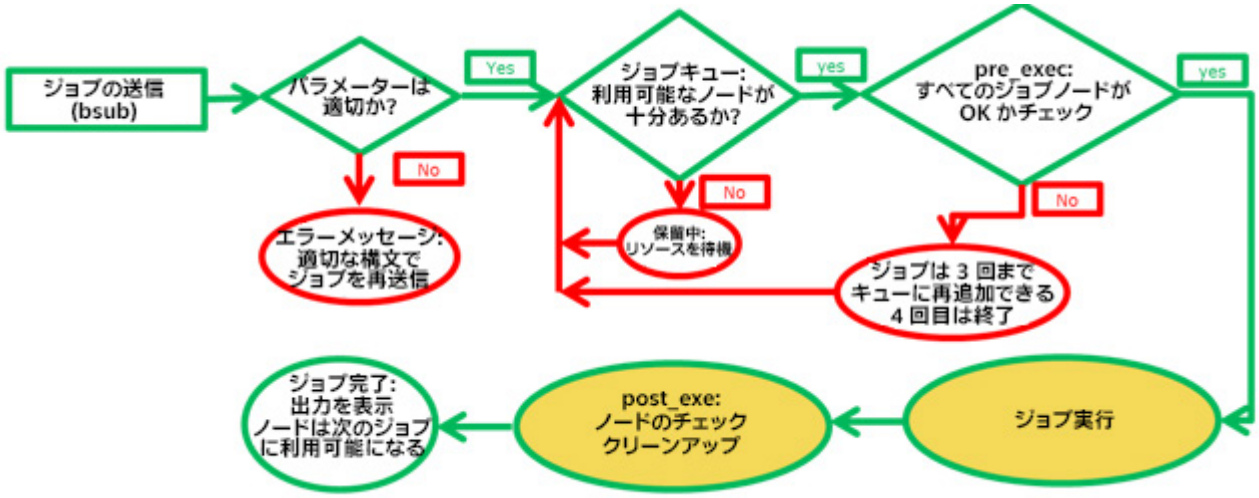
定期的に再構成されており、2006 年以降 **TOP500 SuperComputer Sites** (英語) に常にランクインしています。**図 1** は、Endeavor クラスターのレイアウトです。



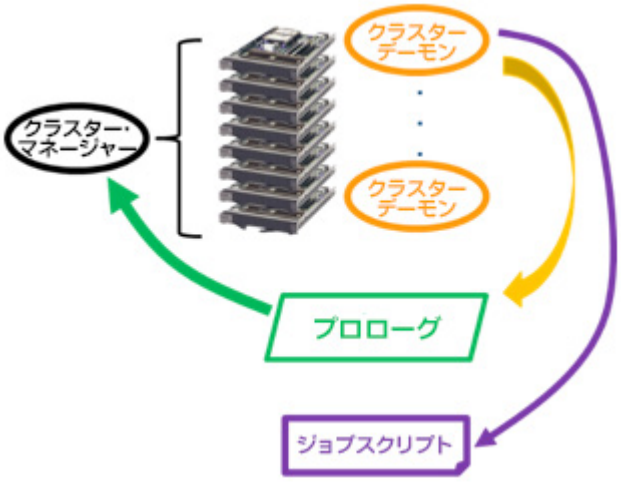
1 インテルの Endeavor クラスターのレイアウト

ユーザーは通常、ログインノードの 1 つに接続して、ワークロードをジョブスクリプトとしてパッケージ化し、クラスター・マネージャーにジョブを送信します。クラスター・マネージャー (例: Altair* PBS Professional*、Bright Cluster Manager*、IBM* LSF*、Slurm*) は、クラスターリソースを効率良く割り当てるためのスケジューリング・ツールです。クラスター・マネージャーにとって、各ジョブは X 個の計算ノードを Y 期間使用するという要求に過ぎません。

ユーザーがジョブを送信すると、システムは必要なすべてのパラメーターが適切な範囲内であるかチェックし、必要な種類の十分なリソースが解放されるまで待機します (**図 2**)。十分なノードが割り当て可能になると、クラスター・マネージャーはプロローグと呼ばれる特別なプログラムを実行します (**図 3**)。このプログラムは通常、ジョブに割り当てられた最初のノード (ヘッドノード) で実行されます。プロローグにはさまざまな用途がありますが、例えば、ジョブに割り当てられたすべてのノードが健全な状態にあることを確認できます。プロローグが正常に完了すると、クラスター・マネージャーはノードで実際のジョブを開始します。ほとんどのケースでは、これはヘッドノードで実行されるシェルスクリプトです。このスクリプトが終了すると、クラスター・マネージャーはノードをクリーンアップして、エピローグプログラムを実行し、次のジョブに向けてノードを準備します。



