インテル® VTune™ プロファイラー パフォーマンス解析クックブック

2021 年 6 月 30 日

インテル® VTune™ プロファイラー は、開発者がコードを解析し、非効率なアルゴリズムおよびハードウェアの利用状況を特定して、適切なパフォーマンス・チューニングのアドバイスを得られるように支援する、パフォーマンス・プロファイル・ツールです。

注

- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
- ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。

このクックブックは、インテル® VTune™ プロファイラーで提供される解析タイプを使用して、パフォーマンスの手法とレシピを紹介します。

- 手法
  最初に、チューニング手法、パフォーマンス・メトリック、統計を収集するハードウェア・ソリューションを理解して、パフォーマンス解析を開始します。その後、インテル® VTune™ プロファイラーと従来のインテル® VTune™ Amplifier で利用可能な特定のチューニングや設定レシピにドリルダウンできます。

- 設定レシピ
  特定のコード環境でパフォーマンス解析を行うため、システムとインテル® VTune™ プロファイラーと従来のインテル® VTune™ Amplifier を設定する方法を詳しく説明します。

- チューニング・レシピ
  インテル® VTune™ プロファイラーと従来のインテル® VTune™ Amplifier で検出可能な最も一般的なパフォーマンスの問題を調査し、パフォーマンスを最適化するためのステップを提供します。

- 法務上の注意書き

手法

最初に、チューニング手法、パフォーマンス・メトリック、統計を収集するハードウェア・ソリューションを理解して、パフォーマンス解析を開始します。その後、インテル® VTune™ プロファイラーと従来のインテル® VTune™ Amplifier で利用可能な特定のチューニングや設定レシピにドリルダウンできます。

- トップダウン・マイクロアーキテクチャー解析法
  アプリケーションが利用可能なハードウェア・リソースをどのように使用しているかを把握し、CPU マイクロアーキテクチャーの利点を活用できるようにします。この情報を得る 1 つの手段として、オンチップのパフォーマンス・モニタリング・ユニット (PMU) を使用する方法があります。

- OpenMP* コード解析
  OpenMP* または OpenMP* - MPI ハイブリッド・アプリケーションの CPU 利用率を解析して、潜在的な非効率性の原因を特定します。

- インテル® GPU 向けのソフトウェア最適化
  インテル® VTune™ プロファイラーを使用して、インテル® GPU へオフロードする場合のオーバーヘッドを予測します。GPU にオフロードされた計算タスクのパフォーマンスを解析します。

- DPDK アプリケーションのコア使用率
  このレシピは、DPDK ベースのアプリケーションにおけるパケット受信のコア使用率を特徴付けるメトリックを調査します。

- DPDK アプリケーションの PCIe* トラフィック
  インテル® VTune™ Amplifier の PCIe* 帯域幅メトリックを使用して、パケット・フォワーディングを行う DPDK ベースのアプリケーションの PCIe* トラフィックを調査します。

- DPDK イベント・デバイス・プロファイル
  DPDK ベースのアプリケーションの DPDK イベント・デバイス・パイプラインの利用効率を解析して、不均衡な負荷分散やワーカーコアが十分に利用されていない問題を特定します。

- インテル® データ・ダイレクト I/O テクノロジーの効果的な利用
  インテル® VTune™ プロファイラーを使用して、インテル® Xeon® プロセッサーのハードウェア機能であるインテル® データ・ダイレクト I/O テクノロジー (インテル® DDIO) の利用効率を明らかにする方法を示します。

- 最新の命令セットを使用して最適化されたポータブルなバイナリーをコンパイルする
  移植性を維持しながら最新の命令セットを使用してバイナリーをコンパイルするさまざまな方法を学びます。

トップダウン・マイクロアーキテクチャー解析法

アプリケーションが利用可能なハードウェア・リソースをどのように使用しているかを把握し、CPU マイクロアーキテクチャーの利点を活用できるようにします。この情報を得る 1 つの手段として、オンチップのパフォーマンス・モニタリング・ユニット (PMU) を使用する方法があります。

コンテンツ・エキスパート: Jackson Marusarz (英語)、Dmitry Ryabtsev (英語)

PMU は、システム上で発生した特定のハードウェア・イベントをカウントする CPU コア内部の専用ロジックです。これらのイベントには、キャッシュミスや分岐予測ミスなどがあります。これらのイベントを組み合わせることで、命令あたりのサイクル数 (CPI) などの高レベルのメトリックを構成して監視できます。

特定のマイクロアーキテクチャーでは、PMU を介して数百ものイベントを利用できます。特定のパフォーマンスの問題を検出して解決する際に、どのイベントが有用であるかを判断することは容易ではありません。生のイベントデータから有用な情報を得るには、マイクロアーキテクチャーの設計と PMU 仕様の両方に関する深い知識が求められます。しかし、事前定義されたイベントとメトリックを使用することにより、トップダウン特性化方法論の利点を活用してデータを実用的な情報に変換することができます。

PMU 解析レシピでは、その手法とインテル® VTune™ プロファイラーでの使用方法を説明します。

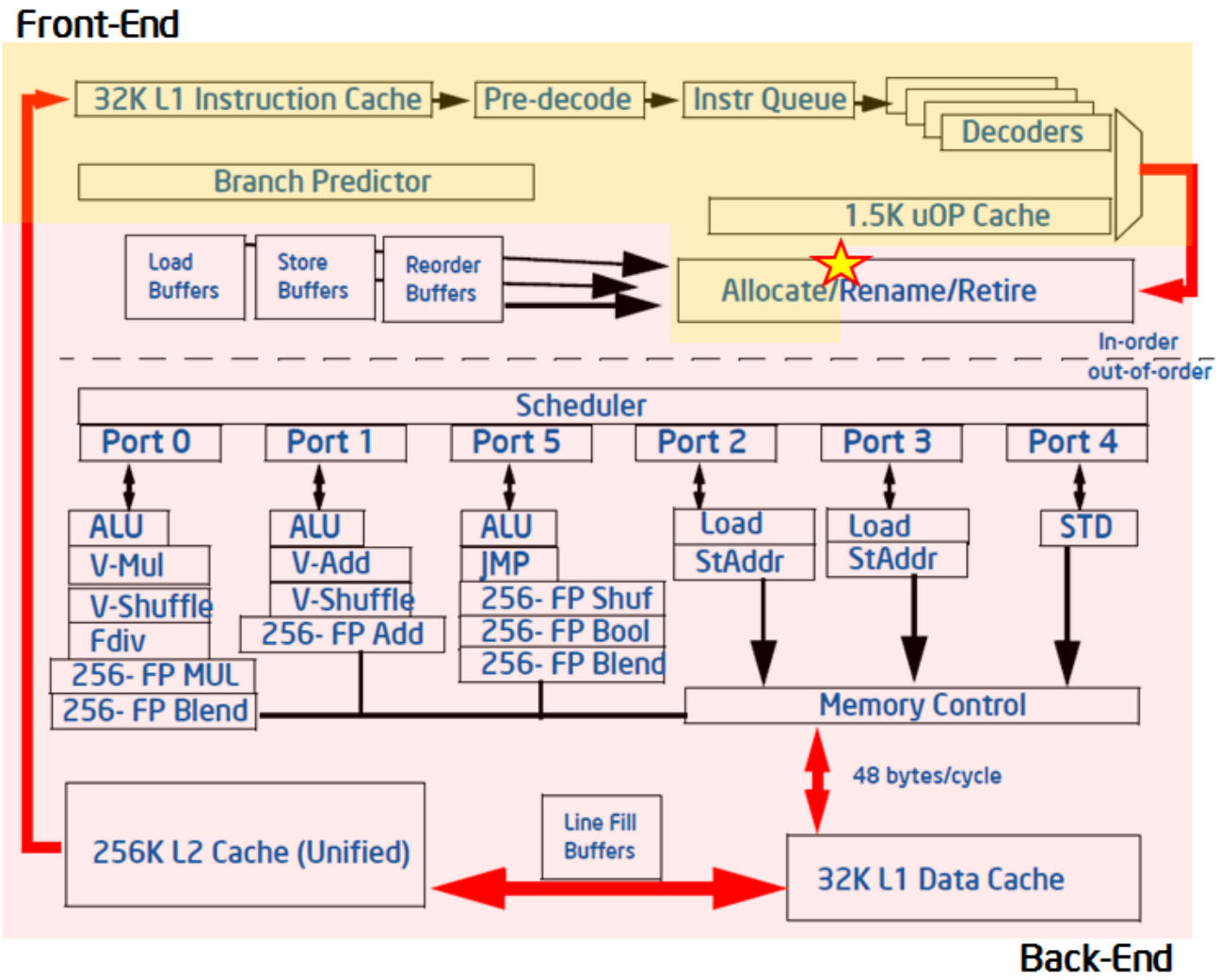
- 使用するもの:
  - トップダウン・マイクロアーキテクチャー解析法 (TMAM) の概要
  - インテル® VTune™ プロファイラーによるトップダウン解析法
  - マイクロアーキテクチャー・チューニングの方法論
- 手順:
  - バックエンド依存カテゴリーのチューニング
  - フロントエンド依存カテゴリーのチューニング
  - 投機の問題カテゴリーのチューニング
  - リタイアカテゴリーのチューニング
- 関連クックブック・レシピ

トップダウン・マイクロアーキテクチャー解析法の概要

現代の CPU は、リソースを可能な限り有効に活用するため、ハードウェアによるスレッド化、アウトオブオーダー実行、命令レベルの並列性などの技術とともにパイプラインを採用しています。しかし、ソフトウェア・パターンとアルゴリズムの中には、依然として非効率なものがあります。例えば、リンクデータ構造はソフトウェアで良く利用されますが、これはハードウェア・プリフェッチャーの有効性を損ねる間接アドレスの原因となります。多くの場合、このような振る舞いはパイプラインに無益な隙間 (パイプライン・バブル) を作り、データが取得されるまで実行するほかの命令がない状態となります。リンクデータ構造は、ソフトウェアの問題に対しては適切なソリューションですが、非効率な結果となるでしょう。このほかにも、CPU パイプラインに関連して重要な意味を持つ多くのソフトウェア・レベルの例があります。トップダウン・マイクロアーキテクチャー解析法は、トップダウン特性化方法論をベースとして、アルゴリズムとデータ構造が適切な選択を行っているかどうか、詳しい情報を提供します。トップダウン・マイクロアーキテクチャー解析法の詳細は、『インテル® 64 および IA-32 アーキテクチャー最適化リファレンス・マニュアル、付録 B.1』を参照してください。

トップダウン特性化は、アプリケーションのパフォーマンス・ボトルネックを特定するイベントベースのメトリックを階層的に構成します。これは、CPU がアプリケーションを実行する際のパイプラインの利用率の平均を示します。以前のフレームワークは、CPU のクロックティックをカウントする方法を使用して、どの要素の CPU のクロックティックが、どの操作 (L2 キャッシュミスによるデータ取得など) に費やされたか判別して、イベントを解釈していました。このフレームワークは、その代わりに、パイプラインのリソースをカウントする方法を使用します。トップダウン特性化を理解するため、高レベルのマイクロアーキテクチャーの概念を調査します。マイクロアーキテクチャーの詳細の多くはこのフレームワークで抽象化されており、ハードウェアのエキスパートでなくても理解して、使用することができます。

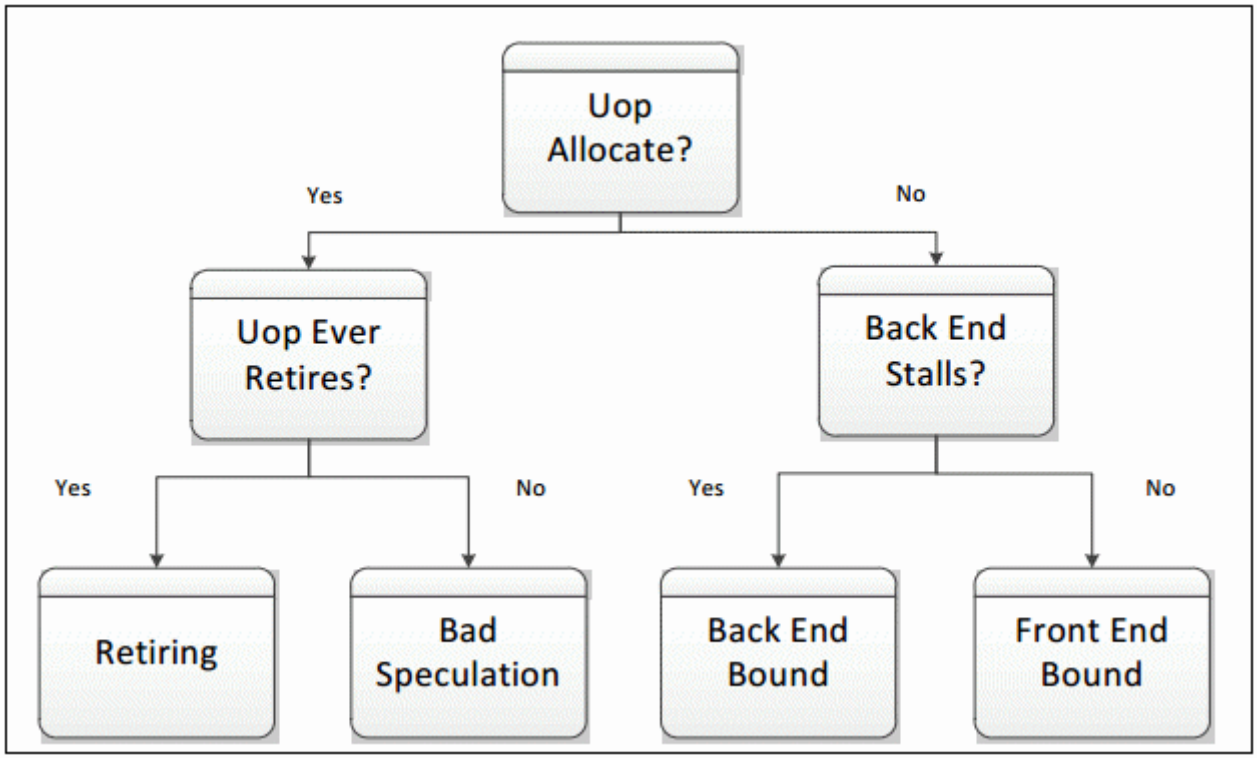
現代のハイパフォーマンス CPU のパイプラインは、非常に複雑です。以下の図に示すように、パイプラインは概念的にフロントエンドとバックエンドの 2 つに分割できます。フロントエンドは、アーキテクチャーの命令を表すプログラムコードをフェッチして、それらを μop (マイクロオペレーション) と呼ばれる 1 つ以上の低レベルのハードウェア操作にデコードします。μop は、パイプラインの「アロケーション」と呼ばれる過程でバックエンドへ送られます。アロケーションが行われると、バックエンドは μop のデータオペランドが利用可能になるのを監視し、利用可能な実行ユニットで μop を実行する役割を担います。μop の実行完了は「リタイア」と呼ばれ、μop の実行結果はアーキテクチャー状態にコミットされます (CPU レジスターやメモリーへの書き戻し)。通常、ほとんどの μop はパイプラインを通過してリタイアしますが、投機的にフェッチされた μop はリタイアする前に取り消される場合があります (分岐予測ミスのようなケース)。



最近のインテル® マイクロアーキテクチャーでは、パイプラインのフロントエンドはサイクルごとに 4 つの µop をアロケートでき、バックエンドはサイクルごとに 4 つの µop をリタイアできます。これらの機能を考慮して、パイプライン・スロットの抽象化の概念を定義します。パイプライン・スロットは、1 つのマイクロオペレーションを操作するために必要なハードウェアのリソースを表します。トップダウン特性化では、それぞれの CPU コアには、各クロックサイクルで利用可能な 4 つのパイプライン・スロットがあると仮定します。そして、特殊な PMU イベントを使用して、これらのパイプライン・スロットがどの程度効率良く利用されているか測定します。パイプライン・スロットの状態は、アロケーションの時点 (上記の図で★印が記された場所) で取得され、µop はフロントエンドからバックエンドへ移ります。アプリケーションの実行中に利用可能な各パイプライン・スロットは、上記のパイプラインの図に基づいて 4 つのカテゴリーのいずれかに分類されます。

すべてのサイクルで、パイプライン・スロットは空であるか、µop で埋められているかのどちらかです。スロットが 1 クロックサイクルで空の場合、ストールに分類されます。このパイプラインを分類するために必要な次のステップは、パイプラインのフロントエンドまたはバックエンドのどちらがストールの原因であるか判断することです。この処理は、指定された PMU イベントと式を使用して行われます。トップダウン特性化の目的は重要なボトルネックを特定することですが、フロントエンドもしくはバックエンドのどちらかがストールの原因であるかは重要な考慮点です。一般に、ストールの原因がフロントエンドによる µop 供給の能力不足であれば、このサイクルはフロントエンド依存 (Front End Bound) スロットとして分類され、パフォーマンスはフロントエンド依存カテゴリーのいくつかのボトルネックで制限されます。バックエンドで µop を処理する準備ができていないためにフロントエンドが µop を渡せない場合、空のパイプライン・スロットはバックエンド依存 (Back End Bound) として分類されます。バックエンド・ストールは、一般にバックエンドがいずれかのリソース (ロードバッファーなど) を使い果たしていることにより生じます。そして、フロントエンドとバックエンドの両方がストールしている場合、そのスロットはバックエンド依存として分類されます。これは、そのような場合にフロントエンドのストールを解決しても、アプリケーションのパフォーマンス改善に直結することが少ないためです。バックエンドがボトルネックであれば、フロントエンドの問題を解決する前にボトルネックを排除する必要があります。

プロセッサーがストールしていない場合、パイプライン・スロットはアロケーションの時点で µop で満たされます。この場合、スロットをどのように分類するか決定する要因は、µop が最終的にリタイアするかどうかです。リタイアすれば、そのスロットはリタイアに分類されます。そうでなければ、フロントエンドによる不適切な分岐予測、もしくは自己修正コードによるパイプライン・フラッシュなどのクリアイベントによって、そのスロットは投機の問題に分類されます。これら 4 つのカテゴリーが、トップダウン特性化のトップレベルを形成します。アプリケーションを特性化するため、それぞれのパイプライン・スロットは 4 つのカテゴリーの 1 つに分類されます。



これら 4 つのカテゴリーにおけるパイプライン・スロットの分布は非常に有用です。イベントベースのメトリックは長年使用されてきましたが、この特性化以前は最もパフォーマンスに影響を与える問題を識別する手法はありませんでした。パフォーマンス・メトリックをこのフレームワークに当てはめると、開発者は最初に取り組むべき問題を知ることができます。4 つのカテゴリーにパイプライン・スロットを分類するイベントは、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge (第 2 世代インテル® Core™ プロセッサー・ファミリーとインテル® Xeon® プロセッサー E5 ファミリー) から利用できます。以降のマイクロアーキテクチャーでは、これらのハイレベルのカテゴリーを、より詳細なパフォーマンス・メトリックに分割できます。

インテル® VTune™ プロファイラーによるトップダウン解析法

インテル® VTune™ プロファイラーは、インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge 以降のトップダウン特性化で定義されたイベント収集のための事前定義を含む [Microarchitecture Exploration (マイクロアーキテクチャー全般)] 解析タイプ (英語) を提供します。マイクロアーキテクチャー全般はまた、その他の有用なパフォーマンス・メトリックの計算に必要なイベントを収集します。マイクロアーキテクチャー全般解析の結果は、デフォルトでは [Microarchitecture Exploration (マイクロアーキテクチャー全般)] ビューポイント (英語) に表示されます。

マイクロアーキテクチャー全般の結果は、トップダウン特性化を補足する階層的なカラムで表示されます。**[Summary (サマリー)]** ウィンドウは、アプリケーション全体の各カテゴリーのパイプライン・スロットの比率を表示します。結果はさまざまな方法で調査できます。最も一般的な方法は、関数レベルで表示されるメトリックを調査することです。

Grouping: Function / Call Stack

Function / Call Stack	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound		Retiring
					Memory Bound	Core Bound	
price_out_impl	62,556,093,834	1.261	2.2%	7.4%	64.2%	8.4%	17.8%
refresh_potential	17,836,026,754	3.589	3.0%	8.1%	73.2%	9.6%	6.1%
primal_bea_mpp	38,108,057,162	1.393	5.6%	24.3%	34.4%	21.0%	14.7%
update_tree	4,092,006,138	3.373	7.2%	11.5%	62.3%	11.8%	7.2%
sort_basket	12,246,018,369	1.037	20.7%	50.4%	3.8%	4.6%	20.6%
primal_iminus	5,324,007,986	2.148	7.1%	6.7%	55.0%	20.5%	10.8%
primal_net_simple:	266,000,399	2.466	17.4%	43.4%	13.1%	13.1%	13.0%

それぞれの関数で、各カテゴリーのパイプライン・スロットが表示されます。例えば、上記で選択されている `price_out_impl` 関数は、パイプライン・スロットの 2.2% がフロントエンド依存、7.4% が投機の問題、64.2% がメモリー依存、8.4% がコア依存、そして 17.8% がリタイアカテゴリーです。各カテゴリーを展開すると、そのカテゴリーに属するメトリックが表示されます。自動ハイライト機能により、開発者の注意を促す潜在的な問題領域が強調されます。ここでは、`price_out_impl` のメモリー依存のパイプライン・スロットの比率がハイライトされています。

マイクロアーキテクチャー・チューニングの方法論

パフォーマンス・チューニングを行う際は、常にアプリケーションの上位の hotspot に注目します。「hotspot」は最も CPU 時間を消費している関数です。これらのスポットに注目し、アプリケーションのパフォーマンス全体に影響する最適化を特定します。インテル® VTune™ プロファイラーには、ユーザーモードのサンプリングとハードウェア・イベントベースのサンプリングの 2 つの収集モードを備えた hotspot 解析機能があります。マイクロアーキテクチャー全般ビューポイント内の hotspot は、CPU クロックティック数を測定して、最も高いクロックティック・イベントの関数やモジュールを決定することで特定できます。マイクロアーキテクチャーのチューニングから最大限の利益を得るため、並列処理の追加などのアルゴリズムの最適化がすでに行われていることを確認します。一般にシステムのチューニングが最初に行われ、その後アプリケーション・レベルのアルゴリズムのチューニング、アーキテクチャーとマイクロアーキテクチャーのチューニングへと続きます。この手順は、トップダウンのソフトウェア・チューニング方法論と同様に、「トップダウン」と呼ばれます。ワークロードの選択などその他の重要なパフォーマンス・チューニングについては、「謎めいたソフトウェア・パフォーマンスの最適化を紐解く」(英語) の記事で説明されています。

1. hotspot の (アプリケーションの合計クロックティックの大半を占める) 関数を特定します。
2. トップダウン法と以下に示すガイドラインを使用して hotspot の効率を評価します。
3. 非効率であれば、最も重要なボトルネックを示すカテゴリーをドリルダウンし (掘り下げ)、ボトルネックのサブレベルを調べて原因を特定します。
4. 問題を最適化します。インテル® VTune™ プロファイラーのチューニング・ガイドには、各カテゴリーの多くのパフォーマンスの問題に対するチューニングの推奨事項が含まれます。
5. 上位の hotspot をすべて評価するまで上記のステップを繰り返します。

インテル® VTune™ プロファイラーは、hotspot が事前定義されたしきい値の範囲を超えている場合、GUI 上のメトリック値を自動的にハイライト表示します。インテル® VTune™ プロファイラーは、アプリケーション内で取得された合計クロックティックの 5% 以上である関数を hotspot として分類します。パイプライン・スロットが特定のカテゴリーのボトルネックを構成するかどうかの判断はワークロードに関係していますが、以下の表に一般的なガイドラインを示します。

	各カテゴリーのパイプライン・スロットの期待される範囲 (適切に最適化された hotspot)		
カテゴリー	**クライアント/ デスクトップ向け アプリケーション**	**サーバー/データベース/ 分散型アプリケーション**	**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**
リタイア	20-50%	10-30%	30-70%
バックエンド依存	20-40%	20-60%	20-40%
フロントエンド依存	5-10%	10-25%	5-10%
投機の問題	5-10%	5-10%	1-5%

これらのしきい値は、インテルの研究所でいくつかのワークロードを解析した結果に基づいています。hotspot のカテゴリー (リタイアを除く) に費やされた時間が上位にあるか、示される範囲よりも大きい場合、調査が役立つと考えられます。この状況が 1 つ以上のカテゴリーに当てはまる場合、時間が最も上位のカテゴリーを最初に調査します。hotspot が各カテゴリーで時間を費やしていても、その値が通常の範囲内であれば問題は報告されない可能性があることに注意してください。

トップダウン法を適用する際に重要なことは、ボトルネックではないカテゴリーに最適化の時間をかける必要はないということです。最適化を行っても大幅なパフォーマンス向上にはつながりません。

バックエンド依存カテゴリーのチューニング

チューニングされていないアプリケーションのほとんどは、バックエンド依存です。バックエンドの問題を解決するのは、多くの場合、必要以上にリタイアに時間のかかるレイテンシーの原因を解決することです。インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge 向けに、インテル® VTune™ プロファイラーは、高いレイテンシーの原因を検出するためのバックエンド依存メトリックを用意しています。例えば、ラストレベル・キャッシュミス (LLC Miss) メトリックは、データを取得するために DRAM にアクセスする必要があるコード領域を特定し、分割ロード (Split Loads) と分割ストア (Split Stores) メトリックは、パフォーマンスに影響する複数のキャッシュ間のメモリー・アクセス・パターンを指摘します。インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge のメトリックの詳細は、「チューニング・ガイド」を参照してください。インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge (第 3 世代インテル® Core™ プロセッサー・ファミリー) 以降では、バックエンド依存に分類されるイベントが、メモリー依存とコア依存のサブメトリックに区分されています。上位 4 つのカテゴリーに属するメトリックは、パイプライン・スロット・ドメイン以外のドメインを使用する可能性があります。各メトリックは、PMU イベントの最も適切なドメインを使用します。詳細は、各メトリックまたはカテゴリーのドキュメント (英語) を参照してください。

メモリー依存とコア依存のサブメトリックは、トップレベルの分類で使用されるアロケーション・ステージとは対照的に、実行ユニットの使用率に対応するイベントを使用して決定されます。したがって、これらのメトリックの合計は、必ずしもトップレベルで決定されたバックエンド比率と一致するわけではありません (関連性は高い)。

メモリー依存カテゴリーのストールの原因は、メモリー・サブシステムに関連します例えば、キャッシュミスとメモリーアクセスは、メモリー依存のストールの原因となります。コア依存のストールは、各サイクルで CPU の実行ユニットが十分に利用されていないことが原因で発生します。例えば、連続する複数の除算命令は、除算ユ

ニットの競合を引き起こし、コア依存のストールの原因となります。この分類では、ストールで未完了のメモリーアクセスがない場合、そのスロットはコア依存に分類されます。例えば、遅延しているロードがある場合、ロードがまだデータを取得していないことが原因で実行ユニットが待機しているため、サイクルはメモリー依存に分類されます。PMU イベントは、アプリケーションの真のボトルネックを特定するのに役立つ分類が可能になるように、ハードウェアで実装されています。バックエンドの問題のほとんどは、メモリー依存カテゴリーに区分されます。

メモリー依存カテゴリーのほとんどのメトリックは、L1 キャッシュからメモリーまでの、ボトルネックになっているメモリー階層のレベルを特定します。この決定に使用されるイベントは注意深く設計されています。一旦バックエンドがストールすると、メトリックは遅延中の特定のキャッシュレベルへのロード、または実行中のストアのストールを区分しようとします。hotspot が特定のレベルで制限されている場合、そのデータの多くはキャッシュやメモリー階層から取得されていることを意味します。最適化の際は、コアに近い位置にデータを移動することに注目します。ストア依存 (Store Bound) は、パイプラインを進行中のロードが直前のストアに依存しているなどの依存性を示すサブカテゴリーとしても利用されます。これらのカテゴリーには、メモリー依存実行の原因となるアプリケーション固有の動作を特定するメトリックがあります。例えば、ストアフォワードでブロックされたロード (Loads Blocked by Store Forwarding) と 4K エリアス (4K Aliasing) は、アプリケーションが L1 依存 (L1 Bound) であることを示すメトリックです。

コア依存のストールは、バックエンド依存ではそれほど多くはありません。これらのストールは、大量のメモリー要求がないために利用可能な計算リソースが効率的に利用されない場合に発生します。例えば、浮動小数点 (FP) 数値計算を小さなループで行い、データがキャッシュに収まる場合などです。インテル® VTune™ プロファイラーは、このカテゴリーの動作を検出するいくつかのメトリックを用意しています。例えば、除算器 (Divider) メトリックは、除算器ハードウェアが頻繁に使用されたサイクルを特定し、ポート使用率 (Port Utilization) メトリックは実行ユニットの個々の競合を特定します。

Function / Call Stack	Back-End Bound						
	Memory Bound					Core Bound	
	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	Divider	Port Utilization
price_out_impl	4.7%	4.7%	4.4%	44.9%	0.0%	0.0%	7.7%
refresh_potential	1.7%	0.0%	4.6%	75.9%	0.0%	0.0%	10.6%
primal_bea_mpp	0.0%	1.1%	11.7%	26.0%	0.0%	0.0%	23.6%
update_tree	33.3%	16.5%	22.9%	4.1%	0.0%	0.0%	14.6%
sort_basket	11.4%	0.0%	0.0%	0.0%	0.0%	0.0%	13.8%

注

灰色で表示されるメトリック値は、このメトリックで収集されたデータの信頼性が低いことを表します。これは、収集された PMU イベントのサンプル数が非常に少ないことが原因であると考えられます。このデータは無視できますが、収集に戻ってデータ収集時間、サンプリングの間隔、またはワークロードを増やして再度収集することもできます。

フロントエンド依存カテゴリーのチューニング

フロントエンド依存カテゴリーは、ほかのタイプのパイプライン・ストールをいくつかカバーします。パイプラインのフロントエンド部分がアプリケーションのボトルネックになることは、それほど多くはありません。例えば、JIT コードとインタープリターで解釈されるコードは、命令ストリームが動的に生成される (コンパイラーによるコード配置の利点が得られない) ため、フロントエンド・ストールの原因となります。フロントエンド依存カテ

ゴリーのパフォーマンスの改善は、通常、コード配置 (ホットなコードと隣接して配置) とコンパイラーのテクニックに関連します。例えば、分岐の多いコードや大きなフットプリントのコードは、フロントエンド依存カテゴリーで警告されます。コードサイズの最適化やコンパイラーによるプロファイル・ガイド最適化 (PGO) などのテクニックは、多くの場合ストールを軽減するのに有効です。

インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge 以降のトップダウン法では、フロントエンド依存のストールがフロントエンド・レイテンシーとフロントエンド帯域幅の 2 つのカテゴリーに分離されました。フロントエンド・レイテンシー (Front-End Latency) メトリックは、バックエンドの準備ができているにもかかわらず、フロントエンドによって µop が発行されないサイクルを報告します。フロントエンド・クラスターは、サイクルあたり最大 4µop を発行できることを思い出してください。フロントエンド帯域幅 (Front-End Bandwidth) メトリックは、発行された µop が 4 未満のサイクル、つまり、フロントエンドの能力を使い切っていないサイクルを報告します。各カテゴリーの下には、さらにメトリックがあります。

分岐予測ミスは、多くの場合、投機の問題カテゴリーに分類されますが、インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge 以降では、フロントエンドに属する分岐リステア (Branch Resteer) メトリックによって、フロントエンドが非効率になったことが示されます。

Function / Call Stack	Front-End Bound							
	Front-End Latency						Front-End Bandwidth	
	ICache Misses	ITLB ...	Branch Resteers	DSB ...	Length...	MS ...	Front-End Band...	Front-End ...
price_out_impl	0.0%	0.0%	0.9%	0.0%	0.0%	0.1%	3.8%	0.0%
refresh_potential	0.0%	0.0%	1.4%	0.2%	0.0%	0.1%	5.7%	0.1%
primal_bea_mpp	0.0%	0.0%	2.9%	0.1%	0.0%	0.0%	12.2%	0.1%
update_tree	0.0%	0.0%	2.2%	0.6%	0.0%	0.0%	7.2%	1.4%
sort_basket	0.0%	0.0%	11.4%	2.1%	0.0%	0.0%	31.4%	0.6%
primal_iminus	0.0%	0.0%	1.0%	0.7%	0.0%	0.0%	9.0%	0.7%

インテル® VTune™ プロファイラーは、フロントエンド依存の原因を特定するメトリックの一覧を表示します。これらのカテゴリーのいずれかの結果が顕著な場合、メトリックをさらに深く調査して原因を特定し、修正する方法を考えます。例えば、命令トランスレーション・ITLB オーバーヘッド (ITLB Overhead) と命令キャッシュミス (ICache Miss) メトリックは、フロントエンド依存の実行で問題のある領域が分かります。チューニングの推奨事項は、インテル® VTune™ プロファイラーのチューニング・ガイドを参照してください。

投機の問題カテゴリーのチューニング

3 番目のトップレベルのカテゴリー、投機の問題は、パイプラインがリタイアしない命令をフェッチして実行しているためにビジーであることを示します。投機の問題のパイプライン・スロットは、マシンが不適切な投機実行から回復する間、リタイアしないまたはストールする µop が発行されることでスロットが浪費されている状態です。投機の問題は、分岐予測ミスとマシンクリア、および (一般的ではありませんが) 自己修正コードによって引き起こされます。投機の問題は、コンパイラーによるプロファイル・ガイド最適化 (PGO)、間接分岐の回避、およびマシンクリアを引き起こすエラー条件の排除などのテクニックで軽減できます。投機の問題を解決することは、フロントエンド依存のストール数を減らすのに役立ちます。特定のチューニングのテクニックについては、適切なマイクロアーキテクチャー向けのインテル® VTune™ プロファイラーのチューニング・ガイドを参照してください。

Function / Call Stack	Bad Speculation		Back-End Bound
	Branch Mispredict	Machine Clears	
price_out_impl	7.4%	0.0%	72.7%
refresh_potential	8.0%	0.1%	82.8%
primal_bea_mpp	24.3%	0.0%	55.5%
update_tree	11.5%	0.0%	74.1%
sort_basket	50.4%	0.0%	8.4%
primal_iminus	6.7%	0.0%	75.4%
primal_net_simple	0.0%	43.4%	26.1%

リタイアカテゴリーのチューニング

トップレベルの最後のカテゴリー、リタイアは、パイプラインが通常動作の実行でビジーであることを意味します。可能であれば、このカテゴリーにアプリケーションの多くのスロットが分類されるのが理想的です。しかし、パイプライン・スロットでリタイアする大部分がコード領域であれば、改善の余地はあります。リタイアカテゴリーの 1 つのパフォーマンスの問題は、マイクロシーケンサーの高い利用率です。これは、特定の条件が記述された µop の長いストリームを生成することで、フロントエンドをアシストします。この場合、多くの µop がリタイアしますが、いくつかは回避することが可能です。例えば、FP アシストはデノーマルイベントに適用され、多くの場合、コンパイラーのオプション (DAZ や FTZ) によって軽減できます。コード生成の選択もまた、これらの問題を軽減するのに役立ちます。詳細は、インテル® VTune™ プロファイラーのチューニング・ガイドを参照してください。インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge では、アシスト (Assists) はリタイアカテゴリーのメトリックとして分類されます。インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge 以降では、リタイアカテゴリーのパイプラインは、全般リタイア (General Retirement) と呼ばれるサブカテゴリーに分割され、マイクロコード・シーケンサーの µop は別に識別されます。

Function / Call Stack	Retiring				
	General Retirement				Microcode Sequencer
	FP Arithmetic			Other	Assists
	FP x87	FP Scalar	FP Vector		
price_out_impl	0.0%	0.0%	0.0%	100.0%	0.0%
refresh_potential	0.0%	0.0%	0.0%	100.0%	0.0%
primal_bea_mpp	0.0%	0.0%	0.0%	100.0%	0.0%
update_tree	0.0%	0.0%	0.0%	100.0%	0.0%
sort_basket	0.0%	0.0%	0.0%	100.0%	0.0%

まだ行っていない場合、アルゴリズムの並列化やベクトル化によるチューニングは、リタイアカテゴリーでのコード領域のパフォーマンスを改善するのに役立ちます。

まとめ

トップダウン法とインテル® VTune™ プロファイラーにおけるその有効性は、PMU を使用したパフォーマンス・チューニングの新たな方向性を示しています。開発者がこの特性化を習得するために費やす時間は価値あるものです。特性化のサポートは最近の PMU 向けに設計されており、その階層構造は将来のインテル® マイクロアーキテクチャーにも拡張可能です。例えば、特性化は、インテル® マイクロアーキテクチャー開発コー

ド名 Sandy Bridge からインテル® マイクロアーキテクチャー開発コード名 Ivy Bridge の間で大幅に拡張されました。

トップダウン法の目標は、アプリケーション・パフォーマンスのボトルネックの種類を特定することです。インテル® VTune™ プロファイラーの全般解析と可視化機能の目標は、アプリケーションを改善するために適用可能な情報を提供することです。これらを併用することで、アプリケーションのパフォーマンスを大幅に改善できるだけでなく、最適化における生産性も向上できます。

関連クックブック・レシピ

- チューニング・レシピ: フォルス・シェアリング
- チューニング・レシピ: 頻繁な DRAM アクセス
- チューニング・レシピ: 低いポート使用率
- チューニング・レシピ: 命令のキャッシュミス

関連情報

- マイクロアーキテクチャー・パイプ (英語)
- マイクロアーキテクチャー全般ビュー (英語)
- チューニング・ガイドとパフォーマンス解析の記事
- クロックティックとパイプライン・スロット・ベースのメトリック (英語)

OpenMP* コード解析

このレシピは、OpenMP* または OpenMP* - MPI ハイブリッド・アプリケーションの CPU 利用率を解析して、潜在的な非効率性の原因を特定します。

コンテンツ・エキスパート: Dmitry Prohorov (英語)

OpenMP* はフォーク・ジョイン並列モデルであり、OpenMP* プログラムは単一のマスター・シリアルコード・スレッドで実行を開始します。並列領域に到達すると、マスタースレッドは複数のスレッドにフォークして並列領域を実行します。各スレッドは並列領域の最後にあるバリアでジョインして、その後マスタースレッドがシリアルコードの実行を続行します。マスタースレッドが並列領域にフォークし、*barrier* や *single* などの構造でワークを調整する、MPI プログラムのように OpenMP* プログラムを記述することもできます。しかし、シリアルコードが点在する並列領域のシーケンスで構成される OpenMP* プログラムの方が一般的です。

理想的には、並列化されたアプリケーションは利用可能な CPU コアの処理時間を 100% 利用して、実行開始から終了まで有用なワークを実行するワーカースレッドを持ちます。実際には、ワーカースレッドがアクティブスピンで待機している場合 (待機時間は短くなることが予想されます)、または受動的に待機して CPU を消費していない場合は、有効な CPU 利用率は低くなります。ワーカースレッドが待機し、有用なワークを実行していない理由はいくつかあります。

- **シリアル領域の実行 (並列領域外)**: マスタースレッドがシリアル領域を実行している場合、ワーカースレッドは OpenMP* ランタイムで次の並列領域を待機しています。
- **ロード・インバランス:** スレッドは並列領域でワークロードの実行を終了すると、他のスレッドが終了するのをバリアで待機します。
- **並列ワーク不足:** ループの反復回数がワーカースレッド数よりも少ないため、チームのいくつかのスレッドはバリアで待機しており、有用なワークを実行していません。
- **ロックでの同期:** 並列領域内で同期オブジェクトが使用されると、ほかのスレッドとの共有リソースへのアクセス競合を避けるため、スレッドはロックが開放されるまで待機します。

インテル® Composer XE 2013 Update 2 以降とともにインテル® VTune™ プロファイラーを使用すると、アプリケーションが利用可能な CPU をどのように利用し、CPU が未使用である原因を特定できます。

インテル® VTune™ プロファイラーで OpenMP* アプリケーションを解析するには、次の操作を行います。

1. 推奨オプションでコードをコンパイルする
2. OpenMP* 領域解析を設定する
3. アプリケーション・レベルの OpenMP* メトリックを調査する
4. シリアルコードを特定する
5. 潜在的なゲインを予測する
6. 制限事項を理解する

推奨オプションでコードをコンパイルする

コンパイル時に並列領域とソース解析を有効にするには、次の手順に従ってください。

- OpenMP* 並列領域を解析するには、インテル® コンパイラー 13.1 Update 2 以降 (インテル® Composer XE 2013 Update 2 に含まれます) でコードがコンパイルされていることを確認してください。古いバージョンの OpenMP* ランタイム・ライブラリーが検出されると、インテル® VTune™ プロファイラーは警告メッセージを出力します。この場合、収集結果は不完全な可能性があります。

  ドキュメントに記載されている最新の OpenMP* 解析オプションを使用するには、常に最新バージョンのインテル® コンパイラーを使用していることを確認してください。

- Linux* 上で、GCC でコンパイルされた OpenMP* アプリケーションを解析するには、GCC OpenMP* ライブラリー (`libgomp.so`) にシンボル情報が含まれていることを確認してください。これを確認するには、`libgomp.so` を検索して `nm` コマンドでシンボルをチェックします。次に例を示します。

  ```
  nm libgomp.so.1.0.0
  ```

  ライブラリーにシンボル情報が含まれていない場合、シンボル付きの新しいライブラリーをインストールまたはコンパイルするか、ライブラリーのデバッグ情報を生成してください。例えば、Fedora* では `yum` リポジトリーから GCC デバッグ情報をインストールできます。

  ```
  yum install gcc-debuginfo.x86_64
  ```

OpenMP* 解析を設定する

ターゲットの OpenMP* 解析を行うには、次の操作を実行します。

1. インテル® VTune™ プロファイラーのツールバーにある  (スタンドアロン GUI)/  (Visual Studio* IDE) **[Configure Analysis (解析の設定)]** ボタンをクリックします。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

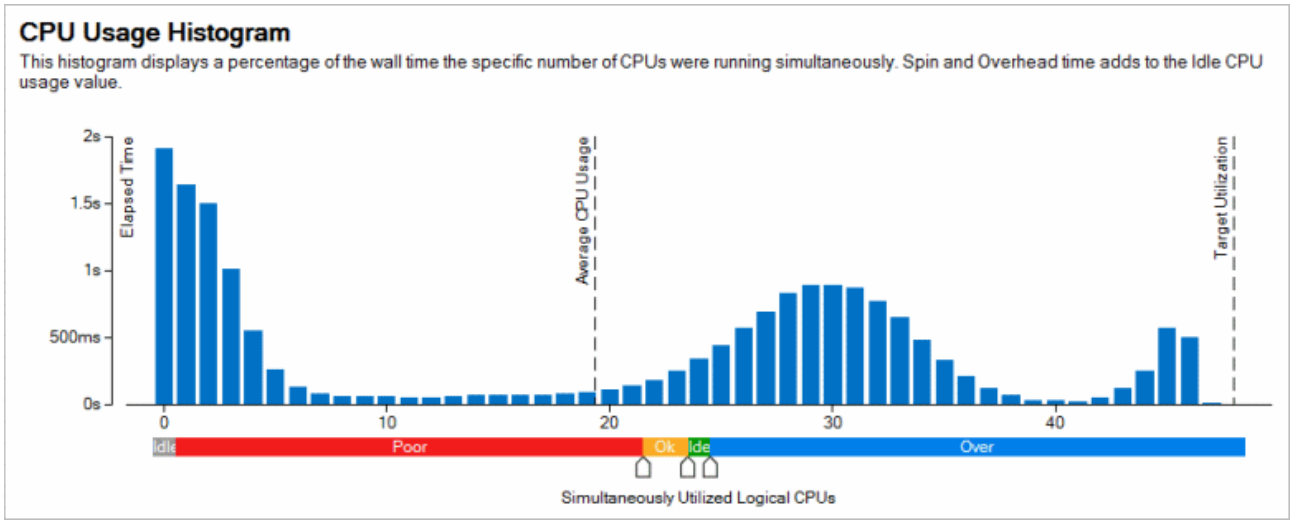
2. **[HOW (どのように)]** ペインで、  [Browse (参照)] ボタンをクリックして OpenMP* 解析をサポートする解析タイプ (スレッド化、HPC パフォーマンス特性、メモリーアクセス、またはカスタム解析タイプ) を選択します。
3. **[Analyze OpenMP regions (OpenMP* 領域を解析)]** オプションが選択されていない場合は選択します (**[Details (詳細)]** セクションを確認)。
4.  [Start (開始)] ボタンをクリックして、解析を実行します。

インテル® コンパイラーの OpenMP* ランタイム・ライブラリーは、プロファイル・モードで実行中のアプリケーション向けに特別なマーカーを提供します。これを利用して、インテル® VTune™ プロファイラーは OpenMP* 並列領域の統計を解読し、アプリケーション・コードのシリアル領域を区別できます。

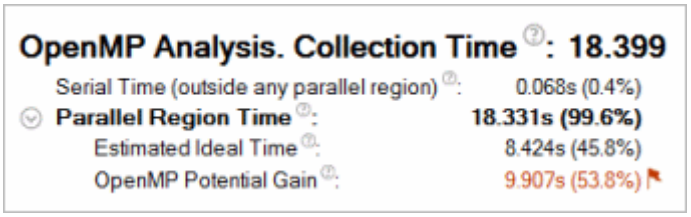
アプリケーション・レベルの OpenMP* メトリックを調査する

解析ターゲットの CPU 利用率を理解してから解析を始めます。HPC パフォーマンス特性 (英語) ビューポイントを使用する場合、[Summary (サマリー)] ウィンドウの **[Effective Physical Core Utilization (効率的な物理コア利用率)]** セクションで、使用されている論理コア数と物理コア数、および CPU 利用率の効率 (パーセント) の予測に注目します。低いコア利用率は、パフォーマンスの問題としてフラグが付けられます。

他のビューポイントでは、アプリケーションの経過時間を CPU 利用率レベルまで細分化した **[CPU Utilization Histogram (CPU 利用率の分布図)]** が表示されます。分布図には有効な利用率しか表示されないため、アプリケーションがスピンループ (アクティブ待機) で CPU を使用した CPU サイクルはカウントされません。利用可能なハードウェア・スレッド数よりも少ない OpenMP* ワーカースレッドを意図的に使用する場合、スライドバーを使用してデフォルトレベルから調整できます。



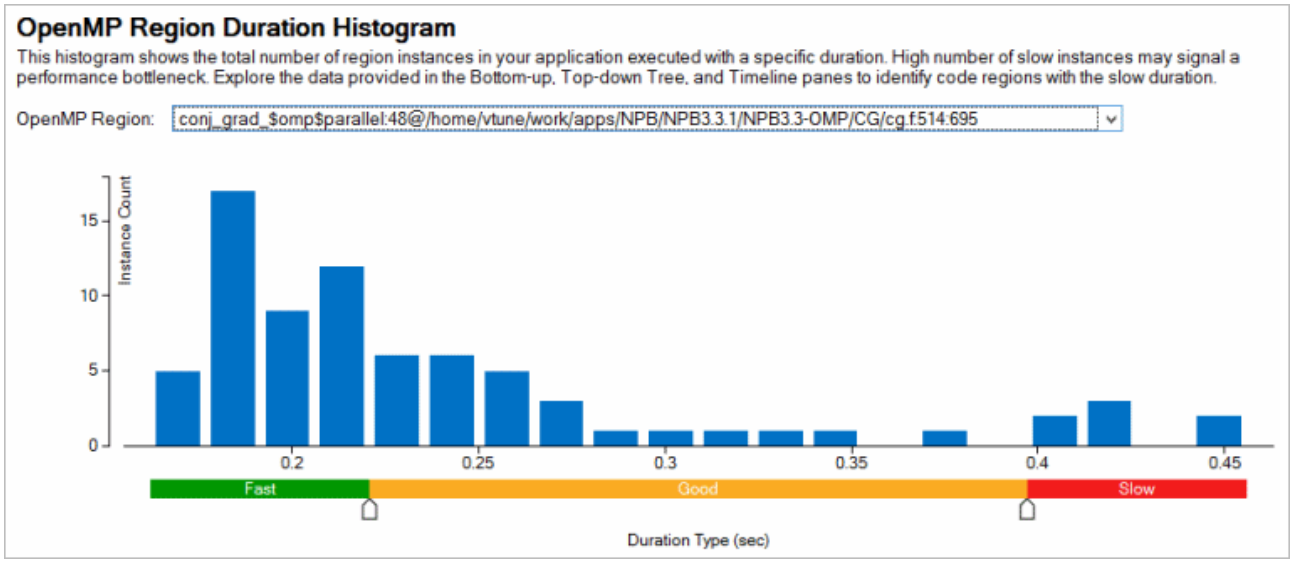
バーが理想的な利用率に近い場合、パフォーマンス向上の可能性を見つけるには、アルゴリズムまたはマイクロアーキテクチャーのチューニングを検討する必要があります。アプリケーションの非効率な並列化については、**[Summary (サマリー)]** ウィンドウの **[OpenMP Analysis (OpenMP* 解析)]** セクションを調査します。



[Summary (サマリー)] ウィンドウのこのセクションには、収集時間とプログラムのシリアル領域 (並列領域外) と並列領域の持続期間が表示されます。シリアル領域が長い場合、さらに並列処理を導入するか、並列化が困難なシリアル領域ではアルゴリズムやマイクロアーキテクチャーのチューニングを行って、シリアル実行を短縮することを検討してください。スレッドカウントの多いマシンのシリアル領域は、潜在的なスケーリングに深刻な悪影響を与えるため (アムダールの法則)、可能な限り最小にすべきです。

[Summary (サマリー)] ウィンドウの **[OpenMP Region Duration Histogram (OpenMP* 領域持続分布図)]** を参照して、OpenMP* 領域のインスタンスを解析し、インスタンスの持続期間の分布を調査して、高速/

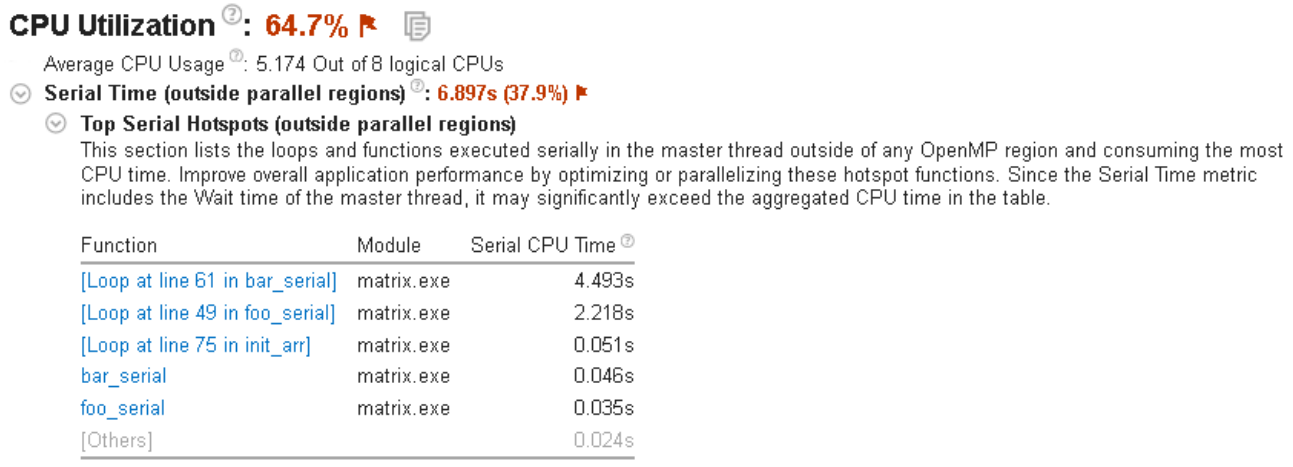
良好/低速領域のインスタンスを識別します。デフォルトの分布図では、最小領域時間と最大領域時間の間が 20/40/20 の比率で高速/良好/低速に分類されます。必要に応じて、しきい値を調整します。



このデータを使用して、**[OpenMP Region/OpenMP Region Duration Type/... (OpenMP* 領域/OpenMP* 領域持続タイプ/...)]** グループ化レベルでグリッドをさらに詳しく解析します。

シリアルコードを特定する

シリアル実行されたコードを解析するには、**[Summary (サマリー)]** ウィンドウの **[Serial Time (outside parallel regions) (シリアル時間 (並列領域外))]** セクションを展開して、**[Top Serial Hotspots (outside parallel regions) (上位のシリアル・ホットスポット (並列領域外))]** を確認します。関数名をクリックすると、**[Bottom-up (ボトムアップ)]** ウィンドウでその関数の詳細を確認できます。



潜在的なゲインを予測する

コードの並列領域で CPU 利用率の効率を予測するには、潜在的なゲイン (英語) メトリックを使用します。このメトリックは、並列領域の実測された経過時間と理想化された経過時間 (スレッドのバランスが完璧で OpenMP* ランタイムのオーバーヘッドがゼロであると仮定) の差を予測します。このデータを使用して、並列実行を改善することで短縮できる最大時間を見積ることができます。

[Summary (サマリー)] ウィンドウには、[Potential Gain (潜在的なゲイン)] メトリック値が最も高い 5 つの並列領域が表示されます。`#pragma omp parallel` で定義された並列領域ごとに 、このメトリックは並列領域のすべてのインスタンスの潜在的なゲインの合計を示します。

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

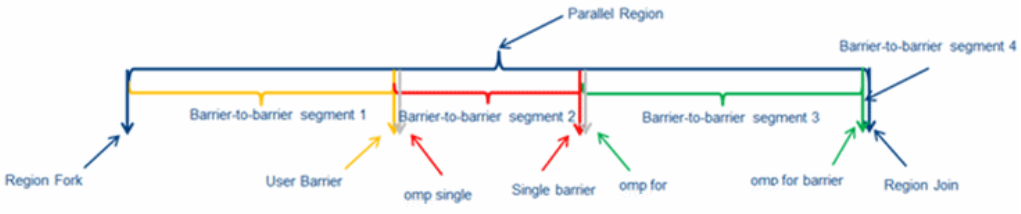
OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
conj_grad_ompparallel:48@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:514:695	9.706s	52.8%	17.649s
MAIN__ompparallel:48@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:185:231	0.173s	0.9%	0.653s
MAIN__ompparallel:48@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:361:365	0.015s	0.1%	0.015s
MAIN__ompparallel:48@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:339:345	0.012s	0.1%	0.013s
MAIN__ompparallel:48@/home/vtune/work/apps/NPB/NPB3.3.1/NPB3.3-OMP/CG/cg.f:263:269	0.000s	0.0%	0.000s
[Others]	N/A*	N/A*	0.000s

*N/A is applied to non-summable metrics.

領域の潜在的なゲインが顕著である場合、領域名のリンクをクリックして **[Bottom-up (ボトムアップ)]** ウィンドウに移動し、バリアによるインバランスなどの非効率なメトリックの詳細な解析を示す **[/OpenMP Region/OpenMP Barrier-to-Barrier Segment/.. (/OpenMP* 領域/OpenMP* バリアからバリアのセグメント/..)]** グループ化を使用して、さらに深く調査できます。

インテル® コンパイラーの OpenMP* ランタイムは、インテル® VTune™ プロファイラー向けにバリアをインストルメントします。インテル® VTune™ プロファイラーは、領域のフォーク位置または以前のバリアからセグメントを定義するバリアまでの、**バリア間**の OpenMP* 領域セグメントの概念を導入しています。

```
#pragma omp parallel
{
    .....
#pragma omp barrier
#pragma omp single
    {
        ....
    }
#pragma omp for
    {
        ...
    }
}
```

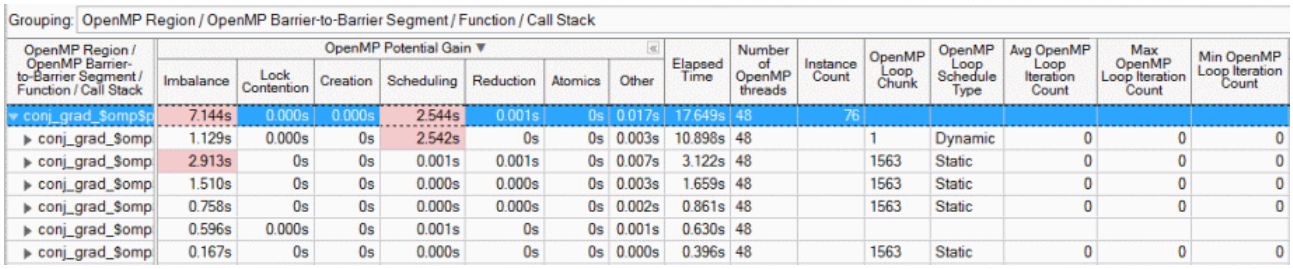


上記の例では、`user barrier`、暗黙的な `single barrier`、暗黙的な `omp for` ループバリアと `region join` バリアとして定義された 4 つのバリアからバリアへのセグメントがあります。

OpenMP* 領域に、並列ループや #pragma single sections などの暗黙のバリア、または明示的なユーザーバリアが複数定義されている場合、特定の構造の影響や非効率なメトリックに対するバリアを解析します。

バリアタイプはセグメント名に組込まれます (例えば、`loop`、`single`、`reduction` など)。また、暗黙のバリアを持つ並列ループについて、ループのスケジュール、チャンクサイズ、および最小/最大/平均ループ反復カウントなど、インバランスやスケジュールのオーバーヘッドを理解するのに役立つ追加の情報を出力します。ループ

反復カウント情報は、外部ループの並列化により、反復回数が少ないワーカースレッドの利用率が低下する問題を識別するのにも役立ちます。この場合、内部ループを並列化するか、collapse 節を使用してワーカースレッドを飽和させることを検討してください。



Grouping: OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack

OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	OpenMP Potential Gain ▼							Elapsed Time	Number of OpenMP threads	Instance Count	OpenMP Loop Chunk	OpenMP Loop Schedule Type	Avg OpenMP Loop Iteration Count	Max OpenMP Loop Iteration Count	Min OpenMP Loop Iteration Count
	Imbalance	Lock Contention	Creation	Scheduling	Reduction	Atomics	Other								
conj_grad_omp2	7.144s	0.000s	0.000s	2.544s	0.001s	0s	0.017s	17.649s	48	76					
conj_grad_$omp	1.129s	0.000s	0s	2.542s	0s	0s	0.003s	10.898s	48		1	Dynamic	0	0	0
conj_grad_$omp	2.913s	0s	0s	0.001s	0.001s	0s	0.007s	3.122s	48		1563	Static	0	0	0
conj_grad_$omp	1.510s	0s	0s	0.000s	0.000s	0s	0.003s	1.659s	48		1563	Static	0	0	0
conj_grad_$omp	0.758s	0s	0s	0.000s	0.000s	0s	0.002s	0.861s	48		1563	Static	0	0	0
conj_grad_$omp	0.596s	0.000s	0s	0.001s	0s	0s	0.001s	0.630s	48						
conj_grad_$omp	0.167s	0s	0s	0.000s	0s	0s	0.000s	0.396s	48		1563	Static	0	0	0

OpenMP* スレッド数で正規化した非効率のコスト (経過時間) を表示することで、領域の潜在的なゲインの内訳を示す **[Potential Gain (潜在的なゲイン)]** カラムのデータを解析します。経過時間コストは、特定の非効率性タイプに対応すべきかどうかを判断するのに役立ちます。インテル® VTune™ プロファイラーは、次の非効率性タイプを認識できます。

- **Imbalance (インバランス)**: スレッドは異なる時間でワークを終え、バリアで待機しています。インバランス時間が顕著である場合、動的なスケジュールの導入を検討してください。インテル® Parallel Studio Composer Edition の OpenMP* ランタイム・ライブラリーはインバランスを正確にレポートし、メトリックはサンプリングに基づいて計算される他の非効率性のように統計の精度に依存しません。
- **Lock Contention (ロック競合)**: スレッドは、競合するロック、または "ordered" 節が指定された並列ループで待機しています。ロック競合の時間が顕著である場合、リダクション操作、スレッド・ローカル・ストレージ、または低コストのアトミック操作を使用することで、並列構造内での同期を回避してください。
- **Creation (生成)**: 並列ワークの配置に関連したオーバーヘッド。並列ワークの配置時間が顕著である場合、並列領域を外部ループに移動して、並列処理の粒度を粗くしてください。
- **Scheduling (スケジュール)**: ワーカースレッドへの並列ワークの割り当てに関連した OpenMP* ランタイム・スケジューラーのオーバーヘッドです。スケジュールの時間が顕著である場合 (動的スケジュールでよく見られます)、大きなチャンクサイズの "dynamic" スケジュール、または "guided" スケジュールを使用します。
- **Atomics (アトミック)**: アトミック操作の実行に関連した OpenMP* ランタイムのオーバーヘッド。
- **Reduction (リダクション)**: リダクション操作で費やされた時間。

インテルの OpenMP* ランタイムのバージョンが古く、**[Potential Gain (潜在的なゲイン)]** カラムを拡張できない場合、対応する CPU 時間メトリックの内訳を解析してください。

パフォーマンスが重要な OpenMP 並列領域のソースを解析するには、**[OpenMP Region/.. (OpenMP* 領域/..)]** グループ化レベルでソートしたグリッドで領域識別子をダブルクリックします。インテル® VTune™ プロファイラーは、ソースビューを開いて、インテル® コンパイラーが生成した疑似関数内の選択された OpenMP* 領域の先頭を表示します。

注

デフォルトでは、インテル® コンパイラーは領域名にソースファイル名を追加しません。そのため、OpenMP* 並列領域名が `[unknown （不明）]` と表示されます。領域名のソースファイル名を取得するには、コンパイルオプションに `-parallel-source-info=2` を追加します。

制限事項を理解する

インテル® VTune™ プロファイラーの並列 OpenMP* 領域の解析には、次のような制限があります。

- サポートされる語彙的な並列領域の最大数は 512 です。512 個を超える並列領域にスコープが到達すると、並列アノテーションは出力されません。
- 入れ子になった並列領域はサポートされません。最上位の項目のみが領域を生成します。
- インテル® VTune™ プロファイラーは、静的リンクされた OpenMP* ライブラリーをサポートしません。

関連クックブック・レシピ

- チューニング・レシピ: OpenMP* インバランスとスケジュール・オーバーヘッド
- チューニング・レシピ: 低いプロセッサー・コア利用率: OpenMP* シリアル時間
- 設定レシピ: MPI アプリケーションのプロファイル

関連情報

- knob analyze-openmp=true (英語) vtune オプション
- MPI コード解析 (英語)

インテル® GPU 向けのソフトウェア最適化

インテル® VTune™ プロファイラーを使用して、インテル® GPU へオフロードする場合のオーバーヘッドを予測します。GPU にオフロードされた計算タスクのパフォーマンスを解析します。

コンテンツ・エキスパート: Alexander Kurylev、Vladimir Tsymbal

ヘテロジニアス・コンピューティングの普及に伴い、パフォーマンスを追求する開発者は、優れたパフォーマンスを発揮するワークロードの種類がハードウェア・アーキテクチャーによって異なることに気付きました。インテルは、CPU、GPU、および FPGA を含む、多くのハイパフォーマンスなアーキテクチャーを提供しています。この記事では、インテル® VTune™ プロファイラーを使用して、インテル® GPU にオフロードされた計算集約型ワークロードをプロファイルおよび最適化する方法を説明します。

インテル® GPU を理解する

- **並列処理の採用:** ワークロード集約型の GPU から優れたパフォーマンスを引き出すには、GPU のアーキテクチャーと機能を理解することから始めます。GPU は、連携して動作するいくつかの小さなプロセシング・コアを利用した高レベルの並列構造を採用しています。GPU は、同時に実行可能なタスクに分割できるワークロードに適しています。GPU のシングルコアのシリアル・パフォーマンスは、CPU よりもかなり低くなります。そのため、アプリケーションは、GPU で利用可能な大規模な並列処理を活用する必要があります。
- **データの適切な移動:** GPU を使用するには、GPU との間でデータを移動する必要があります。データを適切に移動しないと、大きなオーバーヘッドが発生し、パフォーマンスに影響を与えます。GPU の時間と空間的な局所性を活用するように、データを適切に処理します。最高のパフォーマンスを得るには、レジスターとキャッシュを利用してデータを隣接して格納することが重要です。
- **オフロードモデルの使用:** ワークロードの最も重要な部分は GPU を利用して処理されますが、ほかのワークロード・タスクを実行する CPU も重要であることに変わりはありません。**オフロード**モデルで GPU を使用して、ワークロードの一部を GPU (**ターゲット**) デバイスにオフロードします。GPU は、GPU で優れたパフォーマンスを発揮する部分の**アクセラレーター**として機能します。残りのワークロードは CPU (**ホスト**) で実行します。ソフトウェア・パフォーマンスの最適化は、次の 2 つのタスクで行われます。
    - GPU への最適なオフロード
    - GPU 向けの最適化

注

この記事では、汎用 GPU (GPGPU) の計算での利用について取り上げ、計算モデルで GPU を利用する次のポイントを説明します。

- オフロードするコード領域
- オフロードの方法
- GPGPU アルゴリズムの記述方法
- インテル® VTune™ プロファイラーの GPU オフロード解析を使用して GPU オフロードのパフォーマンスを解析する方法

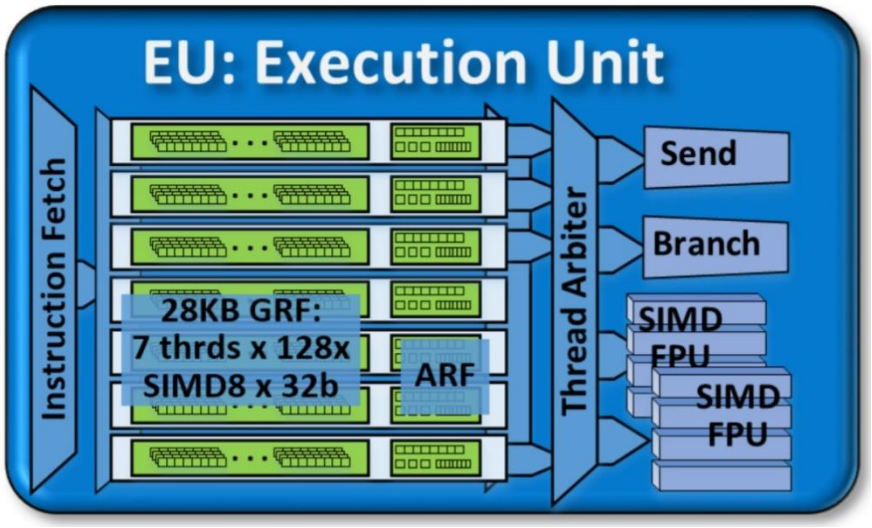
この記事では、インテル® GPU のグラフィックスでの利用方法については取り上げていません。グラフィカル・アプリケーションの解析には、インテル® VTune™ プロファイラーの GPU 計算/メディア・ホットスポット解析やインテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA) を使用してください。

インテル® GPU のアーキテクチャー

GPU オフロードモデルを説明する前に、まずインテル® GPU (**Gen9 GT2 GPU** など) のアーキテクチャーを確認しておきましょう。このデバイスはインテル® マイクロアーキテクチャー (開発コード名 Skylake) に統合されています。この GPU は、OpenCL* や SYCL*/DPC++ などの高水準言語を使用してプログラムできます。

Gen9 GT2 GPU には 24 の実行ユニット (EU) を備えた 1 つのスライスがあります。実行ユニットは、GPU アーキテクチャーの基本的な構成要素です。

Gen9 GT2 GPU の実行ユニット (EU)



EU は、同時マルチスレッディング (SMT) と細粒度のインターリーブ・マルチスレッディング (IMT) を組み合わせたものです。EU は、複数の問題、SIMD ALU (Single Instruction Multiple Data 演算ユニット) を処理するコンピューティング・プロセッサーです。これらの SIMD ALU は、複数のスレッドでパイプライン化されます。SIMD ALU は、高スループットの浮動小数点演算と整数演算に役立ちます。EU の細粒度のスレッド化により、実行の準備ができた命令の継続的なストリームが保証されます。IMT は、メモリーのギャザー/スキャッター、サンプラー要求、その他のシステム通信などの、長い操作のレイテンシーも隠蔽します。

スレッドアービターは、操作の各サイクルでいくつかの命令をディスパッチします。これらの命令が機能ユニットに伝播しない場合、ストールが発生します。ストールの期間は、その状態で経過した実行サイクルの数で測定されます。この測定は、EU の効率を予測するのに役立ちます。**EU Array Stalled (EU 配列ストール)** メトリックは、EU がストールしたが少なくとも 1 つのスレッドがアクティブだった場合のサイクル数をカウントします。ストールは、EU がメモリー・サブシステムからのデータを待機する際に発生する可能性があります。関連する GPU メトリックの詳細は、『インテル® VTune™ プロファイラー・ユーザーガイド』の「GPU メトリック・リファレンス」を参照してください。

効率良いスケジューリングの重要性

超並列マシンの計算能力を最大限に使用するには、GPU のすべての EU に十分な計算を供給する必要があります。そのため、EU には機能処理ユニットよりも多くのハードウェア・スレッドがあります。多くのハードウェア・スレッドがあることで、実行する必要のある命令のオーバーサブスクリプションが発生する可能性がありますが、同時に待機中のデータによるストールを隠蔽するのにも役立ちます。

この方法によるスレッドのスケジューリングは、コストのかかる操作です。スケジューリングを効率良く費用効果の高いものにするには、すべての EU をできるだけビジーにすることが重要です。次のケースでは、スケジューリングの効果は低くなります。

- 計算量が少なすぎる。スケジューリングのオーバーヘッドが、有用な計算を完了するために費やされる時間に匹敵する可能性があります。
- 計算量が多すぎる。スレッド間のワークの分散が不均等になり、GPU のすべての EU の全体的な占有率が低くなる可能性があります。

これらの両方の状況を検出するには、**EU Threads Occupancy (EU スレッドの占有期間)** メトリックを使用します。スレッドの占有率が低いことは、スレッド間でワークロードが効率良く分散されていないことを示します。

それほど一般的ではない状況として、特定の期間 EU 向けのタスクがない場合にも発生することがあります。EU がアイドル状態になり、占有率に悪影響を与える可能性があります。この状況を検出するには、**EU Array Idle (EU アレイアイドル)** メトリックを使用します。

浮動小数点ユニットを使用した SIMD 実行

EU のプライマリー計算ユニットは SIMD 浮動小数点ユニット (FPU) のペアです。これらの FPU は、実際には浮動小数点と整数の両方の計算をサポートしています。次の表は、これらの FPU の SIMD 実行機能を説明したものです。

データサイズ	データ型	SIMD 操作の数
16 ビット	整数	8
16 ビット	浮動小数点	8
32 ビット	整数	4
32 ビット	浮動小数点	4

EU IPC Rate (EU IPC レート) メトリックは、FPU の飽和状態を示す指標です。たとえば、2 つの非ストールスレッドでマシンの浮動小数点計算スループットが飽和する場合、メトリックは 2 です。通常、このメトリックは理論上の最大値である 2 よりも低くなります。

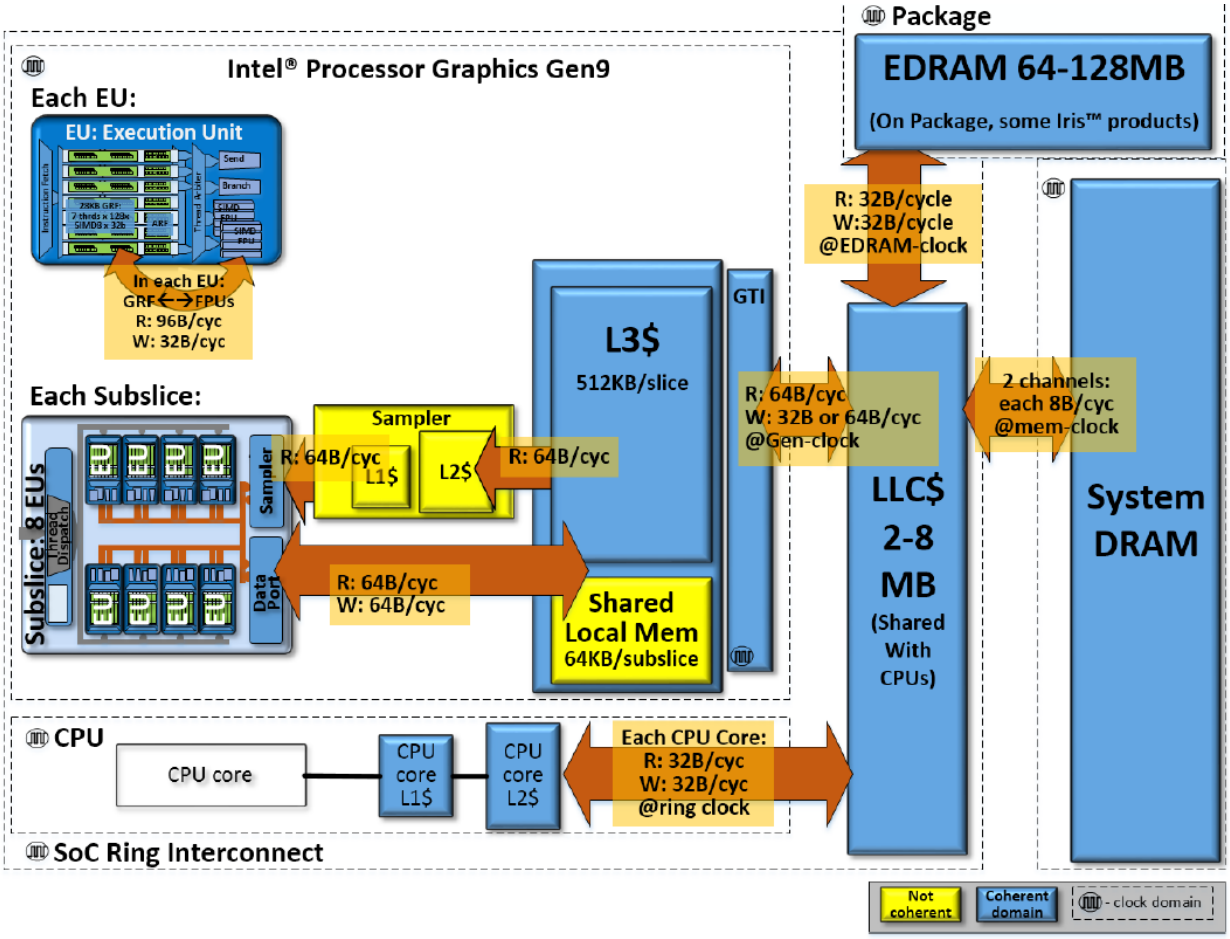
FPU が飽和しておりデータ幅が狭い場合、命令レベルの並列処理は不適切であると言えます。この場合、**SIMD Width (SIMD 幅)** メトリックを確認します。

SIMD 幅の値	意味
4 未満	コンパイラーがループのベクトル化を妨げている原因を確認します。
4 以上	コンパイラーによる命令のベクトル化は成功しています。データの依存関係を削除するか、コードにループアンロール手法を適用すると、この値をデータの局所性とキャッシュの再利用に適した 16 または 32 に増やすことができます。

メモリー・サブシステム

Gen9 GT2 GPU には、**UMA (Unified Memory Architecture)** を備えた独自のメモリー・サブシステムが用意されています。このメモリー・サブシステムは、物理メモリーを CPU と共有して、ゼロコピーバッファー転送を効率良く行います。この機能により、以下のように、CPU と GPU 間のデータ転送が高速化されます。

SOC レベルでのインテル® プロセッサー・グラフィックス Gen9 GT2 GPU のメモリー階層



EU は DRAM/LLC メモリーからデータを受け取ります。GPU L3 または共有ローカルメモリー (SLM) にキャッシュされているデータブロックを再利用できます。大規模な並列処理により、すべての EU がメモリーからデータを要求すると、メモリー・サブブロックの帯域幅が飽和する可能性があります。

ローカル CPU キャッシュへのアクセスは、システムメモリーへのアクセスよりも高速です。理想的な状況では、データアクセスはローカル CPU キャッシュからも発生します。同様に、EU で読み取ったデータは、L3 GPU キャッシュに残しておくことができます。再利用する場合、キャッシュからのデータアクセスのほうが、メインメモリーからデータをフェッチするよりも高速になります。

Gen9 メモリー・アーキテクチャーでは、各スライスはそのスライスの L3 キャッシュにアクセスできます。各スライスには 2 つのサブスライスも含まれます。各サブスライスには、次のものが含まれます。

- ローカル・スレッド・ディスパッチャー
- 命令キャッシュ

- L3 へのデータポート
- 共有ローカルメモリー (SLM)

次のいずれかの方法で、データの局所性を制御できます。

- 特定のデータアクセス。L3 キャッシュにデータを格納するハードウェアを支援します。
- ワークグループでアクセス可能なローカルメモリーを割り当てる特別な API。ハードウェアにより SLM で提供されます。

L3 キャッシュのデータへのアクセスは非常に高速ですが、キャッシュ容量は大きくありません。大きな配列をトラバースすると、データが排出されてキャッシュを有効に利用できない可能性があります。**L3 Cache Miss (L3 キャッシュミス)** メトリックは、GTI の背後のメモリーからデータをフェッチするために必要なデータアクセスの量を示します。データ・ブロッキング手法は、キャッシュミスを減らすのにも役立ちます。例えば、SLM に収まるデータのブロックを保持する場合、サブスライスのローカル・スレッド・ディスパッチャーは最高レベルのデータの局所性を保持できます。インテル® VTune™ プロファイラーを使用して SLM トラフィックを追跡し、転送されたデータの量と転送速度に関する情報を確認できます。

インテル® VTune™ プロファイラーの GPU プロファイル機能

この記事では、GPU 解析をサポートするように調整されたインテル® VTune™ プロファイラーの主要な機能を取り上げています。次のワークフローは、これらの機能の説明です。

1. アプリケーションの **GPU オフロード解析**を実行します。
    - アプリケーションが CPU 依存か GPU 依存かを確認します。
    - GPU 利用率を定義します。
    - GPU EU が実行中にストールしていないか確認します。
    - GPU をビジー状態に保つのに最も適した計算タスクを特定します。これらのタスクは、GPU の効率をさらに解析する際の候補と見なすことができます。
2. **GPU 計算/メディア・ホットスポット**のプロファイルを収集します。次のメトリックを使用して、上位の計算タスクのリストを取得します。
    - 実行時間
    - EU 効率
    - メモリーストール
3. **メモリー階層ダイアグラム**を使用して、最も効率の悪い計算タスクに取り組みます。
    - データ転送/帯域幅メトリックを解析します。
    - 実行のボトルネックの原因となるメモリー/キャッシュユニットを特定します。
    - GPU のマイクロアーキテクチャーの制約に基づいて、アルゴリズムのデータ・アクセス・パターンを決定します。
4. カーネルで **[Instructions Count (命令カウント)]** プリセット解析を実行します。
    - 命令セットと、コンパイラーにより生成された SIMD 命令の選択を確認します。
    - コンパイラーが効率の良い命令を生成するように、特別なコンパイルオプションとプラグマを活用します。
5. 大規模な計算カーネルには、GPU 計算/メディア・ホットスポット解析の **[Basic Block Latency (基本ブロック・レイテンシー)]** プリセットを使用します。
    - 最大の実行レイテンシーの原因であるコード領域を特定します。
    - **[Source (ソース)]** ビューで、ソースコード行に対するレイテンシー・メトリックを調べます。

6. **[Memory Latency (メモリー・レイテンシー)]** プリセットを使用して、重大な実行ストールを示したメモリー・アクセス・コードを調べます。
    - 個々の命令に対するレイテンシーを表示する **[Assembly (アセンブリー)]** ビューのアセンブリー命令から、メモリーアクセスの詳細を調べます。
    - GPU 向けの既知の最適化手法を使用して、メモリーに適したパターンになるようにデータアクセスを再配置します。
7. 目標のパフォーマンス・メトリックになるまで、改善したアルゴリズムで **GPU 計算/メディア・ホットスポット解析**を繰り返します。

インテル® GPU にオフロードする場合の最適化手法

ヘテロジニアス・アプリケーションは通常、アクセラレーターにオフロードするコード領域を特定して設計されます。オフロードするコード領域を特定できない場合は、インテル® Advisor のオフロードのモデル化 (英語) を使用して特定します。

この記事では、GPU にオフロードするコードをすでに特定していると仮定しています。次に、ホスト側でオフロードを実装する最適な方法を紹介します。

ステップ 1: デバイスの使用率を調べる

最適化手法は、CPU コアとアクセラレーター EU でアルゴリズムの実行に費やされる時間を効率良く分散する必要があります。デバイスの使用率を示す (**[CPU Usage (CPU 利用率)]** メトリックおよび **[GPU Usage (GPU 利用率)]** メトリック) は、この効率を早い段階で判断するのに役立ちます。これらの理想値は 100% ですが、実行にギャップや遅延がある場合、インテル® VTune™ プロファイラーを使用して、ギャップや遅延が発生したアプリケーション・コードの場所を特定します。

ステップ 2: GPU でのコード実行の効率を定義する

行列サンプル・アプリケーション (英語) を見てみましょう。このアプリケーションには、密行列 `C = A B` を使用した FP データに対する行列-行列乗算操作が含まれます。

コーディングを簡単にするため、`A`、`B`、`C` は `n × n` の正方行列とします。

注
可読性を高め、コンパクトに表現するため、簡素化されています。ベンチマークの行列乗算タイプはよく知られていて、アクセラレーター向けにも多くの計算最適化手法が開発されています。アルゴリズムの合成ではなく、アルゴリズムの解析を検討します。

```
for (size_t i = 0; i < w; i++)
        for (size_t j = 0; j < w; j++) {
            c[i][j] = T{};
            for (size_t k = 0; k < w; k++)
                c[i][j] += a[i][k] * b[k][j];
    }
```

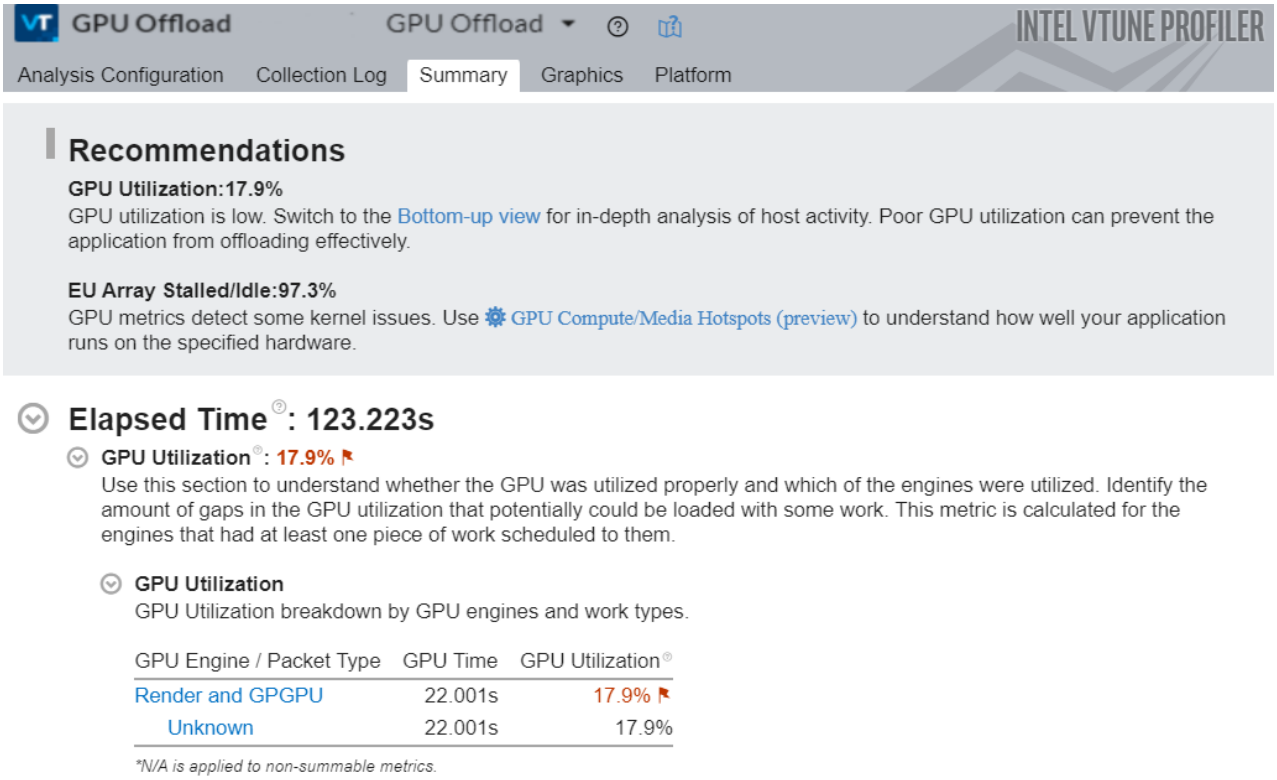
この例では、行列サンプルの簡素化された C++ バージョンを見ていきます。このバージョンでは、キューへのカーネル送信の詳細が省略されています。実際の行列サンプルは、データ並列 C++ (DPC++) で記述され、インテル® oneAPI DPC++/C++ コンパイラーでコンパイルされます。

アクセラレーターにオフロードするコード領域を特定します。通常は、最も外側のループが適切な候補です。しかし、この例では、最も内側のループが計算カーネルである可能性があります。また、このコードの最も内側のループは、必ずしもサンプルの最も内側のループであるとは限りません。高いレベルのライブラリー呼び出しやサードパーティーの機能は、コンピューターの反復構造全体をマスクする可能性があります。そのため、この手法の説明では、最も内側のループをオフロードすることを選択します。

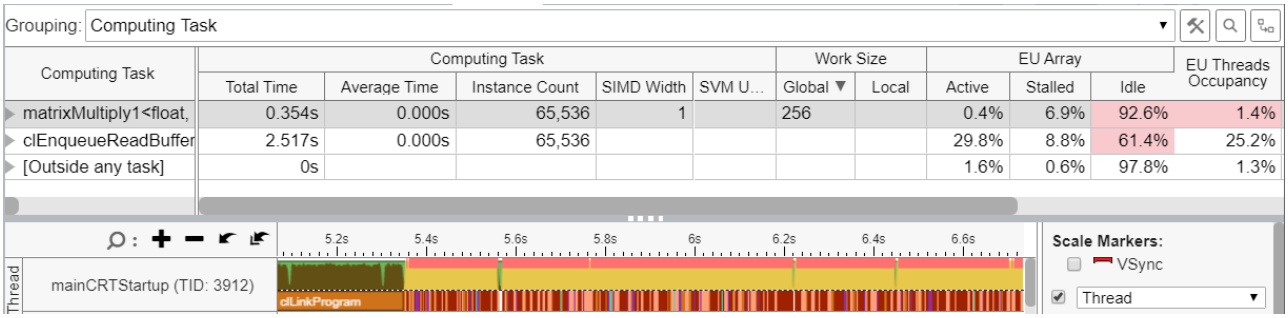
```
for (size_t k = 0; k < w; k++)
                c[i][j] += a[i][k] * b[k][j];
```

ステップ 3: GPU オフロード解析を実行する

インテル® VTune™ プロファイラーの GPU オフロード解析を使用して、GPU にオフロードするホットな計算タスクを素早く特定します。これらのタスクを送信する際に、CPU アクティビティーを明確にすることもできます。以下の例では、1 つのアクティブな計算タスクに注目しています。そのため、ここでは CPU を無視できます。GPU オフロード解析を使用して、GPU での計算タスクの実行に関する情報を収集します。



解析が完了すると、**[Summary (サマリー)]** ウィンドウに GPU 利用率と EU ストールの測定値に関する情報が表示されます。推奨事項に従って、まず低い GPU 利用率の原因となる可能性のあるホスト・アクティビティーを調べます。**[Graphics (グラフィックス)]** タブに切り替えて、**[Bottom-Up (ボトムアップ)]** ビューを開きます。



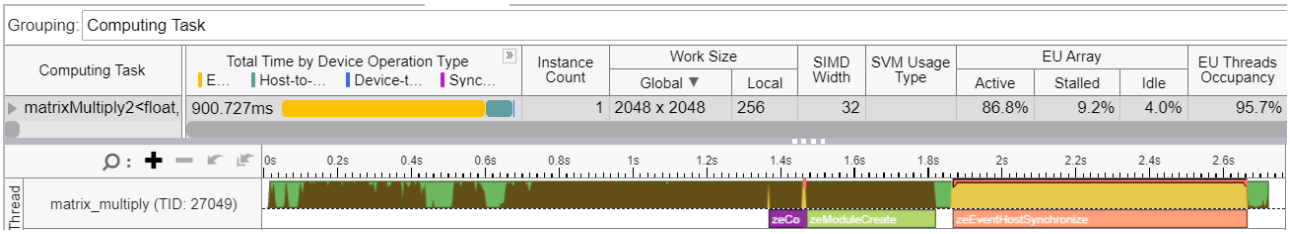
上の図の `matrixMultiply1` カーネルの結果を見ます。

すばやくワークを解析するため、このバージョンのカーネルは 256 X 256 次元の行列を使用しています。**[Instance Count (インスタンス・カウント)]** カラムは、カーネルが 65,536 回呼び出されたことを示しています。各インスタンスは非常に小さいため、カーネルの平均時間はゼロ秒に四捨五入されています。タイムラインのパターンは、カーネルの呼び出し速度が速いことも示しています。この場合、ほとんどの時間は小さなカーネルの作成に費やされます。**[EU Array (EU アレイ)]** セクションの **[Idle (アイドル)]** カラムは、EU が 92.6% の時間でアイドル状態だったことを示しています。非常に多くの短いカーネルを呼び出していることは、ワークの非効率性を示す重要な指標です。

[Work Size (ワークサイズ)] セクションから、ワークの分散が非効率だったことが分かります。外側のループをオフロードしてみましょう。

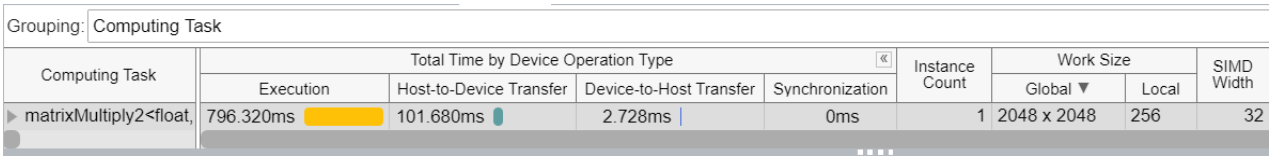
この処理により、計算カーネルのインスタンス数が 1 つ (`matrixMultiply2`) に減り、パフォーマンスが向上するはずです。次の図は、この改良バージョンのカーネルの GPU ホットスポット解析を示しています。このバージョンは、**ナイーブ実装**とも呼ばれます。

行列乗算サンプルのナイーブ実装のための GPU ホットスポット解析



このケースでは、行列のサイズは 2048 X 2048 に増加し、ウォールクロック・パフォーマンスは 10 倍以上高速でした。**[EU Threads Occupancy (EU スレッドの占有期間)]** メトリックは 95.7% と高い値です。これは、実行ユニットで利用できる十分なワークがあることを示しています。

デバイスの操作によるタスク時間の特徴付け



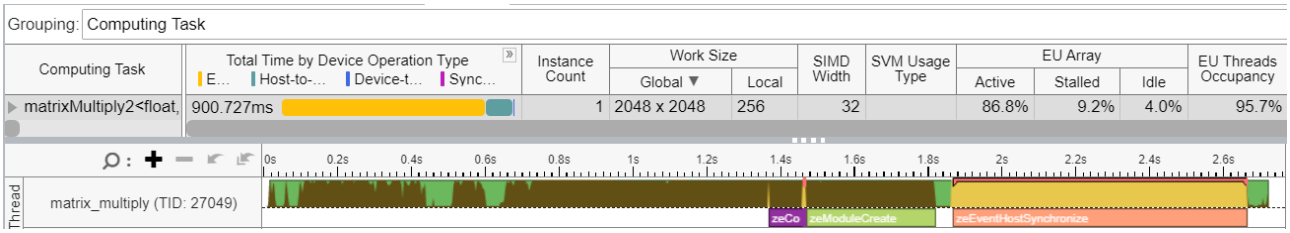
上の図のタイムラインを見ると、データ転送が 100 ミリ秒しかかかっていないのに対して、実行に 800 ミリ秒近くかかっている計算タスクが分かります。このデータ実行とデータ転送の比率は、アルゴリズムを改良することで改善できます。

コンパイラーが SIMD 幅 32 の SIMD 命令を生成していることに注意してください。データアクセスが変更された結果、前回の実行ではほぼゼロであった EU のアクティブな時間が、86.8% になりました。この例は、カーネルの各呼び出し内で十分なワークを提供することの重要性を示しています。

ステップ 4: GPU 計算/メディア・ホットスポット解析を実行する

行列乗算サンプルのナイーブ実装は初期バージョンよりも高速ですが、さらにパフォーマンスを向上できる可能性があります。

インテル® VTune™ プロファイラーは、高い **[EU Threads Occupancy (EU スレッドの占有期間)]** メトリック (95.7%) を示しています。この値から、EU 間でワークが適切に分散されていることが分かります。しかし、**[EU Array Stalled (EU 配列ストール)]** メトリックはわずか 9.2% であり、`matrixMultiply2` カーネルで実行エンジンが十分に活用されていないことが推測できます。



カーネルを制限している要因を調査するため、GPU 計算/メディア・ホットスポット解析を実行します。この解析では、GPU でのカーネル実行に関する詳細な情報を確認できます。

最初に、カーネルが計算依存なのかメモリー依存なのかを特定します。

GPU ホットスポット解析には、いくつかの事前定義済みプロファイルやプリセットが用意されています。これらのプリセットを使用して、メモリーアクセスや計算効率に関連する異なるメトリックを収集できます。カーネル実行を適切に理解できるように、**[Full Compute (完全な計算)]** プリセットを使用しました。このプリセットの情報から、カーネル `matrixMultiply2` の実行で、EU FPU が 63.5% の時間しかアクティブでなかったことが分かります。

計算カーネルの FPU アクティビティー

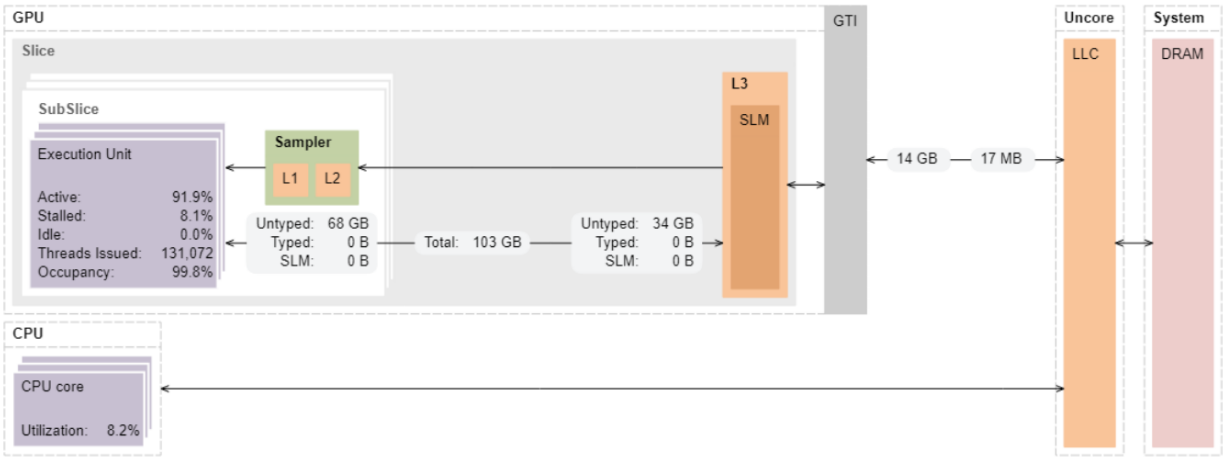
Computing Task	EU Instructions			L3 Bandwidth, GB/sec	Untyped Memory Bandwidth, GB/sec		Shared Local Mem...	
	IPC Rate	2 FPUs active	Send active		Read	Write	Read	Write
matrixMultiply2<float,	1.705	63.5%	20.0%	132.179	88.105	44.074	0.000	0.000

つまり、このカーネルは計算依存ではなく**メモリー依存**です。

次に、**メモリー階層ダイアグラム**を確認します。このダイアグラムは、EU とメモリーユニット間のデータ転送情報を提供します。この情報は、カーネルのコードの最適化ステップを定義するのに役立ちます。

[Overview (概要)] プリセットを選択すると、メモリー階層ダイアグラムに、メモリーユニット (GPU L3 キャッシュ、GTI インターフェイス、LLC および DRAM など) と EU 間のリンクの帯域幅の値と、転送された合計データが表示されます。

GPU メモリー・サブシステムのカーネルデータ転送



EU に転送されたデータの総量 (~68GB) と GTI インターフェイスを経由して LLC/DRAM から取得されたデータ (14GB) に注目します。

これらのデータサイズを各行列データ配列のサイズ (2048x2048x4=16MB) と比較すると、転送された量は非常に多くなっています。この状態では、グローバルメモリーにアクセスするため、実行は非効率になります。効率の良いデータアクセス (配列のユニット・ストライド・アクセスなど) やグローバルメモリーへの最小限のアクセスで、この問題に対処します。

ステップ 5: 追加のカーネルコードの最適化

グローバルメモリーからのデータのフェッチは、GPU の一般的な性能制限要因です。ディスクリート GPU では、この問題は悪化します。PCIe* バスで、帯域幅やレイテンシーがさらに制限されます。一般的ですが次善の最適化アプローチは、データの局所性と再利用を増やすことです。この最適化は、行列領域をブロックして、実行ユニットに近いキャッシュメモリーに収まる小さなブロック内で積算演算を完了します。次のいずれかの方法で、この最適化を実装できます。

- ハードウェアが頻繁にアクセスされるデータを認識し、そのデータを自動的にキャッシュ内で保持することを許可します。
- 共有ローカルメモリーに最も使用されるデータを配置して、データブロックへのアクセスを手動で制御します。

2 つ目の方法を次の条件で実装するときは注意してください。

- スレッドの管理が最適でない (SLM アクセスがスライスからスレッドに制限される) 場合。
- データアクセスが遅い (データの読み取り/書き込みの比率がしきい値未満の) 場合。

注
読み取り/書き込み比率の GPU パフォーマンスへの影響は GPU ハードウェアによって異なります。読み取り/書き込みの比率のしきい値は微妙で、GPU ハードウェアに依存します。しかし、書き込み操作の数が増えると、パフォーマンスが低下する可能性も高くなります。

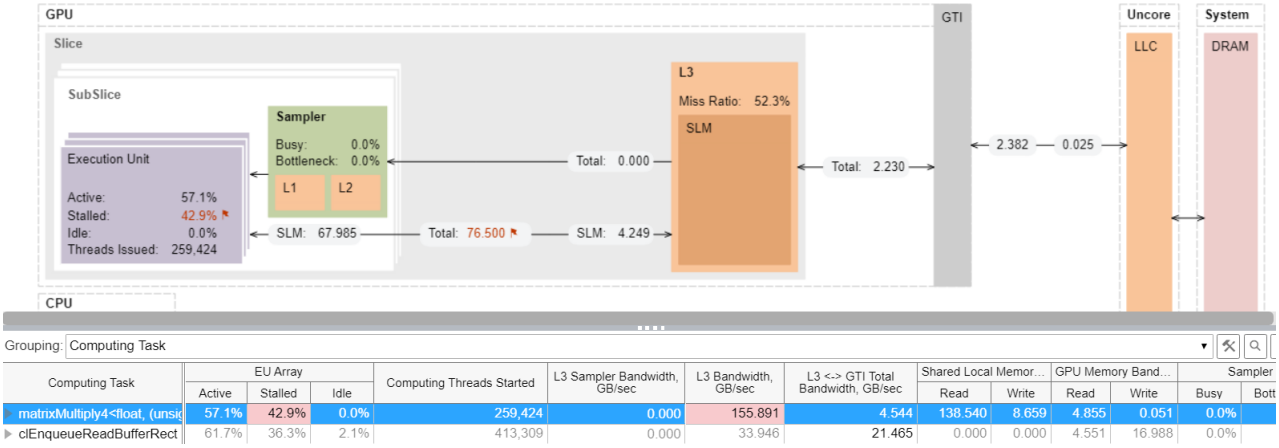
行列乗算アルゴリズムで SLM を使用する 1 つのアプローチは、行列のグローバル・ワークセットをブロックまたはタイルに分割し、タイルでドット積演算を別々に実行することです。このアプローチでは、タイル全体が SLM 領域に収まるように、グローバル・メモリー・アクセスの数を減らす必要があります。データ配列への最適なアクセスは有効になりませんが、データの局所性が得られるため、アクセスは高速になります。

以下のコードで、擬似コードはローカル・インデックス空間のタイルへのデータアクセスのアイデアを示しています。

```
i, j // global idx
    for (size_t tidx = 0; tidx < TILE_COUNT; tidx++)
         ti, tj // local idx
         ai, aj, bi, bj // global to local idx
         ta[ti, tj] = a[ai, aj]
         tb[ti, tj] = b[ai, aj]

         for (size_t tk = 0; tk < TILE_SIZE; tk++)
                c[i][j] += ta[ti][tk] * tb[tk][tj];
     }
```

タイル化された乗算の実装により、データフローは再配布されます。`matrixMultiply4` カーネルの解析 (次のメモリー階層ダイアグラムを参照) から、次のことが分かります。

共有ローカルメモリー (SLM) のタイル化されたカーネルデータ転送



- GTI インターフェイスで LLC から得られたデータは ~2GB で、その多くは L3/SLM から得られています。
- L3 Bandwidth (L3 帯域幅) メトリック (上の表で強調) は、155GB/秒に達しています。これは最大 L3 帯域幅の 70% 以上です。
- EU の約 43% はまだストールしています。

これらの結果から、高速なキャッシュメモリーを使用しているにもかかわらず、アルゴリズム実行は**まだメモリー依存**であることが分かります。この実装により、カーネルの実行速度はナイーブ実装の約 5 倍になりました。

次に、計算タスクの合計時間を調べます。

タイル化されたカーネルのタイミング

Grouping: Computing Task

Computing Task	Work Size		Computing Task				
	Global	Local	Total Time ▼	Average Time	Instance Count	SIMD Width	SVM Usag...
matrixMultiply4<float, (unsig	2048 x 2048	16 x 16	490.727ms	490.727ms	1	16	
clEnqueueReadBufferRect			3.116ms	3.116ms	1		
[Outside any task]			0ms	0ms	0		

高速な実装のための方法はいくつかあります。

- **最適な方法で高レベルのデータアクセスを整理します。**

  データ分布にサブグループを使用すると、独自のローカルメモリーにアクセスする GPU のサブスライスを活用できます。

- **特定の GPU アーキテクチャー向けの低レベルの最適化を使用し、インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) などの最適化ライブラリーを使用します。**

  これらのステップは、GPU で最大のパフォーマンスを達成するのに役立ちます。しかし、GPU には、いくつかの既知の特性を使用して計算できる、パフォーマンスの理論上の制限があります。

  例えば、Gen9 GPU でのアルゴリズム実行の理論上の最小時間を計算してみましょう。Gen9 GT2 GPU アーキテクチャーのパラメーターから、この GPU には 24 の EU が含まれていることが分かります。各 EU には 2 つの FPU (SIMD-4) があります。各 FPU は 2 つの演算 (乗算+加算) を実行できます。最大コア周波数は 1.20GHz で、最大 FP パフォーマンスは次のとおりです。

  24 * 2 * 4 * 2 = 384 FLOP/サイクル (32 ビット浮動小数点)

  384 * 1.2 = 460.8 GFLOPS

  ナイーブ行列乗算実装の FP 演算の数は $2*N^3$ (N=2048 の場合、約 17.2 GOPS) です。

  理論的には、データアクセスの非効率性および帯域幅の制約による制限がなかった場合、アルゴリズムは 17.2 / 460.8 = 0.037 秒 (37 ミリ秒) で計算できます。インテル® VTune™ プロファイラーの結果では、カーネルで実行された最良の時間は 490 ミリ秒であり、理論上の計算時間よりも 10 倍以上遅くなっています。つまり、パフォーマンスを向上する余地はまだあります。

スケーリング・パフォーマンス

行列乗算サンプルのような高度に並列化されたアプリケーションは、GPU リソースを使用することにより効率が向上します。スケーリングがメモリーのボトルネックによって制限されない限り、追加の計算リソースを使用するとパフォーマンスも向上するはずです。

Gen9 シリーズの GPU には、48 の EU を含む GT3 と 72 の EU を含む GT4 があります。しかし、組込み型の GPU には基本的な領域の制限があり、スケーリングを高めるために EU を追加したり、データアクセスを高速化するためにキャッシュブロックを大きくしたりすることはできません。ディスクリート GPU では、領域や電力の制約による制限は少なくなります。システムで 1 つまたは複数の GPU との統合が許可されている場合、アクセラレーターのパフォーマンスをスケールアップできます。

しかし、メイン CPU、メモリー、および GPU の間には、通信インターフェイス (PCIe* バスなど) が存在することに注意してください。通信インターフェイスには、帯域幅、レイテンシー、およびデータの一貫性に独自の制約がある場合があります。

PCIe* ディスクリート・グラフィックス・カード (開発コード名 DG1) と呼ばれていた、インテル® Iris® Xᵉ MAX GPU を見てみましょう。

インテル® Iris® Xᵉ MAX マイクロアーキテクチャーの高レベルのビュー

同じタイル化されたカーネル実装の解析結果は次のとおりです。

タイル化された matrixMultiply カーネルの GPU ホットスポット解析の結果

Computing Task	Work Size		Computing Task			
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width
matrixMultiply4<	2048 x 2048	16 x 16	111.455ms	111.455ms	1	16
clEnqueueRead			159.791ms	159.791ms	1	
[Outside any tas			0ms			

カーネル実行は約 4 倍高速化されました。

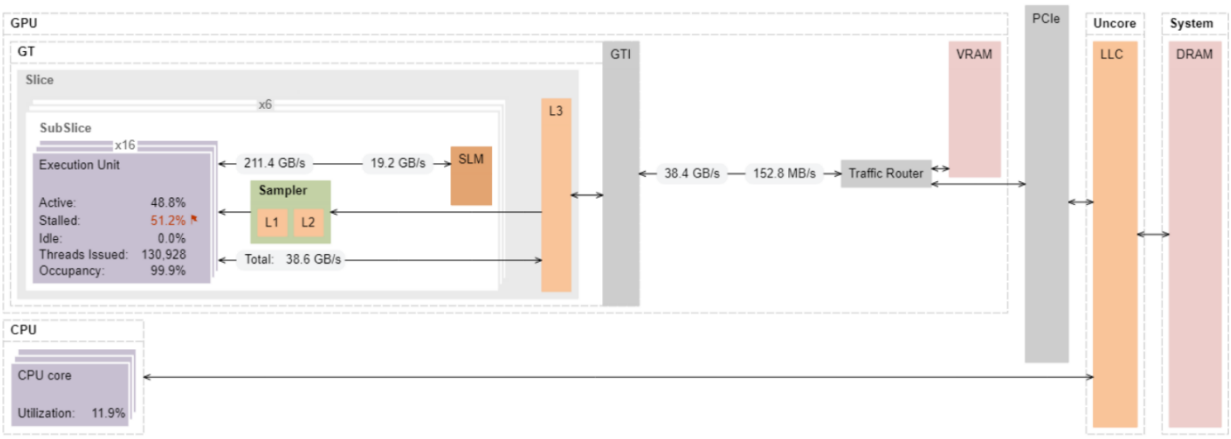
Gen9 GPU の 24 の EU に対してインテル® Iris® Xᵉ MAX GPU には 96 の EU が含まれていることから、これは予想どおりの結果と言えます。しかし、次の表を見ると、EU が実行中に 51% の時間でまだストールしていることが分かります。これは、(一般的な行列アルゴリズムでよく知られている) メモリーからのデータの待機が原因であると考えられます。具体的な原因を調べてみましょう。

タイル化された `matrixMultiply4` カーネルの EU Array (EU アレイ) メトリック

Computing Task	EU Array			EU Threads Occupancy	Computing Threads Started	L3 Bandwidth, GB/sec	Shared Local Memory Bandwidth, GB/sec		GPU Memory Bandwidth, GB/sec	
	Active	Stalled	Idle				Read	Write	Read	Write
matrixMultiply4<float,	48.8%	51.2%	0.0%	99.9%	130,928	6.5%	17.8%	19.220	19.4%	0.153
clEnqueueReadBuffer	0.3%	99.6%	0.0%	94.7%	261,730	0.2%	0.0%	0.033	0.2%	0.419
[Outside any task]	0.0%	2.7%	97.3%	2.5%	49,710	0.0%	0.0%	0.000	0.0%	0.174

結果グリッドでモードを切り替えて最大帯域幅の割合を表示すると、L3 と GPU メモリーの帯域幅は最大値からかなり離れているため、ボトルネックではないことが分かります。データ転送の全体像を把握するため、メモリー階層ダイアグラムを見てみましょう。

データ転送メトリックを含むメモリー階層ダイアグラム



データは、GTI インターフェイスを経由して VRAM やメイン DRAM から取得されます。CPU 側で行列データを準備すると、行列 *a* と行列 *b* のデータが PCIe* 経由で GTI に転送されます。測定された GTI の帯域幅は、PCIe* インターフェイスに必要なデータ転送速度のおおよその指標です。測定されたデータ読み取り速度は GTI インターフェイスで 38GB/秒ですが、PCIe* 3.0x16 の理論上の最大値は片方向で 16GB/秒です。つまり、PCIe* の帯域幅に制限されていることが分かります。インテル® VTune™ プロファイラーを使用して PCIe* のデータ・トラフィックを測定するには、PCIe* パフォーマンス・カウンターを備えたサーバー・プラットフォームが必要です。

サーバーベースのセットアップでは、PCIe* の帯域幅はバスの制限よりもはるかに低くなります。次のように結論付けることができます。

- すべてのデータは VRAM と EU ストールからフェッチされています。これは、ビデオメモリーから EU へのデータ移動のレイテンシーで定義されます。
- EU と L3 間のデータ・トラフィックは GTI と外部トラフィック・ルーター間のデータ・トラフィックと同じであるため、L3 キャッシュを適切に再利用することにより、パフォーマンスをさらに最適化できます。例えば、各 GPU スライスの L3 キャッシュに収まるブロックサイズで、セカンドレベルの行列タイリングを行います。

まとめ

一般に、ヘテロジニアス・アプリケーションでは、特定のワークロードをアクセラレーターにオフロードする場合、GPU などの超並列アクセラレーター・マシンに十分な計算タスクを提供することが不可欠です。

- オフロードしたタスクのデータ転送とタスク・スケジューリングのオーバーヘッドを予測することにより、GPU の効率を向上します。
- インテル® VTune™ プロファイラーの GPU オフロード解析の **[GPU Utilization (GPU 利用率)]** メトリックおよび **[GPU Occupancy (GPU 占有率)]** メトリックを使用して、GPU 使用の非効率性を予測します。
- 計算タスク実行のパフォーマンスは、実行ユニットの不足、メモリー・サブシステムまたはインターフェイスのボトルネックの存在など、いくつかのマイクロアーキテクチャーの要因によって制限される場合があります。**GPU 計算/メディア・ホットスポット解析**を実行して、これらの制限を特定します。GPU メモリー階層ダイアグラムのボトルネックと、すべての計算タスクの詳細なマイクロアーキテクチャーのメトリックをハイライトします。複雑なカーネルの場合、レイテンシー解析を使用して、カーネル内の最も重要なコードを特定します。

関連情報

DPDK アプリケーションのコア使用率
インテル® Advisor ユーザー向けオフロードのモデル化のリソース (英語)
インテル® VTune™ プロファイラー・ユーザーガイド日本語版
DPC++ アプリケーションのプロファイル
インテル® VTune™ プロファイラーを使用したインテル® GPU 向けアプリケーションの最適化
コマンドライン・インターフェイスを使用した GPU で実行している DPC++ アプリケーションのパフォーマンスの解析
インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL)

DPDK アプリケーションのコア使用率

このレシピは、DPDK ベースのアプリケーションにおけるパケット受信のコア使用率を特徴付けるメトリックを調査します。

コンテンツ・エキスパート: Ilia Kurakin (英語)、Roman Khatko (英語)

高速なパケット処理が求められるデータ・プレーン・アプリケーションでは、DPDK は特定の論理コアにピニングされた無限ループでパケットを受信するため特定のポートをポーリングします。このようなパケット受信ポーリングモデルは、有効なコア使用率を測定する上で課題となります。ポーリングループを実行するコアの CPU 時間は、DPDK がアイドルのループサイクル数に関係なく、常に 100% 近くになります。そのため、CPU 時間からパケット受信のコア使用率は分かりません。しかし、このポーリングモデルでは、**[Rx Spin Time - % of wasted polling loop cycles (Rx スピン時間 - 無駄なポーリング・ループ・サイクルの %)]** からコア使用率が分かります。Wasted Cycles (無駄なサイクル) とは、DPDK がパケットを受信しなかった反復を指します。