2 インテルの Endeavor クラスターにおける LSF* ジョブのフローチャート



3 プロローグプログラム

ほとんどのクラスター・マネージャーで実装されるこのワークフローは、プロローグプロセスの一環としてノードを再起動しなければならない場合、問題に直面します。以下の理由から、再起動が必要になる可能性があります。

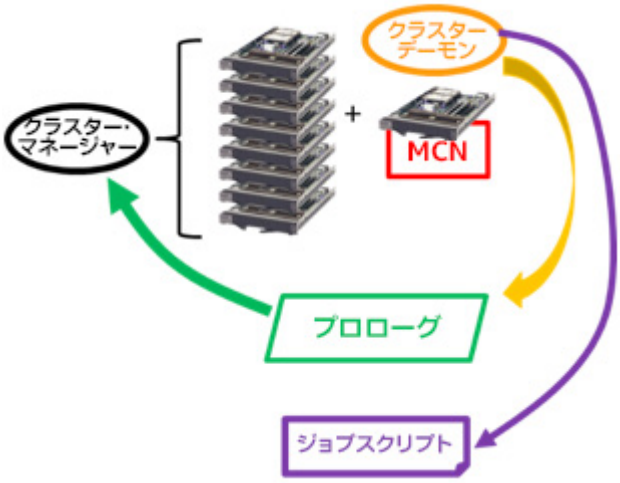
- **インテル® Xeon Phi™ プロセッサー**は、標準メモリーブロックまたはタイルとメモリー間の中間キャッシュとして利用可能な MCDRAM メモリーを内蔵しています。キャッシュとして利用するとプログラムは自動的にスピードアップしますが、標準メモリーとして利用することでさらにスピードアップする可能性があります。これらのモードを切り替えるには、BIOS 設定を変更してシステムを再起動する必要があります。

- **インテル® Xeon® プロセッサー**は、コア、キャッシュ、メモリー、PCIe* コントローラー間の通信にメッシュを使用します。BIOS 設定の Sub-NUMA Clustering を使用すると、ユーザーは CPU を再構成して仮想ソケットに分割できます。
- 最近の Linux* カーネルは、NO_HZ コアの概念を認識しています。通常、カーネルの処理やタスクスイッチのため、100ms ~ 1,000ms ごとにコアに割り込みが発生します。典型的な HPC ワークロードでは、この動作により生産性が低下します。NO_HZ パラメーターを使用して、カーネルが 1 秒あたり 1 回のみ割り込みをスケジュールするように設定できます。そうすることで、OS のノイズが軽減され、スケーラビリティーが向上します。パフォーマンスも向上しますが、そのためにはカーネルの変更とシステムの再起動が必要です。

プロローグスクリプトの標準的なアプローチでは、再起動の必要性はジレンマにつながります。ヘッドノードが再起動すると、クラスター・マネージャーはプロローグが終了したと仮定して、ジョブの前処理を停止し、再スケジュールします。ジョブにノードの新しいセットが割り当てられ、新しいヘッドノードが選択され、プロローグが実行されますが、同じ結果になります。

解決策を見つける

インテルの HPC ベンチマーク・クラスター Endeavor は、現在 IBM* LSF* を導入しています。このクラスターにおける単純かつ移植可能な唯一の解決策は、各ジョブにマスター・コントロール・ノード (MCN) を追加することでした (**図 4**)。MCN は、自動的にジョブのヘッドノードになります。プロローグを実行して、syscfg で BIOS 設定を変更し、IPMI (Intelligent Platform Management Interface) を使用して計算ノードを再起動します。そして、ノードが再起動を完了したら、すべてが適切であることをチェックして、プロローグを終了します。プロローグが正常に完了すると、ジョブが開始します。



4 マスター・コントロール・ノード

ユーザー側では、わずかな変更が必要になります。通常、ユーザーは、ヘッドノードを含め同じタイプのノードを要求します。しかし、再設定後のジョブでは、ヘッドノードが異なるタイプになります。ユーザーは、次のいずれかを選択することができます。