このレシピは、次のステップに従って、DPDK ベースのワークロードでパケット受信の効率を解析します。

1. 使用するもの
2. 手順:
    a. 入力と出力解析を実行する
    b. DPDK Rx スピン時間メトリックを使用してコア使用率を解析する
    c. DPDK Rx バッチ統計ヒストグラムを使用してパケット受信を解析する
    d. Rx 操作を理解して Rx ピークを調査する

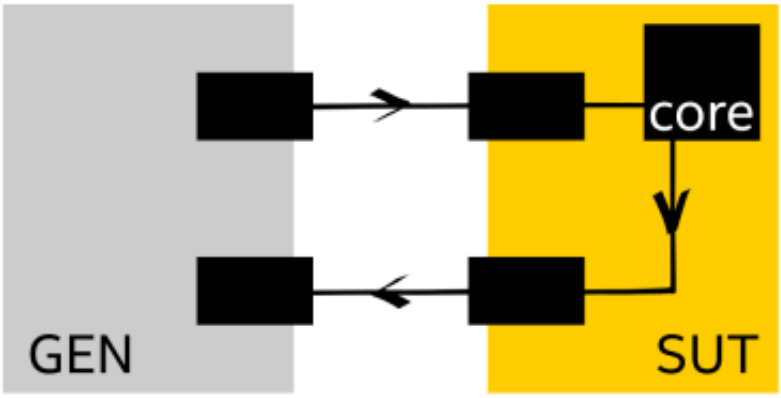
使用するもの

- **アプリケーション**: シングルコアで L2 フォワーディングを実行する DPDK `testpmd` アプリケーション。インテル® VTune™ Amplifier のプロファイル・サポートが有効な DPDK でコンパイルされています。
- **ツール:**
    - **インテル® VTune™ Amplifier のプロファイル・サポートが有効な DPDK**。インテル® VTune™ Amplifier 18.11 以降では、プロファイル・サポートが DPDK に統合されます。それよりも古いバージョンを使用する場合、パッチ (17.11、18.02、および 18.05 で利用可能) を適用してください。DPDK でプロファイルを有効にするには、インテル® VTune™ Amplifier が DPDK ポーリングサイクルにアタッチするように、(`config/common_base config` ファイルで) `CONFIG_RTE_ETHDEV_RXTX_CALLBACKS` フラグと `CONFIG_RTE_ETHDEV_PROFILE_WITH_VTUNE` フラグを有効にして、DPDK (とターゲット・アプリケーション) を再構成し再コンパイルします。
    - **インテル® VTune™ Amplifier 2019**: 入力と出力解析

注

- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。

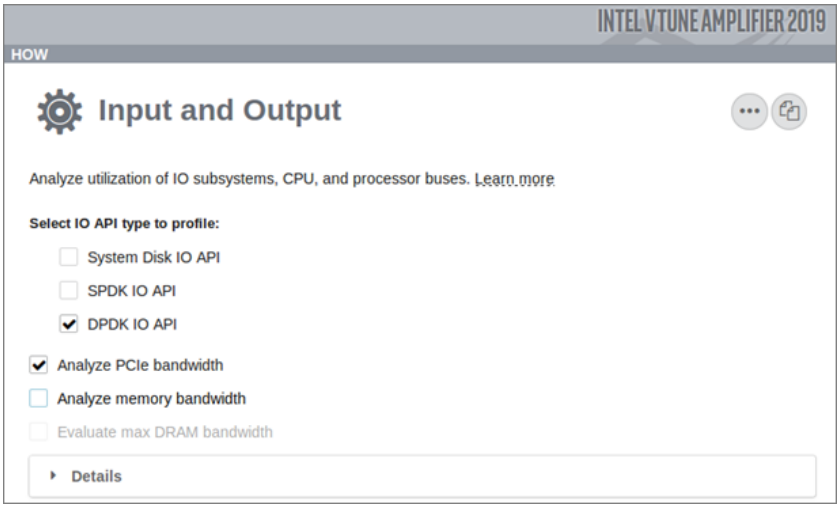
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム**: 40GbE リンクを介して接続された、64 バイトのフレームを生成するトラフィック・ジェネレーター (以下の図では GEN) とパケットレシーバー (SUT: System Under Test) で構成されたテストシステム。SUT はパケットの L2 フォワーディングを実行します。



- **CPU**: インテル® Xeon® Platinum 8180 プロセッサー (38.5MB キャッシュ、2.50GHz、28 コア)

入力と出力解析を実行する

DPDK 解析では、インテル® VTune™ Amplifier GUI で入力と出力解析を選択し、**[DPDK IO API]** を有効にします。



DPDK Rx Spin Time (DPDK Rx スピン時間) などの API 固有メトリックとハードウェア・イベントやハードウェア・イベントベース・メトリックを関連付けることができます。例えば、DPDK Rx スピン時間と **[Analyze PCIe bandwidth (PCIe* 帯域幅を解析)]** が有効な場合に収集される PCIe* 帯域幅の間には依存関係があります。

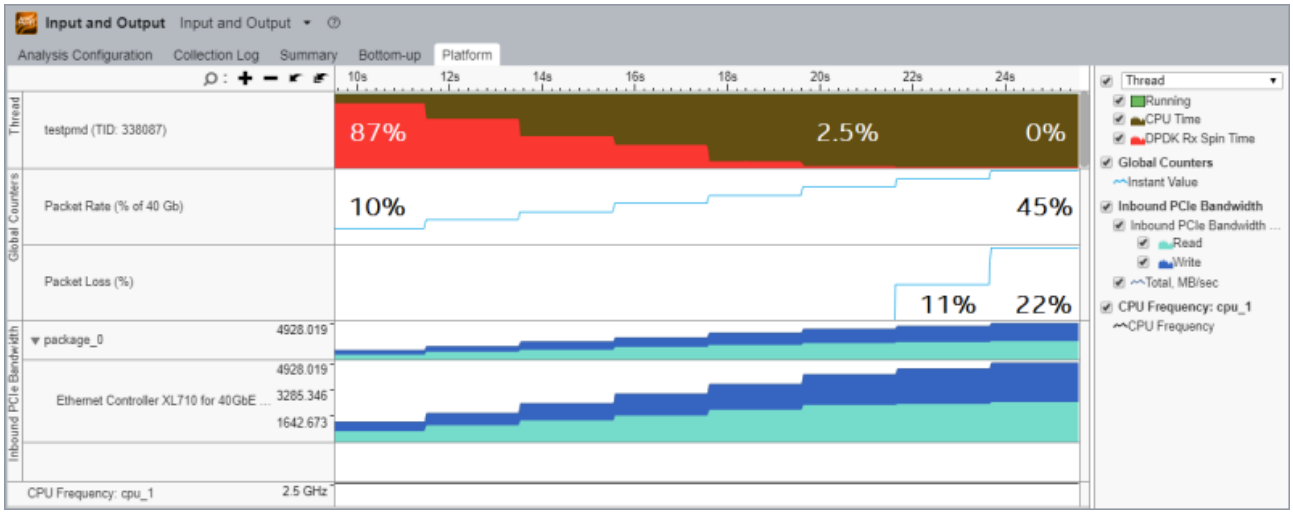
コマンドラインから入力と出力解析を実行して PCIe* 帯域幅と DPDK メトリックを取得するには、解読可能な名前でデバイスごとの PCIe* 帯域幅を取得できるように、root 権限で次のコマンドを実行します。

```
amplxe-cl -collect io -knob kernel-stack=false -knob dpdk=true -knob collect-pcie-bandwidth=true -knob collect-memory-bandwidth=false -knob dram-bandwidth-limits=false --target-process=testpmd
```

DPDK Rx スピン時間メトリックを使用してコア使用率を解析する

データが収集されたら、**[Platform (プラットフォーム)]** タブから始め、スレッドの **[DPDK Rx Spin Time (DPDK Rx スピン時間)]** オーバータイム・メトリックを調査します。このメトリックは、ゼロパケット返す `rte_eth_rx_burst(...)` 関数呼び出しの割合を (スレッドごとに) 示します。これは、パケットを提供しないポーリングループ反復の割合と同じです。

$$DPDK\ Rx\ Spin\ Time = \frac{num\ iterations\ that\ give\ 0\ packets}{total\ num\ iterations}$$



注

ここで紹介した結果は合成されたものです。

上記の **[Platform (プラットフォーム)]** ビューでは、ポーリングスレッドの **[CPU Time (CPU 時間)]** (茶色) は常に 100% 近くです。**[DPDK Rx Spin Time (DPDK Rx スピン時間)]** (赤色) は、パケット受信のスレッド使用率です。マウスでグラフをポイントすると、その時点の値がツールヒントに表示されます。

この例では、トラフィック・ジェネレーターを自動化して、2 秒ごとに 40Gbps の 5% ずつトラフィック・レートを増やしてパケット損失データを収集しました。適切にフォーマットされた `*.csv` ファイル形式のオーバータイム・データは、インテル® VTune™ Amplifier プロジェクトにインポート (英語) してタイムラインに表示できます。

デフォルトでは、インテル® VTune™ Amplifier は上記の **[Global Counters (グローバルカウンター)]** セクションに表示されている **[Packet Rate (パケットレート)]** メトリックと **[Packet Loss (パケット損失)]** メトリックを収集できません。このレシピでは、これらのメトリックは別途収集され、インテル® VTune™ Amplifier によって収集された結果に手動でインポートされました。別の方法として、インテル® VTune™ Amplifier のカ

スタムコレクター (英語) 機能を使用して追加のメトリックを含む `csv` ファイルをインポートできます。カスタムコレクターは、収集の開始/停止/一時停止時にインテル® VTune™ Amplifier によって実行される追加のプロセスです。カスタムコレクターを使用して、すべてのシステム自動化を実装し、追加のメトリックを収集できます。これにより、実験が再現可能となり、結果を比較できるようになります。これは、パフォーマンス・チューニングに役立ちます。

[Platform (プラットフォーム)] ビューの下部では、タイムラインで **[Inbound PCIe Bandwidth (インバウンド PCIe* 帯域幅)]** の変化を確認できます。この解析はインテル® マイクロアーキテクチャー (開発コード名 Skylake) 上で root 権限で実行されたため、PCIe* 帯域幅が PCIe* デバイス別に人間が解読できる名前で表示されています。

上記の入力と出力解析の **[Platform (プラットフォーム)]** ビューでは、すべてのメトリックに相関性があります。トラフィック生成レートが上昇すると、**[Inbound PCIe Bandwidth (インバウンド PCIe* 帯域幅)]** は増加し、**[DPDK Rx Spin Time (DPDK Rx スピン時間)]** は減少します。ある時点で、テストシステムはオーバーロードとなり、非ゼロの **[Packet Loss (パケット損失)]** 値が見られるようになります。

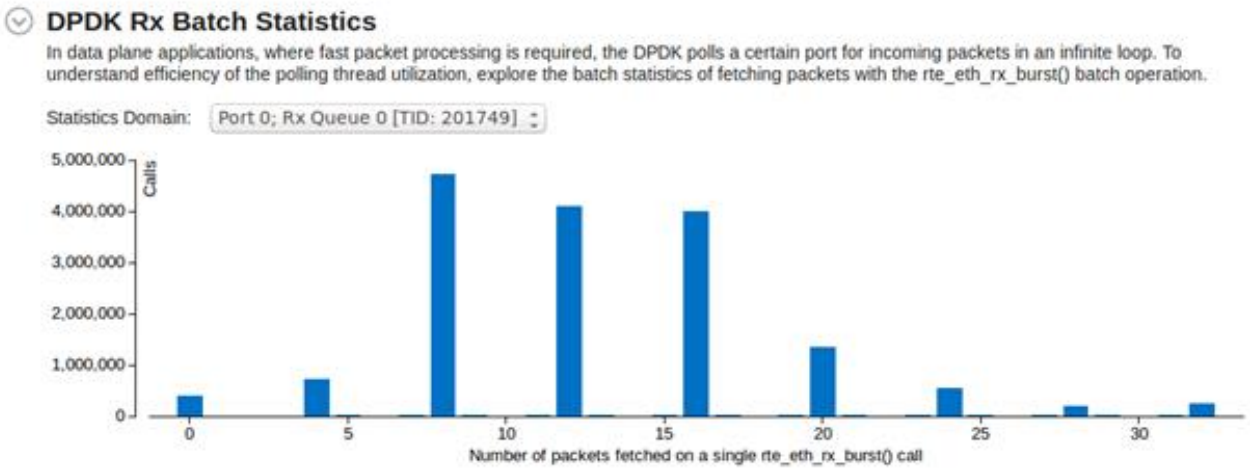
注

スレッドが複数の Rx キューを処理する場合、**[DPDK Rx Spin Time (DPDK Rx スピン時間)]** メトリックは複合統計を表します。

DPDK Rx バッチ統計ヒストグラムを使用してパケット受信を解析する

DPDK は、`rte_eth_rx_burst(...)` 関数を使用して NIC からパケットのバッチを受け取ります。区間 (0, `MAX_NB_PKTS`) の任意の数のパケットを受信できます。ここで、`MAX_NB_PKTS` は定数値 (通常 32) です。したがって、固定の **[Rx Spin Time (Rx スピン時間)]** では、コアのトラフィック処理量が大きく異なる可能性があるため、**[Rx Spin Time (Rx スピン時間)]** は全体像を表していません。

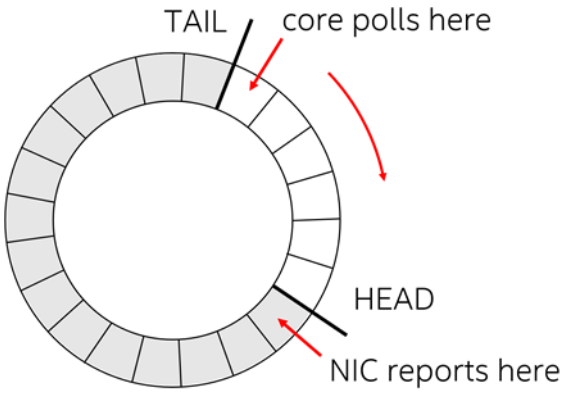
パケット受信のサマリー統計を表示し、Rx のコア使用率を完全に把握するには、**[Summary (サマリー)]** タブに切り替えて、**[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** ヒストグラムを調査します。



ヒストグラムは、選択した **[Port / Rx Queue / TID (ポート/Rx キュー/TID)]** グループの受信バッチパケットに関する統計を表します。この例では、すべてのピークが 4 の倍数の値を示しています。これは偶然ではなく、根本的な原因を調査するにはパケット受信の背景を理解する必要があります。

Rx 操作を理解して Rx ピークを調査する

パケットを受信するため、実行コアは Rx 記述子を介して NIC と通信します。Rx 記述子は、アドレスやサイズなどパケットに関する情報を保持するデータ構造で、Rx キューと呼ばれるリングバッファーに結合されます。簡単に言えば、パケット受信はリングバッファー内のレースであり、NIC はリングバッファーの **[Head (先頭)]** に Rx 記述子を追加し、実行コアは **[Tail (末尾)]** から Rx 記述子をポーリング、処理、そして解放します。



コアは Rx 記述子を解放すると、Tail ポインターを前方に移動します。Tail が Head に到達すると、`rte_eth_rx_burst()` は 0 パケットを返します。逆に、Head が Tail に到達すると、Rx キューに利用可能な Rx 記述子がなく、パケット損失が発生する可能性があります。

新しいパケットを提供するため、NIC は Rx キューの Head にある Rx 記述子を読み取り、記述子のコアで指定されたメモリーアドレスにパケットを転送します。そして、Rx 記述子を書き戻して、コアに新しいパケットの到着を通知する必要があります。

このレシピのセットアップに使用したインテル® イーサネット・コントローラー XL710 シリーズは、16 バイトと 32 バイトの Rx 記述子をサポートします。どちらもキャッシュラインのサイズよりも小さいため、NIC は PCIe* 帯域幅を抑えるため整数のキャッシュラインへ Rx 記述子をパックして書き込みをまとめる、記述子の書き戻しポリシーを採用しています。主に、インテル® イーサネット・コントローラー XL710 シリーズは、次の条件を満たす場合、完了した Rx 記述子を書き戻します。

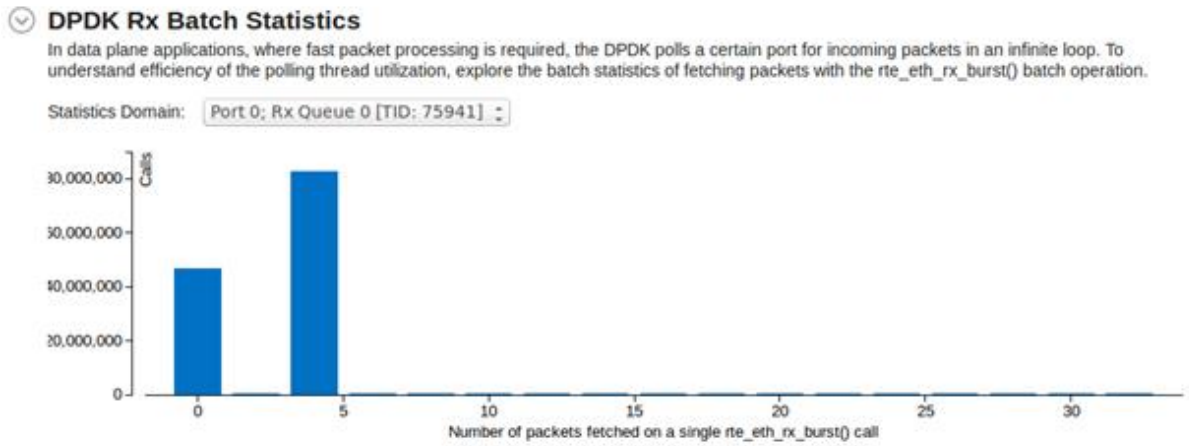
- **4 x 32 バイト** の記述子または **8 x 16 バイト**の記述子が完了した場合
- 内部 NIC キャッシュで記述子が無効にされた場合

詳細は、インテル® イーサネット・コントローラー X710/XXV710/XL710 シリーズのデータシート (英語) を参照してください。

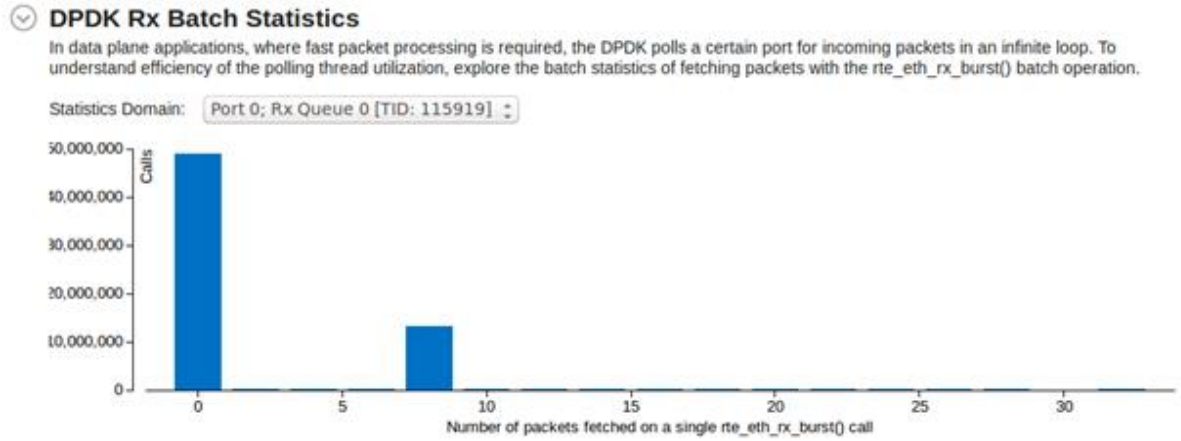
このレシピでは、システムが 32 バイトの Rx 記述子を使用しているため、**[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** のほとんどのピークは 4 の倍数になっています。

DPDK では Rx 記述子のサイズを切り替えることができます。以下は、`testpmd` を 32 バイトと 16 バイトの Rx 記述子を使用して中程度の負荷で実行した場合の **[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** の変化です。

- 32 バイトの Rx 記述子: ほとんどの `rte_eth_rx_burst()` 呼び出しは 4 パケットを受け取ります。



- 16 バイトの Rx 記述子: ほとんどの `rte_eth_rx_burst()` 呼び出しは 8 パケットを受け取ります。



関連情報

- DPDK アプリケーションの PCIe*トラフィック
- カスタムコレクターの使用 (英語)
- 外部データを含む CSV ファイルの作成 (英語)
- 外部データのインポート (英語)

DPDK アプリケーションの PCIe*トラフィック

インテル® VTune™ Amplifier の PCIe* 帯域幅メトリックを使用して、パケット・フォワーディングを行う DPDK ベースのアプリケーションの PCIe* トラフィックを調査します。

コンテンツ・エキスパート: Ilia Kurakin (英語)、Roman Khatko (英語)

10/40 GbE NIC を搭載したシステムで実行するデータ・プレーン・アプリケーションは通常、プラットフォーム I/O 機能を多く利用します。特に、CPU とネットワーク・インターフェイス・カード (NIC) 間のインターフェイスである PCIe* リンクの帯域幅を集中的に消費します。このようなワークロードでは、パケット転送と通信制御のバランスを保つことにより、PCIe* リンクを効率的に利用することが重要です。PCIe* 転送を理解することは、パフォーマンス問題の特定と解決に役立ちます。

DPDK ベースのワークロードにおける PCIe* パフォーマンス解析の方法論の詳細は、https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf (英語) を参照してください。

このレシピでは、DPDK によるパケット・フォワーディングの段階と PCIe* 帯域幅消費の理論的な推定値を調べた後、理論的な推定値とインテル® VTune™ Amplifier で収集したデータを比較します。

使用するもの

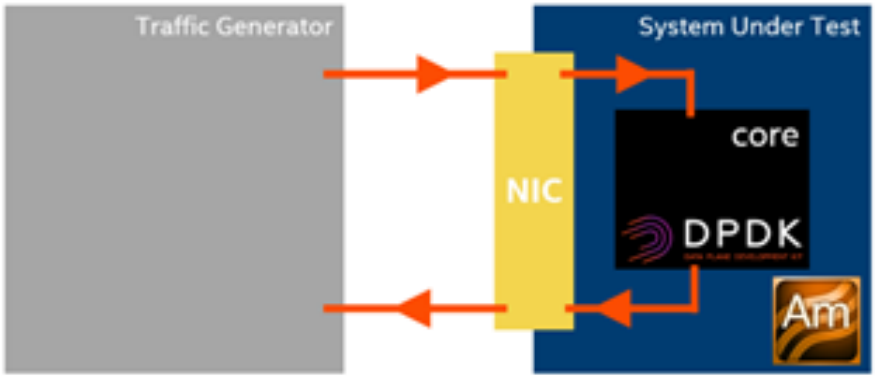
以下は、このレシピで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション**: シングルコアで L2 フォワーディングを実行する DPDK `testpmd` アプリケーション。インテル® VTune™ Amplifier のプロファイル・サポートが有効な DPDK でコンパイルされています。
- **パフォーマンス解析ツール:**
    - インテル® VTune™ Amplifier 2019 Update 3: 入力と出力解析

注

- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。

    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **システムの設定**: トラフィック・ジェネレーターおよび `testpmd` アプリケーションがパケット・フォワーディングを実行し、インテル® VTune™ Amplifier がパフォーマンス・データを収集する被試験システム (SUT)。



- **CPU**: インテル® Xeon® Platinum 8180 プロセッサー (38.5MB キャッシュ、2.50GHz、28 コア)

インバウンド/アウトバウンド PCIe* 帯域幅メトリックを理解する

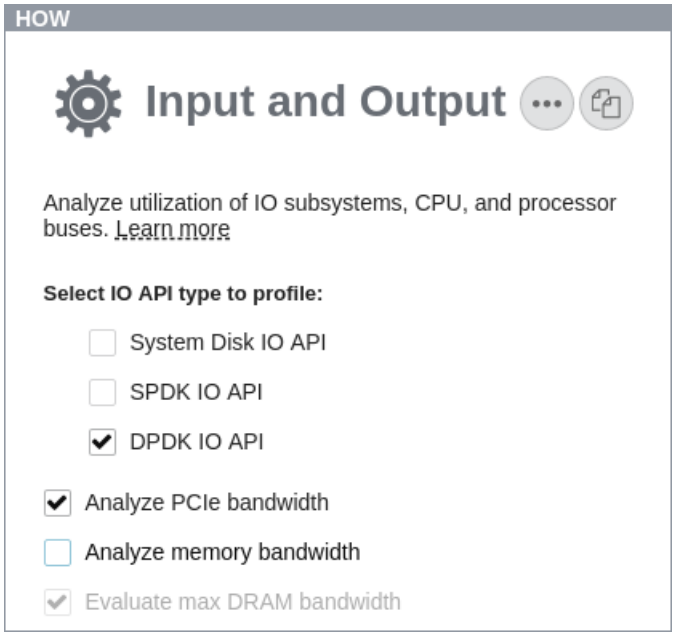
PCIe* 転送は PCIe* デバイス (NIC など) と CPU の両方により開始されます。そのため、インテル® VTune™ Amplifier は、次の帯域幅タイプで、PCIe* 帯域幅メトリックを識別します。

- **[Inbound PCIe Bandwidth (インバウンド PCIe* 帯域幅)]**: システムメモリーをターゲットとするデバイスのトランザクションが消費する帯域幅を示します。
    - **[Read (リード)]**: メモリーからデバイスへの読み取りを示します。
    - **[Write (ライト)]**: デバイスからメモリーへの書き込みを示します。
- **[Outbound PCIe Bandwidth (アウトバウンド PCIe* 帯域幅)]**: デバイスの MMIO 空間をターゲットとする CPU のトランザクションが消費する帯域幅を示します。
    - **[Read (リード)]**: デバイスの MMIO 空間から CPU への読み取りを示します。
    - **[Write (ライト)]**: CPU からデバイスの MMIO 空間への書き込みを示します。

入力と出力解析を設定して実行する

[Inbound PCIe Bandwidth (インバウンド PCIe* 帯域幅)] および **[Outbound PCIe Bandwidth (アウトバウンド PCIe* 帯域幅)]** データを収集するには、入力と出力解析を使用します。

GUI で、プロジェクトを作成し、**[HOW (どのように)]** ペインで **[Input and Output (入力と出力)]** 解析を選択して、**[Analyze PCIe bandwidth (PCIe* 帯域幅を解析)]** オプションを有効にします。



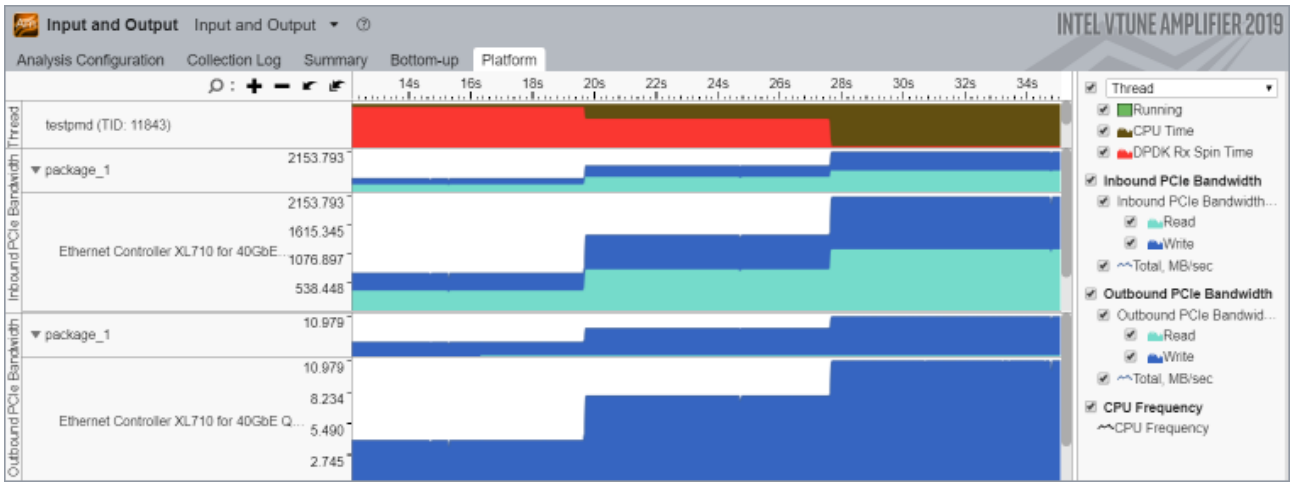
コマンドラインで、`collect-pcie-bandwidth` knob (デフォルトで `true` に設定) を使用します。例えば、次のコマンドは DPDK メトリックを使用して PCIe* 帯域幅データの収集を開始します。

```
amplxe-cl -collect io -knob kernel-stack=false -knob dpdk=true -knob collect-memory-bandwidth=false --target-process my_process
```

結果が収集されたら、GUI で **[Platform (プラットフォーム)]** タブに移動し、[Inbound PCIe Bandwidth (インバウンド PCIe* 帯域幅)] および [Outbound PCIe Bandwidth (アウトバウンド PCIe* 帯域幅)] セクションに注目します。

注

インテル® マイクロアーキテクチャー開発コード名 Skylake 以降のサーバー・プラットフォームでは、デバイスごとに PCIe* 帯域幅メトリックを収集できます。root 権限が必要です。



パケット・フォワーディングに必要な PCIe* 転送を理解する

DPDK によるパケット・フォワーディングは、パケットを受信 (`rx_burst` DPDK ルーチン) した後に、パケットを送信 (`tx_burst`) します。「DPDK アプリケーションのコア使用率」レシピの、Rx 記述子を含む Rx キューを利用したパケット受信の詳細が参考になります。DPDK によるパケット送信はパケット受信と同じように動作します。パケットを受信するため、実行コアは Tx 記述子 (パケットアドレス、サイズ、その他の制御情報を格納する 16 バイトのデータ構造) を使用します。Tx 記述子のバッファーは連続するメモリーのコアにより割り当てられ、Tx キューと呼ばれます。Tx キューはリングバッファーとして処理され、長さ、Head、Tail で定義されます。Tx キューからのパケット送信はパケット受信に非常に似ています。コアは Tx キューの Tail で新しい Tx 記述子を準備し、NIC は Head から処理します。

Rx キューと Tx キューの Tail ポインターは、新しい記述子が利用可能であることを通知するため、ソフトウェアにより更新されます。Tail ポインターは MMIO 空間にマップされる NIC レジスターに格納されます。つまり、Tail ポインターはアウトバウンド書き込み (MMIO 書き込み) で更新されます。MMIO アドレス空間はキャッシュできないため、アウトバウンド書き込みとアウトバウンド読み取りには非常にコストがかかります。そのため、これらのトランザクションは最小限にするべきです。

パケット・フォワーディングを行う場合、PCIe* トランザクションのワークフローは次のようになります。

1. コアが Rx キューを準備して、Rx キューの Tail のポーリングを開始します。
2. NIC が Rx キューの Head の Rx 記述子を読み取ります **(インバウンド・リード)**。
3. NIC が Rx 記述子で指定されたアドレスにパケットを送ります **(インバウンド・ライト)**。
4. NIC が Rx 記述子を書き戻して、コアに新しいパケットが到着したことを通知します **(インバウンド・ライト)**。
5. コアがパケットを処理します。
6. コアが Rx 記述子を解放して、Rx キューの Tail ポインターを移動します **(アウトバウンド・ライト)**。
7. コアが Tx キューの Tail の Tx 記述子を更新します。
8. コアが Tx キューの Tail ポインターを移動します **(アウトバウンド・ライト)**。
9. NIC が Tx 記述子を読み取ります **(インバウンド・リード)**。
10. NIC がパケットを読み取ります **(インバウンド・リード)**。
11. NIC が Tx 記述子を書き戻して、コアにパケットが送信され Tx 記述子を解放できることを通知します **(インバウンド・ライト)**。

PCIe* トラフィックの最適化を理解する

最大パケットレートを増やしてレイテンシーを軽減するため、DPDK は次の最適化を使用します。

- **アウトバウンド・リードを行わない。**Rx および Tx キューの Head の位置を把握するためにコストのかかるアウトバウンド・リード (MMIO 読み取り) を行いません。代わりに、NIC は Rx および Tx 記述子を書き戻してソフトウェアに Head の位置が移動したことを通知します。
- **Tx 記述子に関連するインバウンド・ライト帯域幅を減らす。**コアに Tx キューの Head の位置と再利用できる Tx 記述子を通知するには Tx 記述子の書き戻しが必要です。パケット受信では、できるだけ早く Rx 記述子を書き戻してコアに新しく到着したパケットの情報を通知することが重要です。パケット送信では、Tx 記述子を書き戻す必要はありません。コアにパケット送信の成功を定期的に (例えば、32 パケットごとに) 通知すれば、その前のすべてのパケットも正常に送信されたことが伝わります。NIC は Tx 記述子の RS (レポートステータス) ビットがセットされると Tx 記述子を書き戻します。

DPDK 側には、RS ビットしきい値 (英語) があります。この値は、RS ビットがセットされる頻度、つまり NIC が TX 記述子を書き戻す頻度を定義します。この最適化は、Tx 記述子に関連するインバウンド・ライトを減らします。

- **アウトバウンド・ライトを平均化する。**DPDK はパケット受信と送信をバッチで行い、アプリケーションはパケットのバッチが処理された後に Tail ポインターを更新します。`rx_burst` の一部の実装は、Rx 解放しきい値 (英語) を使用しています。このしきい値を使用すると、アプリケーションが Rx キューの Tail ポインターを更新する前に処理される Rx 記述子の数を設定できます (このしきい値はバッチサイズよりも大きい場合にのみ有効になることに注意してください)。アウトバウンド書き込みはパケット間で平均化されます。

プラットフォーム・レベルでは、トランザクションはキャッシュラインの粒度で行われるため、ハードウェアは、常に読み取りと書き込みをまとめて、一部のキャッシュラインが転送されないように試みます。

PCIe* 帯域幅消費を推定する

最適化されたパケット・フォワーディングは、次の式を適用して指定されたパケットレートの PCIe* 帯域幅消費を推定できます。

$$Inbound\ Write = Pkt\ Rate \cdot \left(Rx\ Desc\ Size + Pkt\ Size + \frac{Tx\ Desc\ Size}{rs\ bit\ threshold} \right)$$

$$Inbound\ Read = Pkt\ Rate \cdot (Rx\ Desc\ Size + Pkt\ Size + Tx\ Desc\ Size)$$

$$Outbound\ Write = Pkt\ Rate \cdot \left(\frac{Rx\ Tail\ Ptr\ Size}{\max(rx\ free\ threshold, rx\ batch\ size)} + \frac{Tx\ Tail\ Ptr\ Size}{tx\ batch\ size} \right)$$

$$Outbound\ Read = 0$$

上記のアウトバウンド・ライト帯域幅の式は、パケットが同じサイズのバッチで処理された場合にのみ有効です。パケットが複数のサイズのバッチで送信された場合、式の精度は低くなります。

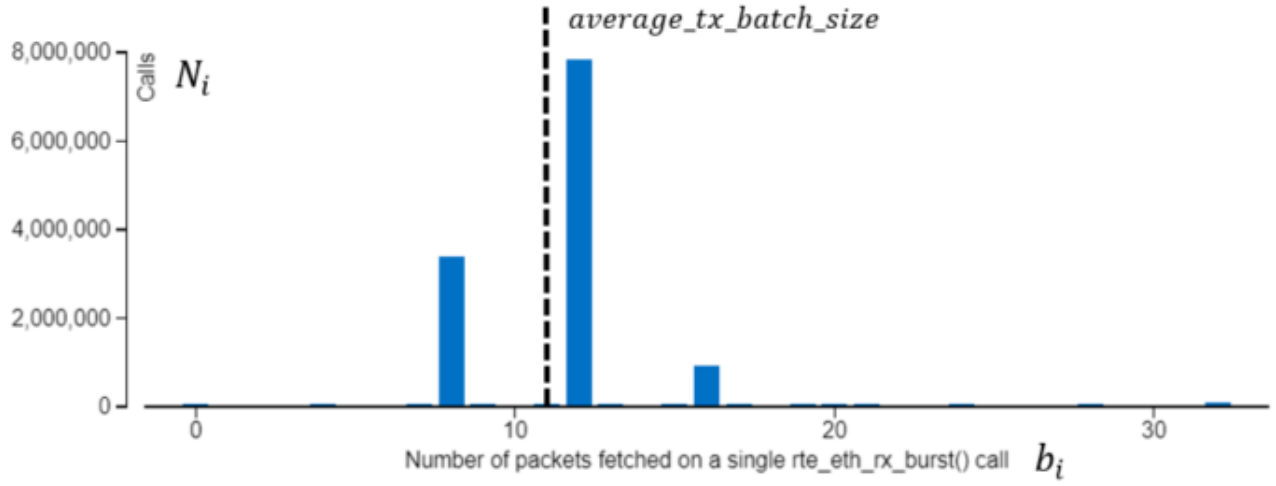
単純なフォワーディングでは、コアは受信したパケットをすべて送信します。`testpmd` は完了までまとめて実行するモデルで設計されたアプリケーションであるため、`tx_burst` は `rx_burst` と同じパケットのバッチで動作すると推測できます。つまり、**[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** (「DPDK アプリケーションのコア使用率」レシピを参照) はパケット受信とパケット送信の両方の統計値を反映します。そのため、**[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** を使用して汎用的なケースのアウトバウンド・ライト帯域幅を推定できます。

Tx バッチサイズの代わりに、「平均」Tx バッチサイズを考えます。

$$average\ tx\ batch\ size = \sum_i b_i \frac{n_i}{n} = \frac{\sum_i b_i^2 N_i}{\sum_i b_i N_i}$$

ここで、b_i はバッチサイズ、N_i は `rx_burst` 呼び出しの回数 (バッチサイズ b_i)、$n_i = b_i N_i$ は **[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** の i 番目のピークのパケット数、$n = \sum_i b_i N_i$ はフォワードされたパケット

の総数です。次の図はこの計算の例です。この例では、バッチ統計にはバッチサイズ 8、10、12 の 3 つのピークがあり、計算された平均バッチサイズは 11 です。

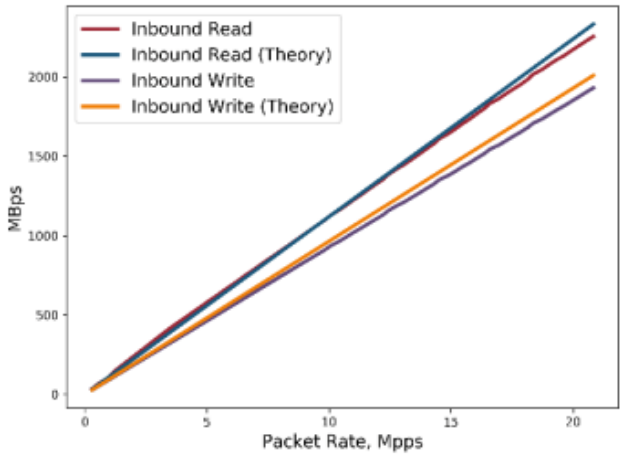


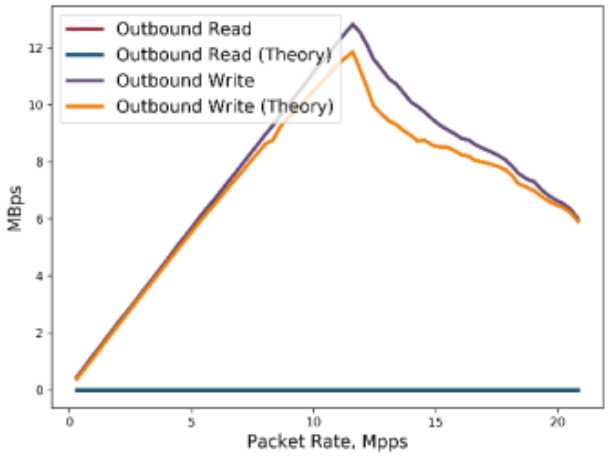
単純にするため、Rx 解放のしきい値がその最大 Rx バッチサイズよりも大きいと考えます。アウトバウンド・ライト帯域幅の最終的な計算は次のようになります。

$$Outbound\ Write = Pkt\ Rate \cdot \left(\frac{Rx\ Tail\ Ptr\ Size}{rx\ free\ threshold} + \frac{Tx\ Tail\ Ptr\ Size}{average\ tx\ batch\ size} \right)$$

推定値と解析結果を比較する

次の 2 つの図は、下記の表にリストされた値で設定された `testpmd` アプリケーションの、PCIe* 帯域幅の理論的な推定値とインテル® VTune™ Amplifier で収集された PCIe* 帯域幅を示しています。

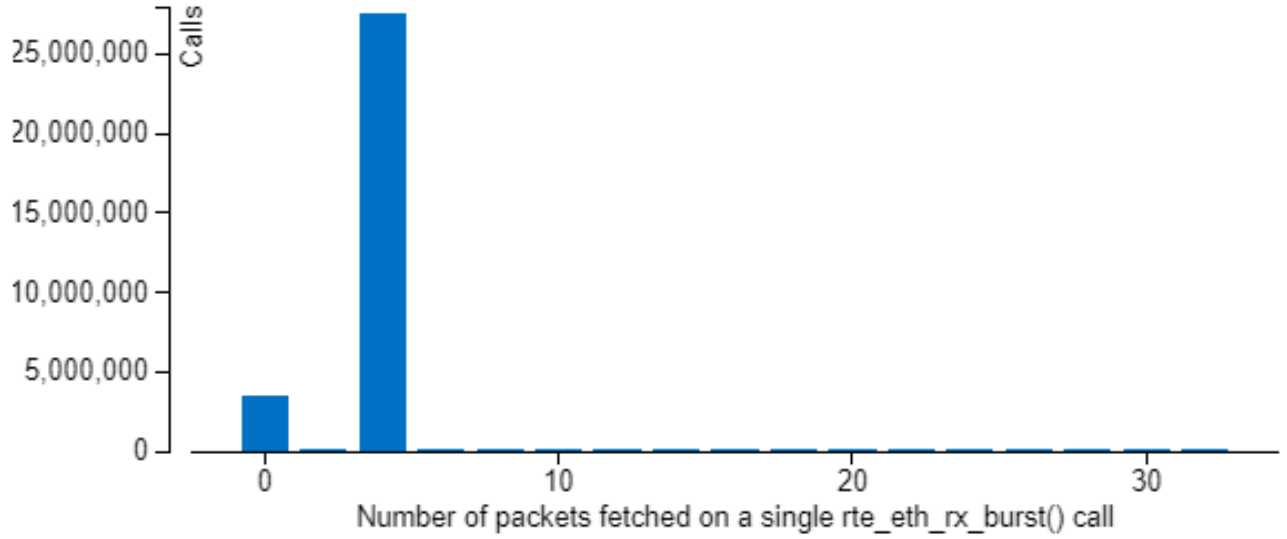




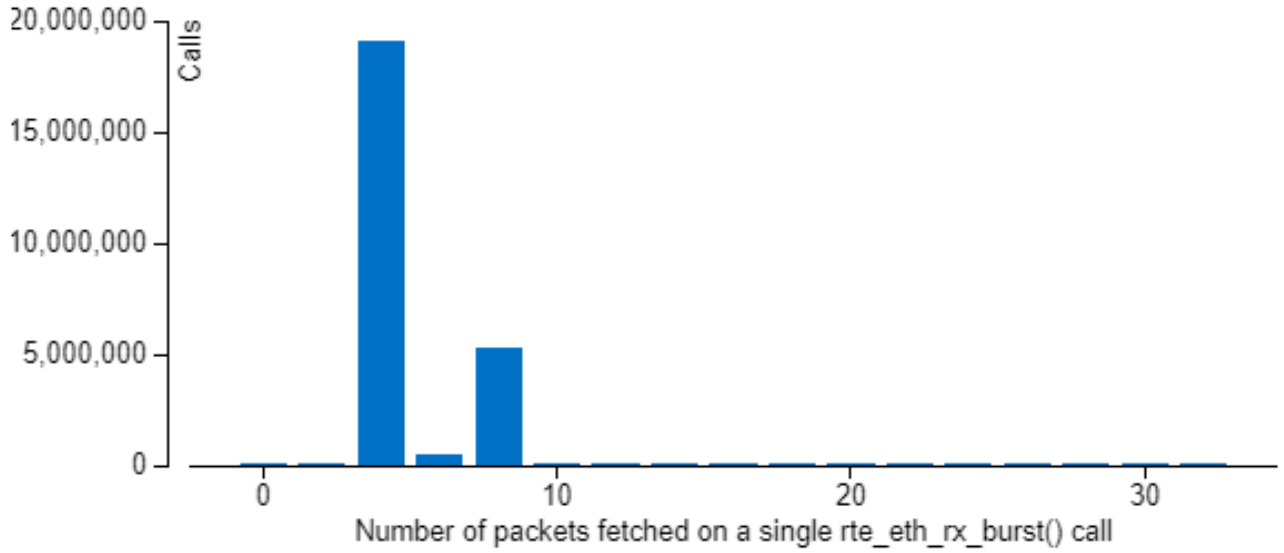
パケットサイズ、B	64
Rx 記述子サイズ、B	32
RS ビットしきい値	32
Rx 解放しきい値	32

アウトバウンド・ライトの値はほぼ中央のポイントから低下しています。この低下は、処理するパケット数の増加による **[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** の変更が原因です。このポイント以前は **[DPDK Rx Batch Statistics (DPDK Rx バッチ統計)]** にはバッチサイズ 0 と 4 の 2 つのピークがあり、このポイント以降はバッチサイズ 8 に新しいピークが現れています (下記の 2 つのヒストグラムを参照)。上記の式に当てはめると、平均バッチサイズは増え、**アウトバウンド・ライト帯域幅**は減ります。

10 Mpps:



13 Mpps:



式で考慮していない要素が原因と思われる多少の違いはありますが、概して、理論的な推定値はインテル® VTune™ Amplifier でレポートされたデータに非常に近くなっています。

このレシピで使用したデータ・プレーン・アプリケーションは、すでに適切に最適化されていました。しかし、実際のアプリケーションで I/O 中心のパフォーマンス解析を行う際は、このレシピを利用することを推奨します。

関連情報

- i40e ドライバーの 16/32 バイト Rx 記述子の切り替え (英語)
- testpmd で利用可能なしきい値およびしきい値の変更方法 (英語)
- https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf (英語)

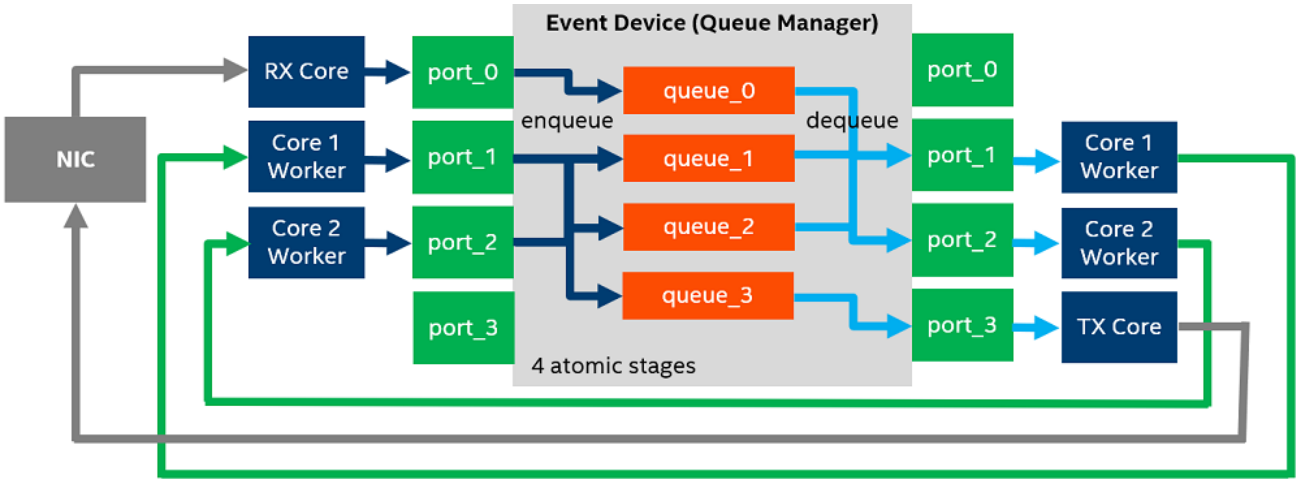
DPDK イベント・デバイス・プロファイル

インテル® VTune™ プロファイラーを使用して、DPDK ベースのアプリケーションの DPDK イベント・デバイス・パイプラインの利用効率を解析して、不均衡な負荷分散やワーカーコアが十分に利用されていない問題を特定します。

コンテンツ・エキスパート: Eugeny Parshutin、Kurakin Ilia

Data Plane Development Kit (DPDK) は、さまざまな CPU アーキテクチャー上で動作するパケット処理ワークロードを高速化する、複数のライブラリーで構成されるフレームワークです。その 1 つが、アプリケーションでイベントベースのモデルを使用することでシステムの負荷分散を向上させる `eventdev` ライブラリーです。イベントベースのアプローチでは、システムが行うべき作業をイベントと呼ばれる個別のユニットで示します。DPDK でイベントベースのプログラミング・モデルを使用する一般的な例として、ネットワーク・パケット処理パイプラインがあり、各パケットがイベントの役割を果たします。

次の図は、`eventdev` パイプラインの構成例です。



ここでは、各ブロックが次のユニットを表しています。

- **Event Device (イベントデバイス)** – ハードウェアまたはソフトウェアで実装されたイベント・スケジュール機能を備えたデバイス。
- **Queue (キュー)** – スケジュール・タイプ (アトミック、順序付き、または並列) に関連付けられた異なるフローのイベントを含む処理パイプラインの論理ステージ。
- **Ports (ポート)** – `eventdev` キューにイベントをエンキュー/デキューする、コアと `eventdev` ライブラリーの接点。
- **Worker Cores (ワーカーコア)** – アプリケーションが作業を実行するのに利用可能な CPU コア。
- **Rx Core (Rx コア)** – NIC からパケットを受信する CPU コア。
- **Tx Core (Tx コア)** – NIC にパケットを送信する CPU コア。
- **NIC** – ネットワーク・インターフェイス・カード。

この例は、4 つのイベントキューを表す 4 つのアトミックステージを管理するイベントデバイスを示しています。

- **queue_0** は新たに受信したパケットを保持するために使用されます。Rx コアのみがこのキューにパケット (イベント) をエンキューします。
- **queue_1** と **queue_2** は、送信先アドレスの設定、暗号処理、圧縮など、特定のイベント処理ステージに使用されます。ワーカーコアはこれらのタスクを実行して、キュー 0、1、2、3 の間でパケットを転送します。
- **queue_3** は、送信準備が整ったパケットを保持するために使用されます。Tx コアのみがこのキューからパケットをデキューします。

デキュー操作は、`rte_event_dequeue_burst()` ルーチンを使用して無限ループで行われます。そのため、ワーカーコアはイベント・デバイス・ポートを継続的にポーリングして、処理すべきイベントのバッチを探します。バッチサイズは、全体の負荷と各ステージのパフォーマンスに依存します。最大バッチサイズは、ワークロードによって定義されます。

インテル® VTune™ プロファイラーが提供するワーカーごとのデキュー統計を利用して、負荷分散の詳細を明らかにし、パイプライン構成の効率を解析し、パイプラインのボトルネックを特定できます。

このレシピは、次の手順に従って、DPDK ベースのアプリケーションのパイプライン処理モデルの効率を解析します。

使用するもの

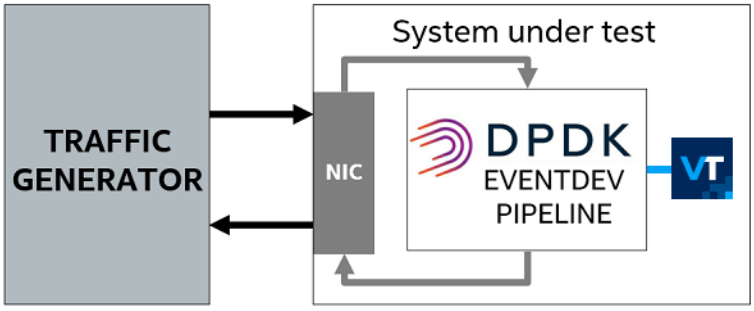
以下は、このパフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** DPDK `eventdev_pipeline` アプリケーション。`eventdev` API の使用法を示し、パイプラインを設定し、イベント処理を実行するワーカーコアを割り当てる方法を示します。インテル® VTune™ プロファイラーのサポートが有効な DPDK でコンパイルされています。
- **ツール:**
  - DPDK。インテル® VTune™ プロファイラーのサポートを有効にしてコンパイルされています。DPDK 側で `eventdev` プロファイルを有効にするには、`dpdk_eventdev_vtune_profiling.patch` パッチを適用して DPDK とターゲット DPDK アプリケーションを再コンパイルする必要があります。DPDK イベントデバイスのプロファイル・パッチはこちら (英語) からダウンロードしてください。
  - インテル® VTune™ プロファイラー 2020: 入力および出力解析

    注

    - バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。

        - インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
        - 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
            - インテル® VTune™ プロファイラー製品ページ
            - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)
- **システムの設定:**
    - **トラフィック・ジェネレーター:** テストするシステムのトラフィックを生成するシステム。
    - **テスト対象のシステム:** パケット処理用の `eventdev_pipeline` アプリケーションと、パフォーマンス・データを収集するインテル® VTune™ プロファイラーが動作しているシステム。



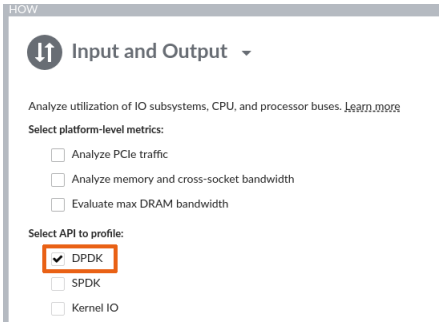
- **CPU:** インテル® Xeon® Platinum 8168 プロセッサー (開発コード名: Skylake)。
- **オペレーティング・システム:** Linux*。

入力と出力解析を実行する

DPDK `eventdev` デキュー統計を収集するには、インテル® VTune™ プロファイラーの入力および出力解析を使用します。

GUI から解析を実行するには、次の操作を行います。

1. インテル® VTune™ プロファイラー GUI を起動して、新しいプロジェクトを作成します。
2. **[HOW (どのように)]** ペインで **[Input and Output (入力および出力)]** 解析を選択します。
3. **[Select IO API type to profile (プロファイルする I/O API タイプを選択)]** で **[DPDK IO API (DPDK I/O API)]** を選択します。



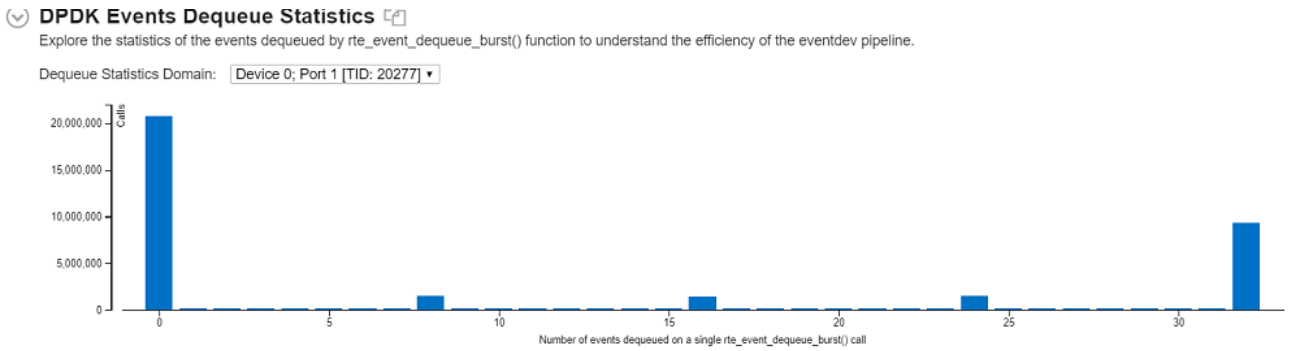
4. **[Start (開始)]** ボタンをクリックします。

コマンドラインから DPDK プロファイルを使用した入力および出力解析を実行するには、次のコマンドを使用します。

```
vtune -collect io -knob kernel-stack=false -knob dpdk=true --target-process=eventdev_pipeline
```

ステージごとの負荷を解析する

DPDK `eventdev` パイプライン利用の全体的な特徴を取得するには、**[Summary (サマリー)]** タブから開始して **[DPDK Events Dequeue Statistics (DPDK イベントデキュー統計)]** ヒストグラムを調査します。

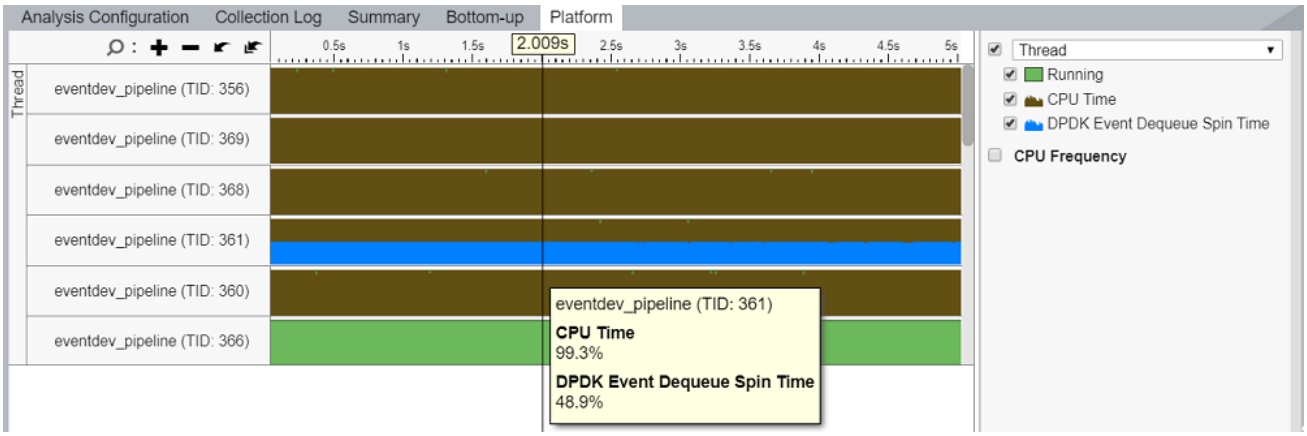


このヒストグラムは、`eventdev` ポートごと、つまりイベントデバイスをポーリングするワーカースレッドごとに、デキューイベント数の統計を表します。ヒストグラムのそれぞれの領域を調べることで、不均衡な負荷分散、オーバーサブスクリプション、十分に利用されていないワーカーを特定できます。

ワーカースレッドの負荷分散に不均衡が見られる場合は、これを避けるようにパイプラインを再構成し、解析を再度実行してください。

CPU 利用を解析する

イベントデキュー操作を実行するワーカーの CPU 利用状況を理解するには、**[Platform (プラットフォーム)]** タブに移動して、ワーカースレッドに関連付けられている **[DPDK Event Dequeue Spin Time (DPDK イベント・デキュー・スピン時間)]** メトリックを調べます。



スレッドごとの **[DPDK Event Dequeue Spin Time (DPDK イベント・デキュー・スピン時間)]** メトリックは、空のデキューサイクルの比率を示しています。これは、デキュー呼び出しの総数に対して、ゼロイベントを返し

た `rte_event_dequeue_burst()` 呼び出しの比率です。このメトリックを使用してワーカースレッドの負荷を推定し、アプリケーションのコアが十分に活用されていないか、リソースを増やす必要があるかを判断します。

関連情報

- クックブック: DPDK アプリケーションのコア使用率
- クックブック: DPDK アプリケーションの PCIe* トラフィック
- DPDK イベント・デバイス・ライブラリー (英語)
- Data Plane Development Kit (DPDK) Eventdev ライブラリーの概要 (英語)
- ソフトウェア・ネットワーク・データ・プレーンのベンチマークと解析 (英語)

インテル® データ・ダイレクト I/O テクノロジーの効果的な利用

このレシピは、インテル® VTune™ プロファイラーを使用して、インテル® Xeon® プロセッサーのハードウェア機能であるインテル® データ・ダイレクト I/O テクノロジー (インテル® DDIO) の利用効率を明らかにする方法を示します。

コンテンツ・エキスパート: Ilia Kurakin (英語)、Perry Taylor (英語)

従来、インバウンドの PCIe* トランザクションはメインメモリーをターゲットにしており、I/O デバイスから消費コアへのデータ移動には複数の DRAM アクセスを必要とします。ソフトウェア・データ・プレーンのような I/O を多用するユースケースでは、この方式は適用できません。

例えば、100G ビット の NIC が 64B のパケットと 20B のイーサネット・オーバーヘッドでフル稼働している場合、新しいパケットは平均して 6.72 ナノ秒ごとに到着します。パケットパス上のいずれかのコンポーネントが、このわずかな時間を上回って個々のパケットを処理すると、パケットロスが発生します。3GHz で動作するコアの場合、6.72 ナノ秒は 20 クロックサイクルにしかなりませんが、DRAM のレイテンシーは平均して 5 ~ 10 倍になります。これは従来の DMA アプローチの主なボトルネックとなっています。

インテル® Xeon® プロセッサーのハードウェア機能であるインテル® DDIO テクノロジー (英語) は、PCIe* デバイスが L3 キャッシュ (LLC-ラストレベルキャッシュ) との間で直接リード/ライト操作を行うことにより、このボトルネックを解消します。これにより、受信データをできるだけコアの近くに配置できます。インテル® DDIO テクノロジーを適切に活用することで、L3 キャッシュのみでコアと I/O デバイス間の相互作用に対応して、DRAM へのアクセスを完全に排除することができ、以下の利点が得られます。

- 高いスループットを可能にする、低レイテンシーのインバウンド・リード/ライト。
- DRAM 帯域幅と消費電力の軽減。

インテル® DDIO は、常に有効でソフトウェアに対して透過的なハードウェア機能ですが、最適なパフォーマンスが得られない場合があります。

インテル® DDIO の利用率を最適化する主なソフトウェア・チューニング手法は 2 つあります。

- **トポロジー設定:** 複数のソケットを持つシステムでは、I/O デバイス、I/O デバイスと相互作用するコア、およびメモリーが同じ NUMA モードであることが重要です。
- **L3 キャッシュ管理:** 高度なチューニングは、必要なデータを適切なタイミングで L3 キャッシュに保持することで、L3 キャッシュの使用を最適化します。

このレシピは、インテル® VTune™ プロファイラーの入力および出力解析 (英語) を使用して、インテル® DDIO テクノロジーの非効率な利用を検出する方法を紹介します。

- 使用するもの

手順:

- アーキテクチャーを理解する
- インテル® DDIO トラフィックを解析する

使用するもの

- **システム:** 2 ソケットの第 2 世代インテル® Xeon® スケーラブル・プロセッサー・ベースのシステム。
- **アプリケーション:** DPDK `testpmd` (英語) アプリケーション。シングルコアで動作し、ソケット 1 に装着された 40G ネットワーク・インターフェイス・カードの 1 ポートを使ってパケット転送を行うように構成されています。
- **パフォーマンス解析ツール:** インテル® VTune™ プロファイラー 2020 Update 2: 入力および出力解析 (英語)。

  注

  - バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
  - インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
  - 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
    - インテル® VTune™ プロファイラー製品ページ
    - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)

アーキテクチャーを理解する

インテル® Xeon® スケーラブル・プロセッサー (英語) では、L3 キャッシュは 1 つのソケット内のすべてのコアとすべてのインテグレーテッド I/O コントローラー (IIO) (英語) で共有されるリソースです。L3 キャッシュとコア間のデータ転送は、キャッシュライン単位 (64B) で行われます。PCIe* デバイスがシステムメモリーに要求を送ると、IIO はこの要求を 1 つまたは複数のキャッシュライン要求に変換し、ローカルソケット上の L3 キャッシュに発行します。ローカル L3 キャッシュへの要求は、次の方法で行うことができます。

- **インバウンド PCIe* ライト要求**
  - **インバウンド PCIe* ライト L3 ヒット** — 理想的なシナリオ — ライト要求の対象となるアドレスが、すでにローカルの L3 キャッシュにある場合に発生します。L3 のキャッシュラインに新しいデータが上書きされます。
  - **インバウンド PCIe* ライト L3 ミス** — 理想的ではないシナリオ — ライト要求の対象となるアドレスがローカルの L3 キャッシュにない場合に発生します。この場合、まず I/O データ専用の L3 ウェイからキャッシュラインを退避させます。退避したラインがダーティな状態の場合、DRAM へのライトバックが発生します。そして、退避したラインの代りに、新しいキャッシュラインが割り当てられます。対象となるキャッシュラインがリモートにキャッシュされている場合、コヒーレンシーのルールを適用して、キャッシュラインの割り当てを完了するため、インテル® ウルトラ・パス・インターコネクト (インテル® UPI) を介したクロスソケット・アクセスが必要になります。最後に、キャッシュラインが新しいデータで更新されます。

- **インバウンド PCIe* リード要求**
  - **インバウンド PCIe* リード L3 ヒット** — 理想的なシナリオ — リード要求の対象となるアドレスがローカルの L3 キャッシュにある場合に発生します。データが読み取られ、PCIe* デバイスに送信されます。
  - **インバウンド PCIe* リード L3 ミス** — 理想的ではないシナリオ — リード要求の対象となるアドレスがローカル L3 キャッシュにない場合に発生します。この場合、データはローカル DRAM またはリモートソケットのメモリー・サブシステムから読み取られます。ローカル L3 割り当ては行われません。

第 1 世代および第 2 世代インテル® Xeon® スケーラブル・プロセッサーの場合、インテル® VTune™ プロファイラーの入力および出力解析では、インバウンドの PCIe* リードおよびライトの **L3 ヒット/ミスメトリック**や**平均レイテンシー**が提供され、PCIe* デバイスグループごとにデータの内訳が示されます。これらのグループは、IIO コントローラーとメッシュ (英語) 間のインターフェイスである M2PCIe ユニットによって定義されます。

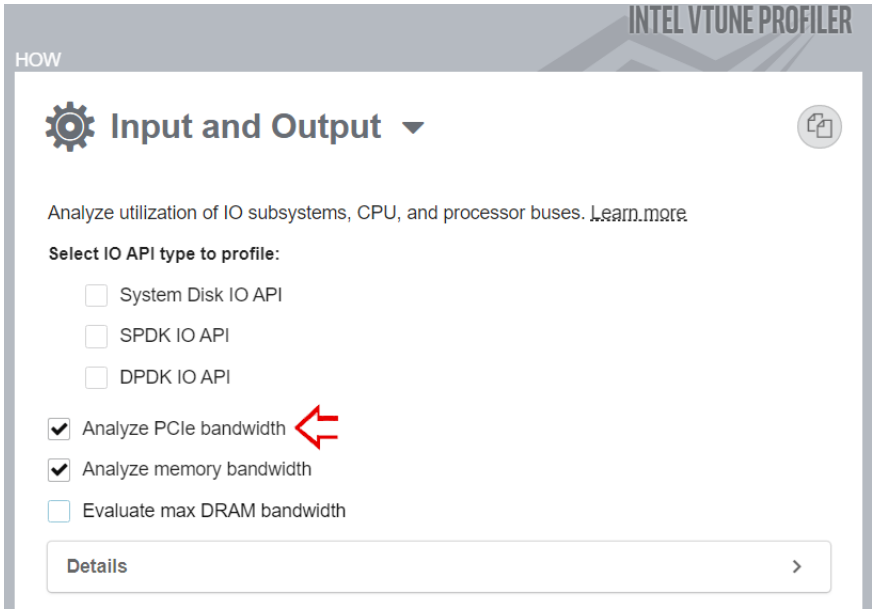
インテル® DDIO トラフィックを解析する

インテル® VTune™ プロファイラーの入力および出力解析を使用して、インテル® DDIO 利用効率メトリックを収集します。

注
解析を実行するには、第 1 世代または第 2 世代インテル® Xeon® スケーラブル・プロセッサーを使用しており、サンプリング・ドライバーをロード済み (英語) でなければなりません。推奨される最小収集時間は 20 秒です。

解析を実行するには、次の操作を行います。

1. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** を選択してアプリケーションのパスと引数を指定するか、**[Attach to Process (プロセスにアタッチ)]** を選択して PID を指定します。さらに、**[Automatically stop collection after (sec) (指定時間後に収集を自動停止 (秒))]** を使用すると、収集時間を自動制御できます。
2. **[HOW (どのように)]** ペインで **[Input and Output (入力および出力)]** 解析を選択します。**[Analyze PCIe traffic (PCIe* トラフィックを解析)]** チェックボックスをオンにして、インテル® DDIO 利用効率メトリックを収集します。



3. [Start (開始)] ボタンをクリックして、解析を実行します。

典型的な例を理解する

典型的なインテル® DDIO テクノロジーの非効率な利用例とその検出に役立つインテル® VTune™ プロファイラーの機能を理解するため、2 ソケットの第 2 世代インテル® Xeon® スケーラブル・プロセッサーを搭載したシステムで DPDK `testpmd` アプリケーションを実行します。このアプリケーションは、シングルコアで動作し、ソケット 1 に装着された 40G の NIC の 1 ポートを使ってパケット転送を行うように構成されています。

トラフィック・ジェネレーターは、1 つのコアが処理できる範囲をはるかに超えるパケットレートで 64B パケットをシステムに供給します。最適化の基準はシステムのスループットであり、スループットは高いほうが良いです。この構成で、シングルコアのアプリケーションのスループットを特定します。

注
ここに示すデータは、パフォーマンス・レポートとして扱われるべきではありません。

以下の 2 つの例では、ソケット 1 のコアがパケット転送を行う構成をベースラインとして使用しています。この構成は、コアと PCIe* デバイスが同じソケット上に存在するため、ローカルと呼ばれます。

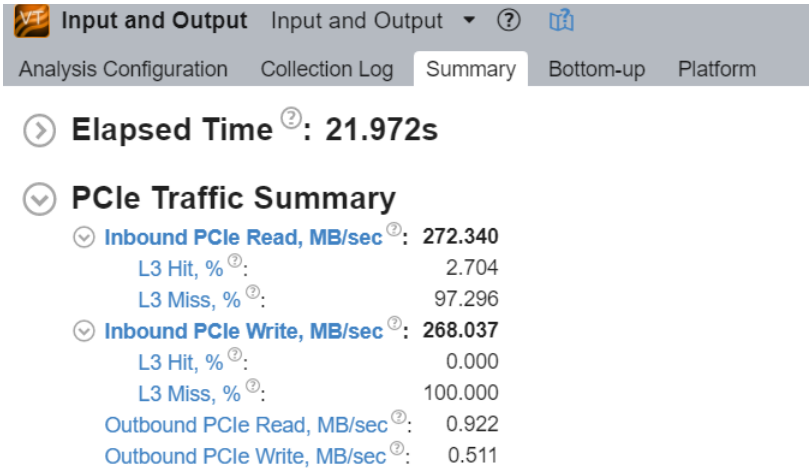
```
# ./testpmd -n 4 -l 24,25 -- -i
testpmd> set fwd mac retry
testpmd> start
```

`mac` 転送モードを使用します。このモードでは、コアがパケットの送信元と送信先のイーサネット・アドレスを変更するため、送信コアはパケット記述子にアクセスし、各パケットにタッチします。

インテル® VTune™ プロファイラーの入力および出力解析を実行して、**[Summary (サマリー)]** ウィンドウの **[Platform Diagram (プラットフォーム・ダイアグラム)]** セクションを使用して結果の調査を開始します。

プラットフォーム・ダイアグラムにはシステムのトポロジーが表示され、物理コア (解析するワークロードの計算ごと)、DRAM、インテル® UPI、PCIe*リンクなどのハードウェア・リソースの平均利用率が示されます。PCIe*

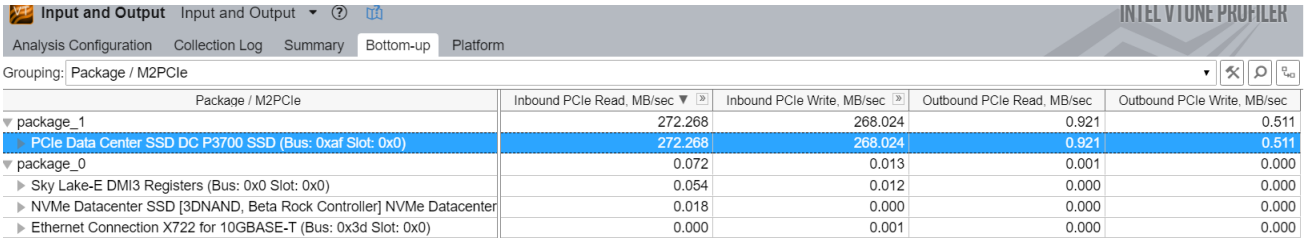
デバイスのメトリックは、ペイロードの転送に消費される物理帯域幅の割合として計算される**有効リンク利用率**を示し、オーバーヘッドは考慮されません。詳細は、ユーザーガイドの「入力および出力解析」(英語) を参照してください。



[Summary (サマリー)] タブの **[PCIe Traffic Summary (PCIe* トラフィックのサマリー)]** セクションで、第 1 レベルのメトリックであるインバウンドとアウトバウンドの PCIe* リードおよびライト・トラフィックの合計と、第 2 レベルのメトリックであるインテル® DDIO の利用効率を確認します。

- **L3 ヒット/ミス率**は、L3 キャッシュにヒット/ミスしたインバウンド要求の割合を示します。
- **平均レイテンシー**は、キャッシュラインに対するインバウンド要求の処理にプラットフォームが費やした平均時間を示します。
- **コア/I/O 競合**メトリックは、キャッシュライン競合が発生したインバウンド・ライト比率を示します。検出されると、インテル® VTune™ プロファイラーは可能なチューニングの方向性を提案します。

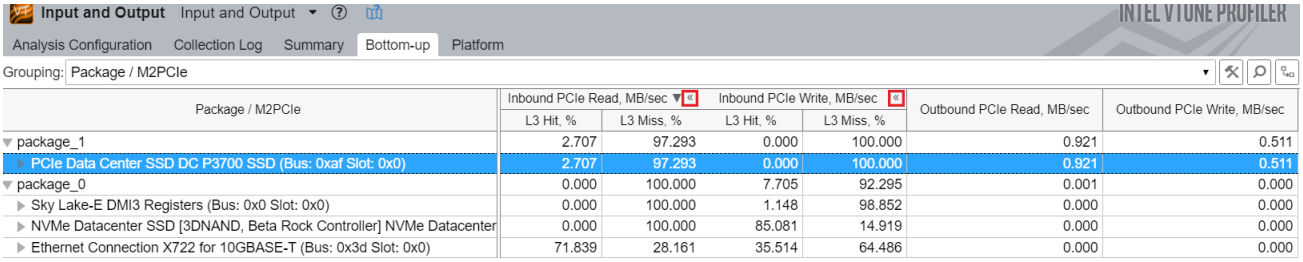
詳細な I/O メトリックを表示するには、**[Bottom-up (ボトムアップ)]** ペインの **[PCIe Traffic Summary (PCIe* トラフィックのサマリー)]** セクションでメトリックをクリックします。



Package / M2PCIe	Inbound PCIe Read, MB/sec ▼	Inbound PCIe Write, MB/sec	Outbound PCIe Read, MB/sec	Outbound PCIe Write, MB/sec
package_1	272.268	268.024	0.921	0.511
PCIe Data Center SSD DC P3700 SSD (Bus: 0xaf Slot: 0x0)	272.268	268.024	0.921	0.511
package_0	0.072	0.013	0.001	0.000
Sky Lake-E DMI3 Registers (Bus: 0x0 Slot: 0x0)	0.054	0.012	0.000	0.000
NVMe Datacenter SSD [3DNAND, Beta Rock Controller] NVMe Datacenter	0.018	0.000	0.000	0.000
Ethernet Connection X722 for 10GBASE-T (Bus: 0x3d Slot: 0x0)	0.000	0.001	0.000	0.000

PCIe* メトリックを表示するには、**[Grouping (グループ化)]** ドロップダウン・メニューから **[Package/M2PCIe (パッケージ/M2PCIe)]** グループを選択します。このグループは、サービスを提供する PCIe* デバイスによって名前が付けられたソケットと M2PCIe ブロックごとにメトリックを分類します。M2PCIe が複数のデバイスを管理している場合、デバイス名はカンマ区切りのリストで示されます。セルにホバーするとすべてのデバイスが表示されます。

インテル® DDIO 利用効率メトリックを表示するには、各カラムの **[Expand (展開)]** ボタンをクリックして第 2 レベルを展開します。



Package / M2PCIe	Inbound PCIe Read, MB/sec		Inbound PCIe Write, MB/sec		Outbound PCIe Read, MB/sec	Outbound PCIe Write, MB/sec
	L3 Hit, %	L3 Miss, %	L3 Hit, %	L3 Miss, %		
package_1	2.707	97.293	0.000	100.000	0.921	0.511
PCIe Data Center SSD DC P3700 SSD (Bus: 0xaf Slot: 0x0)	2.707	97.293	0.000	100.000	0.921	0.511
package_0	0.000	100.000	7.705	92.295	0.001	0.000
Sky Lake-E DMI3 Registers (Bus: 0x0 Slot: 0x0)	0.000	100.000	1.148	98.852	0.000	0.000
NVMe Datacenter SSD [3DNAND, Beta Rock Controller] NVMe Datacenter	0.000	100.000	85.081	14.919	0.000	0.000
Ethernet Connection X722 for 10GBASE-T (Bus: 0x3d Slot: 0x0)	71.839	28.161	35.514	64.486	0.000	0.000

[Platform (プラットフォーム)] タブで DRAM とインテル® UPI 帯域幅の詳細を確認します。

リモート・ソケット・アクセス

最初の例は、インテル® DDIO ミス率とインテル® DDIO レイテンシーが高く、DRAM とインテル® UPI トラフィックを発生させる、最適でないアプリケーション・トポロジーを示します。これらの要因が重なると、パフォーマンスが低下します。

これは、送信コアと NIC が別々のソケットに存在するリモート構成を使用した、最適でないトポロジーの例です。

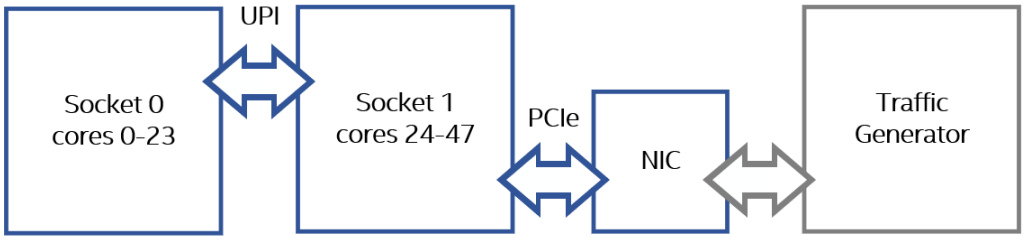
```
# ./testpmd -n 4 -l 0,1 -- -i
testpmd> set fwd mac retry
testpmd> start
```

GUI またはコマンドラインから **[Attach to Process (プロセスにアタッチ)]** モードで解析を再度実行します。

```
# vtune -collect io --duration 20 --target-process testpmd
```

[Platform Diagram (プラットフォーム・ダイアグラム)] を確認すると、トポロジーの問題がすぐに分かります。

- 別のソケットの NIC に関連したコア利用率
- 非ゼロの DRAM およびインテル® UPI 利用率

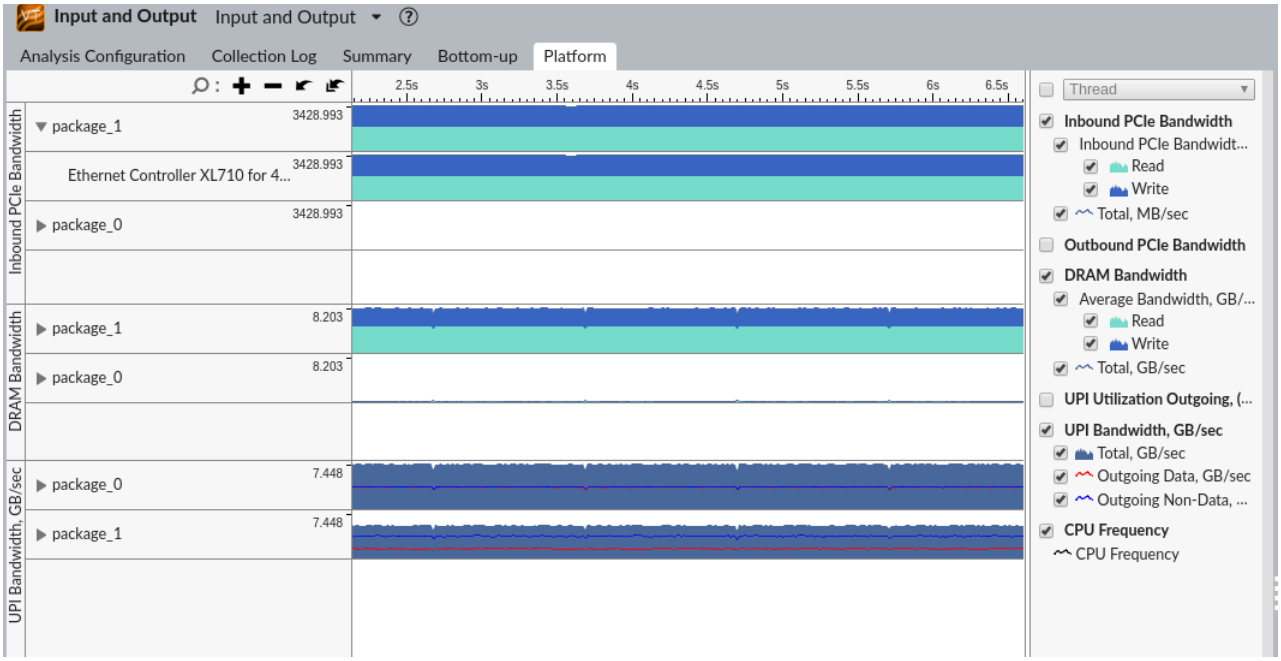


結果を調査します。

送信コア ID	スループット (Mpps)	インバウンド PCIe* リード L3 ミス (%)	平均インバウンド PCIe* リード・レイテンシー (ナノ秒)	インバウンド PCIe* ライト L3 ミス (%)	平均インバウンド PCIe* ライト・レイテンシー (ナノ秒)
25	21.1	0	112	0	135
1	17.1	100	320	100	240

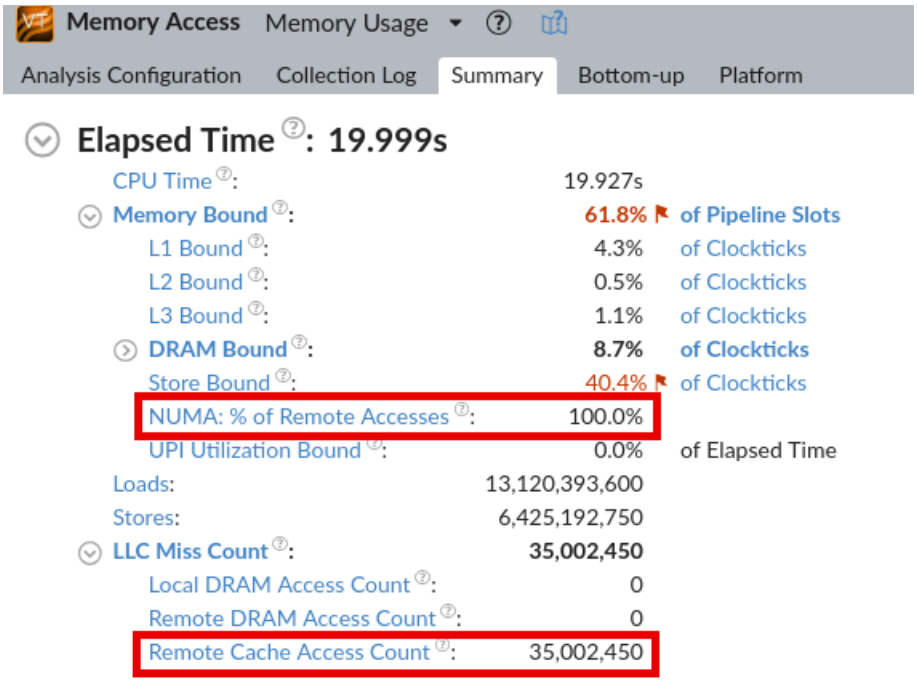
送信コアと NIC が別々のソケットに存在する構成では、L3 ミスが 100% となり、インバウンド PCIe* 要求レイテンシーが高くなり、パフォーマンスが低下します。

リモートケースでの高いミス率の原因を理解するために、解析結果の **[Platform (プラットフォーム)]** ペインに移動します。

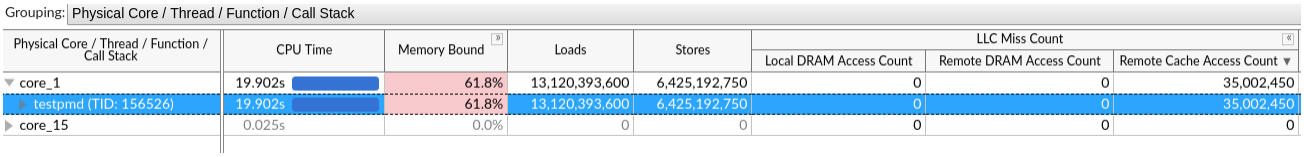


インテル® UPI および DRAM 帯域幅が高いことが分かります。システム上で起こっていることを全体的に把握するには、メモリーアクセス解析を使用してコアの観点から構成を解析します。

```
# vtune -collect memory-access -knob dram-bandwidth-limits=false --duration
20 --target-process testpmd
```



インテル® VTune™ プロファイラーのレポートから、リモート構成では、リモート L3 キャッシュで解決される LLC ミスが原因で生じるリモートアクセスによって CPU コア利用率が制限されていることが分かります。**[Bottom-up (ボトムアップ)]** ペインに移動して、リモート LLC にアクセスするコア、プロセス、スレッド、関数を特定します。



Grouping: Physical Core / Thread / Function / Call Stack

Physical Core / Thread / Function / Call Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count		
					Local DRAM Access Count	Remote DRAM Access Count	Remote Cache Access Count
core_1	19.902s	61.8%	13,120,393,600	6,425,192,750	0	0	35,002,450
testpmd (TID: 156526)	19.902s	61.8%	13,120,393,600	6,425,192,750	0	0	35,002,450
core_15	0.025s	0.0%	0	0	0	0	0

すべての LLC アクセスは、ソケット 0 のコア 1 で実行している `testpmd` アプリケーションで発生していることが分かります。

これで、リモート構成で起こっていることを再現できます。ソケット 0 の DRAM 帯域幅はゼロであるため、記述子やパケットリングに使用されるアプリケーションの消費メモリーはすべてソケット 1 で NIC にローカルに割り当てられます。ソケット 0 の送信コアが記述子とパケットにアクセスすると、ソケット 0 の LLC ミスが発生し、ソケット 1 からデータを取得するためスヌープ要求が送信され、インテル® UPI トラフィックが発生します。これらの要求がソケット 1 の LLC に変更済みデータを見つけると、DRAM ライトバックが発生して DRAM 帯域幅に影響を与えます。

デバイスが同じ場所に再度アクセスすると、データは最後に使用されたソケット 0 のコアにあるため、ソケット 1 で L3 キャッシュミスが発生します。そして、メモリー・ディレクトリーにアクセスしてアドレスがキャッシュされているソケット特定するため、DRAM 帯域幅が発生します。この場合、スヌープ要求はソケット 1 からソケット 0 へ送信され、コヒーレンシー・ルールを適用して I/O 要求を完了します。

その結果、インテル® VTune™ プロファイラーの入力および出力解析とメモリーアクセス解析で次の値が得られました。

送信コア ID	スループット (Mpps)	インバウンド PCIe* リード L3 ミス (%)	平均インバウンド PCIe* リード・レイテンシー (ナノ秒)	インバウンド PCIe* ライト L3 ミス (%)	平均インバウンド PCIe* ライト・レイテンシー (ナノ秒)	testpmd: LLC ミス数	testpmd: リモート・キャッシュ・アクセス数	合計 DRAM 帯域幅 (ソケット 1) (GB/秒)	合計インテル® UPI 帯域幅 (GB/秒)
25	21.1	0	112	0	135	0	0	0	0
1	17.1	100	320	100	240	35M	35M	8	12.6

要求レイテンシーが高くなることによるスループットの低下に加えて、最適でないアプリケーションのトポロジーは、システムの DRAM 帯域幅、インテル® UPI 帯域幅、プラットフォームの電力を無駄にします。

最適でない L3 キャッシュ管理

2 つ目の例は、さまざまなパフォーマンスの問題を含んでいます。リモート・ソケット・アクセスがない場合でも、DDIO ミスに見られるように、LLC の I/O データ管理が最適でないためにパフォーマンスが制限されます。

DPDK `testpmd` を使用した例を示すため、パケットリングとして使用されるメモリープールのソフトウェア・レベルのキャッシュ (DPDK メモリープール・ライブラリー (英語)) を無効にします。このデフォルトのキャッシュメカニズムを使用すると、コアは新しいパケットを受信し、データのデスティネーションとして、ほとんどの場合ハードウェア・キャッシュに存在する (英語) 「warm」メモリープール要素を使用します。そのため、パケット・リ

ング・サイズが L3 容量を超える場合であっても、インバウンド PCIe* リードおよびライトで L3 ミスが発生することはありません。

メモリープールのソフトウェア・キャッシュを無効にする `--mbcache=0` を使用して `testpmd` を実行します。

```
# ./testpmd -n 4 -l 24,25 -- -i --mbcache=0
testpmd> set fwd mac retry
testpmd> start
```

最初のローカル構成と、同じ構成でメモリープールのソフトウェア・キャッシュを無効にした場合の `testpmd` のパフォーマンスを比較します。

メモリープール・キャッシュ	スループット (Mpps)	インバウンド PCIe* リード L3 ミス (%)	平均インバウンド PCIe* リード・レイテンシー (ナノ秒)	インバウンド PCIe* ライト L3 ミス (%)	平均インバウンド PCIe* ライト・レイテンシー (ナノ秒)	合計 DRAM 帯域幅 (ソケット 1) (GB/秒)
有効	21.1	0	112	0	135	0
無効	20.2	0	115	54	178	4.7

メモリープール・キャッシュの最適化なしでアプリケーションを実行すると、インバウンド PCIe* ライト要求の大部分で L3 ミスが発生します。

1 パケットを転送するため、NIC とコアはパケット記述子とパケットリングを介して通信します。データパスで NIC はインバウンド PCIe* ライトを使用してパケットを書き込み、パケット記述子を更新します (詳細は「DPDK アプリケーションの PCIe* トラフィック」を参照)。

記述子リングは常に I/O の前にコアによってアクセスされるため、記述子アクセスで I/O L3 ミスが発生する確率は低いです。しかし、パケットリングは最初に NIC によってアクセスされるため、すべてのインバウンド PCIe* ライト L3 ミスは、NIC が Rx ステージでパケットリングへパケットを書き込むことで発生します。同時に、DPDK はネットワーク・データのゼロコピーポリシーに従っているため、インバウンド PCIe* リード L3 ミスは発生しません。NIC がパケットを取得して Tx を実行しようとすると、パケットはすでにキャッシュされています。

この結論は、パケットリングのソフトウェア・キャッシュを無効にした場合、パケットリングのアクセスはハードウェア・キャッシュ・ミスになるということから容易に導き出されます。このレシピは、実際のシナリオで、どのデータが I/O でアクセスされ、結果的にインテル® DDIO ミスになったかを理解する方法を示しています。

この場合、インバウンド PCIe* 要求の L3 ミスは、L3 からのライトバック、L3 割り当て、メモリー・ディレクトリー・アクセスによって生じる DRAM 帯域幅を意味します。

まとめ

インテル® DDIO テクノロジーは、ソフトウェアが高速 I/O デバイスを完全に活用できるようにします。しかし、NUMA システムでアプリケーションのトポロジーが最適でない場合や、L3 キャッシュのデータ管理が最適でない場合、L3 アクセス・レイテンシーが高くなり、不要な DRAM トラフィックが発生して、ソフトウェアがインテル® DDIO の恩恵を十分に受けられません。インテル® VTune™ プロファイラーの各種解析タイプ (入力および出力、メモリーアクセス、マイクロアーキテクチャー全般) はこのような非効率性を検出して、コアと I/O の両方の観点から全体像を提供します。

アプリケーションのトポロジーが最適でないという問題には明らかな解決策がありますが、効率的な L3 利用スキームの開発は容易ではないかもしれません。このようなスキームを設計して、パフォーマンスを向上するいくつかのアプローチがあります。

- LLC 容量よりも小さいバッファーサイズを選択する
- バッファー要素を再利用する
- デバイスが使用する場所をソフトウェア・プリフェッチする
- キャッシュ・アロケーション・テクノロジー (CAT) (英語) を使用して L3 をパーティショニングする

注
このレシピの情報は、インテル® VTune™ プロファイラー・デベロッパー・フォーラムを参照してください。

関連情報

- ユーザーガイドの「入力および出力解析」セクション (英語)
- インテル® データ・ダイレクト I/O テクノロジーの概要 (英語)
- DPDK アプリケーションの PCIe* トラフィック
- インテル® Xeon® スケーラブル・プロセッサー・ファミリーの技術概要 (英語)
- 第 2 世代インテル® Xeon® スケーラブル・プロセッサーの技術概要
- インテル® Xeon® スケーラブル・プロセッサー・ファミリーの IIO パフォーマンス・モニタリング・イベントの活用 (英語)
- インテル® Xeon® スケーラブル・プロセッサー・ファミリーのアンコア・リファレンス・マニュアル (英語)
- マルチスレッド Data Plane Development Kit (DPDK) アプリケーションのメモリー使用の最適化 (英語)
- ソフトウェア・データ・プレーンのベンチマークと解析ホワイトペーパー (英語)
- What Every Programmer Should Know About Memory by Ulrich Drepper of Red Hat, Inc. の Ulrich Drepper 氏によるメモリーについてすべてのプログラマーが知っておくべきこと (英語)

最新の命令セットを使用して最適化されたポータブルなバイナリーをコンパイルする

移植性を維持しながら最新の命令セットを使用してバイナリーをコンパイルするさまざまな方法を学びます。

コンテンツ・エキスパート: Roman Khatko

最近のインテル® プロセッサーは、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® AVX2、およびインテル® AVX など、異なるバージョンの命令セット拡張をサポートしています。

アプリケーションをコンパイルするとき、アプリケーションの使用目的に基づいて 3 つのオプションを検討します。

- **汎用バイナリー:** 汎用 x86 命令セット向けにアプリケーションをコンパイルします。アプリケーションはすべての x86 プロセッサーで動作しますが、新しいプロセッサーの能力を最大限に活用できません。
- **ネイティブバイナリー:** 特定のプロセッサー向けにアプリケーションをコンパイルします。アプリケーションはターゲット・プロセッサーのすべての機能を活用できますが、古いプロセッサーでは動作しません。
- **ポータブルバイナリー:** コンパイラー・オプションや関数の属性を使用して、異なるプロセッサーをターゲットとする関数の複数のバージョンを含む、最適化されたポータブルなバイナリーをコンパイルします。生成されるバイナリーは、特定のプロセッサー向けにコンパイルされたアプリケーション (ネイティブバイナリー) のパフォーマンス特性を備えつつ、古いプロセッサーでも動作します。

このレシピは、汎用バイナリーの移植性を維持しながら、ネイティブバイナリーのパフォーマンス特性を備えたポータブルバイナリーをコンパイルする方法を示します。このレシピでは、最初に汎用バイナリーとネイティブバイナリーの両方をコンパイルして、パフォーマンスの向上がバイナリーサイズの増加に見合うものかどうかを判断します。

このレシピでは、インテル® C++ コンパイラー・クラシックと GNU* コンパイラー・コレクション (GCC) を取り上げます。

このレシピでは、`CPUID` プロセッサー命令を使用した手動ディスパッチ、プロセッサー・ターゲット・コンパイラー・オプション (英語)、およびターゲットの関数属性 (英語) は取り上げていません。

使用するもの

以下は、このレシピで使用するシステムとツールのリストです。

- **プロセッサー:** インテル® Xeon® プロセッサー (開発コード名 Cascade Lake)
- **オペレーティング・システム**: Fedora* 32
- **コンパイラー:**
  - インテル® C++ コンパイラー・クラシック 2021.1.2
  - GCC 10.1.1
- **解析ツール:** インテル® VTune™ プロファイラー 2021.1.2

サンプル・アプリケーション

次のコードをソースファイル `fma.c` に保存します。

```
// fma.c
#include <stdio.h>
#include <stdlib.h>

void init(float *a, float *b, float *c, int size)
{
    for (int i = 0; i < size; i++)
    {
        a[i] = (float) (i % 10);
        b[i] = a[i] * 1.1f;
        c[i] = a[i] * 1.2f;
    }
}

void my_fma(float *a, float *b, float *c, int size)
{
    for (int i = 0; i < size; i++)
    {
        c[i] += a[i]*b[i];
    }
}

#define ITERATIONS 10000000
#define SIZE 2048

int main()
{
    float *a = malloc(SIZE*sizeof(float));
    float *b = malloc(SIZE*sizeof(float));
    float *c = malloc(SIZE*sizeof(float));

    for (int i = 0; i < ITERATIONS; i++)
    {
        init(a, b, c, SIZE);
        my_fma(a, b, c, SIZE);
    }
    printf("%f", c[5]); // use the data

    free(a);
    free(b);
    free(c);
    return 0;
}
```

汎用の最適化されたバイナリーのコンパイル

『インテル® VTune™ プロファイラー・ユーザーガイド』の説明 (Linux* および Windows*) に従ってバイナリーをコンパイルします。

インテル® C++ コンパイラー・クラシック

デバッグ情報を含め、-O3 最適化レベルを指定してバイナリーをコンパイルします。

```
icc -g -O3 -debug inline-debug-info fma.c -o fma_generic
```

GNU* コンパイラー・コレクション

デバッグ情報を含め、-O2 最適化レベルを指定してバイナリーをコンパイルします。

```
gcc -g -O2 fma.c -o fma_generic_O2
```

コードがインテル® VTune™ プロファイラーの HPC パフォーマンス特性解析タイプを使用してベクトル化されているか確認します。

確認するには、次の解析を実行します。

```
vtune -c hpc-performance -r fma_generic_O2_hpc ./fma_generic_O2
```

インテル® VTune™ プロファイラー GUI で結果を開きます。

```
vtune-gui fma_generic_O2_hpc
```

解析結果を開き、**[Summary (サマリー)]** タブの **[Top Loops/Functions with FPU Usage by CPU Time (CPU 時間による FPU を使用する上位のループ/関数)]** セクションを確認します。

Top Loops/Functions with FPU Usage by CPU Time
This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
[Loop at line 6 in init]	25.334s	10.7%	0.0%	100.0%		
[Loop at line 16 in my_fma]	16.524s	34.0%	0.0%	100.0%		

*N/A is applied to non-summable metrics.

[FP Ops: Scalar (FP 操作: スカラー)] 値が 100% で **[Vector Instruction Set (ベクトル命令セット)]** カラムが空の場合、GCC は -O2 最適化レベルでコードをベクトル化していません。-O2 -ftree-vectorize または -O3 オプションを使用してベクトル化を有効にします。

-O3 最適化レベルを指定して fma_generic バイナリーをコンパイルします。

```
gcc -g -O3 fma.c -o fma_generic
```

ネイティブバイナリーのコンパイル

インテル® C++ コンパイラー・クラシックを使用してネイティブバイナリーをコンパイルする

-xHost オプションは、コンパイルを行うプロセッサーで利用可能な最上位の命令セットを使用したコードを生成するようにコンパイラーに指示します。-x{Arch} オプション ({Arch} はアーキテクチャーの開発コード名) を使用すると、特定のアーキテクチャーのプロセッサー機能をターゲットにするようにコンパイラーに指示できます。

`-xHost` オプションを使用して `fma_native` バイナリーをコンパイルします。

```
icc -g -O3 -debug inline-debug-info -xHost fma.c -o fma_native
```

GNU* コンパイラー・コレクションを使用してネイティブバイナリーをコンパイルする

`-march=native` オプションを使用して `fma_native` バイナリーをコンパイルします。

```
gcc -g -O3 -march=native fma.c -o fma_native
```

プロセッサーがインテル® AVX-512 命令セットをサポートしている場合、`mprefer-vector-width=512` オプションを使用することを検討してください。

汎用バイナリーとネイティブバイナリーの比較

両方のバイナリーの HPC パフォーマンス・スナップショット解析データを収集します。

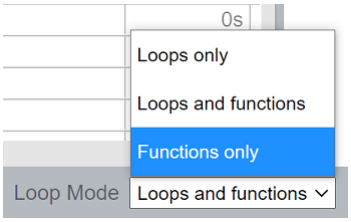
```
vtune -c hpc-performance -r fma_generic_hpc ./fma_generic
```

```
vtune -c hpc-performance -r fma_native_hpc ./fma_native
```

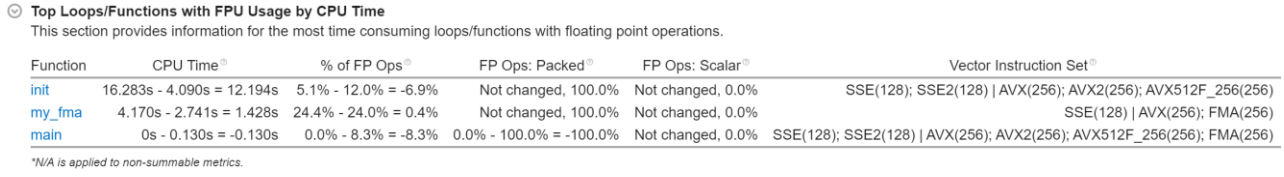
次のコマンドを使用して結果を比較します。

```
vtune-gui fma_generic_hpc fma_native_hpc
```

インテル® VTune™ プロファイラー GUI で、**[Bottom-up (ボトムアップ)]** タブに切り替えて **[Loop Mode (ループモード)]** を **[Functions only (関数のみ)]** に設定します。



[Summary (サマリー)] タブに切り替えて **[Top Loops/Functions with FPU Usage by CPU Time (CPU 時間による FPU を使用する上位のループ/関数)]** セクションまでスクロールします。



Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set
init	16.283s - 4.090s = 12.194s	5.1% - 12.0% = -6.9%	Not changed, 100.0%	Not changed, 0.0%	SSE(128); SSE2(128) | AVX(256); AVX2(256); AVX512F_256(256)
my_fma	4.170s - 2.741s = 1.428s	24.4% - 24.0% = 0.4%	Not changed, 100.0%	Not changed, 0.0%	SSE(128) | AVX(256); FMA(256)
main	0s - 0.130s = -0.130s	0.0% - 8.3% = -8.3%	0.0% - 100.0% = -100.0%	Not changed, 0.0%	SSE(128); SSE2(128) | AVX(256); AVX2(256); AVX512F_256(256); FMA(256)

*N/A is applied to non-summable metrics.

[CPU Time (CPU 時間)] および **[Vector Instruction Set (ベクトル命令セット)]** カラムを確認します。

汎用バイナリーとネイティブバイナリーのパフォーマンスの差を考慮します。複数のコードパスを使用してポータブルバイナリーをコンパイルすることが適切かどうかを判断します。

このサンプル・アプリケーションは、コンパイラーにより自動ベクトル化されました。アプリケーションのベクトル化の可能性を詳しく調べるには、インテル® Advisor を使用します。

ポータブルバイナリーのコンパイル

汎用バイナリーとネイティブバイナリーを比較してパフォーマンスが向上した場合 (例えば、**[CPU Time (CPU 時間)]** が向上した場合)、ポータブルバイナリーをコンパイルします。

インテル® C++ コンパイラー・クラシックを使用してポータブルバイナリーをコンパイルする

`-ax` (Windows* の場合は `/Qax`) オプションを使用して、インテル® プロセッサー向けに複数の機能固有の自動ディスパッチ・コードを生成するようにコンパイラーに指示します。

`-ax` オプションを使用して `fma_portable` バイナリーをコンパイルします。

```
icc -g -O3 -debug inline-debug-info -axCOMMON-AVX512,CORE-AVX2,AVX,SSE4.2,
TREMONT fma.c -o fma_portable
```

GNU* コンパイラー・コレクションを使用してポータブルバイナリーをコンパイルする

汎用バイナリーとネイティブバイナリーの結果を比較します。**[CPU Time (CPU 時間)]** が向上し、ネイティブバイナリーの結果で特定の関数に追加の **[Vector Instruction Set (ベクトル命令セット)]** が使用された場合、この関数に `target_clones` 属性を追加します。

関数がほかの関数を呼び出している場合、`target_clones` 属性は再帰的でないため、`flatten` 属性を追加してインライン展開を行います。

`fma.c` ソースファイルのコンテンツを新しいファイル `fma_portable.c` にコピーして、`TARGET_CLONE` プリプロセッサー・マクロを追加します。

```
#define TARGET_CLONES
__attribute__((flatten,target_clones("default,sse4.2,avx,"\
    "avx2,avx512f,arch=skylake,arch=tremont,arch=skylake-avx512,"\
    "arch=cascadelake,arch=cooperlake,arch=tigerlake,arch=icelake-server")))
```

サポートされているアーキテクチャーの一覧は、GCC マニュアルの x86 オプション (英語) を参照してください。

複数のバージョンの関数を作成すると、バイナリーのサイズは増加します。各ターゲットのパフォーマンス向上とコードサイズのトレードオフを考慮してください。インテル® VTune™ プロファイラーを使用してデータを収集して結果を比較することにより、データ駆動型の決定を行い、新しい命令で高速に実行される関数にのみ `TARGET_CLONES` マクロを適用できます。

`my_fma` 関数定義と `init` 関数の前に `TARGET_CLONES` マクロを追加し、`fma_portable.c` を保存します。

```
TARGET_CLONES
void my_fma(float *a, float *b, float *c, const int size)
```

`fma_portable` バイナリーをコンパイルします。

```
gcc -g -O3 fma_portable.c -o fma_portable
```

ポータブルバイナリーとネイティブバイナリーの比較

ポータブルバイナリーとネイティブバイナリーのパフォーマンスを比較するため、`fma_portable` バイナリーの HPC パフォーマンス特性データを収集します。

```
vtune -c hpc-performance -r fma_portable_hpc ./fma_portable
```

インテル® VTune™ プロファイラー GUI で比較を開きます。

```
vtune-gui fma_portable_hpc fma_native_hpc
```

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set
init	3.383s - 4.090s = -0.707s	8.6% - 12.0% = -3.4%	Not changed, 100.0%	Not changed, 0.0%	AVX(128); AVX(256); AVX2(256); AVX512F_256(256); AVX512F_512(512) | AVX(256); AVX2(256); AVX512F_256(256)
my_fma	1.899s - 2.741s = -0.842s	32.9% - 24.0% = 8.9%	Not changed, 100.0%	Not changed, 0.0%	AVX(128); AVX(256); AVX512F_512(512); FMA(256) | AVX(256); FMA(256)
main	0s - 0.130s = -0.130s	0.0% - 8.3% = -8.3%	0.0% - 100.0% = -100.0%	Not changed, 0.0%	undefined | AVX(256); AVX2(256); AVX512F_256(256); FMA(256)

*N/A is applied to non-summable metrics.