1. **ノードリストの最初のノードにログインして**、通常通りに処理します。
2. **最近の MPI バージョンの機能を使用して**、ジョブの処理に使用されるノードリストからヘッドノードを除外します。

後者には、いくつかの利点があります。

- 通常、MPI プログラムのヘッドノードは追加のプロセスを起動する必要があります。例えば、プログラムの MPI プロセスごとに ssh プロセスを起動します。この追加のロードは MCN に残るため、すべての計算ノードでシステムロードが同じになります。
- インテル® Xeon Phi™ プロセッサー・ベースのクラスターでは、MCN が標準のインテル® Xeon® プロセッサー・ベースのサーバーの場合、起動スクリプトと MPI の初期化をインテル® Xeon Phi™ プロセッサー・ベースのシステムよりも高速に処理できます。

実装

Endeavor では、プロローグとエピローグでノードの健全性のチェックに加えて、ユーザーにルート権限が必要なパラメーターの変更も許可しています。プロローグとエピローグのほとんどのコードは同一であるため、ここでは同じスクリプトを使用して、必要に応じてコードパスを切り替えています。

準備

ジョブの送信

標準の LSF* 構文を使用するため、**bsub** 用の小さなラッパースクリプトを作成し、**-l** オプションでコマンドを拡張します。LSF* は、ユーザーの環境をジョブだけでなく、プロローグとエピローグにも転送するため、すべての特別な要求は一意の環境変数に変換されます。

必要に応じて、ラッパーは自動的にリソース要件を拡張してコントロール・ノードを含めます。

```
$ bsub -R '2*{select[ekf]span[ptile=1]}' -l KNL_MEMMODE=1 run.sh
Warning '-l KNL_MEMMODE=1' will reboot compute nodes
Resource_List_KNL_MEMMODE=1
bsub.orig -R '1*{select[rebootctrl]} + 2*{select[ekf] span[ptile=1]}'  run.sh
…
```

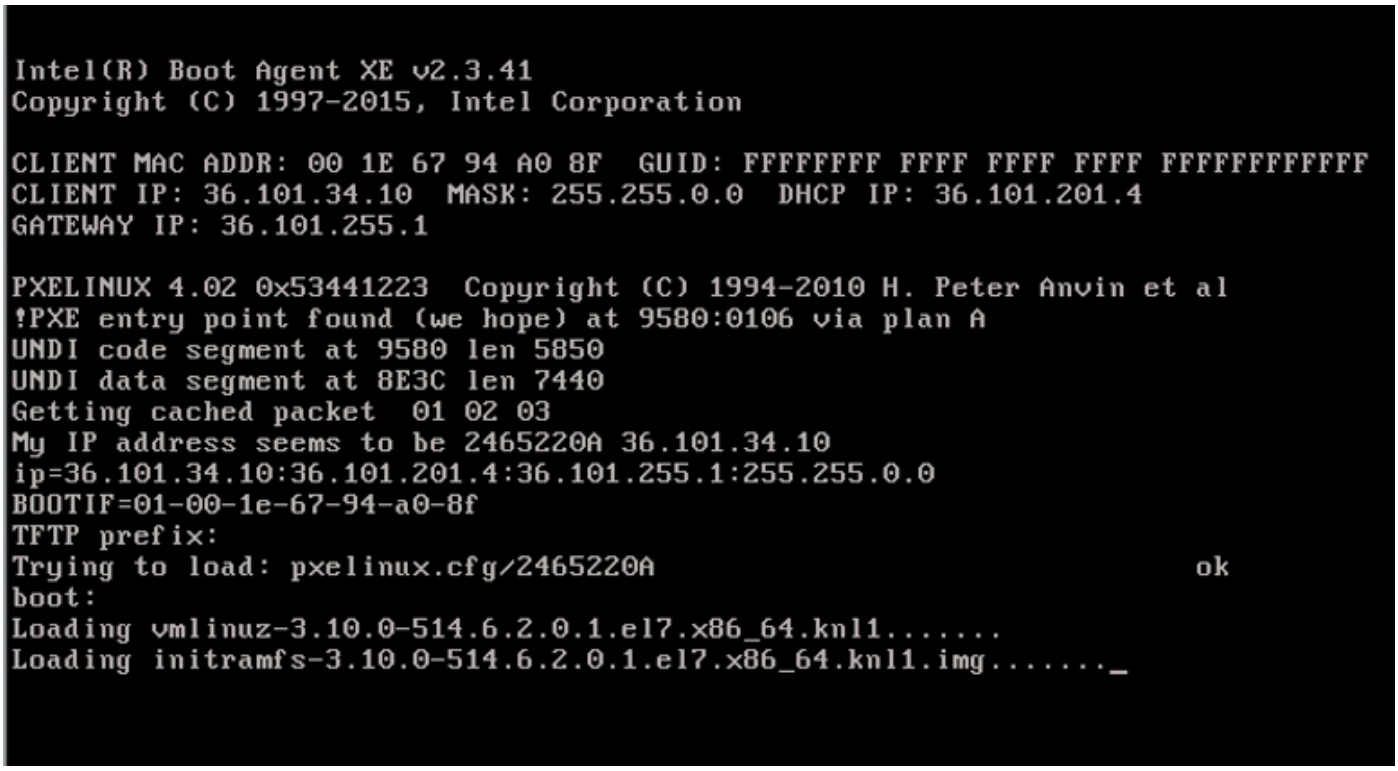
ユーザーがタイプ ekf の 2 つのノードを要求し、スクリプトによって対応する環境変数が設定され、コントロール・ノードを含めるように選択文字列が拡張されます。その後、オリジナルの LSF* バイナリー (bsub.orig に名前を変更) を呼び出します。

リモートからのノードの起動

ネットワーク経由でノードを再起動するには、IPMI を使用します。

PXE (Preboot eXecution Environment) を利用したクラスターノードの起動

PXE は、すでにほとんどのクラスターで使用されています。再設定では、pxelinux.0 バイナリーが tftp サーバーに起動設定ファイルを問い合わせる方法を使用しました (**図 5**)。



5 PXE (Preboot eXecution Environment)

最初のクエリーは、MAC アドレス (01-00-1e-67-94-a0-8f) と同じ名前のファイルを問い合わせます。2 つ目のクエリーは、DHCP サーバーにより割り当てられた 16 進数表記の IP アドレス (2465220A) と同じ名前のファイルを問い合わせます。ここでは、2 つ目のクエリーをデフォルトの起動に使用します。そうすることで、1 つ目のタイプの適切なファイルを一時的に作成して、特定のジョブの起動設定をオーバーライドします。

この方法は、体系的な PXE 設定に依存します。各ノードについて、リンクノード名は 16 進数表記の IP アドレスを指します。この 2 つ目のファイルは、デフォルトの設定へのリンクです。

```
emhtest329 -> 2465220A
2465220A  -> ol7u3_sda6
```

ユーザーには、事前に用意した特定の組み合わせ special1、special2、...、のみ許可します。PXE の起動ディレクトリーには、デフォルトの設定の次の設定が含まれています。

```
ol7u3_sda6
```

例えば、次のようなファイルです。

```
ol7u3_sda6-k229sp0
ol7u3_sda6-k514sp1
ol7u3_sda6-k514sp2
...
```

それぞれ、起動設定を表します。このケースでは、**k229** と **k514** は異なるカーネルバージョン (**sp1**、**2**、**3**…) を示し、それぞれ特別なカーネルオプション (例: **NOHZ_full**) を使用します。ユーザーが特定の設定を要求した場合、このノードに対応するファイルが存在していなければなりません。ここでは、ファイル **ol7u3_sda6-k514sp2** が存在するため、ノード **emhtest329** を設定 **k515sp2** で再起動できます。しかし、**k514x5** を要求すると失敗します。

BIOS 設定の変更

インテルは、インテル製マザーボード向けに、Linux* から BIOS パラメーターの読み取りと変更を許可する syscfg ユーティリティーを提供しています。すべての OEM が同様のツールを提供しているわけではありません。

クラスター・マネージャーへの統合

LSF* への統合は簡単です。プロローグは、LSF* によってマスター・コントロール・ノードで自動的に実行されます。プロローグでは、チェックやセットアップを行う前に、ジョブのすべての計算ノード (コントロール・ノードは含まない) で再設定スクリプトを実行する必要があります。ノードで再起動が必要な場合、プロローグは IPMI を使用してそのノードをリセットできます。そして、再起動の完了を待機します。最大待機時間は、起動に失敗したノードによってこのスキームが損なわれない値にします。エピローグは、同様のジョブフローでノードをデフォルトの設定に戻して、必要に応じてノードを再起動します。