ポータブルバイナリーは利用可能な最上位の命令セットを使用し、ターゲットシステムで最適なパフォーマンスを得ることができました。

設定レシピ

特定のコード環境でパフォーマンス解析を行うため、システムとインテル® VTune™ プロファイラーと従来のインテル® VTune™ Amplifier を設定する方法を詳しく説明します。

- CPU と FPGA (インテル® Arria® 10 GX) の相互作用を解析する
  このレシピは、インテル® Arria® 10 GX FPGA を例として、CPU と FPGA の相互作用を解析するためプラットフォームを設定する方法を説明します。

- .NET Core アプリケーションのプロファイル
  このレシピは、インテル® VTune™ Amplifier を使用して .NET Core ダイナミックコードをプロファイルし、マネージドコードの hotspot を特定してパフォーマンスが向上するようにアプリケーションを最適化します。

- Amazon Web Services* (AWS*) EC2* インスタンス上のアプリケーションのプロファイル
  このレシピは、インテル® VTune™ プロファイラーを使用してパフォーマンスをプロファイルするため、AWS* で VM インスタンスを設定します。

- GitLab* CI でパフォーマンスをプロファイルする
  このレシピは、インテル® VTune™ プロファイラーを GitLab* CI パイプラインに統合して、ビルドをプロファイルする方法を説明します。

- ハードウェアベースの hotspot 解析向けに Hyper-V* 仮想マシンを設定する
  このレシピは、インテル® VTune™ プロファイラーを使用してハードウェア・パフォーマンスをプロファイルするため、Hyper-V* 環境で仮想マシン・インスタンスを設定します。

- パフォーマンス異常を見つけるアプリケーションのプロファイル
  このレシピは、インテル® VTune™ プロファイラーの異常検出解析を使用して、いくつかの要因によるパフォーマンス異常を特定する方法を紹介します。また、これらの異常を修正するための提案も提供します。

- GPU 上で実行する OpenMP* オフロード・アプリケーションのプロファイル
  このレシピは、インテル® GPU へオフロードされる OpenMP* アプリケーションをコンパイルする方法を示し、インテル® VTune™ プロファイラーで OpenMP* アプリケーションの GPU 解析 (HPC パフォーマンス特性、GPU オフロード、および GPU 計算/メディア・ホットスポット) を実行して、結果を調査する方法を紹介します。

- DPC++ アプリケーションのプロファイル
  このレシピは、DPC++ (データ並列 C++) アプリケーションを作成してコンパイルする方法を紹介します。また、インテル® VTune™ プロファイラーを使用して DPC++ アプリケーションの GPU 解析を実行し、結果を検証する方法も示します。

- コマンドライン・インターフェイスを使用して GPU 上で実行する DPC++ アプリケーションのパフォーマンスを解析
  インテル® VTune™ プロファイラーのコマンドライン・インターフェイス (CLI) を使用して、インテル® GPU にオフロードされたデータ並列 C++ (DPC++) アプリケーションのパフォーマンスを解析する方法を紹介します。また、収集したデータを使用してレポートをカスタマイズする方法も説明します。

- FPGA 上での DPC++ アプリケーションのプロファイル
  このレシピは、FPGA 上で DPC++ (データ並列 C++) アプリケーションをプロファイルします。このレシピでは、インテル® VTune™ プロファイラーの CPU/FPGA 相互作用解析タイプ (プレビュー機能) に統合されている AOCL プロファイラーを使用します。

- インテルのサンプリング・ドライバーを使用しないハードウェアのプロファイル
  この一連のレシピは、インテル® VTune™ プロファイラーでドライバーを使用しない Linux* perf ベースのパフォーマンス・プロファイルを設定して、その利点と制限に対する回避策を理解するのに役立ちます。

- MPI アプリケーションのプロファイル
  このレシピは、インテル® VTune™ Amplifier を使用して MPI アプリケーションのインバランスと通信の問題を特定し、アプリケーション・パフォーマンスを向上します。

- Node.js* の JavaScript* コードのプロファイル
  このレシピは、`Node.js*` をリビルドし、インテル® VTune™ プロファイラーを使用して、JavaScript* フレームとネイティブフレーム (ネイティブコード、例えば、JavaScript* コードから呼び出されたシステム・ライブラリーやネイティブ・ライブラリー) から成る混在モードのコールスタックを含む JavaScript* コードのパフォーマンスを解析するための設定手順を説明します。

- Docker* コンテナーでのプロファイル
  このレシピは、インテル® VTune™ Amplifier の解析向けに Docker* コンテナーを構成して、独立したコンテナー環境で動作しているアプリケーションの hotspot を特定します。

- プロキシーサーバー介したリモートターゲットのプロファイル
  このレシピは、プロキシーサーバーを介してインテル® VTune™ プロファイラーを実行し、リモートターゲットをプロファイルする方法を説明します。

- インテル® VTune™ プロファイラー・サーバーと Visual Studio* Code およびインテル® DevCloud for oneAPI の併用
  このレシピでは、インテル® VTune™ プロファイラーをウェブサーバーとして使用し、リモートの開発マシンでパフォーマンスのチューニングを行う方法を紹介します。例として、リモートマシンにインテル® DevCloud for oneAPI のコンピュート・ノードを使用します。

- Singularity* コンテナーでのプロファイル
  このレシピは、インテル® VTune™ Amplifier の解析向けに Singularity* コンテナーを構成して、独立したコンテナー環境で動作しているアプリケーションの hotspot を特定します。

- Linux*、Android*、および QNX* のシステムブート時のプロファイル
  このレシピは、インテル® VTune™ Amplifier のパフォーマンス解析を Linux*、Android*、および QNX* オペレーティング・システムのブートフローと統合する方法を示します。この解析は、OS ブート

時に CPU コアで予想外に長く実行されるアクティビティーを識別するのに役立ちます。これにより、ブート順序のさらに詳しい調査が可能になります。

- システム・アナライザーによるリアルタイム・モニタリング
  このレシピは、システム・アナライザーの概要を紹介し、ターゲットシステムをリアルタイムにモニタリングして、CPU、GPU、メモリー、ディスク、ネットワークによる制限を特定します。

CPU と FPGA (インテル® Arria® 10 GX) の相互作用を解析する

このレシピは、インテル® Arria® 10 GX FPGA を例として、CPU と FPGA の相互作用を解析するためプラットフォームを設定する方法を説明します。

コンテンツ・エキスパート: JONG IL P. (英語)、Vitaly Slobodskoy (英語)

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション**: 行列乗算 OpenCL* アプリケーション。行列乗算サンプル・アプリケーションは、インテル® FPGA SDK for OpenCL* ウェブサイトからダウンロードできます。
- **ツール**: インテル® FPGA SDK for OpenCL*、インテル® VTune™ Amplifier 2019 以降

注

    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム**: CentOS* 7、Red Hat* Enterprise Linux* 7 以降
- **CPU**: インテル® サーバー・プラットフォーム (開発コード名 Skylake)
- **FPGA**: インテル® Arria® 10 GX

インテル® Arria® 10 GX FPGA とインテル® FPGA SDK for OpenCL* を設定する

1. インテル® Arria® 10 GX FPGA で DIP スイッチを設定して、電源と USB ケーブルを接続します。詳細な手順 (英語) をご覧ください。
2. **インテル® FPGA SDK for OpenCL* (コードビルダー、インテル® Quartus® Prime ソフトウェア、デバイスを含む)** を http://fpgasoftware.intel.com/opencl/ (英語) からダウンロードします。
3. `setup_pro.sh` ファイルを実行して SDK をインストールします。
4. `source init_opencl.sh` を実行して適切な環境変数を設定します。

5. `aocl version` を実行して正しくインストールされたことを確認します。次のようなメッセージが出力されます。

```
aocl 17.1.0.240 (Intel(R) FPGA SDK for OpenCL(TM), Version 17.1.0 Build
240, Copyright (C) 2017 Intel Corporation)
```

6. `aocl install` を実行して FPGA ボードをインストールします。
7. `aocl diagnose` を実行してハードウェアが正しくインストールされたことを確認します。次のようなメッセージが出力されます。

```
Device Name:
acl0

Package Pat:
/home/tce/intelFPGA_pro/17.1/hld/board/a10_ref

Vendor: Intel(R) Corporation

Phys Dev Name  Status   Information

acla10_ref0   Passed   Arria 10 Reference Platform (acla10_ref0)
                        PCIe dev_id = 2494, bus:slot.func = 44:00.00,
Gen3 x4
                        FPGA temperature = 44.3555 degrees C.

DIAGNOSTIC_PASSED
```

サンプル・アプリケーションをビルドして FPGA へフラッシュする

1. デフォルトの `makefile` で `make` を実行してホストの実行ファイルをビルドします。生成される実行ファイルの名前は `host` です。
2. 次のコマンドで FPGA 向けのバイナリーをビルドします。

```
aoc -v -board=a10gx device/matrix_mult.cl -o bin/ matrix_mult.aocx
```

3. フラッシュの USB ドライバーを設定します。
    a. 次のコマンドを実行します。

    ```
    sudo vim /etc/udev/rules.d/51-usbblaster.rules
    ```

    次の行を追加します。

    ```
    # usb blaster
    SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device",
    ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", MODE="0666",
    NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod
    0666 %c"
    SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device",
    ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6002", MODE="0666",
    NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod
    0666 %c"
    SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device",
    ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6003", MODE="0666",
    ```

```
NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod
0666 %c"
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device",
ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6010", MODE="0666",
NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod
0666 %c"
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device",
ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6810", MODE="0666",
NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod
0666 %c"
```

4. 次のコマンドで JTAG のクロック速度を 6MHz に設定します。

   ```
   jtagconfig --setparam 1 JtagClock 6M
   ```

5. 次のコマンドでバイナリーを FPGA へフラッシュします。

   ```
   aocl flash acl0 ./bin/matrix_mult.aocx
   ```

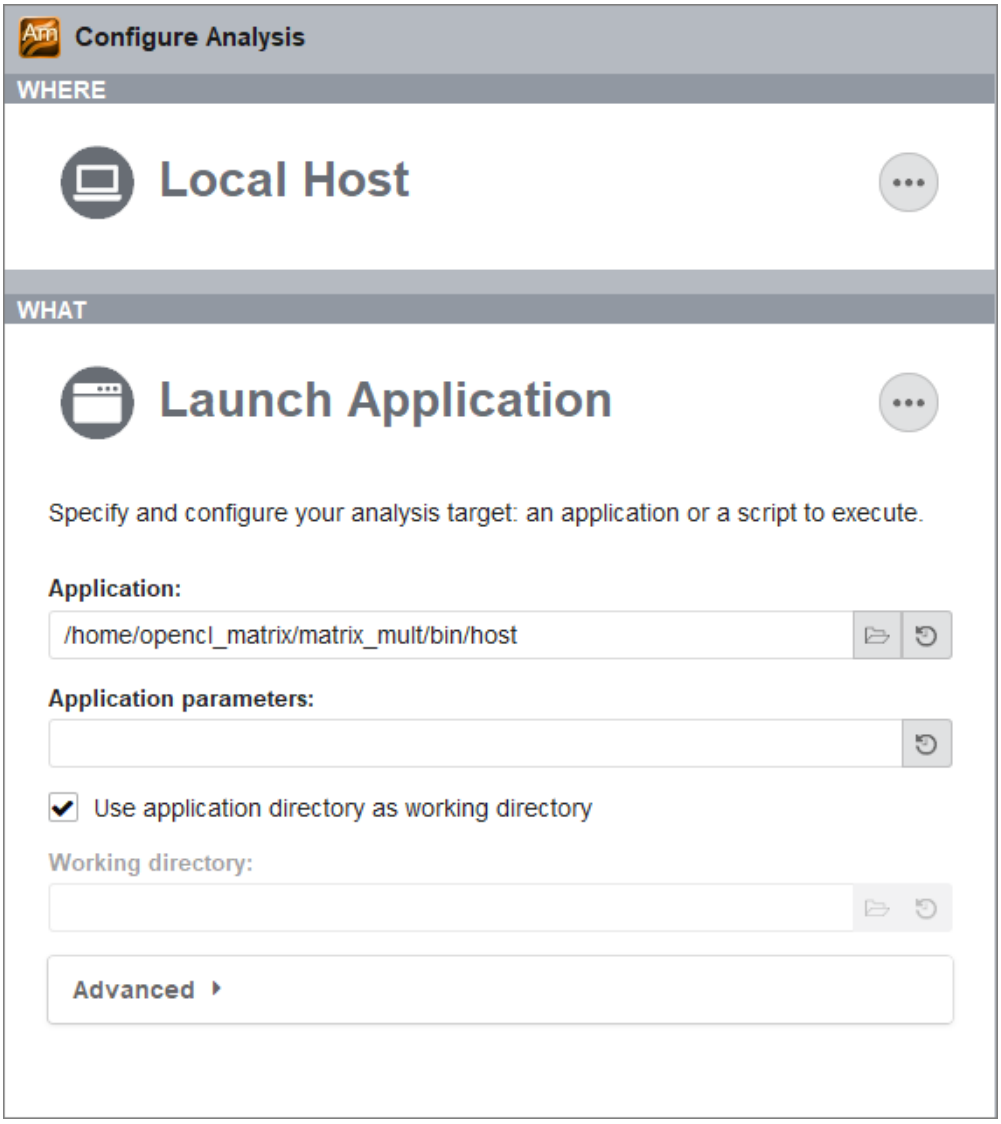
6. FPGA を搭載したホストシステムを再起動します。

CPU/FPGA 相互作用解析を実行する

1. インテル® VTune™ Amplifier を起動します。次に例を示します。

   ```
   /opt/intel/vtune_amplifier_2019/bin64/amplxe-gui
   ```

2. 解析用のプロジェクト (例: `hello_world_opencl`) を作成します。
3. **[Configure Analysis (解析の設定)]** をクリックして新しい解析を開始します。
4. **[CPU/FPGA Interaction (CPU/FPGA 相互作用)]** 解析を設定します。

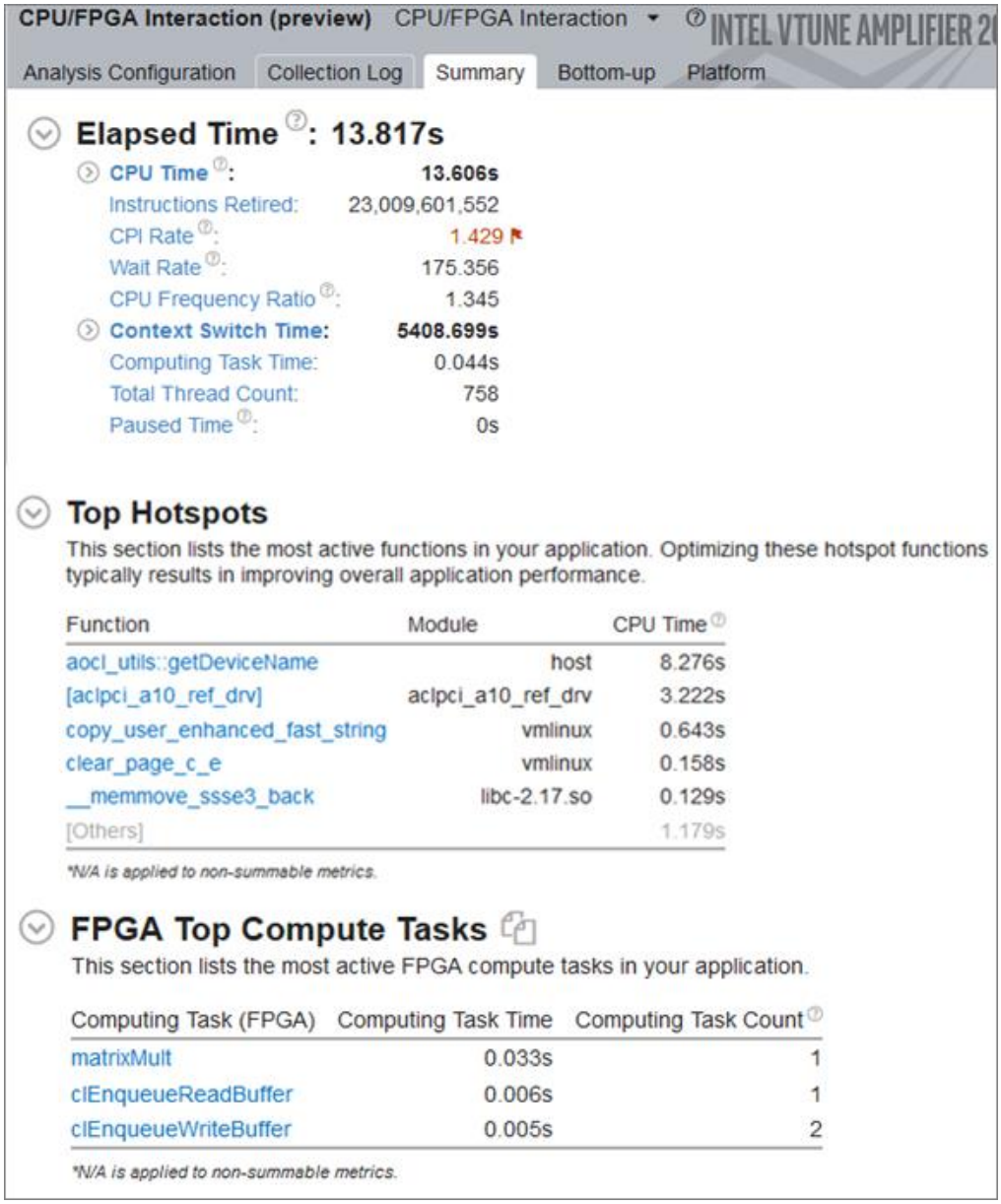


a. **[WHERE (どこを)]** ペインでは、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
b. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、`hello world` アプリケーションを指定します。通常、このアプリケーションは *<sample app>*/bin/host にあります。
c. **[HOW (どのように)]** ペインで、解析タイプから **[CPU/FPGA Interaction (CPU/FPGA 相互作用)]** を選択します。

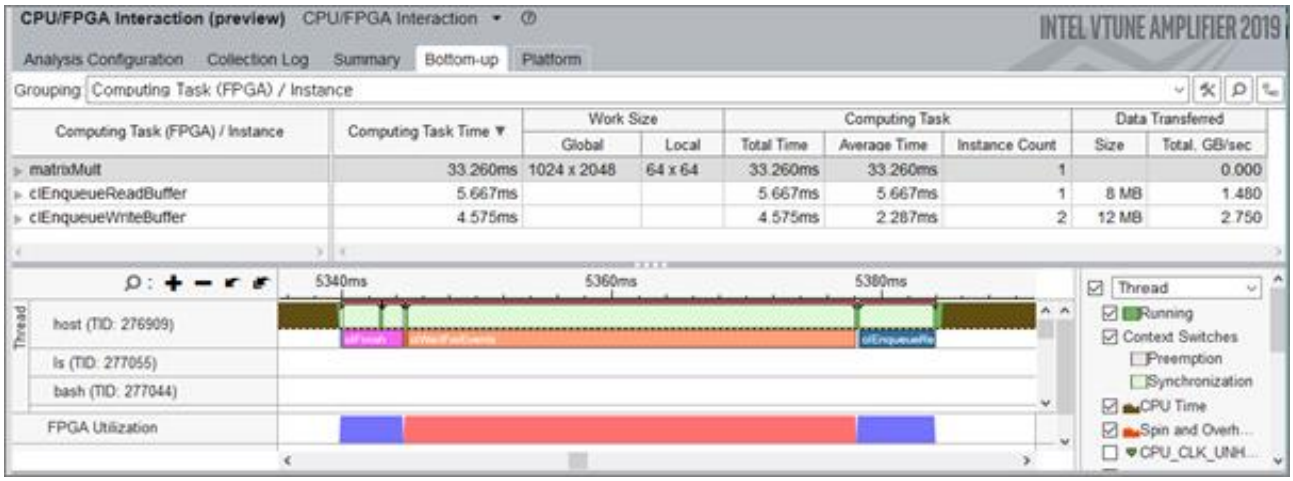
5. **[Start (開始)]** をクリックして解析を開始します。

結果を解釈する

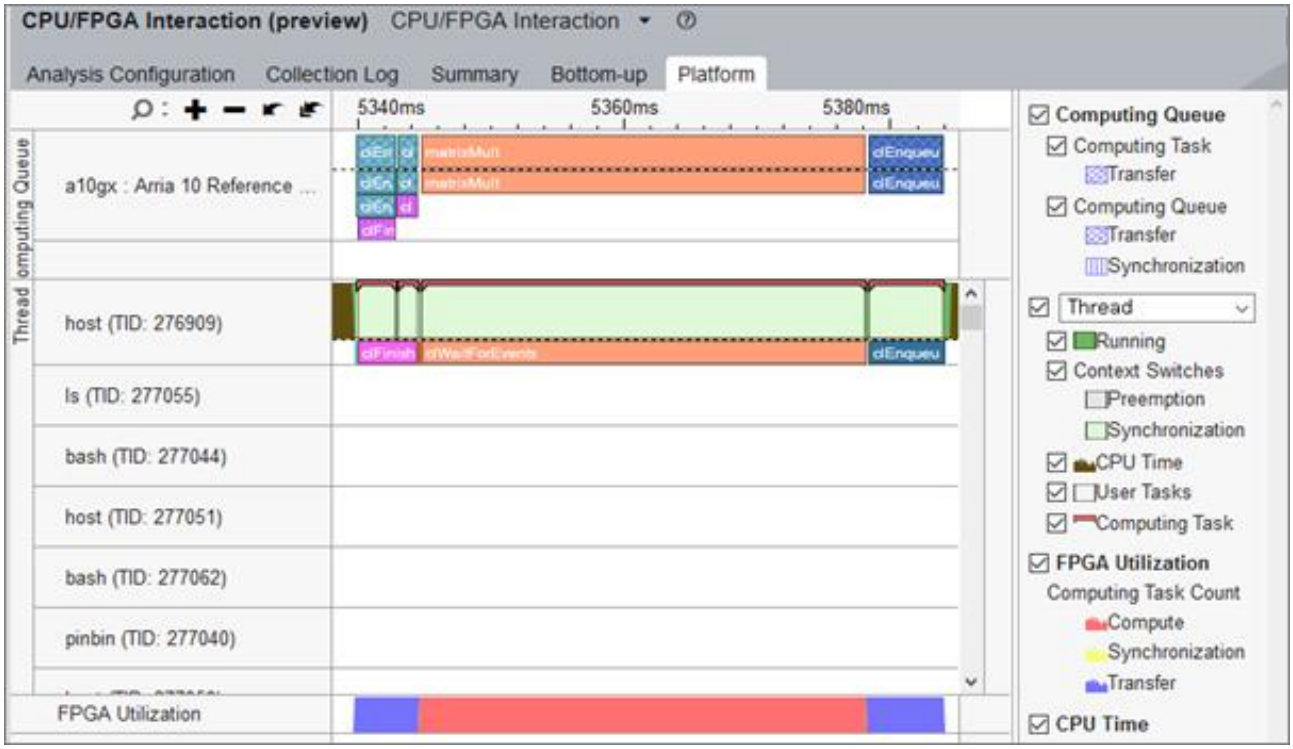
データ収集が完了すると、結果がファイナライズされ **[CPU/FPGA Interaction (CPU/FPGA 相互作用)]** ビューポイントに表示されます。FPGA の上位の計算タスクや CPU の上位のタスクと hotspot が表示される **[Summary (サマリー)]** タブから始めます。



[Bottom-up (ボトムアップ)] タブに切り替えて計算タスクのワークサイズとデータ転送スループットを確認します。[Timeline (タイムライン)] ペインで計算タスクとデータ転送の FPGA 使用率を確認します。



[Platform (プラットフォーム)] タブで FPGA とホスト・アプリケーションの計算キューを確認します。各転送と同期の開始時間と継続期間に関する情報も表示されます。



関連情報

- CPU/FPGA 相互作用解析 (英語)

.NET Core アプリケーションのプロファイル

このレシピは、インテル® VTune™ Amplifier を使用して .NET Core ダイナミックコードをプロファイルし、マネージドコードの hotspot を特定してパフォーマンスが向上するようにアプリケーションを最適化します。

コンテンツ・エキスパート: Denis Pravdin (英語)

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** 整数リストのすべての要素を加算するサンプル C# アプリケーション。このアプリケーションはデモ用であり、ダウンロードすることはできません。
- **ツール:**
    - インテル® VTune™Amplifier 2018

注

        - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
        - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
        - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
    - .NET Core 2.0 SDK (英語)
- **オペレーティング・システム:** Microsoft* Windows* 10
- **CPU:** インテル® プロセッサー (開発コード名 Skylake)

アプリケーションを準備する

1. .NET 環境変数が設定された新しいコマンドウィンドウを開き、.NET Core 2.0 が正しくインストールされていることを確認します。

    ```
    dotnet --version
    ```

2. アプリケーションの新しい `listadd` ディレクトリーを作成します。

```
mkdir C:\listadd
> cd C:\listadd
```

3. `dotnet new console` と入力して、次の構造の新しいスケルトン・プロジェクトを作成します。

```
C:\listadd>dir
 Volume in drive C has no label.
 Volume Serial Number is 0A4A-D42D

 Directory of C:\listadd

09/20/2017  04:09 PM    <DIR>          .
09/20/2017  04:09 PM    <DIR>          ..
09/20/2017  04:08 PM               178 listadd.csproj
09/20/2017  04:08 PM    <DIR>          obj
09/20/2017  04:08 PM               189 Program.cs
               2 File(s)            367 bytes
               3 Dir(s)  36,955,869,184 bytes free

C:\listadd>
```

4. `listadd` フォルダーの `Program.cs` の内容を、整数リストの要素を加算する C# コードに変更します。

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace listadd
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Starting calculation...");
            List<int> numbers = Enumerable.Range(1,10000).ToList();
            for (int i =0; i < 100000; i ++)
            {
                ListAdd(numbers);
            }

            Console.WriteLine("Calculation complete");
        }

        static int ListAdd(List<int> candidateList)
        {
            int result = 0;
            foreach (int item in candidateList)
            {
                result += item;
            }

            return result;
        }
```

```
    }
}
```

5. `listadd.csproj` ファイルの `PropertyGroup` セクションに `<DebugType>pdbonly</DebugType>` フラグを追加してインテル® VTune™ Amplifier のソースコード解析を有効にします。
6. `C:\listadd\bin\Release\netcoreapp2.0` フォルダーに `listadd.dll` を作成します。

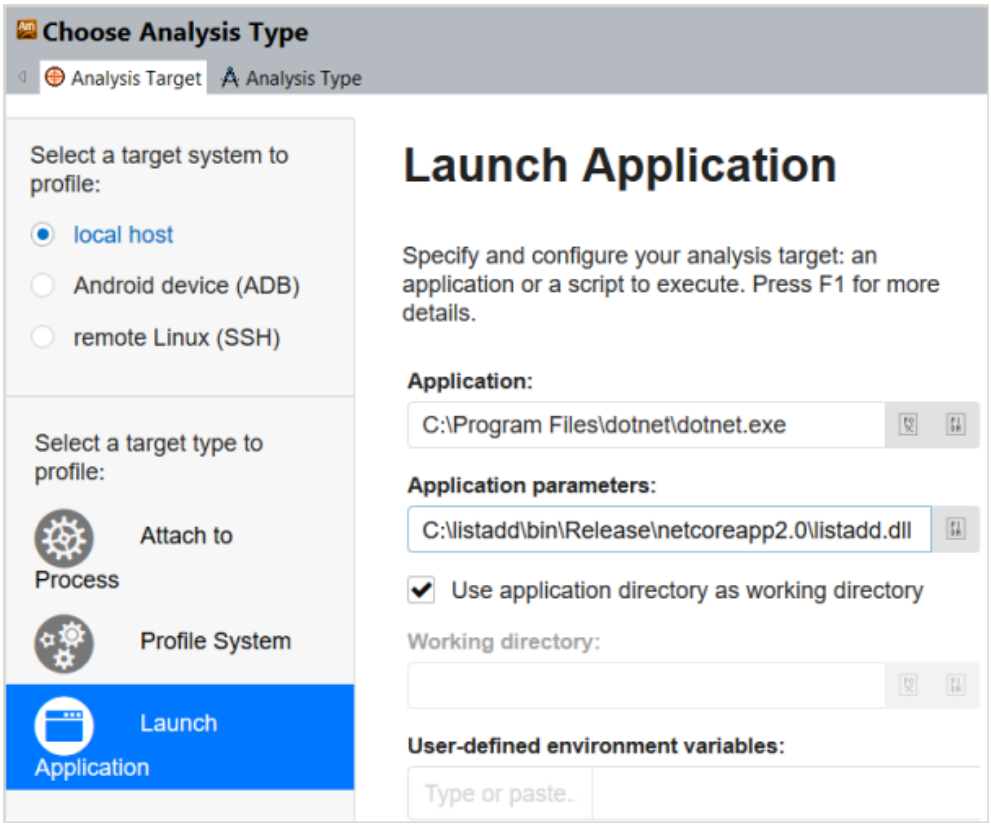
   ```
   dotnet build -c Release
   ```

7. サンプル・アプリケーションを実行します。

   ```
   dotnet C:\listadd\bin\Release\netcoreapp2.0\listadd.dll
   ```

高度な hotspot 解析を実行する

1. 管理者権限でインテル® VTune™ Amplifier を起動します。
2. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `dotnet`) を指定します。
3. **[Analysis Target (解析ターゲット)]** ウィンドウで、左ペインから **[local host (ローカルホスト)]** および **[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択します。
4. **[Launch Application (アプリケーションを起動)]** ペインで、解析するアプリケーションを指定します。
   - **アプリケーション:** `C:\Program Files\dotnet\dotnet.exe`
   - **アプリケーションのパラメーター:** `C:\listadd\bin\Release\netcoreapp2.0\listadd.dll`



注

`dotnet.exe` の場所は環境変数に依存します。`where dotnet` コマンドを使用して確認できます。

5. 右の **[Choose Analysis (解析の選択)]** ボタンをクリックして、左ペインから **[Advanced Hotspots (高度な hotspot)]** 解析を選択します。

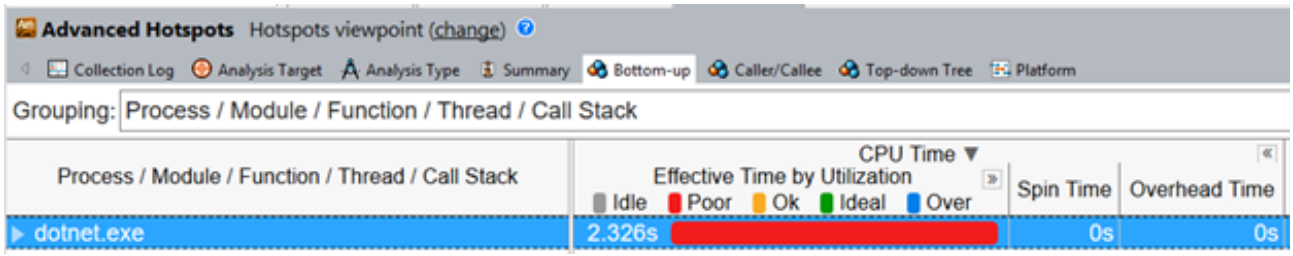
注

高度な hotspot 解析は、インテル® VTune™ Amplifier 2019 で汎用の hotspot 解析 (英語) に統合されました。ハードウェア・イベントベース・サンプリング収集モードで利用できます。

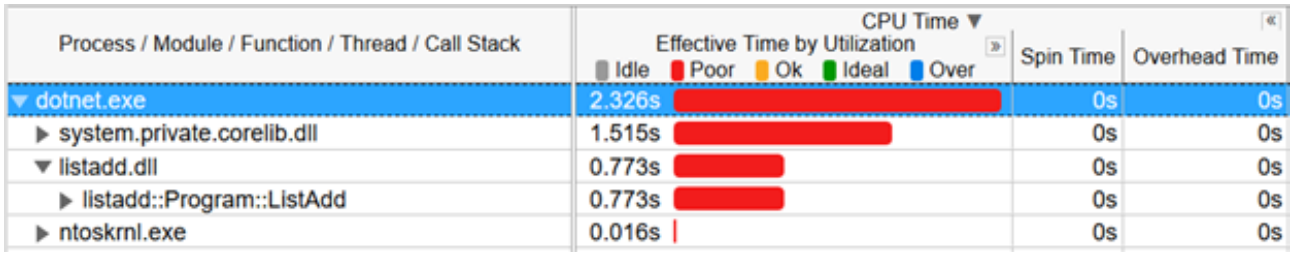
6. **[Start (開始)]** をクリックして解析を開始します。

マネージドコードの hotspot を特定する

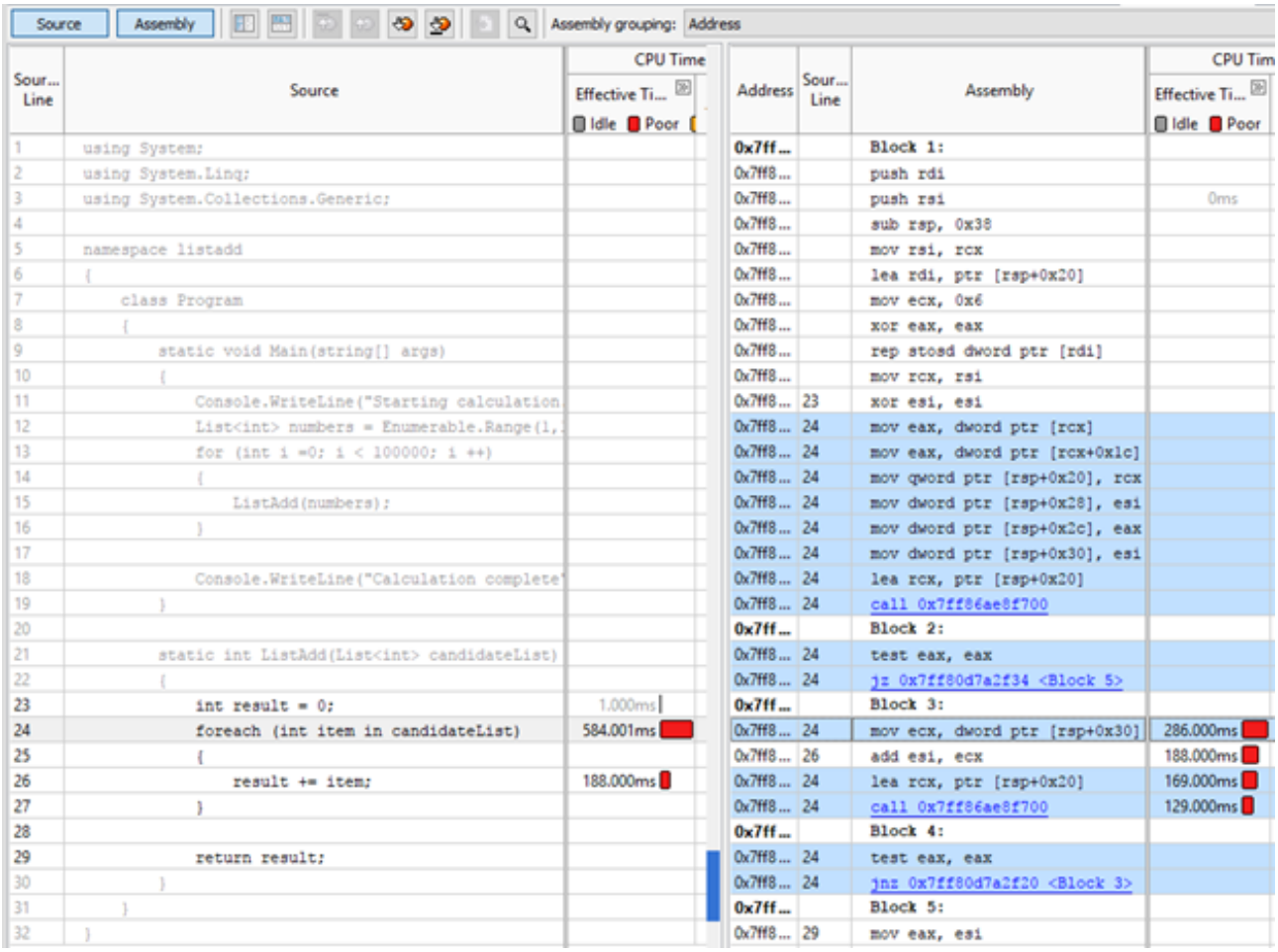
収集した解析結果が表示されたら、**[Bottom-up (ボトムアップ)]** タブに切り替えて **[Process/Module/Function/Thread/Call Stack (プロセス/モジュール/関数/スレッド/コールスタック)]** グループを選択します。



`dotnet.exe` > `listadd.dll` を展開して、最も CPU 時間がかかっているマネージド `listadd::Program::ListAdd` 関数を見つけます。



この hotspot 関数をダブルクリックして [Source (ソース)] ビューを開きます。ソースと逆アセンブルしたコードを並べて表示するには、ツールバーの **[Assembly (アセンブリー)]** ボタンをクリックします。



ソース行/アセンブリー命令ごとの統計を使用して、最も時間を費やしているコード部分 (上記の例では行 24) を特定します。

ループ交換を使用してコードを最適化する

インテル® VTune™ Amplifier は、次のコード行をパフォーマンス・クリティカルとしてハイライトします。

```
foreach (int item in candidateList)
```

`for` ループ文を使用してコードを最適化します。`Program.cs` の内容を次のように変更します。

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace listadd
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Starting calculation...");
            List<int> numbers = Enumerable.Range(1,10000).ToList();
            for (int i =0; i < 100000; i ++)
            {
```

```
                ListAdd(numbers);
            }

            Console.WriteLine("Calculation complete");
        }

        static int ListAdd(List<int> candidateList)
        {
            int result = 0;
            for (int i = 0; i < candidateList.Count; i++)
            {
                result += candidateList[i];
            }

            return result;
        }
    }
}
```

最適化を確認する

更新したコードの最適化を確認するため、高度な hotspot 解析を再度実行します。

最適化前は、サンプル・アプリケーションの CPU 時間は 2.636 秒でした。

Process / Module / Function / Thread / Call Stack	CPU Time ▼		
	Effective Time	Spin Time	Overhead Time
▼ listadd.dll	2.636s	0s	0s
▶ system.private.corelib.dll	1.876s	0s	0s
▼ listadd.dll	0.701s	0s	0s
▶ listadd::Program::ListAdd	0.700s	0s	0s
▶ listadd::Program::Main	0.001s	0s	0s
▶ ntoskrnl.exe	0.030s	0s	0s

最適化後は、サンプル・アプリケーションの CPU 時間が 0.945 秒になり、オリジナルから約 64% パフォーマンスが向上しました。

Process / Module / Function / Thread / Call Stack	CPU Time ▼		
	Effective Time	Spin Time	Overhead Time
▼ listadd.dll	945.215ms	0ms	0ms
▼ listadd.dll	878.058ms	0ms	0ms
▶ listadd::Program::ListAdd	878.058ms	0ms	0ms
▶ listadd::Program::Main	0ms	0ms	0ms
▶ ntoskrnl.exe	23.054ms	0ms	0ms

注

このレシピの情報は、デベロッパー・フォーラム (英語) を参照してください。

関連情報

- .NET コード解析 (英語)

Amazon Web Services* (AWS*) EC2* インスタンス上のアプリケーションのプロファイル

このレシピは、インテル® VTune™ プロファイラーを使用してパフォーマンスをプロファイルするため、AWS* で VM インスタンスを設定します。

コンテンツ・エキスパート: Denis Pravdin (英語)、Jennifer Dimatteo

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** `matrix`。このアプリケーションはデモ用であり、ダウンロードすることはできません。
- **ツール**: インテル® VTune™ プロファイラーまたはインテル® VTune™ Amplifier 2019 - hotspot 解析

注

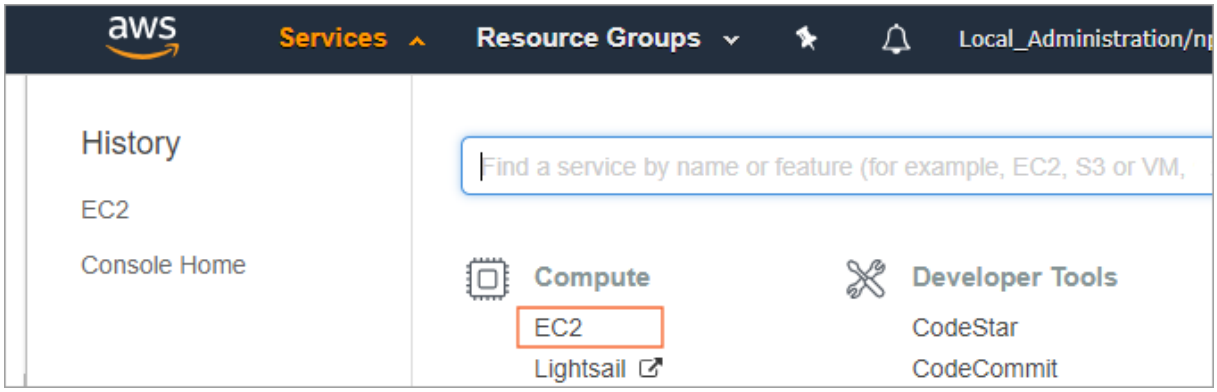
- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
- ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。

仮想マシン・インスタンスを作成して設定する

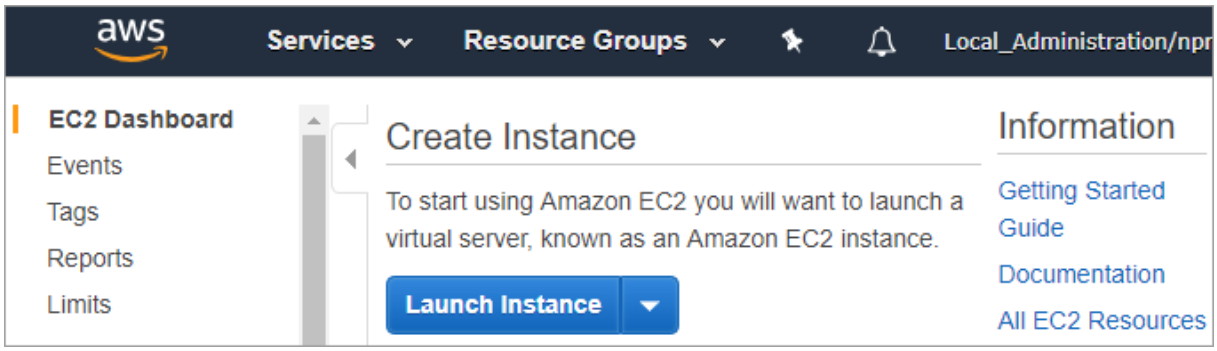
1. AWS* サイトで **[アカウント (Account)] > [AWS マネジメントコンソール (AWS Management Console)]** からアカウントにログインします。

   アカウントをお持ちでない場合は、作成してください。

2. Amazon EC2* ダッシュボードで **[Compute (コンピューティング)] > [EC2]** を選択します。

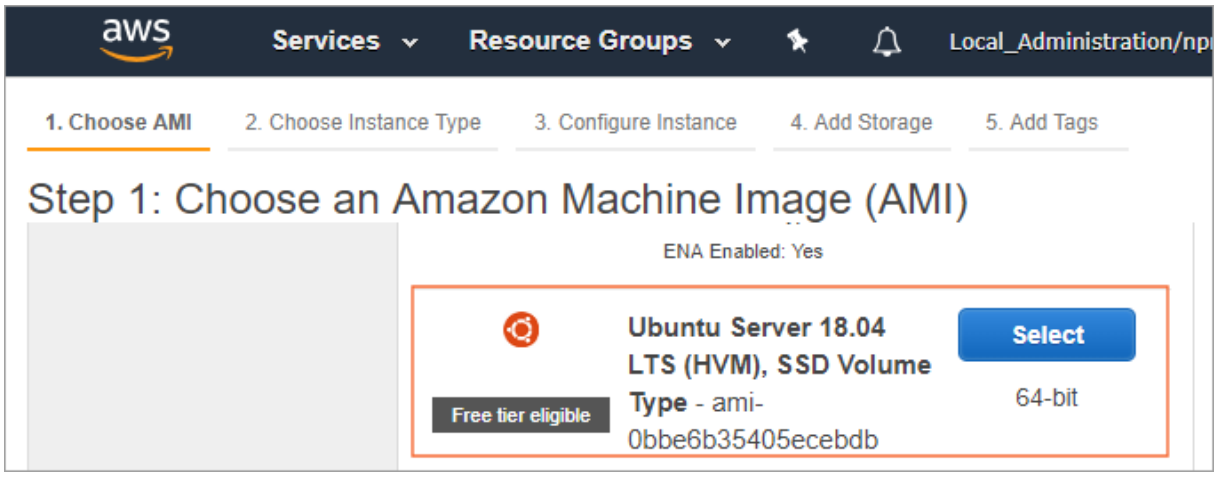


3. **[Launch Instance (インスタンスの作成)]** をクリックして、ウィザードに従って仮想マシンを作成し設定します。

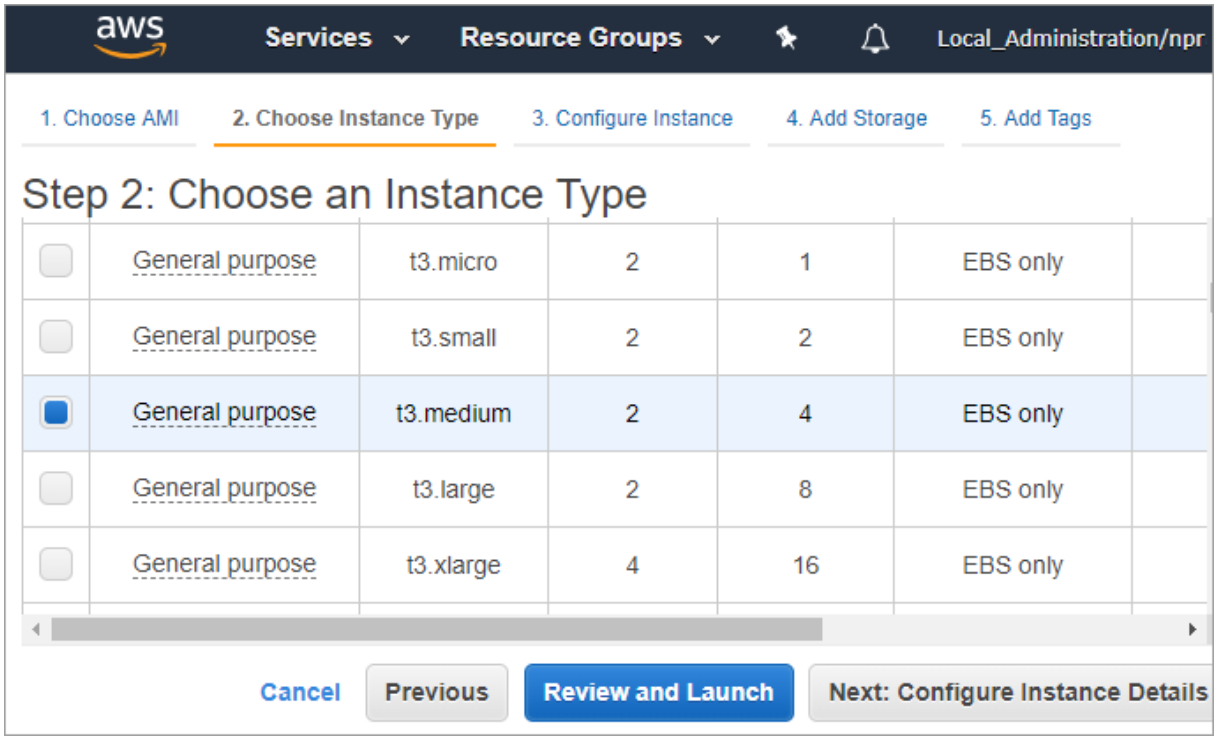


設定ウィザードでは、使用するケースに応じて OS とインスタンスの仕様 (CPU の数、メモリー、ストレージ、ネットワーク容量) を指定できます。

4. **[Step 1: Choose an Amazon Machine Image (AMI) (ステップ 1: Amazon マシンイメージ (AMI))]** では、Ubuntu* Server 18.04 LTS を選択します。



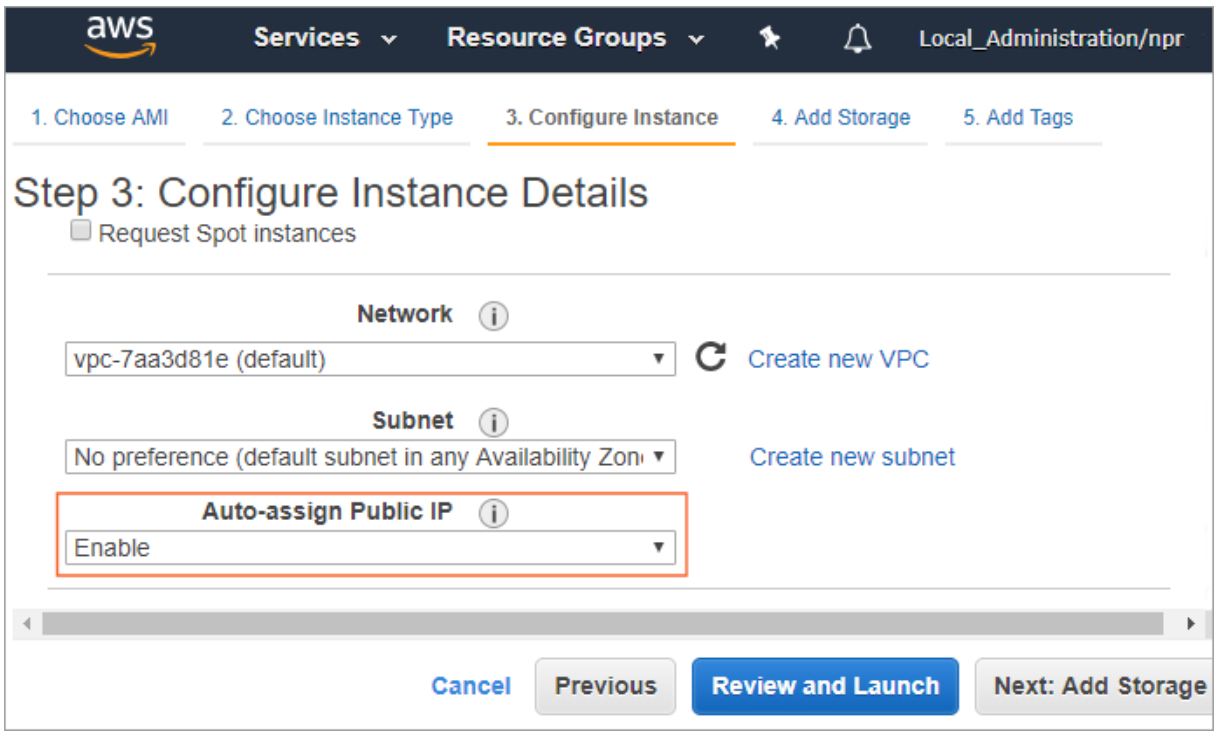
5. **[Step 2: Choose an Instance Type (ステップ 2: インスタンスタイプの選択)]** では、インスタンスを選択して **[Next: Configure Instance Details (次の手順: インスタンスの詳細の設定)]** をクリックします。



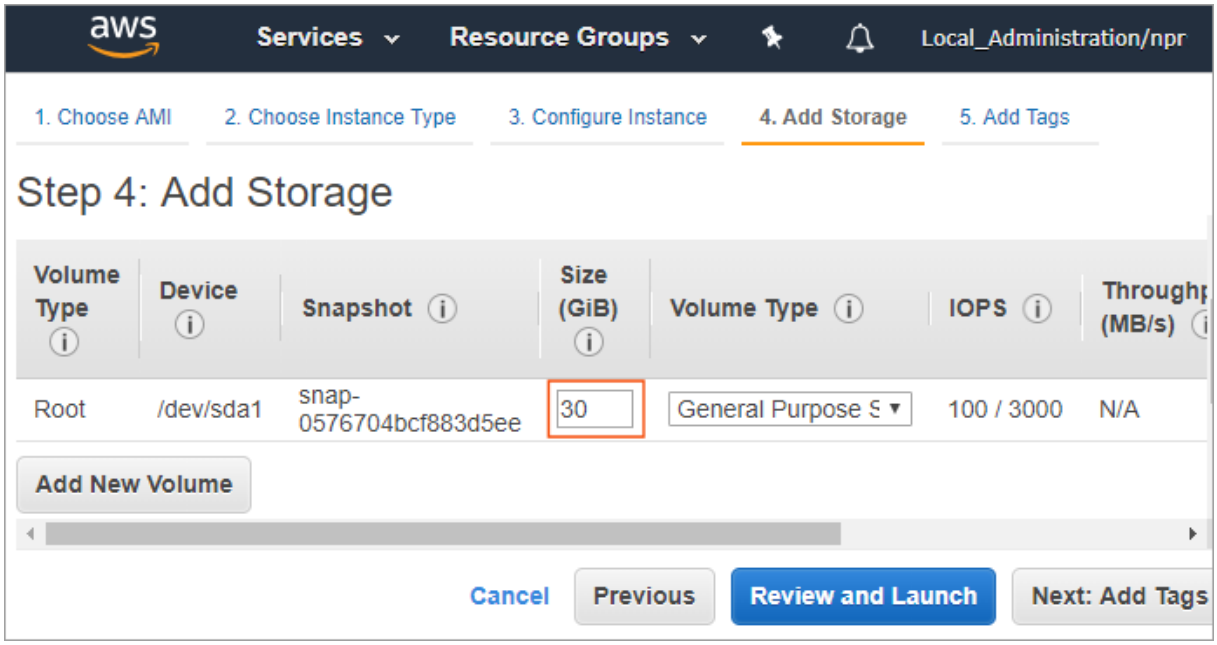
注

インテル® VTune™ プロファイラーの推奨システム要件は 4GB RAM、空きディスク容量 10GB 以上です。

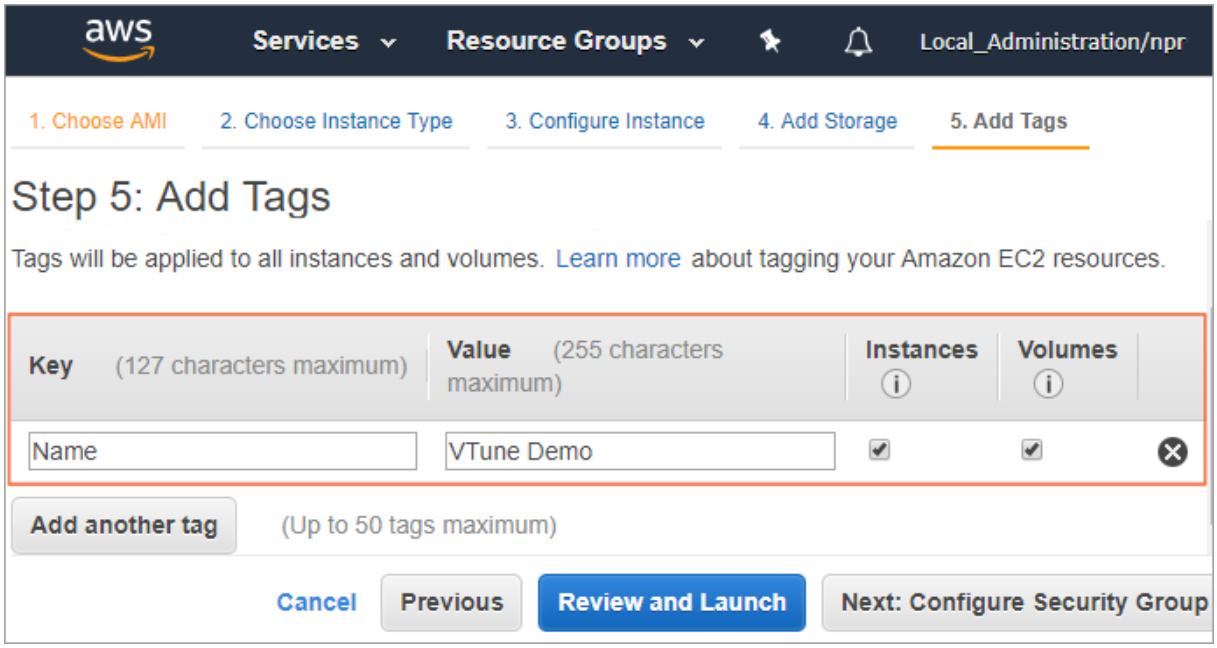
6. **[Step 3: Configure Instance Details (ステップ 3: インスタンスの詳細の設定)]** では、**[Auto-assign Public IP (自動割り当てパブリック IP)]** を有効にして **[Next: Add Storage (次の手順: ストレージの追加)]** をクリックします。



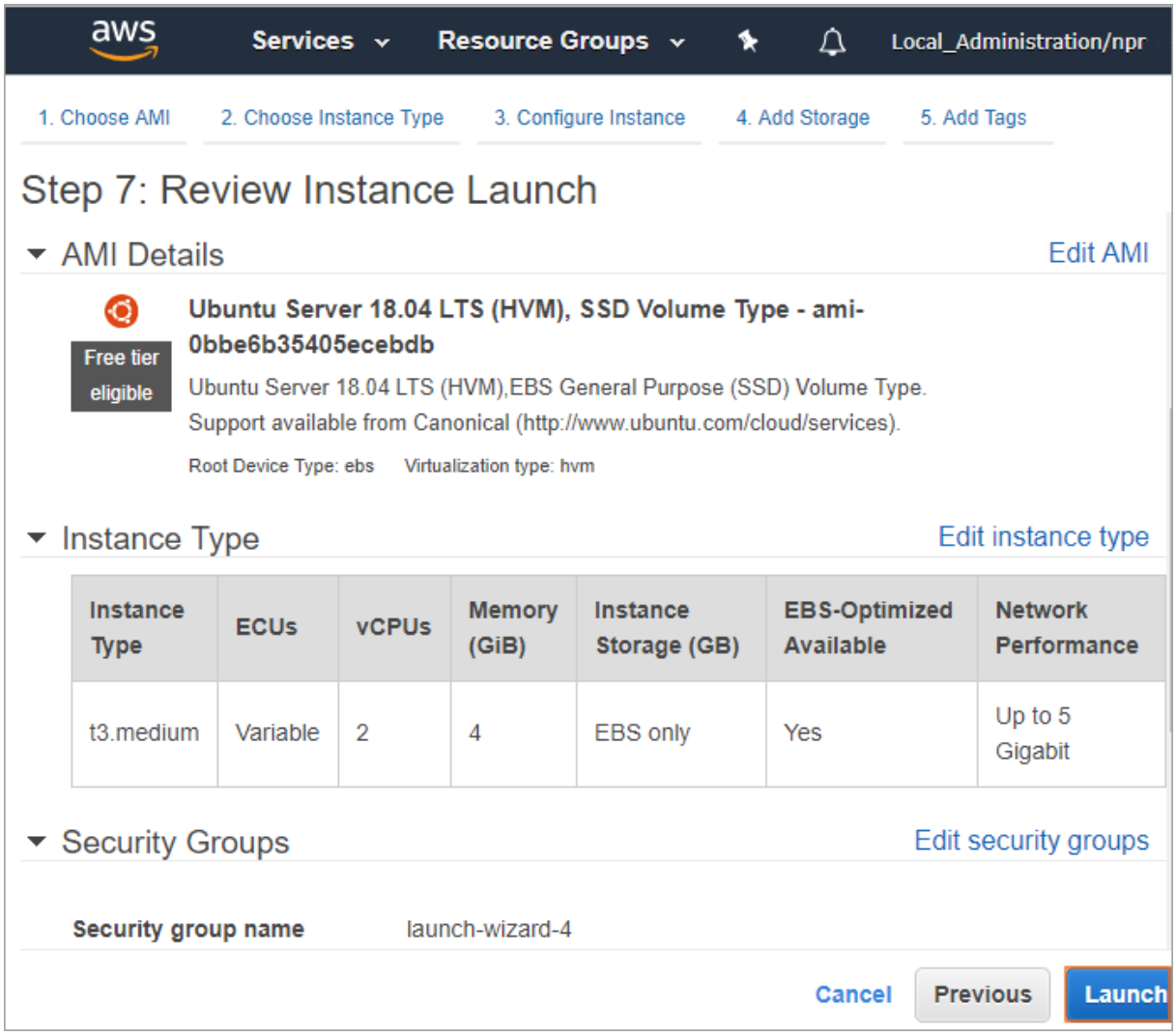
7. **[Step 4: Add Storage (ステップ 4: ストレージの追加)]** では、ストレージデバイスの容量を増やしてから **[Next: Add Tags (次の手順: タグの追加)]** をクリックします。



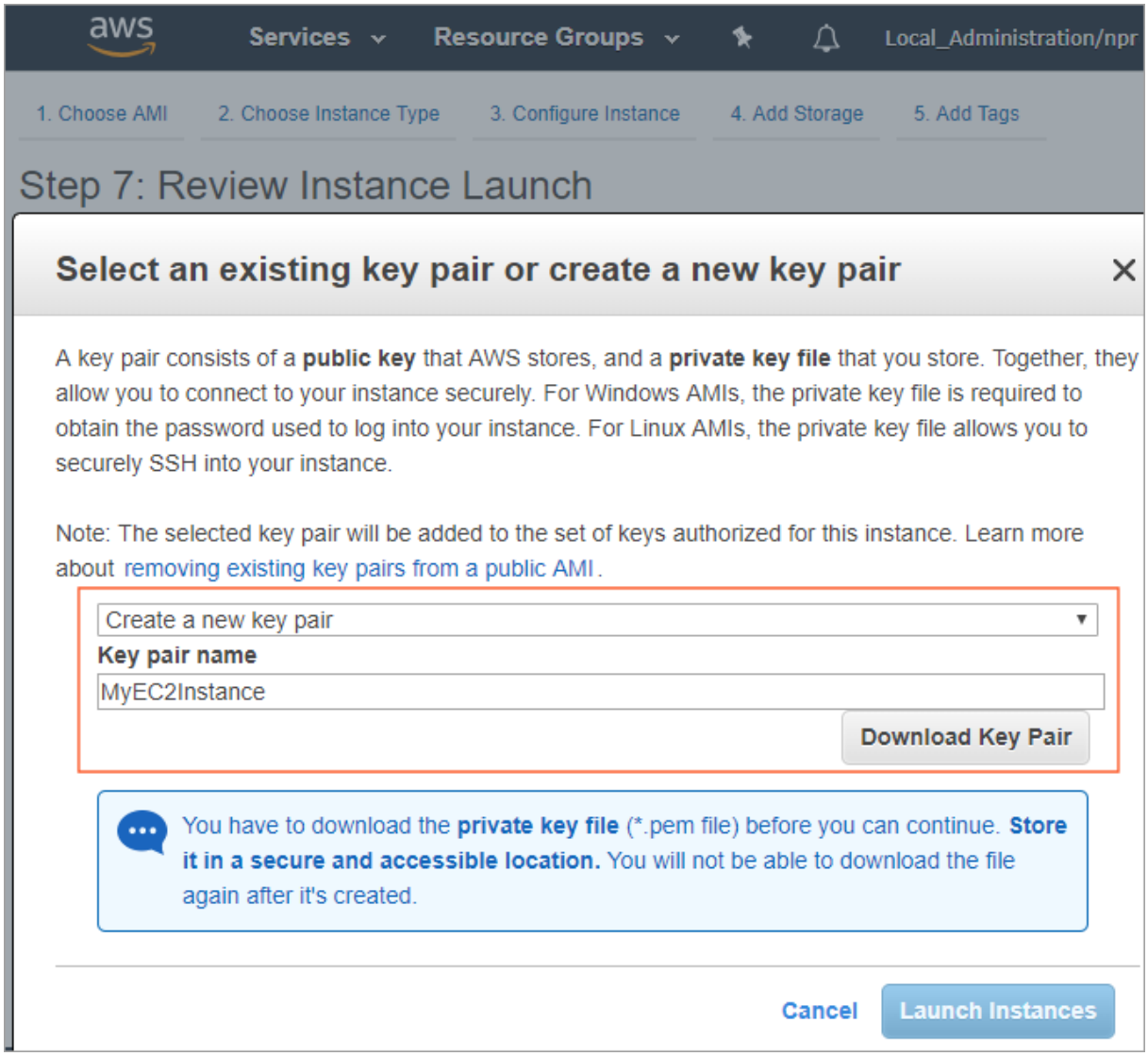
8. (オプション) **[Step 5: Add Tags (ステップ 5: インスタンスのタグ付け)]** では、タグ (例えば **VTune Demo**) を追加して **[Review and Launch (確認と作成)]** をクリックします。