再設定スクリプトの構造

再設定スクリプトは、プロローグによって各ノードで実行され、いくつかの環境変数の影響を受けます。

```
Prologue                                    # プロローグが実行された場合は 1

# bsub からのユーザー要件
Resource_List_KNL_MEMMODE
Resource_List_KNL_CLUSTERMODE
Resource_List_Sub_NUMA_Cluster
Resource_List_SPECIAL_KERNEL
```

初期化では、重要な syscfg バイナリーの場所とファイルの格納場所を指定します。ノードを再起動する必要があるかどうかは、REBOOT 変数で管理します。

```
SYSCFG=/usr/local/bin/syscfg
SAFEDIR=/var/lib/icsmoke3/safe
CURRENTDIR=/var/lib/icsmoke3/current
PXEDIR=/admin/tftpboot/3.0/pxelinux/pxelinux.cfg
REBOOT=no
HOSTNAME=`hostname`
```

ヘルパー関数は、syscfg コマンドの出力を直接入力として利用できない場合に役立ちます。**sed** コマンドは、**syscfg.INI** の次の形式の行を

```
Cluster Mode=Quadrant;Options: All2All=00: SNC-2=01: SNC-4=02: Hemisphere=03:
Quadrant=04: Auto=05
```

関連する数値に変換します。変数 **$I** が適切に設定されていなければなりません。

```
convert_syscfg()
{
 echo “$1” | sed -e “s,${I}=Cache.*,0,” -e “s,${I}=Flat.*,1,” -e
“s,${I}=All2All.*,0,” -e “s,${I}=SNC-2.*,1,” -e “s,${I}=SNC-
4.*,2,” -e “s,${I}=Hemisphere.*,3,” -e “s,${I}=Quadrant.*,4,” -e
“s,${I}=Disabled.*,0,” -e “s,${I}=Enabled.*,1,”
}
```

現在の BIOS 設定をダンプして、現在の設定をすべて取得します。

```
cd $CURRENTDIR
/bin/rm syscfg.INI
$SYSCFG /s INI
```

ジョブが完了したら、すべてのノードでエピローグを実行します。

```
if [ “$prologue” != “1” ]
then
```

最初に、特別な PXE 設定がすでに存在するかどうかチェックします。存在する場合は削除します。ファイルを削除できない場合、スクリプトはエラーになります。

```
 # 特別なカーネルの起動に使用される PXE リンク
 ADDR=”01-`sed -e ‘s,:,-,g’ /sys/class/net/eth0/address`”
 ADDRFILE=”/admin/tftpboot/3.0/pxelinux/pxelinux.cfg/$ADDR”
 if [ -e “$ADDRFILE” ]
 then
  /bin/rm $ADDRFILE
  sleep 1    # クラスター・ファイル・システムの処理が終わるまで待機
  if [ -e “$ADDRFILE” ]
  then
   badmin hclose -C “wrong bootimage, fix $ADDRFILE” $HOSTNAME
   REBOOT=error
   exit 1
  fi
```

BIOS 設定の現在の値と **$SAFEDIR/syscfg.INI** にある想定される値を比較して、異なる場合は修正します。異なる値がある場合は、変数を **$REBOOT=yes** に設定します。

```
for I in “Memory Mode” “Cluster Mode” “Sub_NUMA Cluster” “IMC Interleaving”
do
 CURRENT=`egrep “$I” $ CURRENTDIR/syscfg.INI`
 SAFE=`egrep “$I” $ SAFEDIR/syscfg.INI`
 if [ “$CURRENT” != “$SAFE” ]
 then
  VAL=`convert_syscfg “$SAFE”`
  $SYSCFG /bcs “” “$I”  “$VAL”
  REBOOT=yes
 fi
done
```

カーネルコマンド行には、ノードがデフォルトのカーネルで実行しているか、または再起動が必要なものがあるかを示すヒントが含まれています。

```
grep -q “CRTBOOT=default” /proc/cmdline || REBOOT=yes
```