9. **[Step 7: Review Instance Launch (ステップ 7: インスタンス作成の確認)]** では、**[Launch (作成)]** をクリックします。



10. キーペアを作成します。

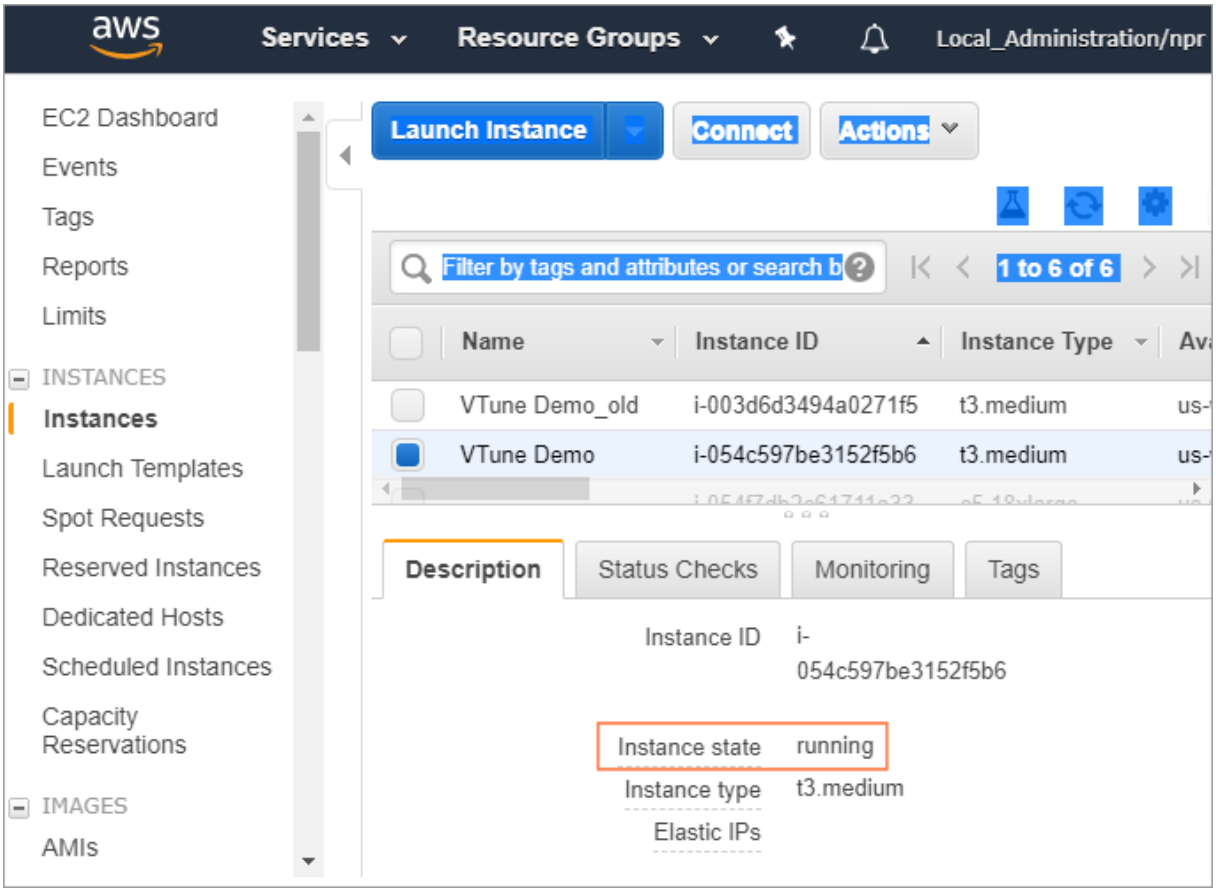


a. **[Create a new key pair (新しいキーペアの作成)]** を選択して、名前 (例えば MyEC2Instance) を割り当てます。
b. **[Download Key Pair (キーペアのダウンロード)]** をクリックして、キーをダウンロードします。
c. **[Launch Instances (インスタンスの作成)]** をクリックして、セットアップを完了します。

   インスタンスが作成されます。

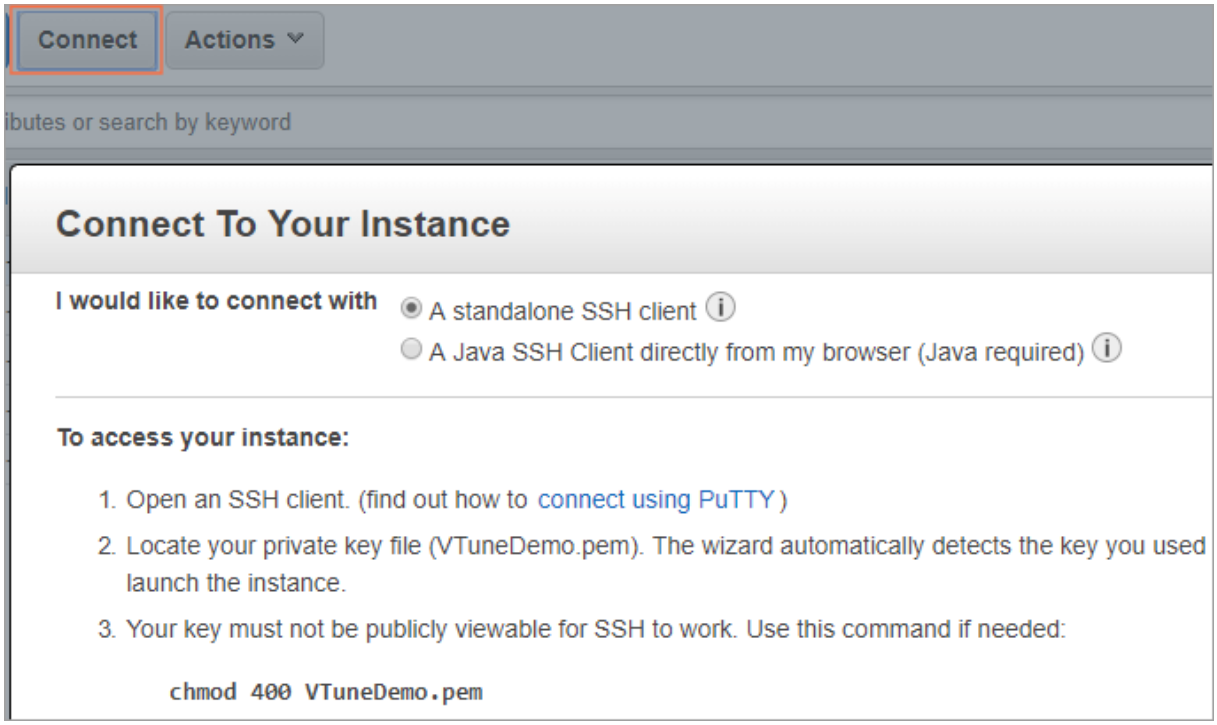
11. **[View Instances (インスタンスの表示)]** をクリックします。

    [Instance State (インスタンスの状態)] が **[running]** に更新されたら次のステップに進みます。



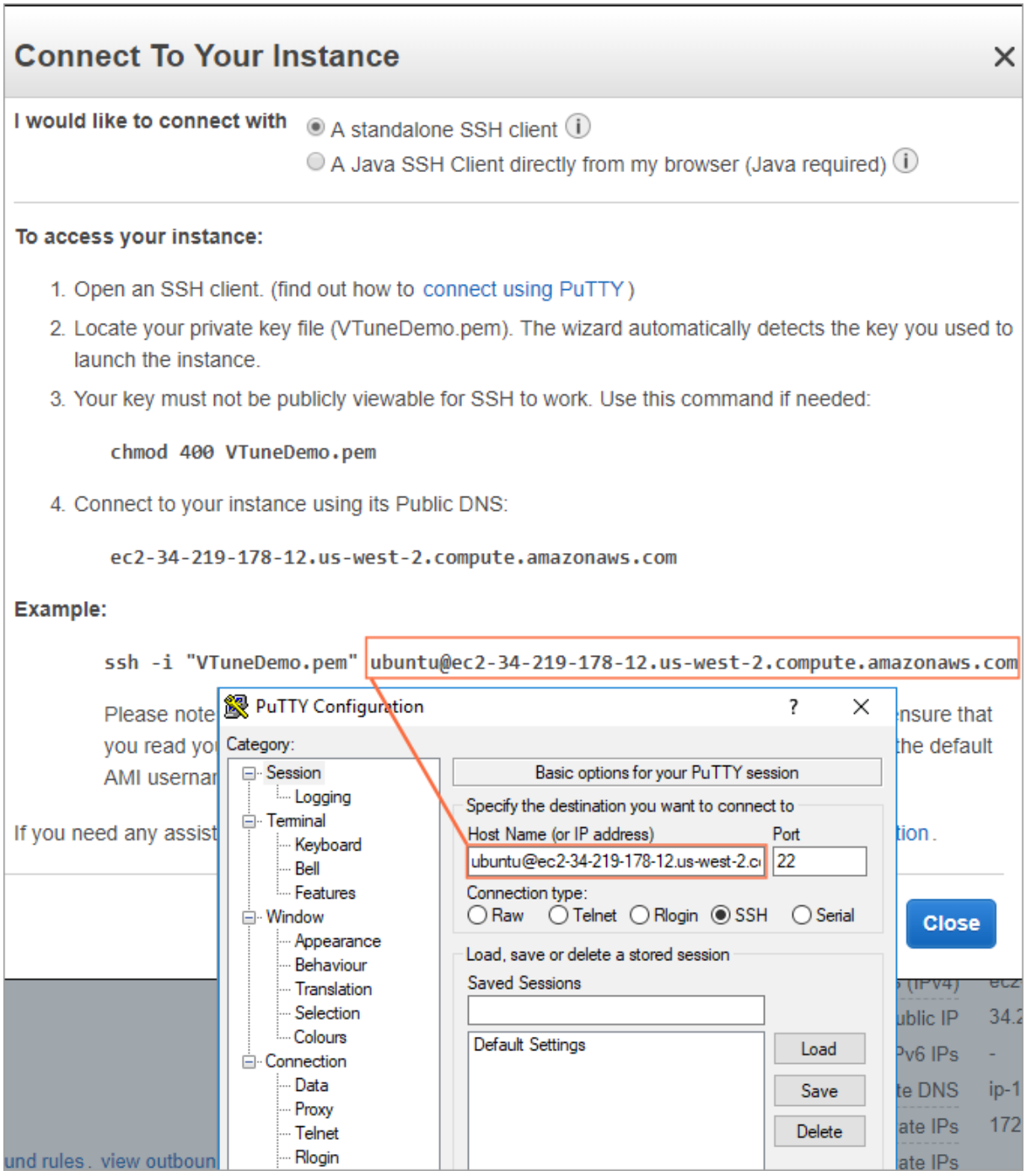
EC2* インスタンスに接続する

作成した EC2* インスタンスを選択し、**[Connect (接続)]** をクリックしてインスタンスへ SSH 接続します。



PuTTY* は、Amazon EC2* によって生成されるプライベート・キー形式 (.pem) をネイティブにサポートしません。PuTTY* を使用してインスタンスに接続する前に、PuTTYgen を使用してプライベート・キー (.pem) を必要な PuTTY* 形式 (.ppk) に変換する必要があります。

1. プライベート・キーを変換します。
    a. PuTTYgen を起動します。
    b. **[Type of key to generate]** で **[SSH-1 (RSA)]** を選択します。
    c. **[Load]** をクリックして .pem ファイルを選択します。
    d. **[Save private key]** をクリックして PuTTY* が使用可能な形式でキーを保存します。
2. PuTTY* を起動して、**[Connection] > [SSH] > [Auth]** でインスタンスのパブリック DNS とプライベート・キー (.ppk) ファイルを指定します。



3. 設定を保存して、**[Open]** をクリックします。

   インスタンスにログインします。

注

実行中のインスタンスに Xubuntu* デスクトップをインストールする必要があります。VNC* 経由でシステムにアクセスする場合は、VNC* サーバーのセットアップも必要です。

プロファイル向けにインスタンスを設定する

`/proc/sys/kernel/yama/ptrace_scope` を 0 に設定して、プロファイル向けにターゲット・インスタンスを準備します。

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

hotspot 解析を実行する

次の任意の方法で hotspot 解析を実行します。

SSH 経由でローカルにインストールされたインテル® VTune™ プロファイラーからリモート収集を実行:

注

リモート・ハードウェア・イベントベース・サンプリング解析は、ベアメタル・インフラストラクチャーとインテル® VTune™ プロファイラーのすべてのプロファイル機能に直接アクセス可能な EC2* ベアメタル・インスタンス (i3.metal インスタンス・ファミリー) を除き、AWS* EC2* 仮想環境では利用できません。

1. インテル® VTune™ プロファイラーでプロジェクトを作成します。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

2. **[WHERE (どこを)]** ペインでは、**[Remote Linux (SSH) (リモート Linux* (SSH))]** を選択して、**[SSH destination (SSH の対象)]** に `ubuntu@<PuTTY* 設定の名前>` と入力します。

   この時点でインテル® VTune™ プロファイラーはリモートシステムに接続して、収集に必要なバイナリーを指定されたフォルダーにインストールしようとします。

3. **[WHAT (何を)]** ペインでは、アプリケーションの場所と作業ディレクトリーを指定します。
4. **[HOW (どのように)]** ペインでは、デフォルトで表示される hotspot 解析の **[User-Mode Sampling (ユーザーモード・サンプリング)]** を選択します。
5.  をクリックして、解析を開始します。

   結果は、解析のため自動的にローカルシステムにコピーされます。

AWS* インスタンスから直接インテル® VTune™ プロファイラーを実行:

1. 「インストール・ガイド」(英語) に従って、インテル® VTune™ プロファイラーをインストールします。
2. インテル® VTune™ プロファイラーを実行します。次に例を示します。

   ```
   sudo <vtune_install_dir>/vtune_profiler/bin64/vtune-gui
   ```

3. プロジェクトを作成します。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

4. **[WHERE (どこを)]** ペインでは、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
5. **[WHAT (何を)]** ペインでは、アプリケーションの場所と作業ディレクトリーを指定します。
6. **[HOW (どのように)]** ペインでは、**[Hardware Event-Based Sampling (ハードウェア・イベントベース・サンプリング)]** など、hotspot 解析の収集モードを選択します。
7.  をクリックして、解析を開始します。

   解析結果は、デフォルトの [Hotspots by CPU Utilization (CPU 使用率によるホットスポット)] ビューポイントに表示されます。

関連情報

- 仮想環境のターゲット (英語)
- パフォーマンス解析 (英語)
- リモート収集向けに SSH アクセスを設定 (英語)

GitLab* CI でパフォーマンスをプロファイルする

このレシピは、インテル® VTune™ プロファイラーを GitLab* CI パイプラインに統合して、ビルドをプロファイルする方法を説明します。

コンテンツ・エキスパート: Dmitry Sivkov (英語)

このレシピでは、インテル® VTune™ プロファイラーを GitLab* Continuous Integration (CI) パイプラインに統合することでビルドを自動プロファイルする方法と、インテル® VTune™ プロファイラーの静的な HTML レポート機能を使用して最新のパフォーマンス解析データへのアクセスをより便利にする方法を示します。

このアプローチは、次の利点をもたらします。

- **自動パフォーマンス解析:** パフォーマンス・リグレッションが検出された場合、テスト環境を設定して手動でパフォーマンス結果を収集することは、単調な作業であり、エンジニアの貴重な時間を消費してしまいます。インテル® VTune™ プロファイラーを GitLab* CI パイプラインのテストステージに統合することで、あらかじめ設定した環境でパフォーマンス・データを自動収集して、結果を自動的にアーティファクトとしてアップロードできます。

  これにより、手作業を省き、ビルド完了時にパフォーマンス結果を利用できるようにすることで、リグレッションの原因究明に専念できます。

- **カスタム設定:** インテル® VTune™ プロファイラーのコマンドライン・インターフェイス (CLI) 機能 (英語) は、柔軟な CI 統合を提供します。インテル® VTune™ プロファイラー の CLI を使用することで、例えば、チームのニーズに合わせて、必要な解析タイプとパラメーターを選択してカスタムプロセスをセットアップできます。

  以下は、いくつかの設定例です。

  - ビルドごとにホットスポット解析 (英語) を自動実行します。
  - ビルドシステムの負荷テストステージでパフォーマンス・リグレッションが検出された場合のみ新しいビルドをプロファイルして、すべての必要な解析タイプのパフォーマンス・データを収集します。

- **HTML 解析レポート:** インテル® VTune™ プロファイラーには、収集結果のサマリーを含む静的な HTML レポートを生成するコマンドライン・オプション (英語) が用意されています。この HTML レポートをブラウザーで表示して、追加の解析が必要かどうかを判断できます。オプションで、この HTML レポートを便利な GitLab* ページとしてホストすることもできます。

  - 使用するもの
  - 手順:
    1. GitLab* Runner をインストールする
    2. 自動データ収集を設定する
    3. 結果の自動アップロードを設定する
    4. 結果データを表示する
    5. (オプション) 新たに発生した問題を解決する

使用するもの

以下は、パフォーマンス解析シナリオで使用するソフトウェアのリストです。

- **インフラストラクチャー:** 以下を含む、GitLab* CI パイプラインを設定済みの GitLab* リポジトリー:
  - プロジェクトの Makefile
  - 事前設定済みの `.gitlab-ci.yml` ファイル
  - インテル® VTune™ プロファイラーがインストールされた GitLab* Runner
- **ツール:** インテル® VTune™ プロファイラー 2020 (スタンドアロンまたはインテル® Parallel Studio XE/インテル® System Studio に含まれるもの)。

注

  - インテル® VTune™ プロファイラーのダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
  - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
  - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。

GitLab* Runner をインストールする

GitLab* Runner が設定されていない場合は、GitLab* 公式ドキュメント (https://docs.gitlab.com/runner/#install-gitlab-runner (英語)) の手順に従って、GitLab* Runner ソフトウェア・パッケージをインストールして設定します。

GitLab* Runner をインストールした後に、インテル® VTune™ プロファイラーをインストールします。手順と利用可能なインストール方法については、『インテル® VTune™ プロファイラー・インストール・ガイド』(英語) を参照してください。

注

- root 権限なしでハードウェア・イベントベース・サンプリング収集を実行できるように、`/proc/sys/kernel/perf_event_paranoid` を 0 に設定してください。インテルのサンプリング・ドライバーまたは root 権限を使用しないハードウェア・プロファイルについては、本クックブックの「インテルのサンプリング・ドライバーを使用しないハードウェア・プロファイル」レシピを参照してください。
- パフォーマンス解析結果は、使用する Runner と解析を実行するマシンによって異なります。

自動データ収集を設定する

`vtune` コマンド呼び出しとアーティファクト処理コマンドを GitLab* パイプラインに追加します。例えば、次のコマンドを `.gitlab-ci.yml` ファイルに追加します。

```
vtune -collect hotspots ./<アプリケーション>
```

このコマンドは、最後のコマンドオプションで指定されたアプリケーションを起動して、ホットスポット解析 (英語) データを収集し、結果を別のディレクトリーに保存します。

インテル® VTune™ プロファイラーの結果と HTML サマリーレポートの任意の組み合わせをアップロードできます。例えば、アプリケーションのパフォーマンスの概要を把握するため、完全な解析結果と静的な HTML サマリーレポートをアップロードできます。

静的な HTML レポートを生成するには、次のコマンドを追加します。

```
vtune -report summary -format=html > hotspots_summary.html
```

インテル® VTune™ プロファイラーの完全な結果をアーティファクトとしてアップロードするには、任意のツールで結果ディレクトリーをパッケージ化する必要があります。例えば、次のように、`tar` を使用して結果ディレクトリーをパッケージ化します。

```
tar -c r00* > vtune_result.tar
```

注

- `vtune` コマンドを使用して、任意のオプションを指定し、実行環境で有効な任意の解析タイプを選択できます。コマンドラインから解析を実行する方法については、『インテル® VTune™ プロファイラー・ユーザーガイド』の「コマンドライン解析の実行」(英語) を参照してください。
- インテル® VTune™ プロファイラーのグラフィカル・ユーザー・インターフェイス (GUI) は、GUI で解析を設定して、必要なすべてのオプションとパラメーターを含むコマンドラインを生成してコピーできる、コマンドライン設定生成機能 (英語) を提供します。この機能を利用してコマンドラインを簡単に生成し、後で使用できます。

手動での `vtune` コマンドの作成については、『インテル® VTune™ プロファイラー・ユーザーガイド』の「コマンドライン・インターフェイス」(英語) を参照してください。

注

特定の環境では、実行可能な解析タイプが制限されます。例えば、マイクロアーキテクチャー全般解析は、特定の仮想マシン・ハイパーバイザーでは利用できません。

結果の自動アップロードを設定する

アップロードするファイルを `.gitlab-ci.yml` ファイルの `artifacts/paths:` セクションに追加します。例えば、結果ディレクトリーの `.tar` ファイルと HTML サマリーをアップロードする場合、次のように指定します。

```
artifacts:
 paths:
  - <プロジェクトの相対パス>/vtune_result.tar
  - <プロジェクトの相対パス>/hotspots_summary.html
```

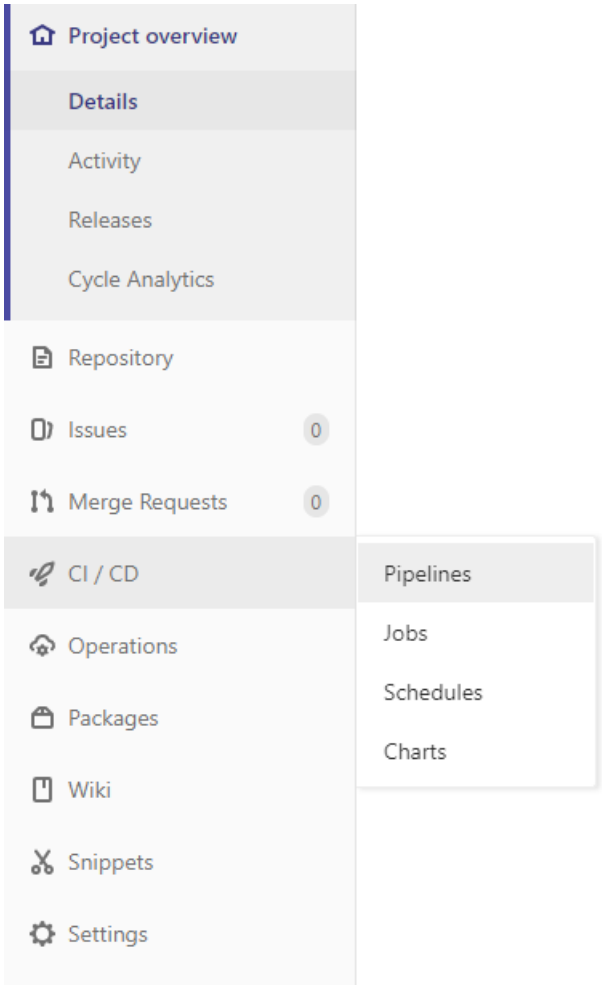
注

結果は、GitLab* アーティファクトとしてアップロードしてください。そうでないと、インテル® VTune™ プロファイラーを実行するマシン上に結果が保存され、手動で取得する必要があります。

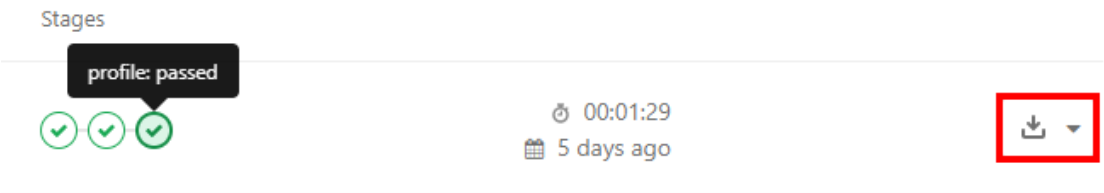
結果データを表示する

ビルドが完了したら、次の手順に従って、GitLab* ウェブ・インターフェイスから解析サマリーページにすぐにアクセスできます。

1. GitLab* で Pipelines ページに移動します。

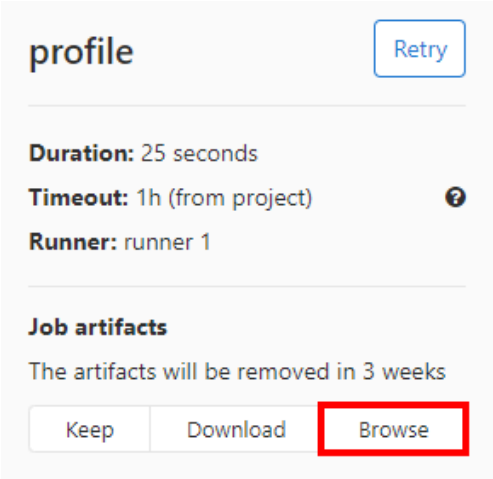


2. このページから、アーティファクト・バンドルをダウンロードするか、個々の HTML ページを参照して、サマリーを基に結果をダウンロードする必要があるかどうかを判断できます。

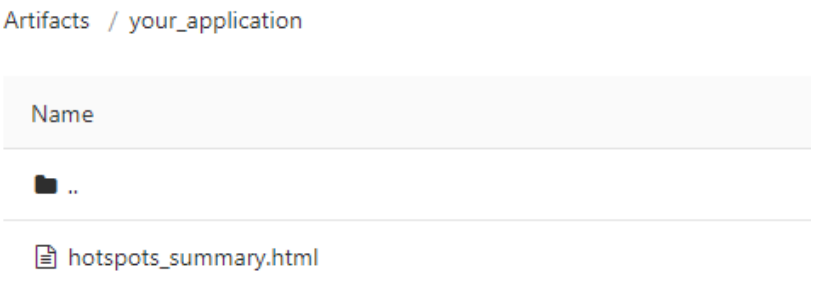


HTML ページを個別にダウンロードするには、次の操作を行います。

1. パイプライン・ステージのレポート・ページに移動して、**[Browse]** をクリックします。



2. HTML ページの場所に移動して、ダウンロードします。



3. 任意のブラウザーでサマリー HTML を開いて、追加の解析が必要かどうかを判断します。

Intel® VTune™ Profiler 2020

Elapsed Time:	1.652s
CPU Time:	1.070s
Effective Time:	1.070s
Idle:	0.010s
Poor:	1.060s
Ok:	0s
Ideal:	0s
Over:	0s
Spin Time:	0s
Overhead Time:	0s
Total Thread Count:	3
Paused Time:	0s

Top Hotspots:

Function	Module	CPU Time
memcmp	libc-dynamic.so	0.084s
OS_BARESYSCALL_DoCallAsmIntel64Linux	libc-dynamic.so	0.082s
[Outside any known module]	[Unknown]	0.073s
llvm::StringMapImpl::LookupBucketFor	libpin3dwarf.so	0.056s
std::priv::_Rb_tree<std::pair<std::string, unsigned long>, std::less<std::pair<std::string, unsigned long>>, std::pair<std::string, unsigned long>, std::priv::_Identity<std::pair<std::string, unsigned long>>, std::priv::_SetTraitsT<std::pair<std::string, unsigned long>>, std::allocator<std::pair<std::string, unsigned long>>>::insert_unique	libtpsstool.so	0.052s
[Others]	N/A	0.722s

Effective Physical Core Utilization: 21.5% (0.861 out of 4)

```
The metric value is low, which may signal a poor physical CPU cores
utilization caused by:
    - load imbalance
    - threading runtime overhead
    - contended synchronization
    - thread/process underutilization
    - incorrect affinity that utilizes logical cores instead of physical
cores
Explore sub-metrics to estimate the efficiency of MPI and OpenMP
parallelism or run the Locks and Waits analysis to identify parallel
bottlenecks for other parallel runtimes.
```

インテル® VTune™ プロファイラーのグラフィカル・ユーザー・インターフェイスを使用して結果データを表示するには、次の操作を行います。

1. `.tar` アーカイブから結果ファイルを展開します。
2. インテル® VTune™ プロファイラー GUI を起動します。
3.  ボタンをクリックして、結果ファイルを参照します。結果データは新しいタブに表示されます。

(オプション) 新たに発生した問題を解決する

パフォーマンス・リグレッションが検出された場合、問題を特定して解決するため、インテル® VTune™ プロファイラーのほかの解析タイプを使用できます。

注

このレシピの情報は、インテル® VTune™ プロファイラー・デベロッパー・フォーラム (英語) を参照してください。

関連情報

- 設定レシピ
- インテル® VTune™ プロファイラーのコマンドライン・インターフェイス (英語)
- インテル® VTune™ プロファイラーの CLI リファレンス (英語)
- コマンドライン設定生成機能 (英語)

ハードウェアベースの hotspot 解析向けに Hyper-V* 仮想マシンを設定する

このレシピは、インテル® VTune™ プロファイラーを使用してハードウェア・パフォーマンスをプロファイルするため、Hyper-V* 環境で仮想マシン・インスタンスを設定します。

コンテンツ・エキスパート: Alexey Kireev (英語)、Denis Pravdin (英語)

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **ツール:**
    - インテル® VTune™ プロファイラー (またはインテル® VTune™ Amplifier 2019)
    - サードパーティー製の Virtual Machine Manager (VMM) も必要になることがあります。
- **CPU**: インテル® バーチャライゼーション・テクノロジー (インテル® VT) 対応のインテル® Xeon® プロセッサー

  次のようなパフォーマンス・モニタリング・ハードウェア搭載のインテル® マイクロアーキテクチャーを使用することもできます。

インテル® マイクロアーキテクチャー開発コード名	PMU バージョン	PEBS	LBR	IPT	PT2GPA
Kaby Lake	4	○	○	○	×
Ice Lake	4	○	○	○	○
Cascade Lake	4	○	○	○	×
Gemini Lake	4	○	○	○	×
Skylake	4	○	○	×	×
Haswell	3	○	○	×	×
Broadwell	3	○	○	×	×

- **オペレーティング・システム:** Microsoft* Windows Server* 2019 または Windows* 10 バージョン 1809 (October 2018 Update) 以降

- インテル® VT 対応の **BIOS**

注

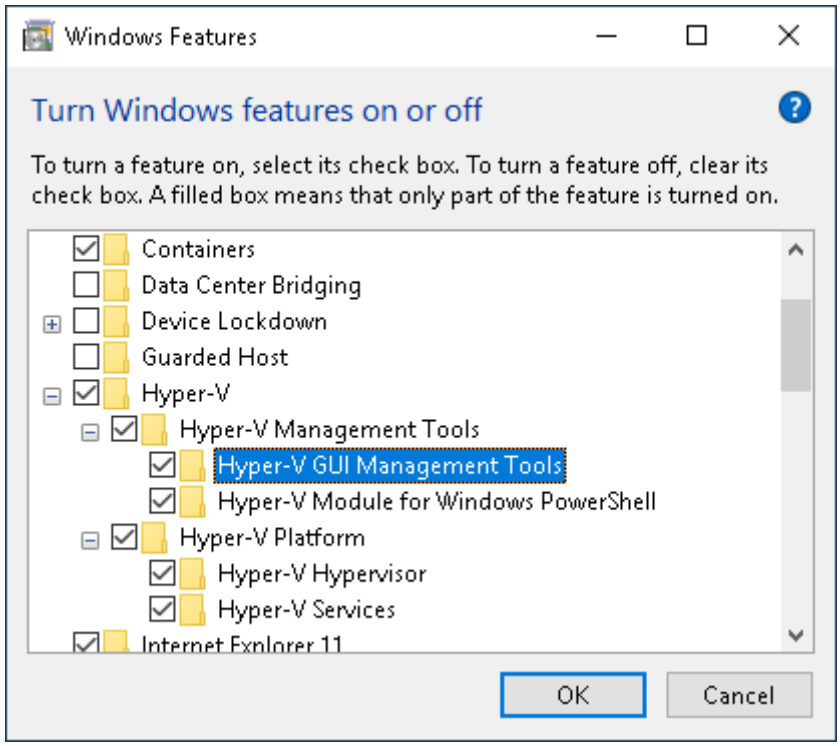
Hyper-V* には、「Hyper-V* 仮想マシンでインテル® パフォーマンス・モニタリング・ハードウェアを有効にする」で説明されている追加の要件があります。

Hyper-V* ホストを設定する

1. サーバーの BIOS セットアップでインテル® VT を有効にします。
    a. 電源投入時の自己テスト (POST) 中に **F2** キーを押してシステムの BIOS にアクセスします。
    b. **[Advanced]** > **[Processor Configuration]** を選択します。
    c. **[Intel® Virtualization Technology]** を選択して **[Enabled]** に変更します。
    d. 変更を保存して再起動します。
2. **[コントロール パネル]** > **[プログラム]** > **[プログラムと機能]** に移動して、左のペインで **[Windows* の機能の有効化または無効化]** をクリックします。

    [Windows* の機能] ダイアログボックスが表示されます。

3. **[Hyper-V]** を展開してすべてのチェックボックスをオンにします。



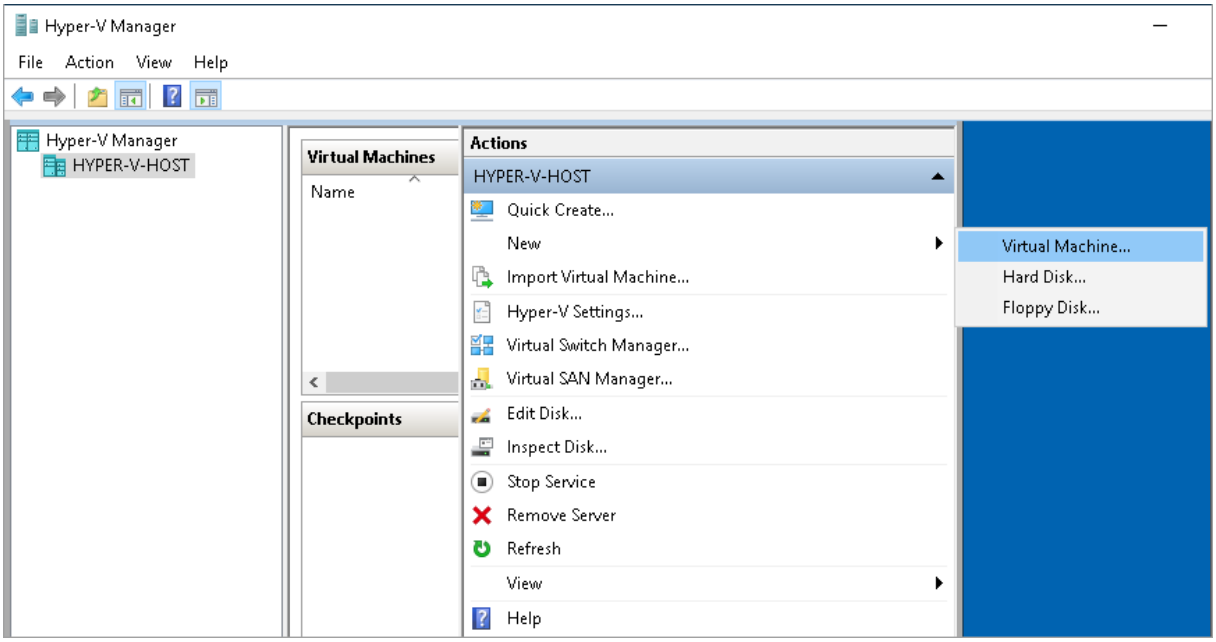
4. **[OK]** をクリックして、インストールします。

    インストールが完了したら **[今すぐ再起動]** をクリックします。

仮想マシンを作成して設定する

Hyper-V* ホスト (以降、「ホスト」) で **[スタート]** メニューから Hyper-V マネージャーを実行して、新しい VM (以降、「VM」) を作成し設定します。

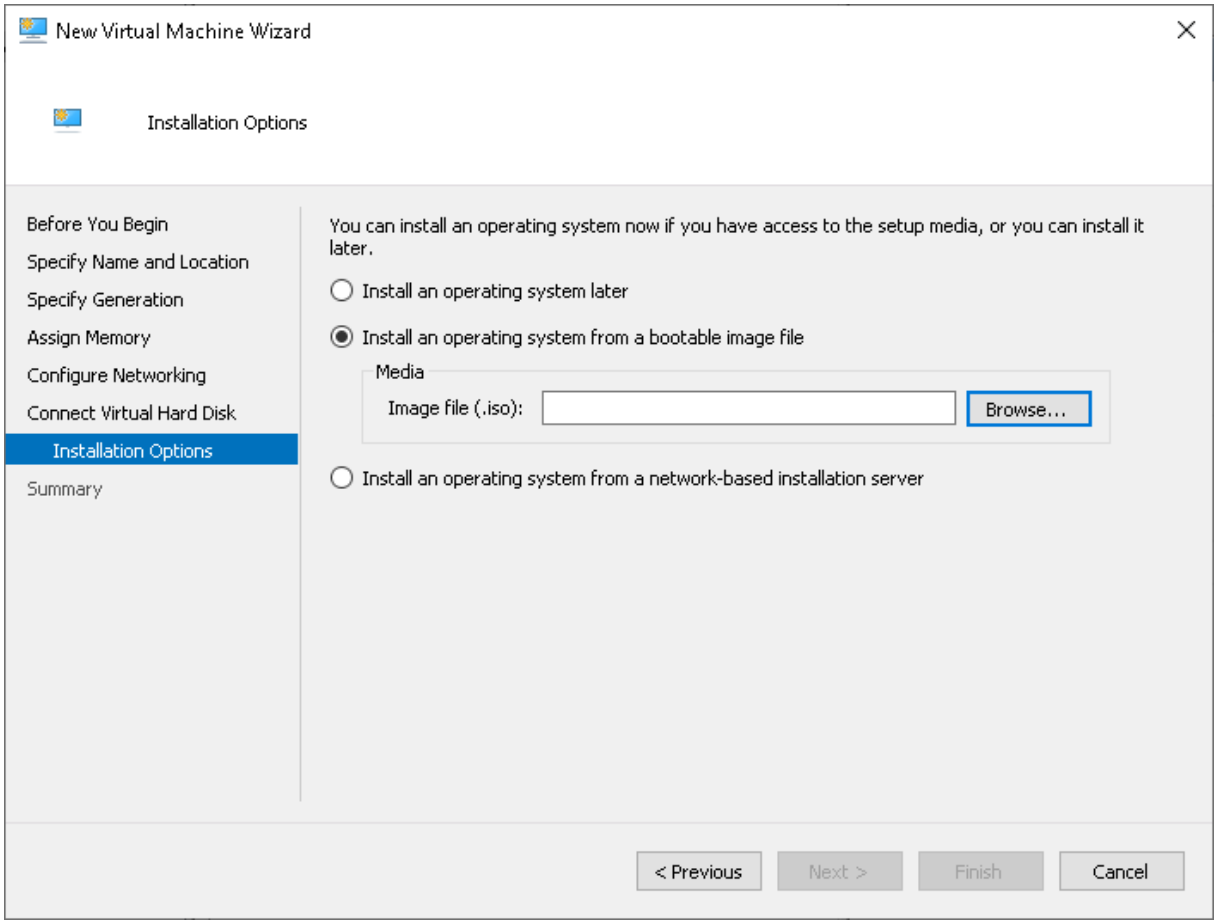
1. **[操作]** パネルで **[新規]** をクリックして、**[仮想マシン...]** を選択します。



2. **[仮想マシンの新規作成ウィザード]** で新しいマシンに必要なすべてのパラメーター (CPU 構成、メモリー、ネットワーク、ハードディスク、ほか) を指定します。

   インテル® VTune™ プロファイラーの推奨システム要件は 4GB RAM、空きディスク容量 10GB 以上です。

3. インストールのソースを設定します。例えば、ローカルのインストール・イメージ (`.iso`) を使用できます。



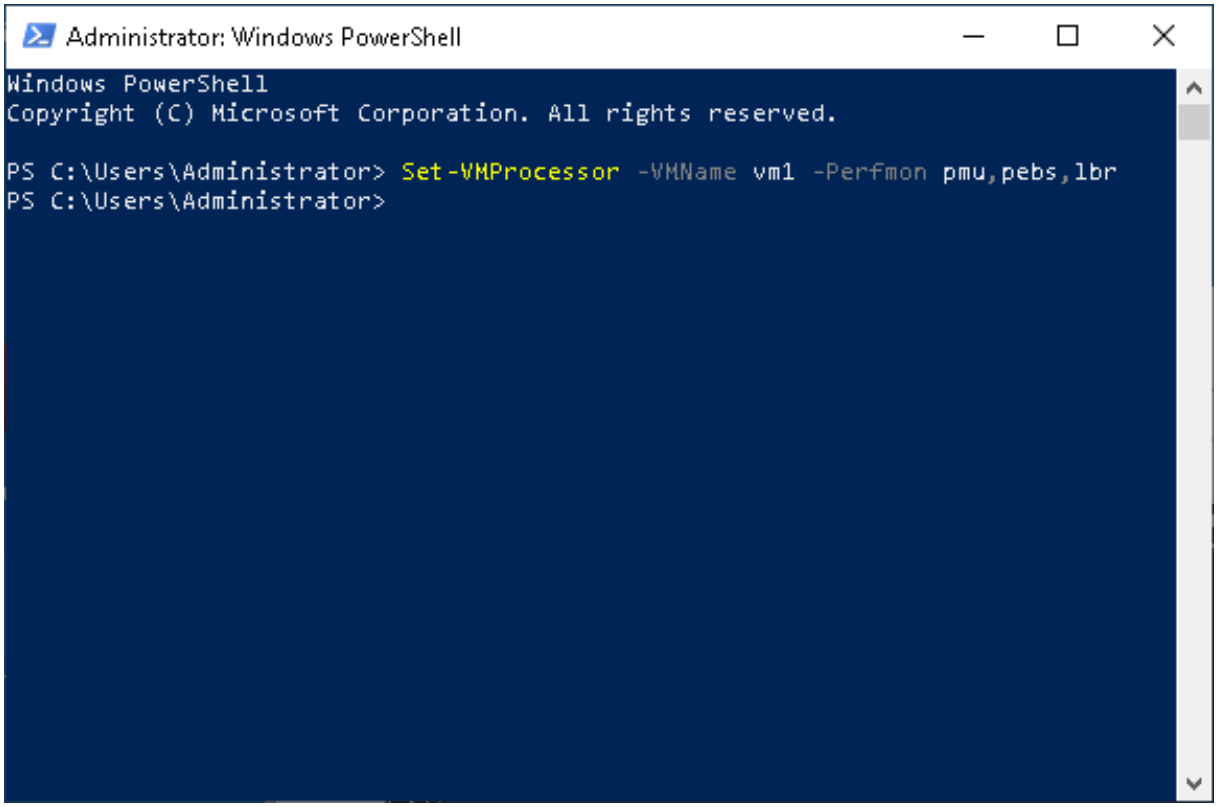
4. 新しく作成した VM を起動して、OS のインストール・プロセスを開始します。

ハードウェア解析向けに仮想マシンを設定する

PMU/PEBS/LBR を VM に公開してプロファイルするターゲット・システム・インスタンスを準備します。

1. VM をシャットダウンして、**[スタート]** メニューから管理者権限でホストの Windows PowerShell* を起動します。
2. PowerShell* で次のコマンドを実行して、PMU、PEBS、LBR が VM に公開されるように設定します。

```
Set-VMProcessor -VMName your_vm_name -Perfmon pmu,pebs,lbr
```



3. VM を起動します。

注

追加の情報は、「Hyper-V* 仮想マシンでインテル® パフォーマンス・モニタリング・ハードウェアを有効にする」を参照してください。

hotspot 解析を実行する

VM から直接インテル® VTune™ プロファイラーを実行します。

1. VM にインテル® VTune™ プロファイラーをインストールします。
2. **[スタート]** メニューからインテル® VTune™ プロファイラーを実行します。
3. プロジェクトを作成します。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

4. **[WHERE (どこを)]** ペインでは、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
5. **[WHAT (何を)]** ペインでは、アプリケーションの場所と作業ディレクトリーを指定します。
6. **[HOW (どのように)]** ペインでは、hotspot 解析の **[Hardware Event-Based Sampling (ハードウェア・イベントベース・サンプリング)]** 収集モードを選択します。

   PMU/PEBS/LBR が正しく公開されている場合、[Analysis Configuration (解析の設定)] ペインにエラーメッセージは表示されず、解析を実行できます。

7.  **[Start (開始)]** をクリックして解析を開始します。

   解析結果は、デフォルトの **[Hotspots by CPU Utilization (CPU 使用率によるホットスポット)]** ビューポイントに表示されます。

注

同時に実行している複数の VM で hotspot 解析を実行することも可能です。

関連情報

- Hyper-V* 環境のターゲットをプロファイル (英語)
- Hyper-V* 仮想マシンでインテル® パフォーマンス・モニタリング・ハードウェアを有効にする
- ハードウェア・イベントベース・サンプリング収集 (英語)

パフォーマンス異常を見つけるアプリケーションのプロファイル

このレシピは、インテル® VTune™ プロファイラーの異常検出解析を使用して、いくつかの要因によるパフォーマンス異常を特定する方法を紹介します。また、これらの異常を修正するための提案も提供します。

コンテンツ・エキスパート: Vasily Starikov

パフォーマンス異常は、無視すると取り返しのつかない損失をもたらす散発的な問題です。望ましくない動作を引き起こす可能性のあるパフォーマンス異常には、いくつかの種類があります。

- ビデオフレームが遅い/スキップする
- 画像のトラッキングに失敗する
- 金融取引に想定外の時間がかかる
- ネットワーク・パケット処理に長い時間がかかる
- ネットワーク・パケットの紛失

これらの動作は、従来のサンプリングベースの手法では見つけることができませんが、異常検出解析タイプを使用して検出できます。この解析では、次の原因による異常を調査します。

- 制御フローの逸脱
- スレッドのコンテキスト・スイッチ
- 想定外のカーネル・アクティビティー (割り込みやページフォールトなど)
- CPU 周波数の低下

異常検出は、インテル® Processor Trace (インテル® PT) テクノロジーを使用して、プロセッサーからの粒度の細かい情報をナノ秒レベルで提供します。

使用するもの

以下は、このパフォーマンス解析の最小ハードウェアおよびソフトウェア要件です。

- **アプリケーション:** 任意のサンプル・アプリケーション。
- **マイクロアーキテクチャー:** インテル® Xeon® プロセッサー開発コード名 Skylake 以降。
- **ツール**: インテル® VTune™ プロファイラー 2021 以降の異常検出解析。

注

- バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
- インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
- 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
  - インテル® VTune™ プロファイラー製品ページ
  - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)

- **オペレーティング・システム:**
  - Linux* Fedora* 31 (Workstation エディション) - 64 ビット・バージョン
  - Windows* 10

インテル® PT の要件

- **オペレーティング・システム:** 任意のバージョンの Windows* または Linux*。
- **マイクロアーキテクチャー:** インテル® プロセッサー開発コード名 Skylake 以降。

解析用にアプリケーションを準備する

通常、ソフトウェアのパフォーマンス解析では大量のデータを収集します。パフォーマンス異常は稀で短期間であるため、これらのデータセットに占める割合はごくわずかであり、気付かれないこともあります。インテルのインストルメンテーションおよびトレーシング・テクノロジー (ITT) API を使用して、特定のコード領域に絞って解析すると良いでしょう。

コード領域を選択してアプリケーションを準備します。

1. サンプル・アプリケーションがあるディレクトリーに移動します。
2. プロファイルするコード領域の名前を登録します。

```
__itt_pt_region region=__itt_pt_region_create("region of interest");
```

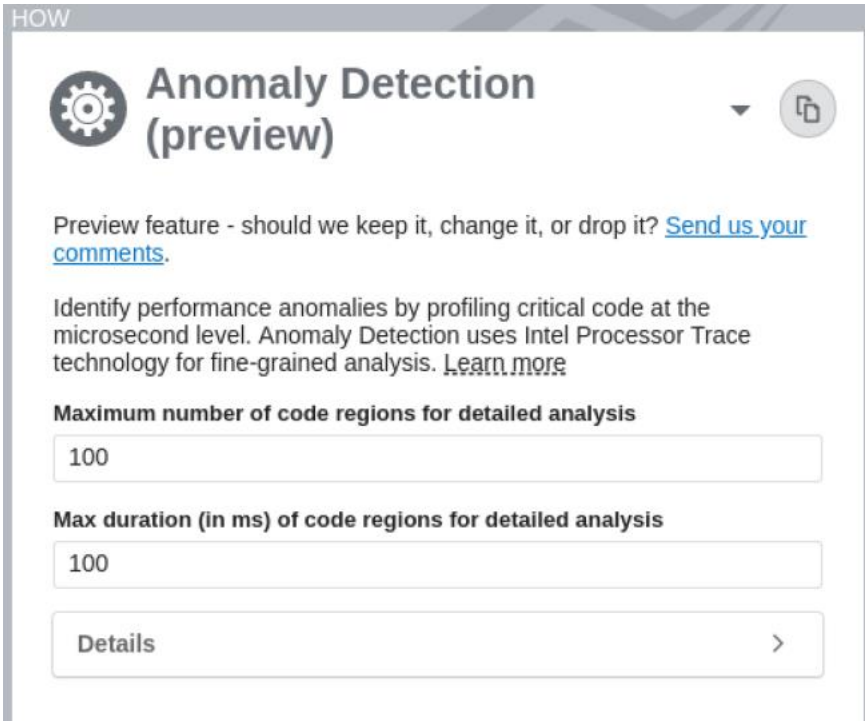
3. サンプルの中で、異常が発生しやすい操作を行うループを探します。begin および end 関数を使用してループの反復をマークします。次に例を示します。

```
double process(std::vector<double> &cache)
{
   double res=0;
   for (size_t i=0; i<ITERATIONS; i++)
   {
     __itt_mark_pt_region_begin(region);
     res+=calculate(i, cache);
     __itt_mark_pt_region_end(region);
   }
return res;
}
```

異常検出を実行する

1. [Welcome (ようこそ)] 画面で **[Configure Analysis (解析の設定)]** をクリックします。
2. 解析ツリーで **[Algorithm (アルゴリズム)]** グループの **[Anomaly Detection (異常検出)]** 解析タイプを選択します。
3. **[WHAT (何を)]** ペインで、アプリケーションと関連するアプリケーション・パラメーターを指定します。
4. **[HOW (どのように)]** ペインで以下のパラメーターを指定して、解析で収集するデータの量を定義します。

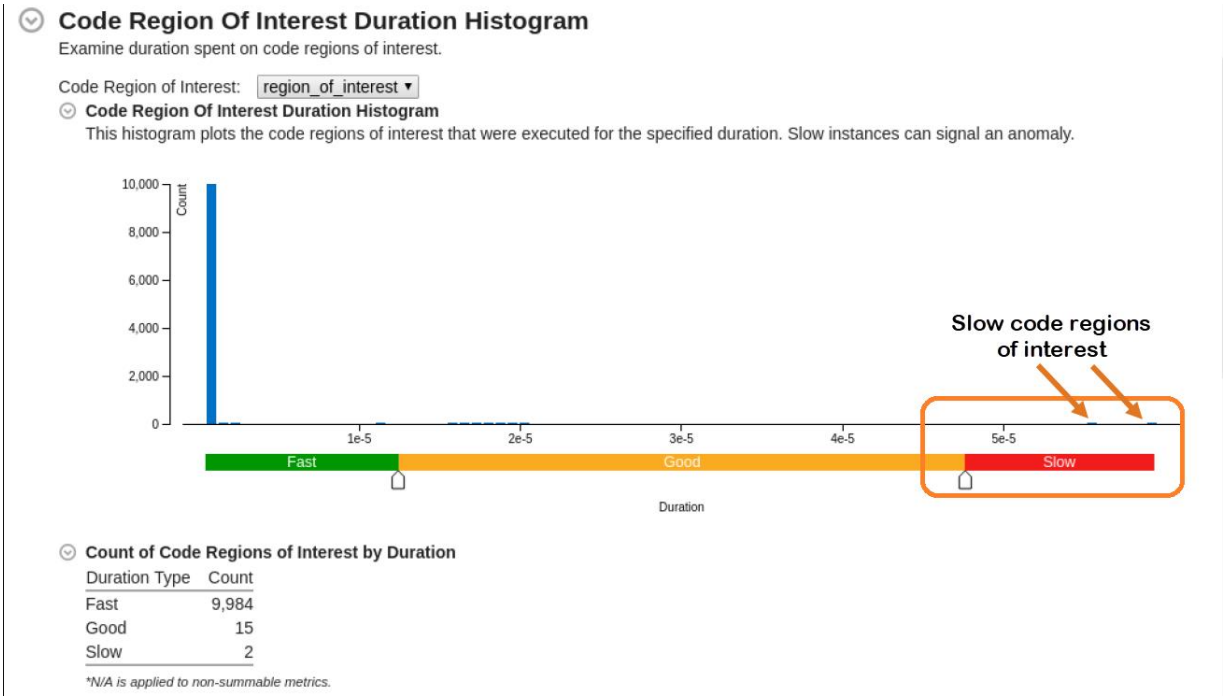
引数	説明	範囲	推奨値
Maximum number of code regions for detailed analysis (詳細に解析するコード領域の最大数)	結果を解析するため、同時に詳細をロードするアプリケーションのコード領域インスタンスの最大数を指定します。	10-5000	詳細をより速く表示するには、1000 以下の値を選択してください。
Maximum duration of code regions for detailed analysis (詳細に解析するコード領域の最大時間)	コード領域の各インスタンスの解析に費やす最大時間 (ミリ秒) を指定します。指定した時間よりも長い時間を必要とするインスタンスは、無視されるか、ロードされません。	0.001-1000	1000 ミリ秒以下の任意の値。また、大量のデータは処理効率を低下させる可能性があるため、データ収集を制限するオプション (英語) も検討してください。



5. **[Start (開始)]** ボタンをクリックして、解析を実行します。

異常を特定する

1. 解析が完了したら **[Summary (サマリー)]** ウィンドウに切り替えます。**[Code Region of Interest Duration Histogram (注目するコード領域の時間ヒストグラム)]** を確認します。

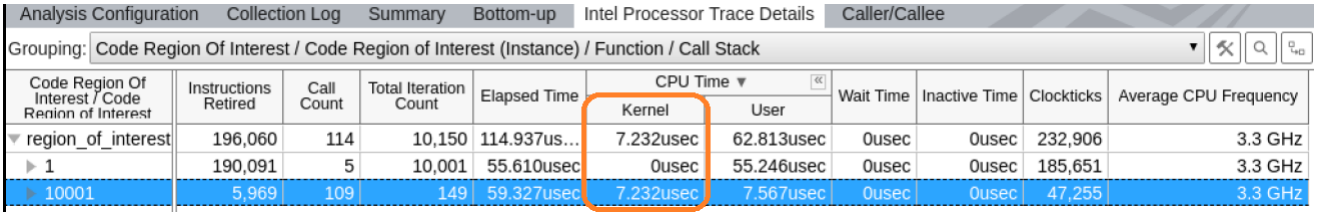


2. パフォーマンスが低い場合は、ヒストグラムのスライダーを動かして、パフォーマンスの異常値を表示します。
3. **[Bottom-up (ボトムアップ)]** ウィンドウに切り替えます。
4. グループ化したテーブルに注目する遅いコード領域の詳細をロードします。
    1. 表示を展開して **[Fast (高速)]** と **[Slow (低速)]** 領域を表示します。
    2. テーブルの **[Slow (低速)]** 領域を右クリックします。
    3. ポップアップ・メニューで **[Load Intel Processor Data by Selection (選択したインテル(R) プロセッサー・データをロード)]** を選択します。