スクリプトの残りの部分は、プロローグでのみ処理されます。

```
else
```

各 BIOS 設定について、ユーザーによって提供された値が許容値かどうかをチェックします。そして、現在の設定とユーザーが要求した設定を比較します。設定の変更が必要な場合は、syscfg で新しい値を設定し、**$REBOOT** 変数を “**yes**” (再起動が必要) に設定します。以下に例を示します。

```
  case “$Resource_List_KNL_MEMMODE”   in 0|1)
   I=”Memory Mode”
   CURRENT=`egrep “$I” $CURRENTDIR/syscfg.INI`
   SAFE=`egrep “$I” $SAFEDIR/syscfg.INI`
   VAL=`convert_syscfg “$CURRENT”`
   test -n “$SAFE” && test “$VAL” != “$Resource_List_KNL_MEMMODE”
 && { $SYSCFG /bcs “” “$I”     “$Resource_List_KNL_MEMMODE” ;
REBOOT=yes; }
  esac
```

ロギングのため、すべての設定の現在の値を表示します。

```
        # 現在の設定を表示
        $SYSCFG /d BIOSSETTINGS “Memory Mode”
        $SYSCFG /d BIOSSETTINGS “Cluster Mode”
        $SYSCFG /d BIOSSETTINGS “Sub_NUMA Cluster”
```

異なるカーネルを設定するには、最初に **$PXEDIR** でこのノードの標準の **PXE-config** ファイルの場所を特定します。そして、**$Resource_List_SPECIAL_KERNEL** でその名前を拡張します。要求された設定ファイルが存在する場合、**01-MACADDRESS** という名前のリンクを作成します。このリンクは、次回起動時に優先されます。**$REBOOT** 変数を “**yes**” に設定します。

```
  if [ -n “$Resource_List_SPECIAL_KERNEL” ]
  then
   echo “configuring for Kernel $Resource_List_SPECIAL_KERNEL”
   if ! `grep -q “CRTBOOT=${Resource_List_SPECIAL_KERNEL}” /proc/cmdline`
   then
    DEFAULT=`readlink -f $PXEDIR/$HOSTNAME`
    DIR=`dirname $DEFAULT`
    BASE=`basename $DEFAULT`
    if [ -e “$DIR/${BASE}-${Resource_List_SPECIAL_KERNEL}” ]
    then
     cd $DIR
     ln -s ${BASE}-${Resource_List_SPECIAL_KERNEL} “$ADDR”
     echo “created $DIR/$ADDR”
     ls -l “$DIR/$ADDR”
     REBOOT=yes
    else
     echo “can not set kernel to $DIR/${BASE}-${Resource_List_SPECIAL_KERNEL}”
    fi
   fi
  fi
fi
```

スクリプトが終了したら、“**REBOOT=yes**” または “**REBOOT=no**” を出力します。

```
echo “REBOOT=$REBOOT”
```

コントロール・ノードで実行中のプロローグスクリプトは、出力を解析し、その結果に応じて、IPMI を介して再起動シーケンスを発行します。

クラスターの最適化

最近のインテル® Xeon® プロセッサー・ベースのシステムと Linux* カーネルは、特定のアプリケーション向けにハードウェアとオペレーティング・システム (OS) を最適化するさまざまな方法を提供しています。この記事では、ジョブ単位で複雑な最適化を行う方法を紹介しました。複雑さが増し、ジョブの起動時間が長くなりますが、インテルの HPC ベンチマーク・クラスター Endeavor では、パフォーマンスの大幅な向上に、この方法が非常に重要な役割を果たしました。ほかのクラスターではさらなる向上が得られるかもしれません。

BLOG HIGHLIGHTS

最新のインテルの HEVC エンコーダー / デコーダーでメディア・アプリケーションの品質とパフォーマンスを向上

TERRY DEEM INTEL CORPORATION

インテル® Media Server Studio Professional Edition (2017 R3) の新しい HEVC テクノロジーにより、メディアおよびビデオ・アプリケーション開発者は、さらに高画質で高速なパフォーマンスが得られるようにチューニングできます。この新しいエディションでは、主要な解析ツールが強化されており、アプリケーションのパフォーマンス特定に関するより正確で詳細な解析データが提供されるため、開発者は最適化にかかる時間を節約できます。インテル® Media Server Professional Edition には、使いやすいビデオ・エンコーディング/デコーディング API、画質とパフォーマンスの解析ツールが含まれており、メディア・アプリケーションを高画質かつ高いフレームレートで配信できるように支援します。

この記事の続きはこちら (英語) でご覧になれます。>

Software

THE PARALLEL UNIVERSE

JPN/1712/PDF/XL/SSG/TT