調査する異常を選択する

データをロードしたら [Intel Processor Trace Details (インテル(R) Processor Trace の詳細)] ビューに切り替えます。遅いコード領域で収集された情報を調査します。

この例では、Inactive Time (インアクティブ時間) と Wait Time (待機時間) メトリックがゼロです。これは、コンテキスト・スイッチが発生しなかったことを示しています。



Code Region Of Interest / Code Region of Interest	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time ▼ Kernel	CPU Time ▼ User	Wait Time	Inactive Time	Clockticks	Average CPU Frequency
region_of_interest	196,060	114	10,150	114.937us…	7.232usec	62.813usec	0usec	0usec	232,906	3.3 GHz
1	190,091	5	10,001	55.610usec	0usec	55.246usec	0usec	0usec	185,651	3.3 GHz
10001	5,969	109	149	59.327usec	7.232usec	7.567usec	0usec	0usec	47,255	3.3 GHz

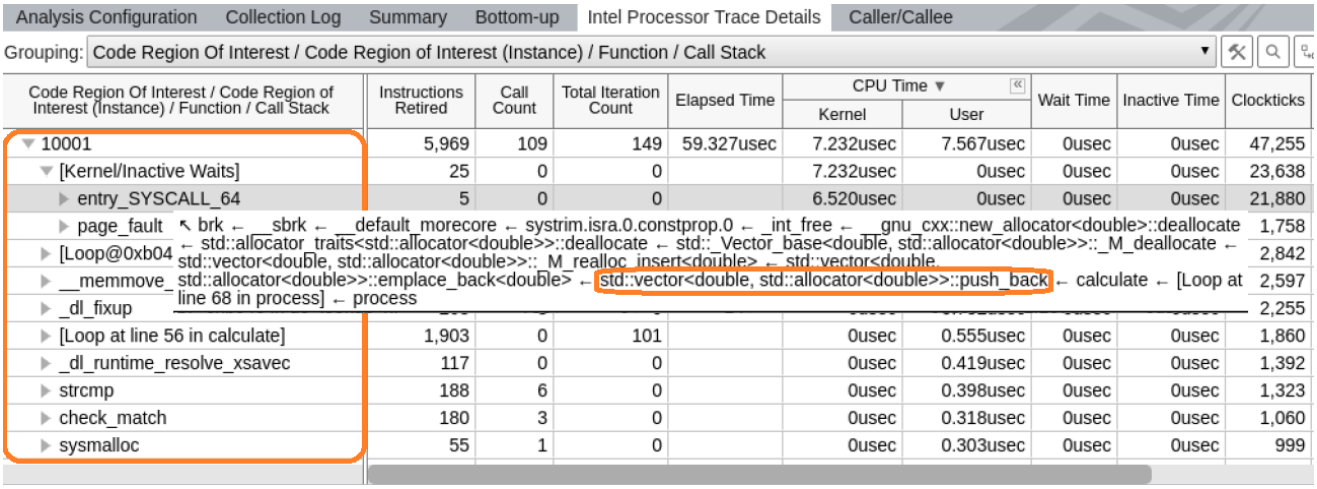
非ゼロのカーネル時間は、予期しないカーネル・アクティビティーを見つける手掛かりになります。

この例では、**[Code Region of Interest Duration Histogram (注目するコード領域の時間ヒストグラム)]** から、2 つの注目する遅いコード領域が見つかりました。カーネル CPU 時間の値が大きいコード領域インスタンス 10001 から調査を開始します。

カーネル・アクティビティーの異常を調査する

最初の異常は領域 10001 にあります。

各コード領域の実行内容を見てみます。テーブルで領域のノードを展開して、ノードで実行された関数のリストを確認します。

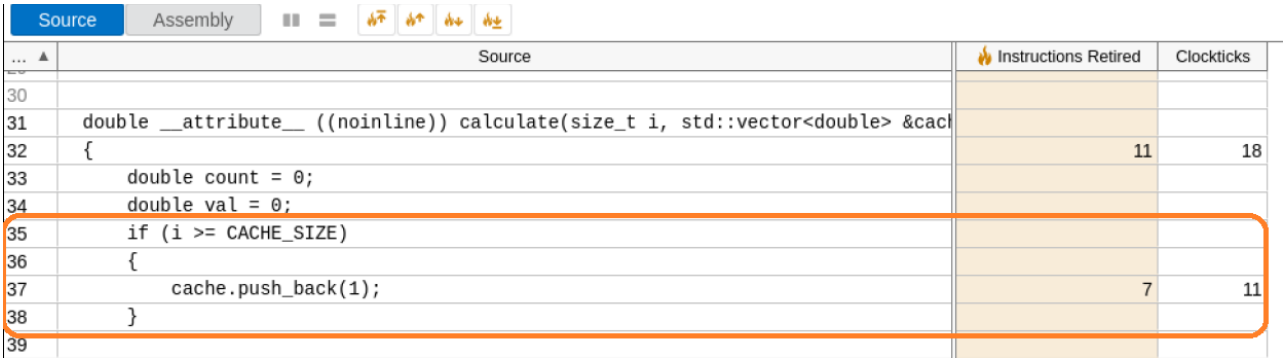


この例では、*Kernel/Inactive Waits* 要素が関数リストのトップにあります。Linux* カーネルは動的なコード変更を採用しているため、カーネルのバイナリーを静的に解析しても、カーネルの制御フローを完全に再構築することはできません。このノードは、特定の注目するコード領域を実行中に発生したカーネル・アクティビティーのすべてのパフォーマンス・データを集約しています。

カーネルのバイナリーは処理されていないため、**Call Count (呼び出し回数)**、**Iteration Count (反復回数)**、**Instructions Retired (リタイアした命令)** などの制御フローメトリックを再構築することはできません。**Call Count (呼び出し回数)** と **Iteration Count (反復回数)** はゼロですが、**Instructions Retired (リタイアした命令)** はカーネルへのエントリー数を示しています。

このノードのスタックには、カーネル・エントリー・ポイントを含む完全な関数呼び出しシーケンスが含まれます。この情報からアプリケーションがカーネルへ制御を移した理由が分かります。

Kernel/Inactive Waits 要素のコールスタックは、`calculate` メソッドから `std::vector` の `push_back` メソッドを呼び出すことで増えています。この関数をダブルクリックして **[Source (ソース)]** ビューで開きます。



詳しく調査することで異常の原因が分かります。

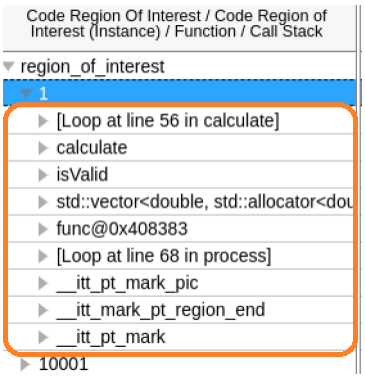
問題
計算が内部ソフトウェア・キャッシュ・サイズを超過して、キャッシュに新しい要素を追加しています。

解決策
ソフトウェア・キャッシュのサイズを増やします。

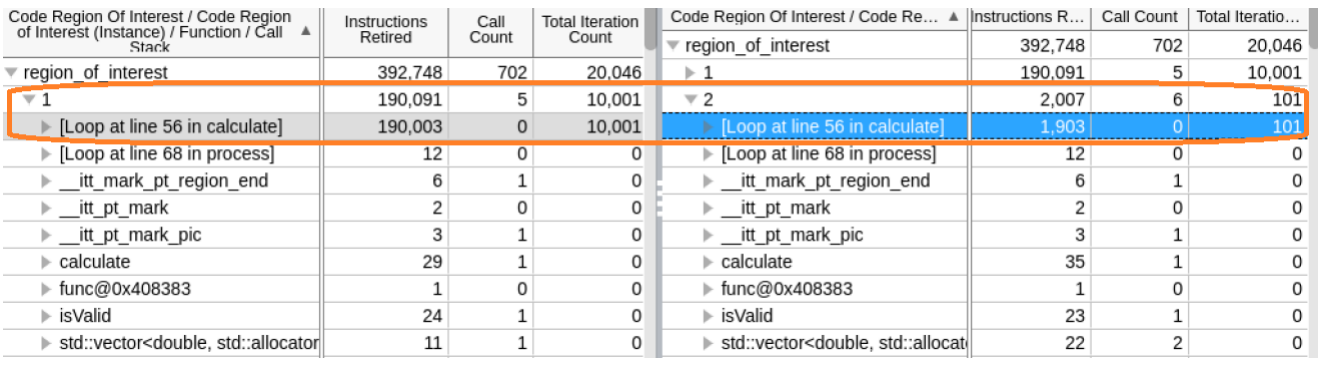
制御フローの逸脱異常を調査する

次に、ヒストグラムで見つけた別の種類の異常を調査します。この場合、**Instruction Retired (リタイアした命令)** メトリックが異常に高いです。

これは、そのコード領域の実行中に制御フローの逸脱があったことを示しています。グリッド内のノードを展開して実行された関数を確認すると、一見何も異常がないように見えます。



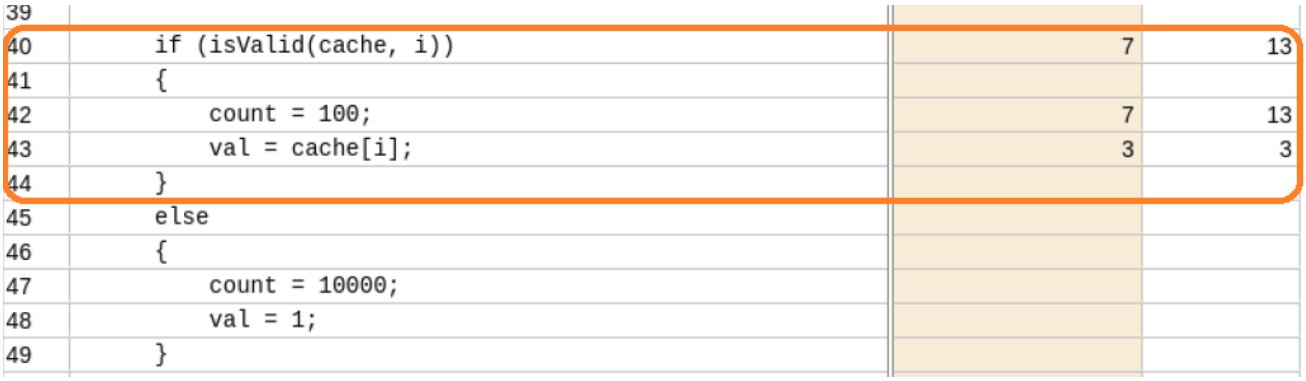
速い反復と遅い反復の詳細を一緒にロードして比較します。



実行された関数のリストは同じですが、異常なインスタンスでは `calculate` 関数のループ反復回数が多いことが分かります。

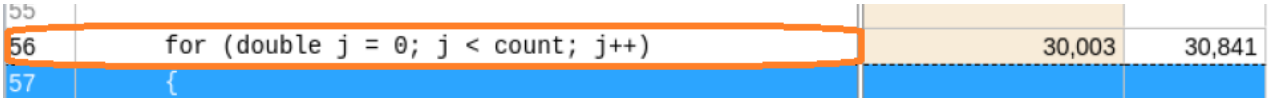
[Source (ソース)] ビューで `calculate` 関数を開いて、速いインスタンスと遅いインスタンスの両方を確認します。

速いインスタンスでは、`isValid` 条件が満たされ、データ要素がキャッシュにあります。



39			
40	`if (isValid(cache, i))`	7	13
41	`{`		
42	`count = 100;`	7	13
43	`val = cache[i];`	3	3
44	`}`		
45	`else`		
46	`{`		
47	`count = 10000;`		
48	`val = 1;`		
49	`}`		

遅いインスタンスでは、`isValid` 条件が満たされず、キャッシュ内のデータ要素の検証に失敗します。`else` 節が有効になり、追加の計算が実行されます。



55			
56	`for (double j = 0; j < count; j++)`	30,003	30,841
57	`{`		

問題
キャッシュに有効なデータがない遅い反復では、追加の計算が発生しています。

解決策
計算を開始する前にキャッシュデータを更新するか、キャッシュ・アルゴリズムを変更します。

注
このレシピの情報は、インテル® VTune™ プロファイラー・デベロッパー・フォーラム (英語) を参照してください。

関連情報

- 異常検出解析 (英語)
- 異常検出ビュー (英語)

GPU 上で実行する OpenMP* オフロード・アプリケーションのプロファイル

このレシピでは、インテル® GPU へオフロードされる OpenMP* アプリケーションをコンパイルする方法を示し、インテル® VTune™ プロファイラーで OpenMP* アプリケーションの GPU 解析 (HPC パフォーマンス特性、GPU オフロード、および GPU 計算/メディア・ホットスポット) を実行して、結果を調査する方法を紹介します。

コンテンツ・エキスパート: Sunny Gogar、Nikita Kiryuhin

使用するもの

以下は、このパフォーマンス解析の最小ハードウェアおよびソフトウェア要件です。

- **アプリケーション:** `iso3dfd_omp_offload` (英語) OpenMP* オフロードサンプル。このサンプル・アプリケーションは、インテル® oneAPI ツールキットのサンプルコード・パッケージ (英語) に含まれています。
- **コンパイラー:** DPC++ アプリケーションをプロファイルするには、インテル® oneAPI HPC ツールキット (英語) に含まれるインテル® oneAPI DPC++/C++ コンパイラー (`icx/icpx`) (英語) が必要です。
- **ツール:** インテル® VTune™ プロファイラー
    - HPC パフォーマンス特性解析
    - GPU オフロード解析
    - GPU 計算/メディア・ホットスポット解析

    注

    - バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
    - インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
    - 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
        - インテル® VTune™ プロファイラー製品ページ
        - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)

- **マイクロアーキテクチャー:**
  - 第 9 世代インテル® プロセッサー・グラフィックス
- **オペレーティング・システム:**
  - Linux* カーネルバージョン 4.14 以降
  - Windows* 10

OpenMP* オフロード・アプリケーションをコンパイルする

Linux*:

1. サンプル・ディレクトリーに移動します。

   ```
   cd <sample_dir>/DirectProgramming/C++/StructuredGrids/iso3dfd_omp_offload
   ```

2. OpenMP* オフロード・アプリケーションをコンパイルします。

   ```
   mkdir build;
   cd build;
   cmake -DVERIFY_RESULTS=0 -DCMAKE_CXX_FLAGS="-g -mllvm -parallel-source-
   info=2"  ..
   make -j
   ```

   `src/iso3dfd` 実行ファイルが生成されます。

   プログラムを削除するには、次のコマンドを実行します。

   ```
   make clean
   ```

   `make` コマンドによって作成された実行ファイルとオブジェクト・ファイルが削除されます。

Windows*:

1. サンプル・ディレクトリーに移動します。

   ```
   cd <sample_dir>/DirectProgramming/C++/StructuredGrids/iso3dfd_omp_offload
   ```

2. OpenMP* オフロード・アプリケーションをコンパイルします。

   ```
   mkdir build
   cd build
   icx /Zi -mllvm -parallel-source-info=2 /std:c++17 /EHsc /Qiopenmp
   /I../include\ /Qopenmp-targets:spir64 /DUSE_BASELINE
   /DEBUG ..\src\iso3dfd.cpp ..\src\iso3dfd_verify.cpp ..\src\utils.cpp
   ```

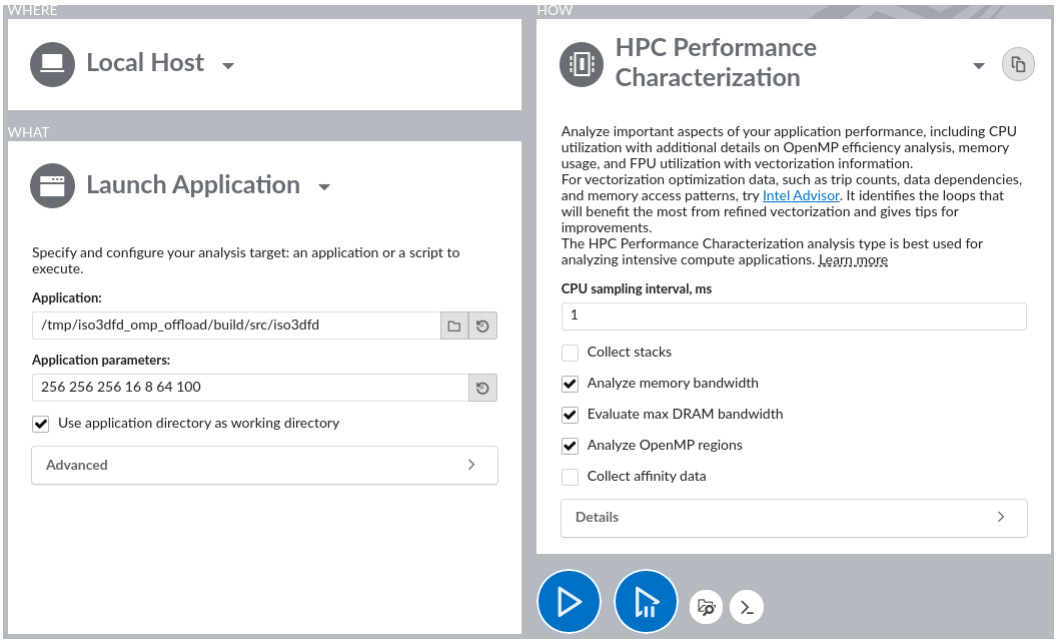
OpenMP* オフロード・アプリケーションの HPC パフォーマンス特性解析を実行する

OpenMP* オフロード・アプリケーションのパフォーマンスに関する高レベルのサマリーを取得するには、HPC パフォーマンス特性解析を実行します。この解析タイプは、アプリケーションが CPU、GPU、およびメモリーをどのように利用しているかを理解するのに役立ちます。また、コードがどの程度ベクトル化されているかも確認できます。

OpenMP* オフロード・アプリケーションでは、HPC パフォーマンス特性解析は各 OpenMP* オフロード領域に関連したハードウェア・メトリックを示します。

必要条件: GPU 解析を実行するシステムを準備します。「GPU 解析用にシステムをセットアップ」(英語) を参照してください。

1. インテル® VTune™ プロファイラーを開き、**[New Project (新規プロジェクト)]** をクリックしてプロジェクトを作成します。
2. [Welcome (ようこそ)] ページで **[Configure Analysis (解析の設定)]** をクリックして解析を設定します。
3. 次の解析設定を選択します。
    - **[WHERE (どこを)]** ペインでは、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
    - **[WHAT (何を)]** ペインでは、**[Launch Application (アプリケーションを起動)]** を選択して、プロファイルするアプリケーションとして `iso3dfd_omp_offload` バイナリーを指定します。
    - **[HOW (どのように)]** ペインでは、解析ツリーの **[Parallelism (並列処理)]** グループから **[HPC Performance Characterization (HPC パフォーマンス特性)]** 解析タイプを選択します。



4. **[Start (開始)]** ボタンをクリックして、解析を実行します。

コマンドラインから解析を実行する

コマンドラインから HPC パフォーマンス特性解析を実行するには、次の操作を行います。

- Linux*:
  1. スクリプトをエクスポートしてインテル® VTune™ プロファイラーの環境変数を設定します。

     ```
     export <install_dir>/vtune-vars.sh
     ```

  2. HPC パフォーマンス特性解析を実行します。

     ```
     vtune -collect hpc-performance - src/iso3dfd 256 256 256 16 8 64 100
     ```

- Windows*:
  1. バッチファイルを実行してインテル® VTune™ プロファイラーの環境変数を設定します。

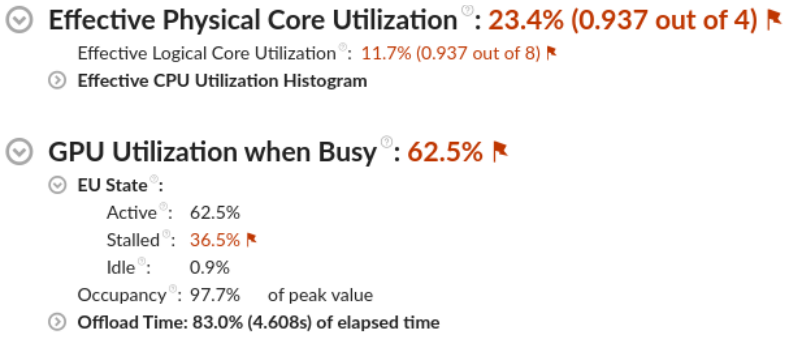
     ```
     <install_dir>\vtune-vars.bat
     ```

  2. HPC パフォーマンス特性解析を実行します。

     ```
     vtune -collect hpc-performance - iso3dfd.exe 256 256 256 16 8 64 100
     ```

HPC パフォーマンス特性データを調査する

[Summary (サマリー)] ペインから調査を開始します。**[Effective Physical Core Utilization (効率的な物理コア利用率)]** (または **[Effective Logical Core Utilization (効率的な論理コア利用率)]**) と **[GPU Utilization when Busy (ビジー時の GPU 利用率)]** セクションでハイライトされている問題を確認します。



[GPU Utilization when Busy (ビジー時の GPU 利用率)] セクションでは、オフロード時間でソートされた OpenMP* オフロード領域の上位に注目します。これらのオフロード領域の GPU 利用率を確認します。

GPU Utilization when Busy: 62.5%

EU State:

Active: 62.5%
Stalled: 36.5%
Idle: 0.9%
Occupancy: 97.7% of peak value

Offload Time: 83.0% (4.608s) of elapsed time

Compute: 98.1% (4.518s) of offload time
Data Transfer: 0.8% (0.038s) of offload time
Overhead: 1.1% (0.052s) of offload time

Top OpenMP Offload Regions

OpenMP Offload Region	Offload Time	Percentage of Elapsed Time	Data Transfer	Overhead	GPU Utilization when Busy
Iso3dfdIterationomptarget$region:dvc=0@/tmp/iso3dfd_omp_offload/src/iso3dfd.cpp:50	4.539s	81.8%	0.000s	0.020s	62.6%
Iso3dfdomptarget$region:dvc=0@/tmp/iso3dfd_omp_offload/src/iso3dfd.cpp:332	0.069s	1.3%	0.037s	0.032s	7.0%
[Outside any OpenMP Offload Region]		0.0%			0.0%

*N/A is applied to non-summable metrics.

完全なデバッグ情報付きでアプリケーションをコンパイルした場合、領域名にソースの場所に関する次の情報が含まれます。

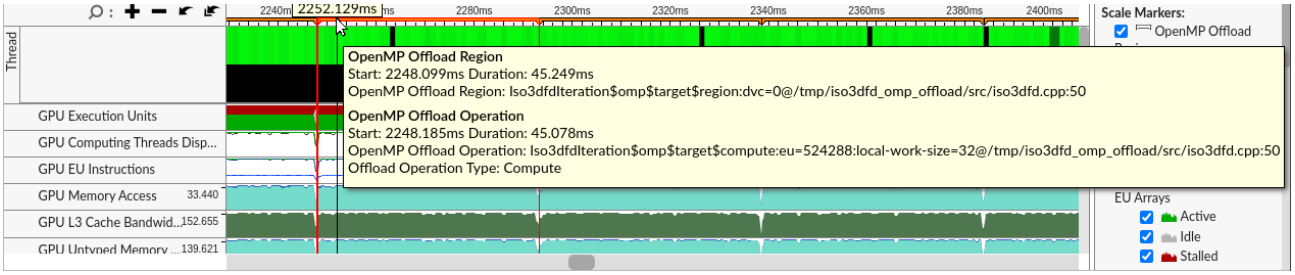
- 関数名
- ソースファイル名
- 行番号

この例では、オフロード・アクティビティーのほぼすべてが **[Compute (計算)]** アクティビティーとして分類されています。また、1 つのオフロード領域がオフロード時間の大部分を消費しています。このオフロード領域名をクリックして **[Bottom-Up (ボトムアップ)]** ビューに切り替えます。OpenMP* オフロード領域の時間、領域インスタンス数、GPU および CPU メトリックを含むグループ化されたテーブルを調査します。

Grouping: OpenMP Offload Region / Function / Call Stack

OpenMP Offload Region / Function / Call Stack	OpenMP Offload Time			Instance Count	GPU				CPU Time
	Compute ▼	Data Transfer	Overhead		EU State			Occupancy	
					Active	Stalled	Idle		
Iso3dfdIterationomptarget$region:dvc=0@/tmp/iso3	4.518s	0.000s	0.020s	100	62.6%	36.6%	0.9%	97.8%	4.391s
Iso3dfdomptarget$region:dvc=0@/tmp/iso3dfd_omp	0s	0.037s	0.032s	2	7.0%	4.0%	89.0%	10.8%	0.041s
[Outside any OpenMP Offload Region]					0.0%	0.1%	99.9%	0.0%	0.911s

タイムライン・ビューの上部にある領域マーカーにホバーすると、各オフロード領域の名前と時間、領域内のオフロード操作が表示されます。タイムラインの GPU メトリックは、オフロード領域の各インスタンスが時間経過とともにどのように動作するのかを理解するのに役立ちます。

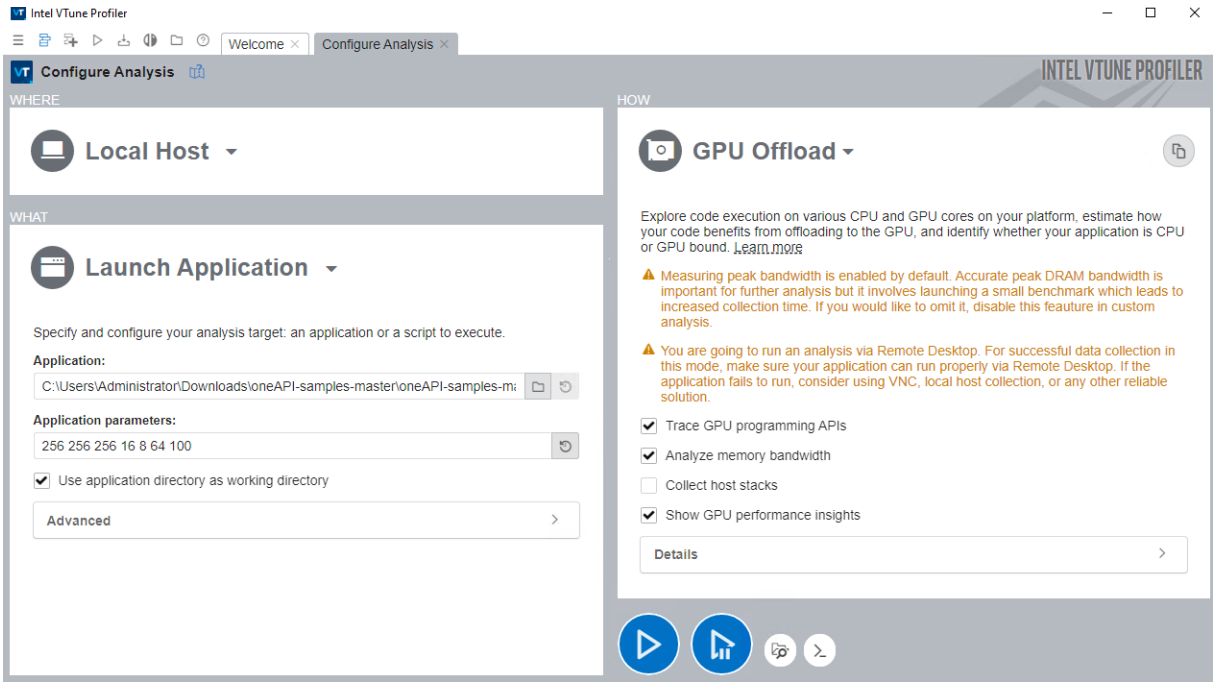


これらの詳細は、このアプリケーションのパフォーマンスにおいて GPU アクティビティーが重要な役割を果たしていることを明確に示しています。次に、GPU オフロード解析に移動して詳細を見てみましょう。

OpenMP* オフロード・アプリケーションの GPU オフロード解析を実行する

必要条件: まだ行っていない場合は、GPU 解析を実行するためシステムを準備します。「GPU 解析用にシステムをセットアップ」(英語) を参照してください。

1. 解析ツリーの **[Accelerators (アクセラレーター)]** グループから **[GPU Offload (GPU オフロード)]** 解析タイプを選択します。
2. 次の解析設定を選択します。



3. **[Start (開始)]** ボタンをクリックして、解析を実行します。

コマンドラインから解析を実行する

コマンドラインから GPU オフロード解析を実行するには、次のコマンドを使用します。

- Linux*:

```
vtune -collect gpu-offload - src/iso3dfd 256 256 256 16 8 64 100
```

- Windows*:

```
vtune -collect gpu-offload - iso3dfd.exe 256 256 256 16 8 64 100
```

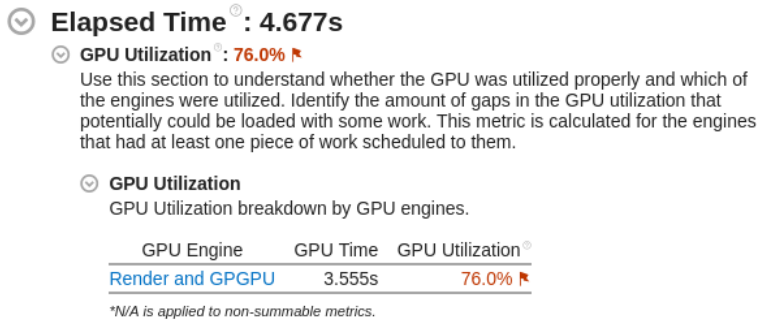
GPU オフロード解析データを調査する

[GPU Offload (GPU オフロード)] ビューポイントから調査を開始します。

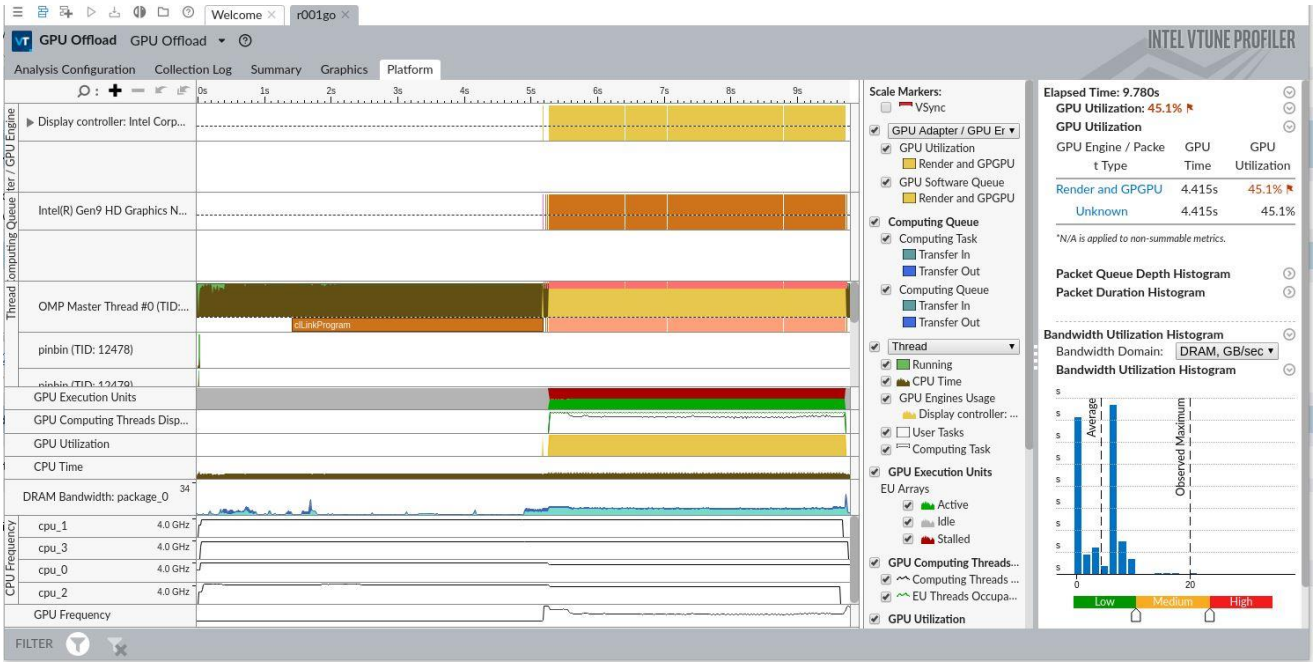
[Summary (サマリー)] ウィンドウで CPU および GPU リソースの利用に関する統計を確認します。このデータからアプリケーションの特性を判断します。

- GPU 依存
- CPU 依存
- システムの計算リソースの非効率的な利用

この例では、アプリケーションは計算集約型の処理に GPU を使用すべきです。しかし、解析結果から、実際の GPU 利用率が低いことが分かります。



[Platform (プラットフォーム)] ウィンドウに切り替えます。ここでは、ソフトウェア・キューの GPU 利用率の解析に役立つ基本的な CPU および GPU メトリックを確認できます。このデータは、タイムラインで CPU 利用率に関連付けられます。



[Platform (プラットフォーム)] ウィンドウの情報は、いくつかの推論に役立ちます。

GPU 依存アプリケーション	CPU 依存アプリケーション
プロファイル時間の大部分で GPU がビジー	プロファイル時間の大部分で CPU がビジー
ビジーである間に小さなアイドル時間がある	ビジーである間に大きなアイドル時間がある
GPU ソフトウェア・キューがほとんどゼロにならない	

注
ほとんどのアプリケーションでは、ここに示すような明らかな状況は発生しません。すべての依存関係を理解するには、詳細な解析が重要です。例えば、ビデオ処理とレンダリングを行う GPU エンジンが交互にロードされる場合、これらのエンジンはシリアルに使用されます。アプリケーション・コードを CPU で実行する場合、GPU のスケジュールが非効率になります。これにより、誤ってアプリケーションが GPU 依存であると解釈される可能性があります。

計算タスク・リファレンスと **[GPU Utilization (GPU 利用率)]** メトリックに基づいて GPU 実行フェーズを特定します。そして、タスクの作成とキューへの配置のオーバーヘッドを定義します。

計算タスクを調査するには、**[Graphics (グラフィックス)]** ウィンドウに切り替えて、スレッドごとに GPU 上で実行しているワークの種類 (レンダリングまたは計算) を調べます。**[Computing Task (計算タスク)]** グループを選択して、テーブルでタスクのパフォーマンス特性を調べます。

Grouping: Computing Task

Computing Task	Computing Task					EU Array			EU Threads Occupancy	Computing Threads Started
	Total Time	Average T...	Insta...	SIMD Width	SVM Usage ...	Active	Stalled ▼	Idle		
Matrix1<float>	297.870ms	297.870ms	1	8		63.2%	36.7%	0.0%	95.1%	131,072
[Outside any task]	0ms					2.5%	7.2%	90.3%	6.4%	0
clEnqueueReadBufferR	0.007ms	0.007ms	1			0.0%	0.0%	100.0%	0.0%	0

`iso3dfd_omp_offload` コードのほかの実装をプロファイルするには、サンプルの README ファイルに従ってください。

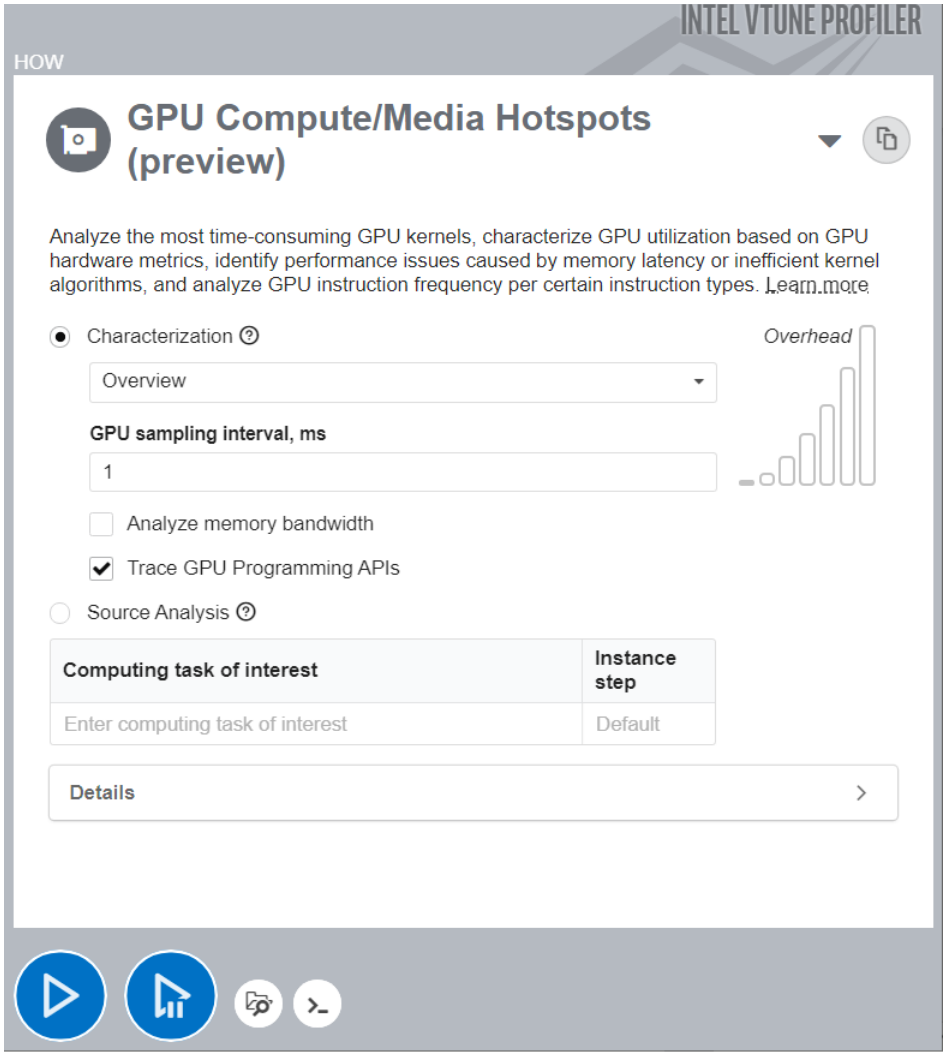
次のセクションでは、「GPU 計算/メディア・ホットスポット解析」(英語) を使用して調査を続けます。

OpenMP* オフロード・アプリケーションの GPU 計算/メディア・ホットスポット解析を実行する

必要条件: まだ行っていない場合は、GPU 解析を実行するためシステムを準備します。「GPU 解析用にシステムをセットアップ」(英語) を参照してください。

解析を実行するには、次の操作を行います。

1. **[Accelerators (アクセラレーター)]** グループで **[GPU Compute/Media Hotspots (GPU 計算/メディア・ホットスポット)]** 解析タイプを選択します。
2. 前のセクションで説明したように解析オプションを設定します。
3. **[Start (開始)]** ボタンをクリックして、解析を実行します。



コマンドラインから解析を実行する

コマンドラインから解析を実行するには、次のコマンドを使用します。

- Linux*:

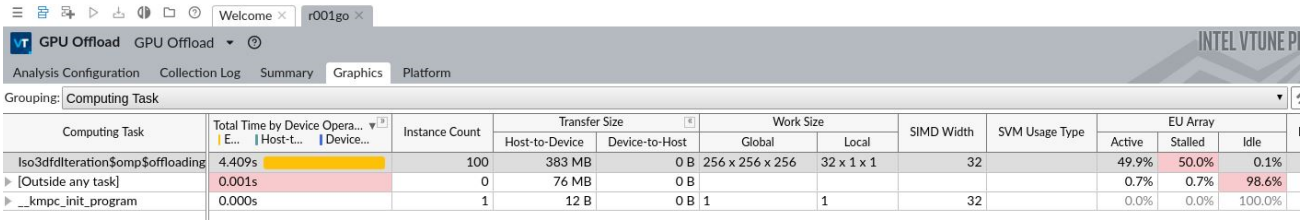
```
vtune -c gpu-hotspots -knob profiling-mode=source-analysis - src/iso3dfd
256 256 256 16 8 64 100
```

- Windows*:

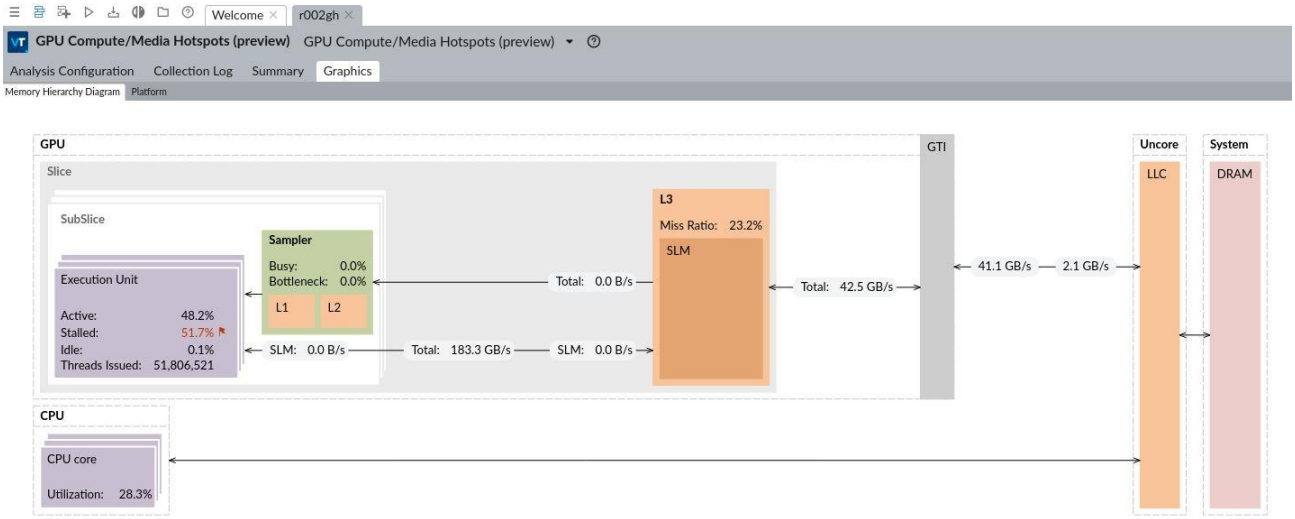
```
vtune -collect gpu-hotspots - iso3dfd.exe 256 256 256 16 8 64 100
```

計算タスクを調査する

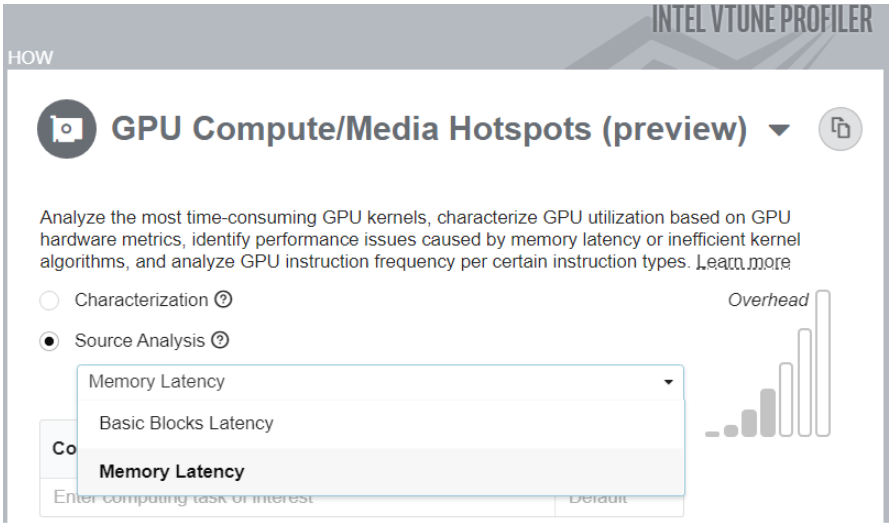
デフォルトの解析設定は、Overview (概要) メトリックセットを含む **[Characterization (特性)]** プロファイルを実行します。**[GPU Offload (GPU オフロード)]** 解析で提供される個々の計算タスクの特性に加えて、GPU メモリー階層ごとに分類されたメモリー帯域幅メトリックを取得できます。



メモリー階層を視覚的に確認するには、**[Memory Hierarchy Diagram (メモリー階層ダイアグラム)]** を使用します。このダイアグラムは、現在の GPU マイクロアーキテクチャーを反映しており、メモリー帯域幅メトリックを提供します。このダイアログからメモリーユニットと実行ユニット間のデータ・トラフィックを理解できます。また、EU ストールの原因となる潜在的なボトルネックも特定できます。



ソースコード・レベルで計算タスクを調査することもできます。例えば、特定のタスクやメモリー・レイテンシーにより費やされた GPU クロックサイクル数を特定できます。これには、**[Source Analysis (ソース解析)]** オプションを使用します。



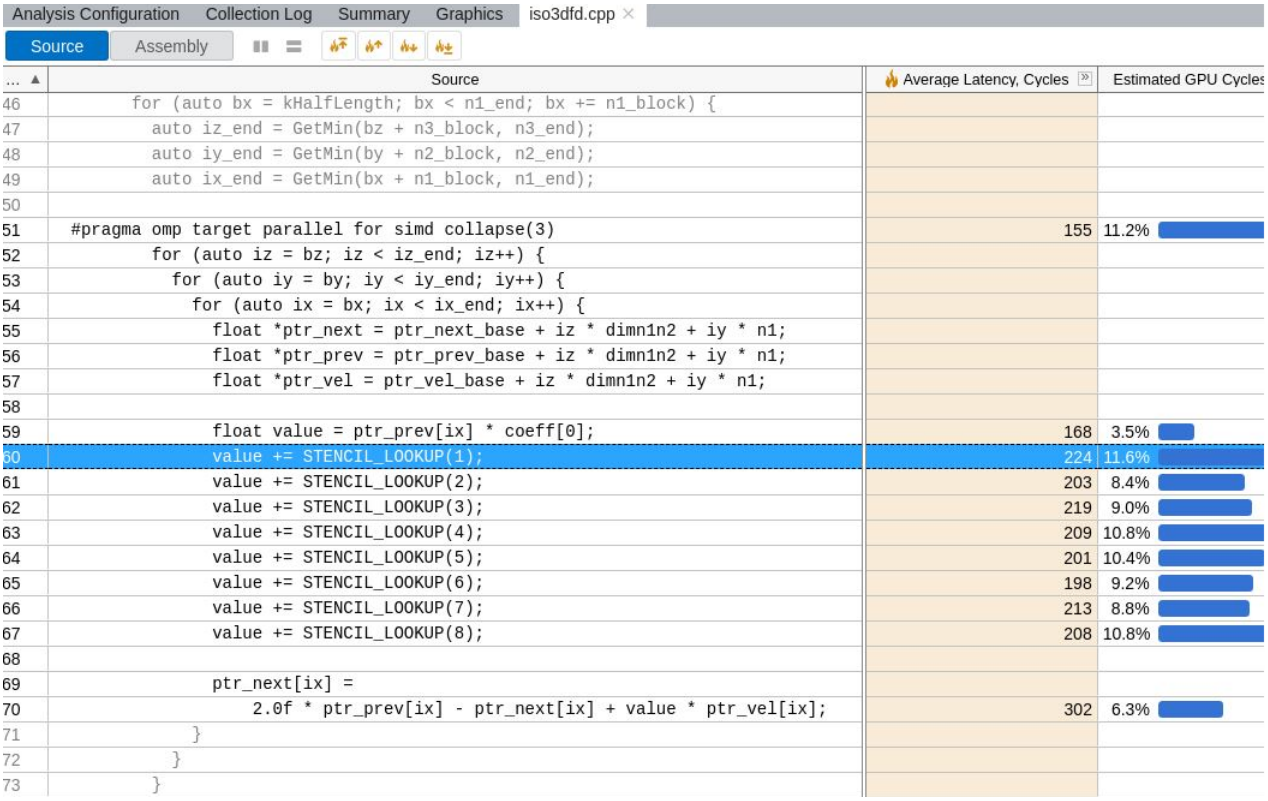
コマンドラインからメモリー・レイテンシーのソース解析を実行する

コマンドラインからメモリー・レイテンシーのソース解析オプションを指定して解析を実行するには、次のコマンドを使用します。

- Linux*:

```
vtune -c gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=mem-latency -r iso_ghs_src-analysis_mem - src/iso3dfd 256 256 256 16 8 64 100
```

ソースビューで、オフロードカーネルの **[Average Latency Cycles (平均レイテンシー・サイクル)]** を調べます。



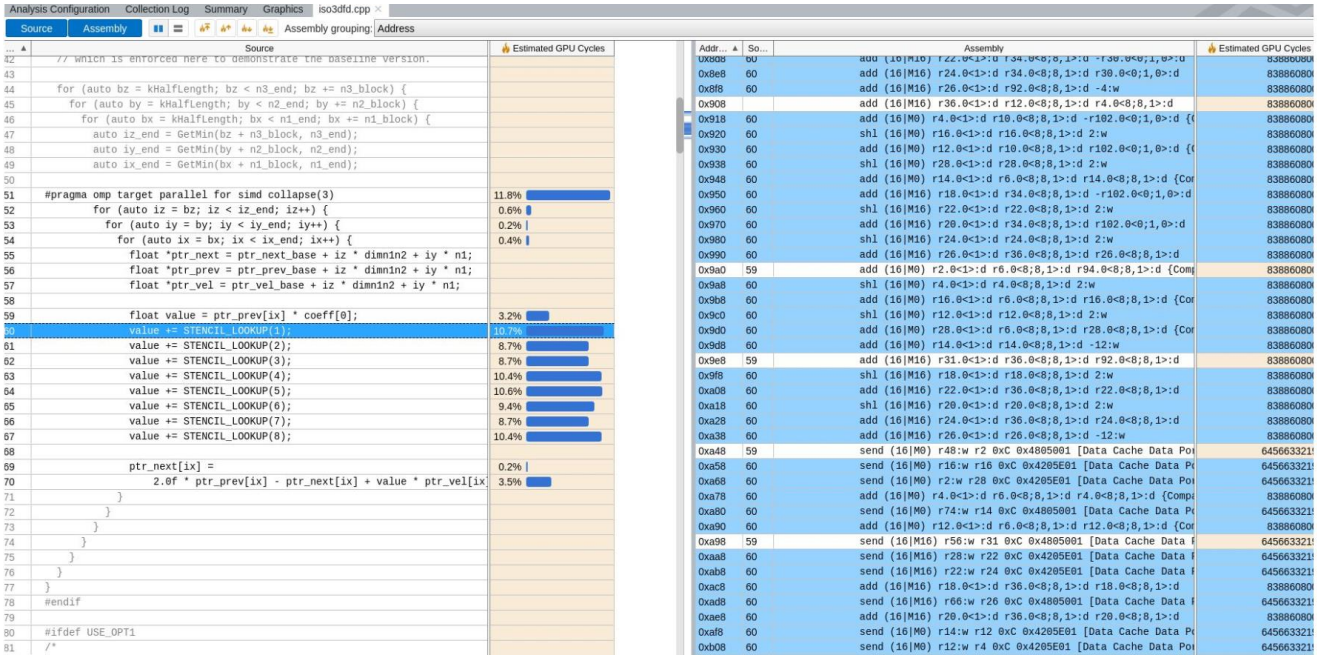
コマンドラインから基本ブロック・レイテンシーのソース解析を実行する

コマンドラインから基本ブロック・レイテンシーのソース解析オプションを指定して解析を実行するには、次のコマンドを使用します。

- Linux*:

```
vtune -c gpu-hotspots -knob profiling-mode=source-analysis -r iso_ghs_src-analysis - src/iso3dfd 256 256 256 16 8 64 100
```

ソースビューで、オフロードカーネルの **[Average Latency Cycles (平均レイテンシー・サイクル)]** を調べます。



注

このレシピの情報は、インテル® VTune™ プロファイラー・デベロッパー・フォーラム (英語) を参照してください。

関連情報

- インテル® VTune™ プロファイラーを使用してインテル® GPU 向けにアプリケーションを最適化 (英語)
- HPC パフォーマンス特性解析 (英語)
- GPU オフロード解析 (英語)
- GPU 計算/メディア・ホットスポット解析 (英語)

DPC++ アプリケーションのプロファイル

このレシピは、DPC++ (データ並列 C++) アプリケーションを作成してコンパイルする方法を紹介します。また、インテル® VTune™ プロファイラーを使用して DPC++ アプリケーションの GPU 解析を実行し、結果を検証する方法も示します。

コンテンツ・エキスパート: Jackson Marusarz (英語)

- 使用するもの
- 手順:
    1. DPC++ アプリケーションを作成してコンパイルする
    2. DPC++ アプリケーションの GPU 解析を実行する
    3. 収集データを解析する

使用するもの

このレシピの最小ハードウェア要件とソフトウェア要件は次のとおりです。

- **アプリケーション:** `matrix_multiply`。このサンプル・アプリケーションは、インテル® oneAPI ツールキットのサンプルコード・パッケージ (英語) に含まれています。
- **コンパイラー:** DPC++ アプリケーションをプロファイルするには、インテル® oneAPI ツールキット (英語) に含まれる `dpcpp` コンパイラーが必要です。
- **ツール:** インテル® VTune™ プロファイラー: GPU 計算/メディア hotspot 解析

注

    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **マイクロアーキテクチャー:**
    - 第 9 世代インテル® プロセッサー・グラフィックス (Gen9)
    - インテル® マイクロアーキテクチャー開発コード名 Kaby Lake または Coffee Lake
- **オペレーティング・システム:** Linux* カーネル 4.14 以降 (GPU ターゲット・プロファイルの実行)
- **グラフィカル・ユーザー・インターフェイス:**
    - GTK+* (2.10 以降、2.18 以降を推奨)
    - Pango* (1.14 以降)
    - X.Org (1.0 以降、1.7 以降を推奨)

DPC++ アプリケーションを作成してコンパイルする

独自の DPC++ アプリケーションをコンパイルする場合、oneAPI DPC++ コンパイラー・オプションの `-gline-tables-only` と `-fdebug-info-for-profiling` を必ず指定してください。これにより、パフォーマンス解析に必要なデバッグ情報が生成されます。

`matrix_multiply` サンプルをビルドするには、次の操作を行います。

1. サンプル・ディレクトリーに移動します。

   ```
   cd <sample_dir>/VtuneProfiler/matrix_multiply
   ```

2. `src` ディレクトリーの `multiply.cpp` ファイルには、行列乗算のいくつかの DPC++ バージョンが含まれています。`multiply.h` で対応する `#define MULTIPLY` 行を編集して、バージョンを選択します。
3. サンプル DPC++ アプリケーションをコンパイルします。

   ```
   cmake .
   make
   ```

   `matrix.dpcpp` 実行ファイルが生成されます。

   プログラムを削除するには、次のコマンドを実行します。

   ```
   make clean
   ```

   `make` コマンドによって作成された実行ファイルとオブジェクト・ファイルが削除されます。

DPC++ アプリケーションの GPU 解析を実行する

必要条件: GPU 解析を実行するシステムを準備します。詳細は、『インテル® VTune™ プロファイラー・ユーザーガイド』 (英語) を参照してください。

1. インテル® VTune™ プロファイラーを起動して、[Welcome (ようこそ)] ページで **[New Project (新規プロジェクト)]** をクリックします。

   [Create a Project (プロジェクトの作成)] ダイアログボックスが表示されます。

2. プロジェクトの名前と場所を指定したら、**[Create Project (プロジェクトの作成)]** ボタンをクリックします。

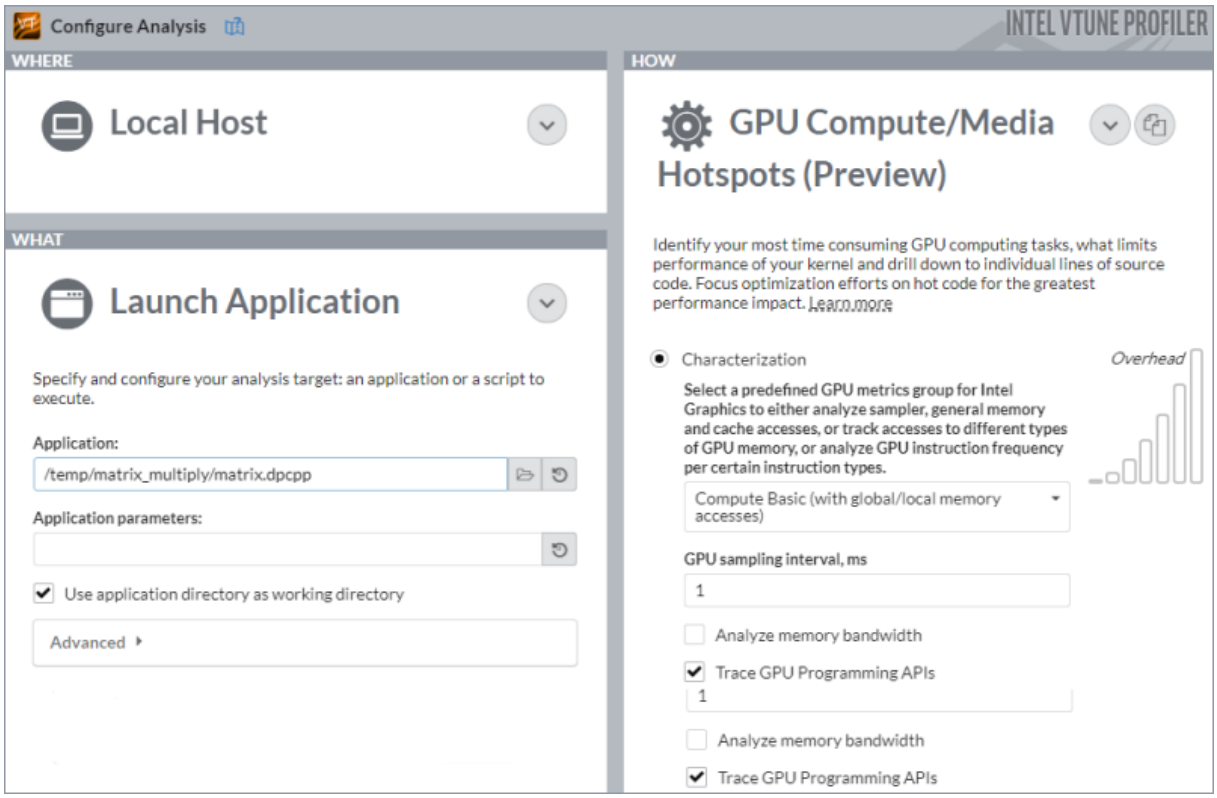
   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

3. **[WHERE (どこを)]** ペインで **[Local Host (ローカルホスト)]** が選択されていることを確認してください。
4. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** ターゲットが選択されていることを確認して、プロファイルする **[Application (アプリケーション)]** として `matrix_multiply` を指定します。

5. **[HOW (どのように)]** ペインで、 ボタンをクリックして **[Platform Analysis (プラットフォーム解析)]** グループの **[GPU Compute/Media Hotspots (GPU 計算/メディア・ホットスポット)]** を選択します。

   これは、インテル® グラフィックスおよびインテル® VTune™ プロファイラーでサポートされるサードパーティー GPU を搭載したプラットフォームで実行中のアプリケーションを最も干渉しない方法で解析します。

6. 最初の GPU 解析では、次のデフォルトのオプションが有効になっていることを確認してください。
   - **[Characterization (特性)]** モードと **[Overview (概要)]** メトリック
   - **[Trace GPU Programming APIs (GPU プログラミング API をトレース)]** チェックボックス



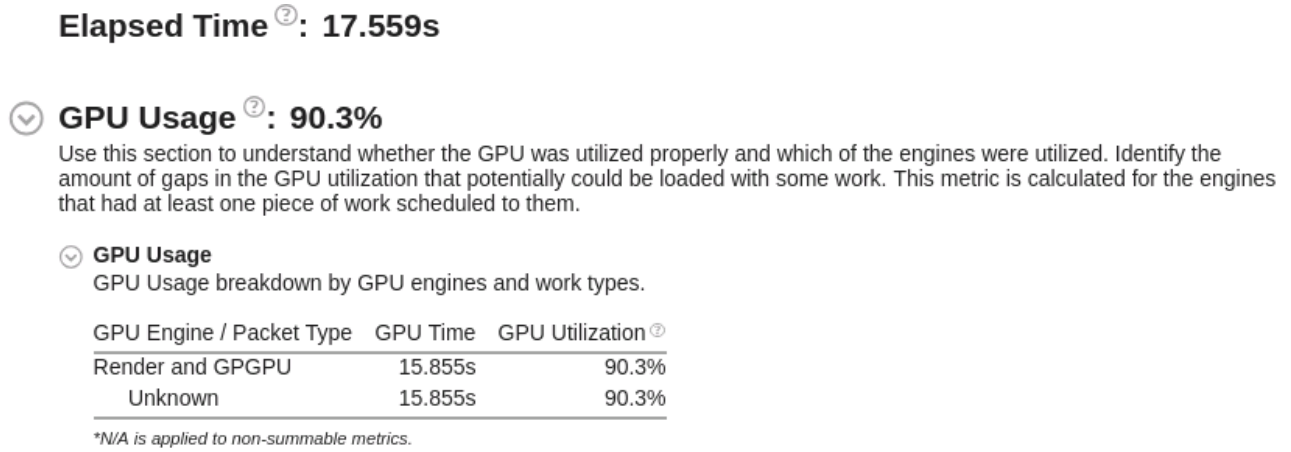
7. ウィンドウの下部にある **[Start (開始)]** ボタンをクリックして解析を開始します。

   コマンドラインからこの設定を実行するには、次のコマンドを入力します。

```
vtune -collect gpu-hotspots -- ./matrix.dpcpp
```

収集データを解析する

[GPU Offload (GPU オフロード)] ビューポイントから解析を開始します。**[Summary (サマリー)]** ウィンドウで、CPU と GPU リソースの利用状況に関する統計を確認して、アプリケーションが GPU 依存かどうかを判断します。この例では、GPU 利用率が非常に高いことが分かります。

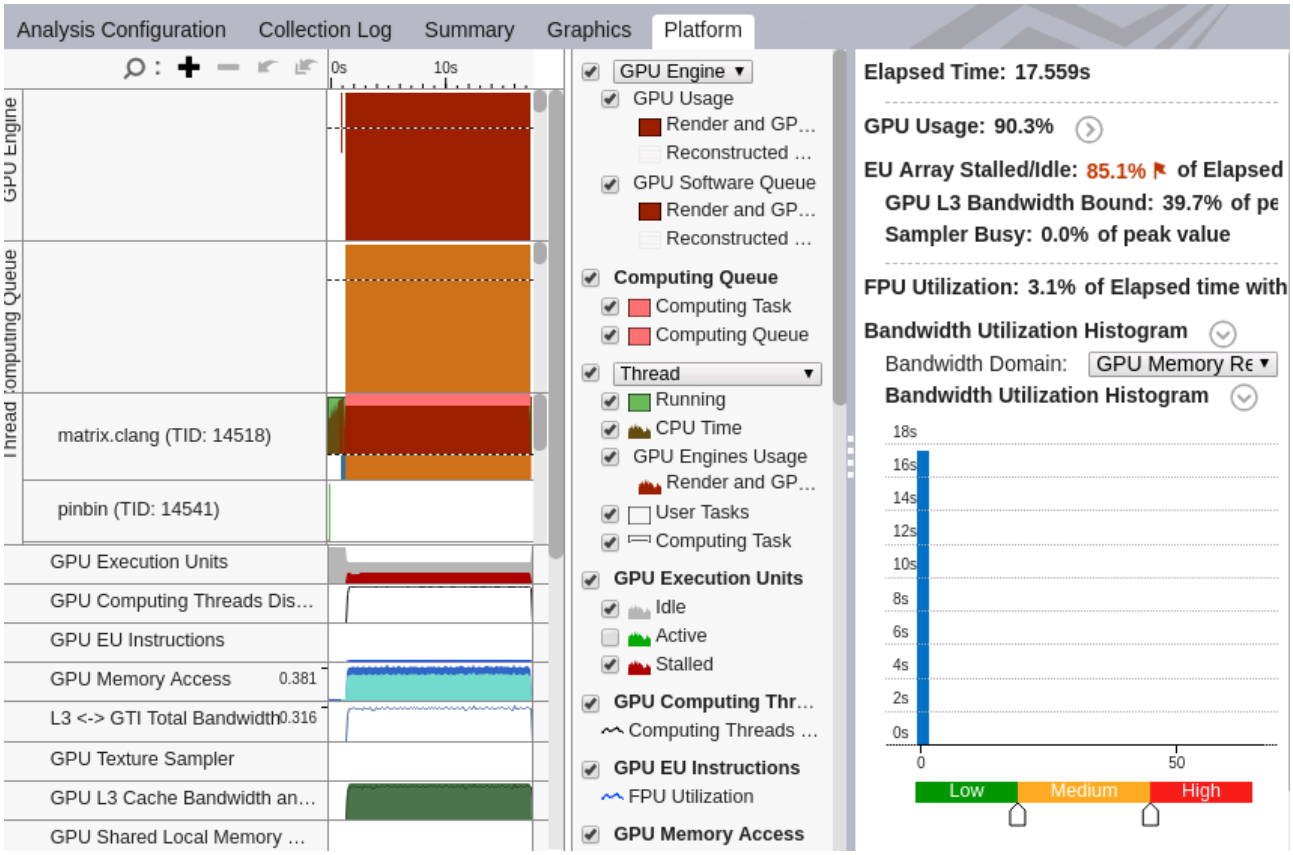


[Platform (プラットフォーム)] ウィンドウに切り替えて、ソフトウェア・キューの GPU 利用率の解析に役立つ基本的な CPU メトリックと GPU メトリックを確認します。このデータは、タイムラインで CPU 利用率に関連付けられます。**[Platform (プラットフォーム)]** ウィンドウで、以下の状況に関する情報を確認します。

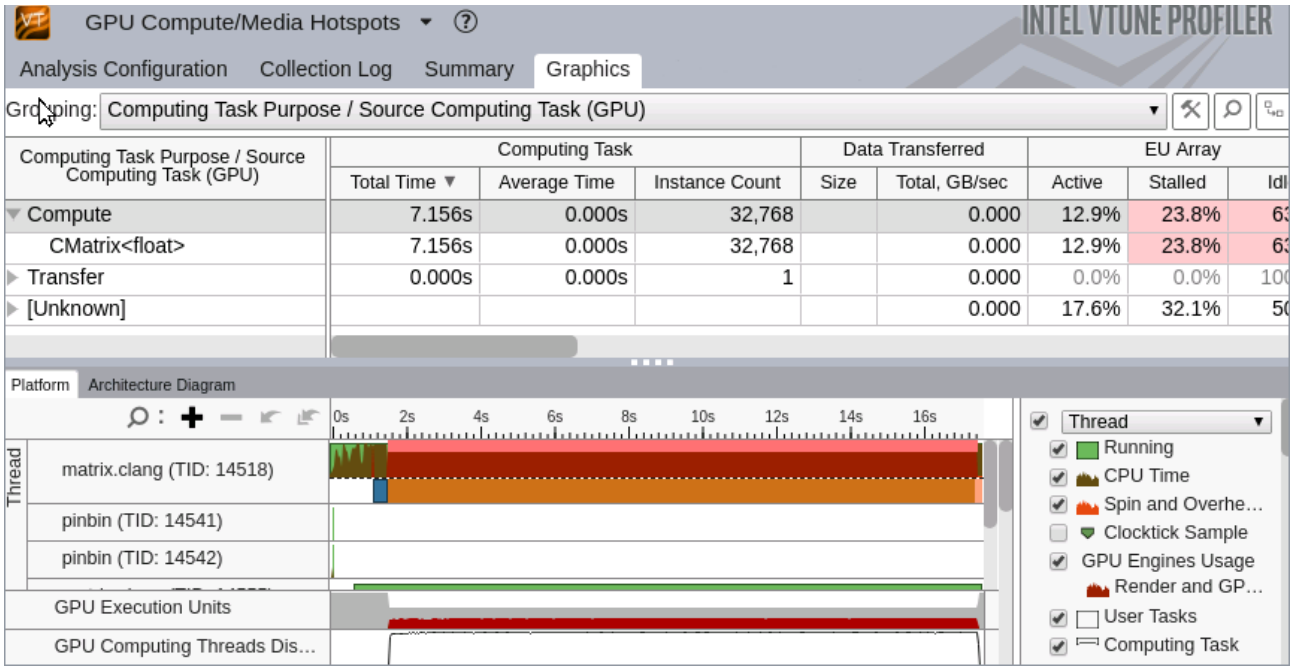
GPU 依存アプリケーション	CPU 依存アプリケーション
プロファイル時間の大部分で GPU がビジー	プロファイル時間の大部分で CPU がビジー
ビジーの間に小さなアイドル時間がある	ビジーの間に大きなアイドル時間がある
GPU ソフトウェア・キューがほとんどゼロにならない	

注

ほとんどのアプリケーションでは、上記のような明らかな状況が見られない場合があります。すべての依存関係を理解するには、詳細な解析が重要です。例えば、ビデオ処理とレンダリングを行う GPU エンジンが交互にロードされる場合、これらのエンジンはシリアルに使用されます。アプリケーション・コードを CPU で実行する場合、GPU のスケジュールが非効率になります。これにより、誤ってアプリケーションが GPU 依存であると解釈される可能性があります。



GPU が長時間にわたってビジーな場合、**[GPU Compute/Media Hotspots (GPU 計算/メディア・ホットスポット)]** ビューポイントに切り替えて、**[Graphics (グラフィックス)]** ウィンドウの **[Platform (プラットフォーム)]** タブでスレッドごとに GPU で実行されているワークの種類 (レンダリングまたは計算) を調べます。



サンプルの README ファイルを使用して、`multiply.cpp` のほかの実装をプロファイルします。

関連情報

- GPU 計算/メディア hotspot 解析 (英語)
- インテル® HD グラフィックスとインテル® Iris® グラフィックスでの GPU アプリケーションの解析 (英語)

コマンドライン・インターフェイスを使用して GPU 上で実行する DPC++ アプリケーションのパフォーマンスを解析

このレシピでは、インテル® VTune™ プロファイラーのコマンドライン・インターフェイス (CLI) を使用して、インテル® GPU にオフロードされたデータ並列 C++ (DPC++) アプリケーションのパフォーマンスを解析する方法を紹介します。また、収集したデータを使用してレポートをカスタマイズする方法も説明します。

コンテンツ・エキスパート: Egor Suldin (英語)、Mariya Petrova (英語)

インテル® VTune™ プロファイラーは、リモート解析、スクリプトコマンド、およびソフトウェアのパフォーマンスを長期的に監視するパフォーマンス・リグレッション・チェック用にコマンドライン・インターフェイス (`vtune` ツール) を提供しています。`vtune` コマンドライン・インターフェイス (CLI) は、GUI で可能なほぼすべてのタスクを実行できる豊富なオプションセットを備えています。コマンドラインから解析を開始して (バックグラウンド・タスクとして、またはリモートシステムで実行して)、結果を表示したり、レポートを生成できます。

このレシピでは、CLI を効率良く使用して、次の目的でホットスポットに関するレポートを生成する方法を説明します。

- `gpu-offload` 解析と `gpu-hotspots` 解析を実行して CPU/GPU 側のホットスポットを調査します。
- 最もホットな GPU 計算タスクを、次の情報とともに表示します。
  - 実行時間
  - データ転送
  - ワークグループ・サイズ
  - SIMD 幅
  - 平均 GPU ハードウェア・メトリック
- ソース/アセンブリー・コード・ビューを生成して、パフォーマンスの問題に関連する可能性のある命令を調査します。

以下は、CLI を効率良く使用して GPU パフォーマンス解析を行うために必要なものと手順です。

- 使用するもの
- 手順:
  1. DPC++ アプリケーションを作成してコンパイルする
  2. GPU 解析の必要条件を確認する
  3. GPU オフロード解析を実行する
  4. GPU 計算/メディア・ホットスポット解析を実行する

使用するもの

以下は、このパフォーマンス解析の最小ハードウェアおよびソフトウェア要件です。

- **アプリケーション:** matrix_multiply_vtune (英語)。このサンプル・アプリケーションは、インテル® oneAPI ツールキットのサンプルコード・パッケージ (英語) に含まれています。
- **コンパイラー:** DPC++ アプリケーションをコンパイルするには、インテル® oneAPI ベース・ツールキット (英語) に含まれるインテル® oneAPI DPC++/C++ コンパイラー (`dpcpp`) (英語) が必要です。
- **ツール:** インテル® VTune™ プロファイラー 2021 - GPU オフロード解析 (英語) および GPU 計算/メディア・ホットスポット解析 (英語)

  注

  - バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
  - インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
  - 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
    - インテル® VTune™ プロファイラー製品ページ
    - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)
- **マイクロアーキテクチャー:**
  - インテル® Iris® Pro グラフィックス 580
  - インテル® マイクロアーキテクチャー開発コード名 Skylake S
- **オペレーティング・システム:**
  - Ubuntu* 20.04 LTS

DPC++ アプリケーションをコンパイルする

1. サンプル・ディレクトリーに移動します。

   ```
   cd <sample_dir>/VtuneProfiler/matrix_multiply_vtune
   ```

2. `src` ディレクトリーの `multiply.cpp` ファイルには、行列乗算のいくつかの DPC++ バージョンが含まれています。`multiply.hpp` の対応する `#define MULTIPLY` 行を編集してバージョンを選択します。
3. サンプル DPC++ アプリケーションをコンパイルします。

   ```
   cmake . && make
   ```

   このコマンドは、`matrix.dpcpp` 実行ファイルを生成します。

   プログラムを削除するには、次のコマンドを実行します。

   ```
   make clean
   ```

このコマンドは、`make` コマンドによって作成された実行ファイルとオブジェクト・ファイルを削除します。

GPU 解析の必要条件を確認する

GPU オフロード解析または **GPU 計算/ホットスポット解析**を実行する前に次のステップを完了します。

1. GPU 解析を実行するためシステムを準備します。「GPU 解析用にシステムをセットアップ」(英語) を参照してください。
2. インテル® ソフトウェア・ツールの環境変数を設定します。

   ```
   source $ONEAPI_ROOT/setvars.sh
   ```

DPC++ アプリケーションの GPU オフロード解析を実行する

GPU オフロード解析を開始点として、アプリケーションが CPU 依存か、GPU 依存かを特定します。データ転送解析により GPU オフロード効率を調査して、パフォーマンス・クリティカルなカーネルを見つけ、さらに詳しく解析して最適化します。

GPU オフロード解析を実行する

CLI で次のコマンドを実行します。

```
vtune -collect gpu-offload -r ./result_gpu-offload -- ./matrix.dpcpp
```

デフォルトでは、インテル® VTune™ プロファイラーはデータ収集後にサマリーレポートを生成します。レポートには、次の情報が含まれます。

- 経過時間
- GPU 利用率
- ホットな計算タスク
- 推奨事項

サマリーレポートを表示するには、次のコマンドを実行します。

```
vtune -report summary -r ./result_gpu-offload
```

データ収集後すぐにサマリーレポートを表示しない場合は、`-finalization-mode` オプションを使用して設定を変更できます。

```
vtune -collect gpu-offload -finalization-mode=none -r ./result_gpu-offload --
./matrix.dpcpp
```

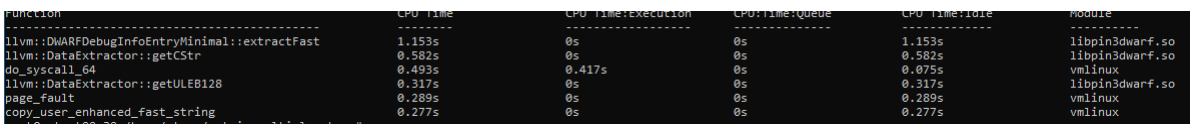
```
root@nntpat99-39:/home/vtune/matrix_multiply_vtune# vtune -report summary -r ./result_gpu-offload
vtune: Using result path `/home/vtune/matrix_multiply_vtune/result_gpu-offload'
vtune: Executing actions 75 % Generating a report                              Elapsed Time: 11.235s
    GPU Utilization: 10.1%
     | GPU utilization is low. Consider offloading more work to the GPU to
     | increase overall application performance.
     |

        GPU Utilization
        GPU Engine        Packet Type  GPU Time     GPU Utilization(%)
        ----------------  -----------  --------  --------------------
        Render and GPGPU  Unknown        1.134s                  10.1%

Hottest GPU Computing Tasks
Computing Task             Total Time  Execution  % of Total Time(%)  Instance Count
-------------------------  ----------  ---------  ------------------  --------------
Matrix1_1<float>              1.146s     1.130s               98.6%               1
zeCommandListAppendBarrier    0.000s         0s                0.0%               0
Collection and Platform Info
    Application Command Line: ./matrix.dpcpp
    Operating System: 5.4.30 DISTRIB_ID=Ubuntu DISTRIB_RELEASE=20.04 DISTRIB_CODENAME=focal DISTRIB_DESCRIPTION="Ubuntu 20.04 LTS"
    Computer Name: nntpat99-39
    Result Size: 98,0 MB
    Collection start time: 12:44:24 20/02/2021 UTC
    Collection stop time: 12:44:36 20/02/2021 UTC
    Collector Type: Event-based sampling driver,Driverless Perf system-wide sampling,User-mode sampling and tracing
    CPU
        Name: Intel(R) Processor code named Skylake
        Frequency: 2.592 GHz
        Logical CPU Count: 8
        Max DRAM Single-Package Bandwidth: 19.000 GB/s
    GPU
        Name: Iris Pro Graphics 580
        Vendor: Intel Corporation
        EU Count: 72
        Max EU Thread Count: 7
        Max Core Frequency: 950.000 MHz
        GPU OpenCL Info
            Version
            Max Compute Units: 72
            Max Work Group Size: 256
            Local Memory: 65,5 KB
            SVM Capabilities

Recommendations:
    GPU Utilization: 10.1%
     | GPU utilization is low. Switch to the for in-depth analysis of host
     | activity. Poor GPU utilization can prevent the application from
     | offloading effectively.
    EU Array Stalled/Idle: 69.8% of Elapsed time with GPU busy
     | GPU metrics detect some kernel issues. Use GPU Compute/Media Hotspots
     | (preview) to understand how well your application runs on the specified
     | hardware.
```

その他のレポートを生成して収集データを表示する

- **CPU ホットスポット・レポート**

  このレポートは、実行された関数のリストを、CPU 時間メトリック、モジュール名、ソースファイル・パス、その他のパラメーターとともに表示します。最もパフォーマンス・クリティカルなものから順にホットなプログラム単位も表示します。データを表形式で表示するには、`-column`、`-filter`、および `-limit` オプションを使用します。

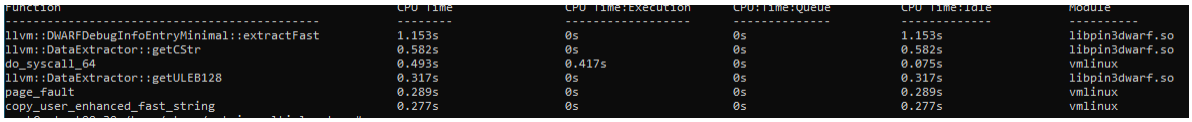
  ```
  vtune -report hotspots -r ./result_gpu-offload
  ```

- **モジュールでフィルター処理され、関数でグループ化された CPU ホットスポット・レポート**

  -filter (英語) オプションを使用して、特定のモジュールなど、レポートの特定の部分に注目できます。そして、-group-by (英語) オプションを使用して、特定のシーケンスで結果をグループ化できます。

```
vtune -report hotspots -r ./result_gpu-offload -group-by=function
-filter module=matrix.dpcpp -q
```



Function	CPU Time	CPU Time:Execution	CPU Time:Queue	CPU Time:Idle	Module
llvm::DWARFDebugInfoEntryMinimal::extractFast	1.153s	0s	0s	1.153s	libpin3dwarf.so
llvm::DataExtractor::getCStr	0.582s	0s	0s	0.582s	libpin3dwarf.so
do_syscall_64	0.493s	0.417s	0s	0.075s	vmlinux
llvm::DataExtractor::getULEB128	0.317s	0s	0s	0.317s	libpin3dwarf.so
page_fault	0.289s	0s	0s	0.289s	vmlinux
copy_user_enhanced_fast_string	0.277s	0s	0s	0.277s	vmlinux

関数名、モジュール、ソースファイル・パス、計算タスクなどで生成されたデータをグループ化できます。

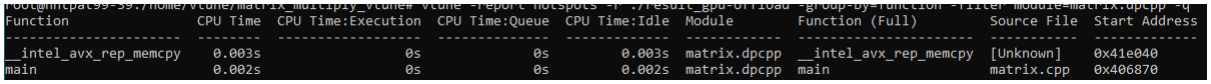
特定の結果で利用可能なグループ (英語) を表示するには、次のコマンドを実行します。

```
vtune -report hotspots -r ./result_gpu-offload -group-by=?
```

- **昇順/降順でソートされた CPU ホットスポット**

  sort-desc (英語) と sort-asc (英語) オプションを使用して、ホットスポットに関する特定の情報を降順または昇順でソートできます。最大 3 つのカラムの順番を指定することが可能です。

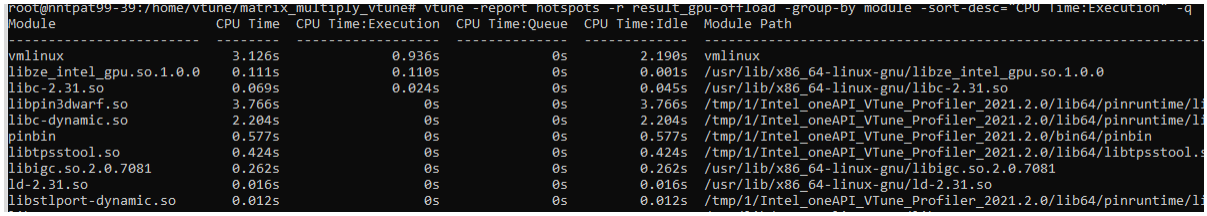
```
vtune -report hotspots -r result_gpu-offload -group-by module
-sort-desc="CPU Time:Execution" -q
```



Function	CPU Time	CPU Time:Execution	CPU Time:Queue	CPU Time:Idle	Module	Function (Full)	Source File	Start Address
__intel_avx_rep_memcpy	0.003s	0s	0s	0.003s	matrix.dpcpp	__intel_avx_rep_memcpy	[Unknown]	0x41e040
main	0.002s	0s	0s	0.002s	matrix.dpcpp	main	matrix.cpp	0x406870

以下に、別の例を示します。

```
vtune -report hotspots -r result_gpu-offload -group-by module
-sort-asc=”CPU Time:Idle” -q
```



Module	CPU Time	CPU Time:Execution	CPU Time:Queue	CPU Time:Idle	Module Path
vmlinux	3.126s	0.936s	0s	2.190s	vmlinux
libze_intel_gpu.so.1.0.0	0.111s	0.110s	0s	0.001s	/usr/lib/x86_64-linux-gnu/libze_intel_gpu.so.1.0.0
libc-2.31.so	0.069s	0.024s	0s	0.045s	/usr/lib/x86_64-linux-gnu/libc-2.31.so
libpin3dwarf.so	3.766s	0s	0s	3.766s	/tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinruntime/li
libc-dynamic.so	2.204s	0s	0s	2.204s	/tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinruntime/li
pinbin	0.577s	0s	0s	0.577s	/tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/bin64/pinbin
libtpsstool.so	0.424s	0s	0s	0.424s	/tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/libtpsstool.s
libigc.so.2.0.7081	0.262s	0s	0s	0.262s	/usr/lib/x86_64-linux-gnu/libigc.so.2.0.7081
ld-2.31.so	0.016s	0s	0s	0.016s	/usr/lib/x86_64-linux-gnu/ld-2.31.so
libstlport-dynamic.so	0.012s	0s	0s	0.012s	/tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinruntime/li

特定の結果で利用可能なカラム (英語) を表示するには、次のコマンドを実行します。

```
Vtune -report hotspots -r ./result_gpu-offload -column=?
```

レポートデータには、**CPU Time:Self (CPU 時間: セルフ)**、**Module (モジュール)**、**Source File (ソースファイル)** などのカラムがあります。

- **上位 ‘n’ 個の時間がかかるプログラムモジュールのレポート**

  limit (英語) オプションを使用して上位 ‘n’ 個のホットスポットに関する情報を表示できます。例えば、アプリケーションの上位 5 個の時間がかかるプログラムモジュールの詳細を理解するには、次のコマンドを実行します。

```
Vtune -report hotspots -r result_gpu-offload -group-by module
-sort-desc="CPU Time" -limit=5 -q
```

```
Module                  CPU Time  CPU Time:Execution  CPU Time:Queue  CPU Time:Idle  Module Path
----------------------  --------  ------------------  --------------  -------------  ----------------------------------------------------------
libze_intel_gpu.so.1.0.0    0.111s             0.110s              0s         0.001s  /usr/lib/x86_64-linux-gnu/libze_intel_gpu.so.1.0.0
[Outside any module]        0.001s                 0s              0s         0.001s  [Unknown]
libstdc++.so.6.0.28         0.001s                 0s              0s         0.001s  /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28
i915.ko                     0.002s                 0s              0s         0.002s  /lib/modules/5.4.30/kernel/drivers/gpu/drm/i915/i915.ko
linker                      0.003s                 0s              0s         0.003s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinrun
libalteracl.so              0.005s                 0s              0s         0.005s  /opt/intel/oneapi/compiler/2021.1.1/linux/lib/oclfpga/ho
libxed.so                   0.005s                 0s              0s         0.005s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinrun
matrix.dpcpp                0.005s                 0s              0s         0.005s  /home/vtune/matrix_multiply_vtune/matrix.dpcpp
libgcc_s.so.1               0.007s                 0s              0s         0.007s  /usr/lib/x86_64-linux-gnu/libgcc_s.so.1
libstlport-dynamic.so       0.012s                 0s              0s         0.012s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinrun
```

- **(GPU にオフロードされた) 計算タスクでグループ化された、転送カラムを含むホットスポット・レポート**

  このコマンドは、GPU 計算タスクでグループ化されたホットスポット情報と、CPU と GPU 間の転送サイズと転送時間の詳細を表示します。

  ```
  vtune -report hotspots -r ./result_gpu-offload -group-by=computing-task
  -column=Transfer -q
  ```

  レポートには、それぞれの計算タスクに起因するデータ転送が含まれます。

```
Module           CPU Time  CPU Time:Execution  CPU Time:Queue  CPU Time:Idle  Module Path
---------------  --------  ------------------  --------------  -------------  ----------------------------------------------------------------------
libpin3dwarf.so     3.766s                 0s              0s         3.766s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinruntime/libpin3dwarf.so
vmlinux             3.126s              0.936s             0s         2.190s  vmlinux
libc-dynamic.so     2.204s                 0s              0s         2.204s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinruntime/libc-dynamic.so
pinbin              0.577s                 0s              0s         0.577s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/bin64/pinbin
libtpsstool.so      0.424s                 0s              0s         0.424s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/libtpsstool.so
```

- **GPU オフロード計算タスクでグループ化された、時間カラムを含むホットスポット・レポート**

  このコマンドは、オフロード計算タスクでグループ化されたホットスポット情報と、CPU と GPU 間の転送時間の詳細を表示します。

  ```
  vtune -report hotspots -r ./result_gpu-offload -group-by=computing-
  task-offload -column='Time' -q
  ```

```
Column filter is ON.
Computing Task                Total Time:Host-to-Device Transfer  Total Time:Device-to-Host Transfer  Transfer Size  Transfer Size:Host-to-Device  Transfer Size:Device-to-Host
----------------------------  ----------------------------------  ----------------------------------  -------------  ----------------------------  ----------------------------
Matrix1_1<float>                                         0.010s                              0.006s       16,8 MB                       0,0 B                      16,8 MB
zeCommandListAppendBarrier                                   0s                                  0s          0,0 B                       0,0 B                       0,0 B
```

GPU 計算/メディア・ホットスポット解析を実行する

次に、**GPU 計算/メディア・ホットスポット**解析を実行します。この解析は、GPU 依存のアプリケーションやそのステージのパフォーマンスをさらに向上するのに役立ちます。

解析を実行するには、次のコマンドを使用します。

```
vtune -collect gpu-hotspots -r ./result_gpu-hotspots -- ./matrix.dpcpp
```

サマリーレポートを表示するには、次のコマンドを実行します。

```
vtune -report summary -r ./result_gpu-hotspots
```

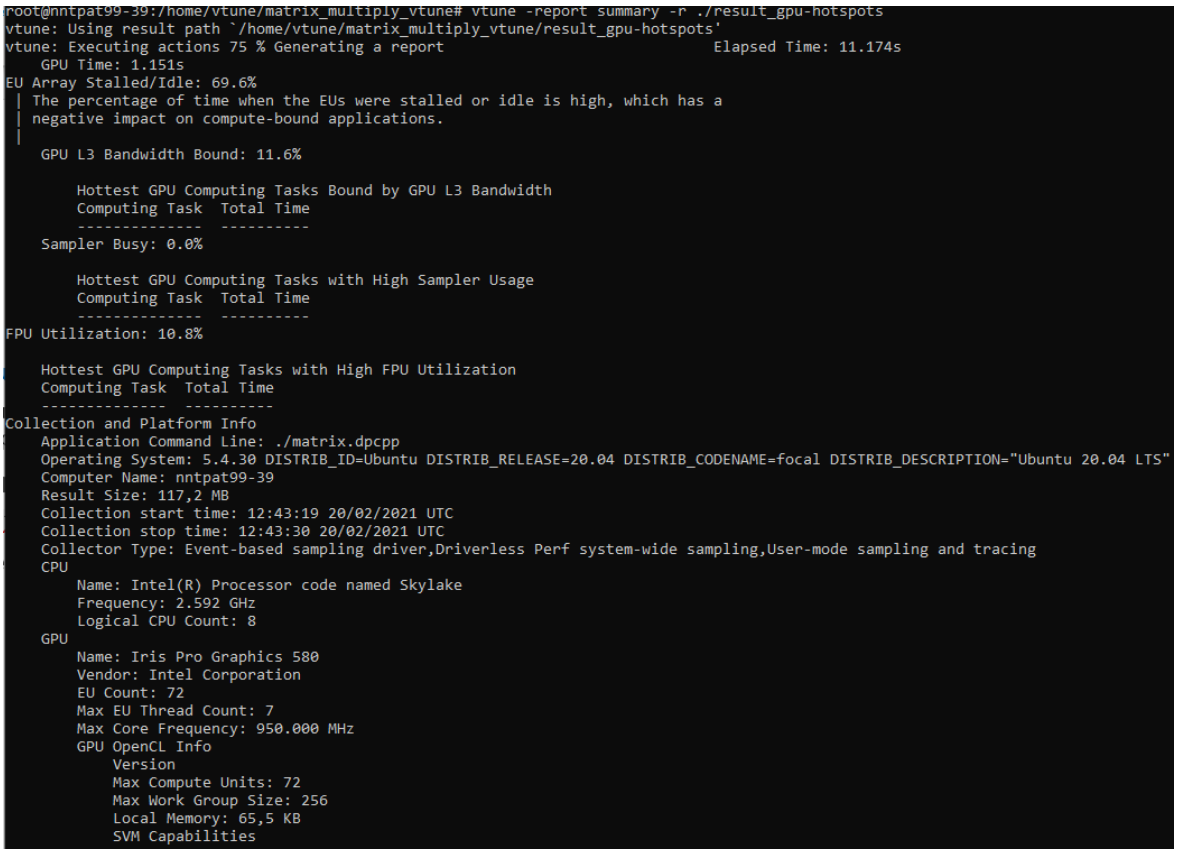
```
Column filter is ON.
Computing Task            Total Time  Total Time:Execution  Total Time:Host-to-Device Transfer  Total Time:Device-to-Host Transfer  Total Time:Synchronization
------------------------  ----------  --------------------  ----------------------------------  ----------------------------------  --------------------------
Matrix1_1<float>             1.146s                1.130s                              0.010s                              0.006s                          0s
zeCommandListAppendBarrier   0.000s                   0s                                 0s                                  0s                       0.000s
```

その他のレポートを生成して収集データを表示する

- **計算タスクと L3 メトリック**

  次のコマンドは計算タスクの L3 メトリックのみをリストするレポートを生成します。

  ```
  vtune -report hotspots -r result_gpu-hotspots -group-by=computing-task
  -column='L3' -q
  ```



- **動的命令数と SIMD 利用率**

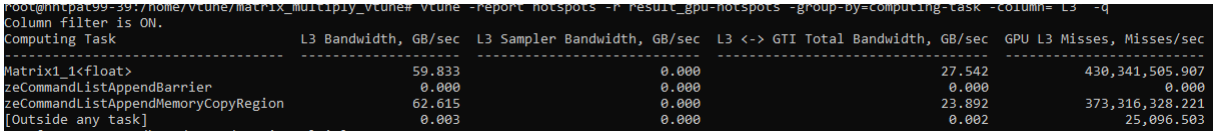
  特性化モードで解析を実行して、動的命令数と SIMD 利用率のデータを収集します。

  ```
  vtune -collect gpu-hotspots -knob characterization-mode=instruction-
  count -r ./result_gpu-hotspots_inst-count -- ./matrix.dpcpp
  ```

- **特定の計算タスクのソースコード**

  次のコマンドは、特定の計算タスクのソースコードを取得します。

  ```
  vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object
  computing-task="Matrix1_1<float>" -group-by=gpu-source-line
  -column="Source","GPU Instructions Executed:Int32 & SP Float" -q
  ```



- **特定の計算タスクのアセンブリー・コード**

  次のコマンドは、特定の計算タスクのアセンブリー・コードを取得します。

  ```
  vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object
  computing-task="Matrix1_1<float>" -group-by=address -limit=5 -q
  ```

```
root@nntpat99-39:/home/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object computing-task=
"Matrix1_1<float>" -group-by=gpu-source-line -column="Source","GPU Instructions Executed:Int32 & SP Float" -q
Column filter is ON.
Source Line  Source                                                                          GPU Instructions Executed:Int32 & SP Float
-----------  ------------------------------------------------------------------------------  ------------------------------------------
84               accessor accessorB(bufferB, h, read_only);
85               accessor accessorC(bufferC, h);
86
87               // Execute matrix multiply in parallel over our matrix_range
88               // ind is an index into this range
89               h.parallel_for<class Matrix1_1<TYPE>>(matrix_range,[=](cl::sycl::id<2> ind) {                                  655,360
90                 int k;
91                 TYPE acc = 0.0;
92                 for (k = 0; k < NUM; k++) {                                                                            301,858,816
93                   // Perform computation ind[0] is row, ind[1] is col
94                   acc += accessorA[ind[0]][k] * accessorB[k][ind[1]];                                                1,073,741,824
```

- **CSV ファイルとしてレポートを保存**

  -report-output (英語) オプションを使用して、生成されたレポートをファイルに保存します。.csv 形式のレポートを生成するには、-format (英語) と -csv-delimiter (英語) オプションを使用します。

  ```
  vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object
  computing-task="Matrix1_1<float>" -group-by=address -limit=5
  -report-output=result.csv -format=csv -csv-delimiter=comma -q
  ```

```
root@nntpat99-39:/home/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object comput
ing-task="Matrix1_1<float>" -group-by=address -limit=5 -q
Address  Assembly                                                    Source Line  GPU Instructions Executed  SIMD Utilization(%)
-------  ----------------------------------------------------------  -----------  -------------------------  -------------------
0        Block 1:
0        (W) mov (8|M0) r101.0<1>:ud r0.0<1;1,0>:ud                  89                           131,072                100.0%
0x10     (W) or (1|M0) cr0.0<1>:ud cr0.0<0;1,0>:ud 0x4C0:uw {Switch}  89                           131,072                100.0%
0x20     (W) mul (1|M0) r10.3<1>:d r14.2<0;1,0>:d r101.6<0;1,0>:d                                 131,072                100.0%
0x30     (W) mul (1|M0) r10.4<1>:d r14.1<0;1,0>:d r101.1<0;1,0>:d                                 131,072                100.0%
```

注
このレシピの情報は、インテル® VTune™ プロファイラー・デベロッパー・フォーラム (英語) を参照してください。

関連情報

- インテル® VTune™ プロファイラーでインテル® GPU 向けにアプリケーションを最適化 (英語)
- GPU オフロード解析 (英語)
- コマンドラインからの GPU オフロード解析 (英語)
- GPU 計算/メディア・ホットスポット解析 (英語)
- コマンドラインからの GPU 計算/メディア・ホットスポット解析 (英語)

FPGA 上での DPC++ アプリケーションのプロファイル

このレシピは、FPGA 上で DPC++ (データ並列 C++) アプリケーションをプロファイルします。このレシピでは、インテル® VTune™ プロファイラーの CPU/FPGA 相互作用解析タイプ (プレビュー機能) に統合されている AOCL プロファイラーを使用します。

コンテンツ・エキスパート: Dmitry Ryabtsev (英語)

- 使用するもの
- 手順:
    1. ツールキットをインストールして設定する
    2. サンプル・アプリケーションをビルドする
    3. CPU/FPGA 相互作用解析を実行する
    4. 結果を解釈する

使用するもの

このレシピの最小ハードウェア要件とソフトウェア要件は次のとおりです。

- **アプリケーション**: `crr`。この FPGA サンプルは、インテル® oneAPI DPC++ コンパイラー・サンプルのリポジトリー (英語) から入手できます。
- **コンパイラー:** DPC++ アプリケーションをプロファイルするには、インテル® oneAPI ツールキット (英語) に含まれる `dpcpp` コンパイラーが必要です。
- **ツール**:
    - インテル® oneAPI ベース・ツールキット (Linux* 版) (英語)
    - oneAPI ベース・ツールキット用インテル® FPGA アドオン (英語)
    - インテル® VTune™ プロファイラー: CPU/FPGA 相互作用解析 (プレビュー機能)

注

- インテル® VTune™ プロファイラーのダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
- ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。

- **オペレーティング・システム**: Ubuntu* 18.04
- **CPU**: インテル® サーバー・プラットフォーム開発コード名 Cascade Lake
- **FPGA**: インテル® プログラマブル・アクセラレーション・カード (インテル® PAC) インテル® Arria® 10 GX FPGA 搭載版または DPC++ 向けインテル® Stratix 10 GX FPGA PAC ボード (およびインストール可能なアドオン)

ツールキットをインストールして設定する

インテル® PAC カードをマシンの PCIe* スロットに装着します。

1. インテル® oneAPI ベース・ツールキット (Linux* 版) (英語) をダウンロードしてインストールします。すべてのデフォルトのオプションを選択して、オンラインまたはオフライン・インストーラーのいずれかを選択します。
2. oneAPI ベース・ツールキット用インテル® FPGA アドオン (英語) をダウンロードします。
3. oneAPI ベース・ツールキット用インテル® FPGA アドオンを展開して、`setup.sh` を実行します。すべてのデフォルトのオプションを選択します。
4. oneAPI 環境をセットアップします。

   ```
   source <oneAPI-install-dir>/setvars.sh
   ```

5. FPGA ボードを装着します。

   ```
   aocl install
   ```

6. 診断コマンドを実行して、すべての診断にパスすることを確認します。

   ```
   aocl diagnose
   ```

サンプル・アプリケーションをビルドする

1. インテル® oneAPI DPC++ コンパイラー・サンプルのリポジトリー (英語) からサンプルコードをダウンロードします。

   ```
   git clone https://github.com/intel/BaseKit-code-samples.git
   ```

2. `crr` サンプルフォルダーを開きます。

   ```
   cd BaseKit-code-samples/FPGAExampleDesigns/crr
   ```

3. `src/CMakeLists.txt` ファイルを開きます。
4. `set(HARDWARE_LINK_FLAGS` から始まる、ハードウェア・フラグをリストしているコード行に移動します。
5. リストに `-Xsprofile` を追加します。
6. サンプルのメイン・ディレクトリーに戻ります。`build` という名前の新しいフォルダーを作成して開きます。

   ```
   mkdir build
   cd build
   ```

7. サンプルをコンパイルします。

   ```
   cmake ..
   make fpga
   ```

   この処理には数時間かかります。処理が終了すると、`crr.fpga` という名前の実行ファイルが生成されます。

これで、FPGA ハードウェア上で `crr.fpga` を実行できます。

CPU/FPGA 相互作用解析を実行する

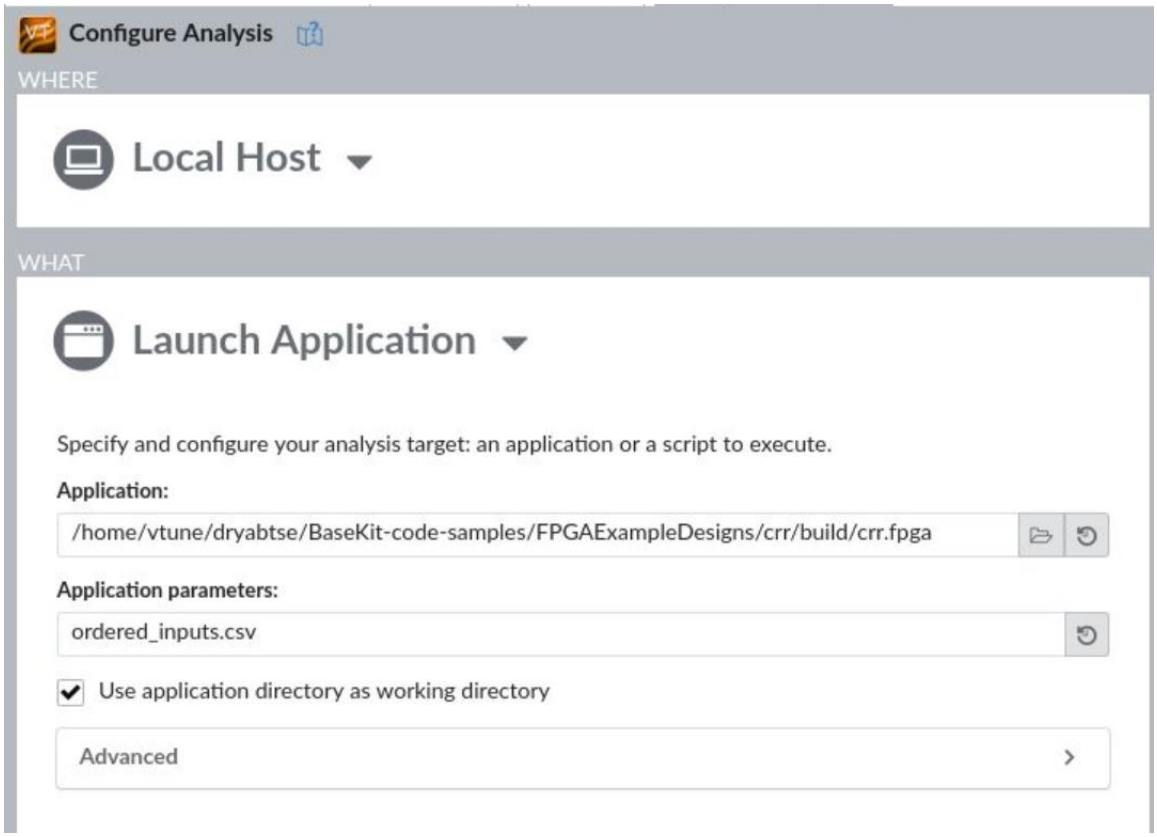
1. インテル® VTune™ プロファイラーを起動して、**[Welcome (ようこそ)]** ページで **[New Project (新規プロジェクト)]** をクリックします。

   [Create a Project (プロジェクトの作成)] ダイアログボックスが表示されます。

2. プロジェクトの名前と場所を指定したら、**[Create Project (プロジェクトの作成)]** ボタンをクリックします。

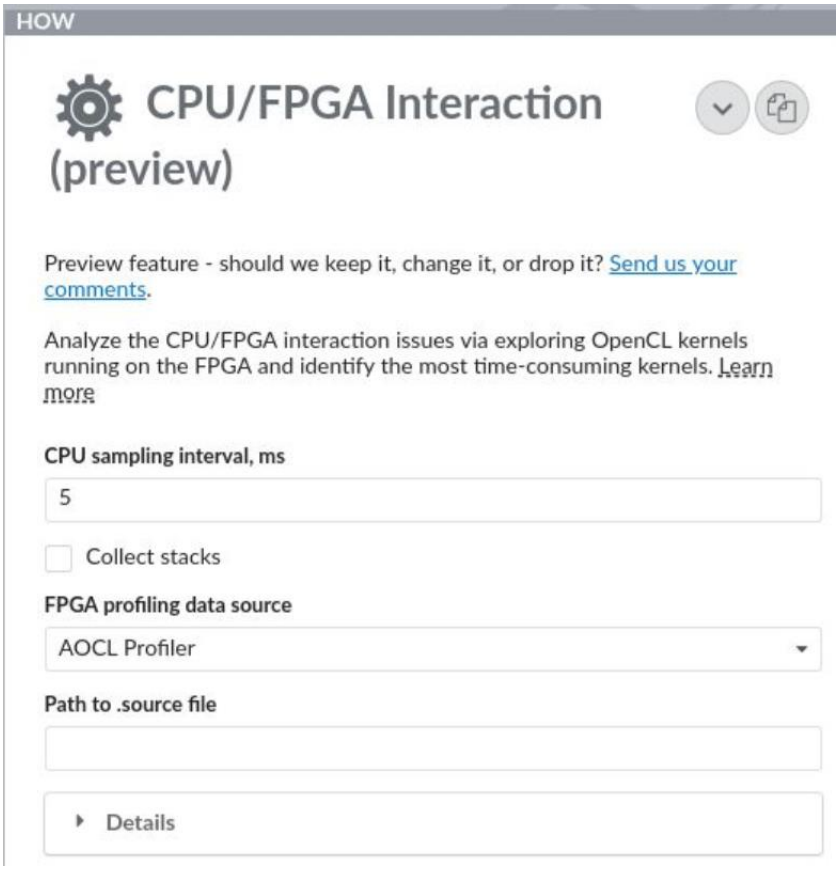
   [Configure Analysis (解析の設定)] ダイアログボックスが開きます。

3. **[WHERE (どこを)]** ペインで、**[Local Host (ローカルホスト)]** を選択します。
4. **[WHAT (何を)]** ペインで、ターゲットとして **[Launch Application (アプリケーションを起動)]** を選択します。
   - **[Application (アプリケーション)]** フィールドに、`crr.fpga` 実行ファイルのパスを指定します。
   - **[Application parameters (アプリケーションのパラメーター)]** フィールドに、`ordered_inputs.csv` を入力します。



5. **[HOW (どのように)]** ペインで、**[Platform Analysis (プラットフォーム解析)]** グループの **[CPU/FPGA Interaction (preview) (CPU/FPGA 相互作用 (プレビュー))]** を選択します。

6. 解析の設定で、**[FPGA profiling data source (FPGA プロファイル・データソース)]** に **[AOCL Profiler (AOCL プロファイラー)]** を選択します。

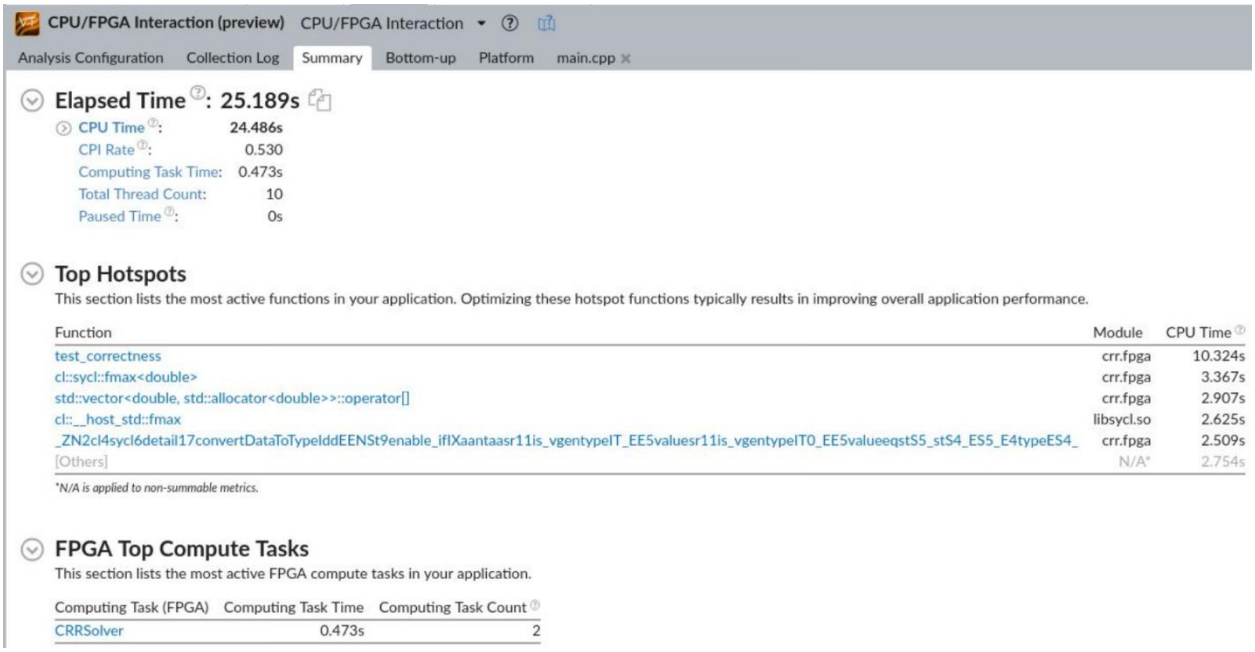


7. ウィンドウの下部にある **[Start (開始)]** ボタンをクリックして解析を開始します。

結果を解釈する

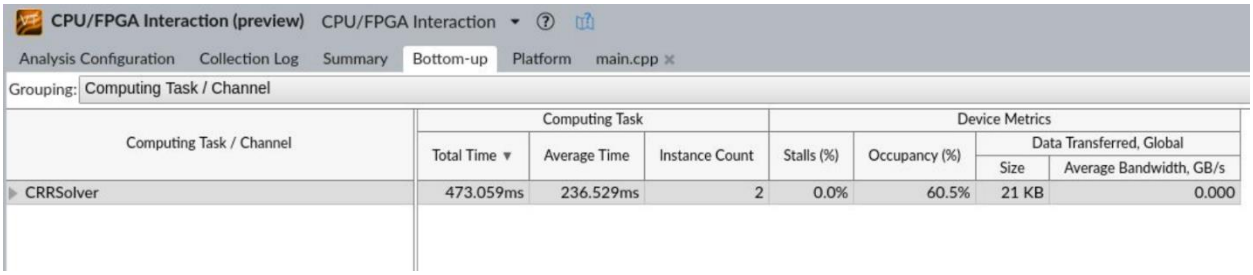
データ収集が完了すると、**[CPU/FPGA Interaction (CPU/FPGA 相互作用)]** ビューポイントでファイナライズされた結果を確認できます。最初に、**[Summary (サマリー)]** ウィンドウで次の詳細を確認します。

- FPGA の上位計算タスク
- CPU の上位のタスクとホットスポット



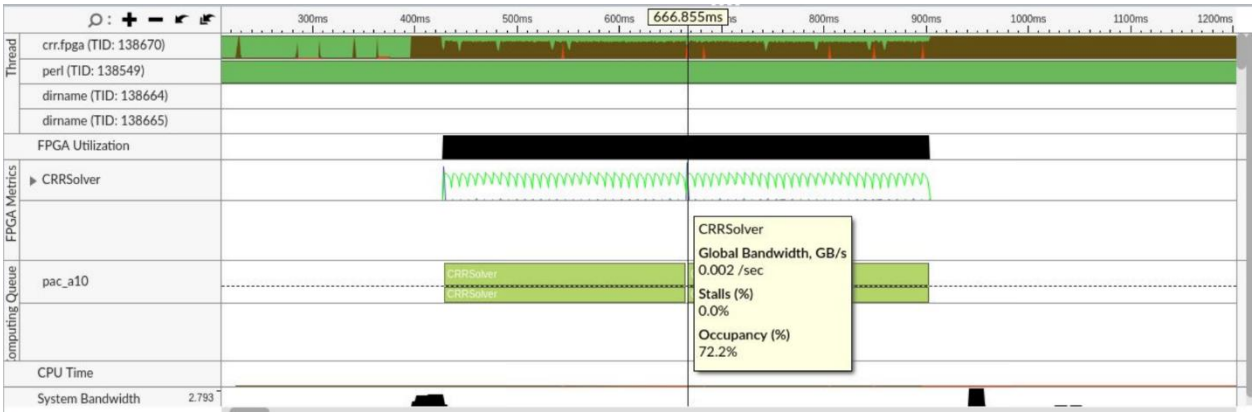
[Bottom-up (ボトムアップ)] ウィンドウに切り替えて、以下を含むカーネルレベルの詳細情報を確認します。

- ストール
- 占有率
- データ転送サイズ
- 転送データの平均帯域幅



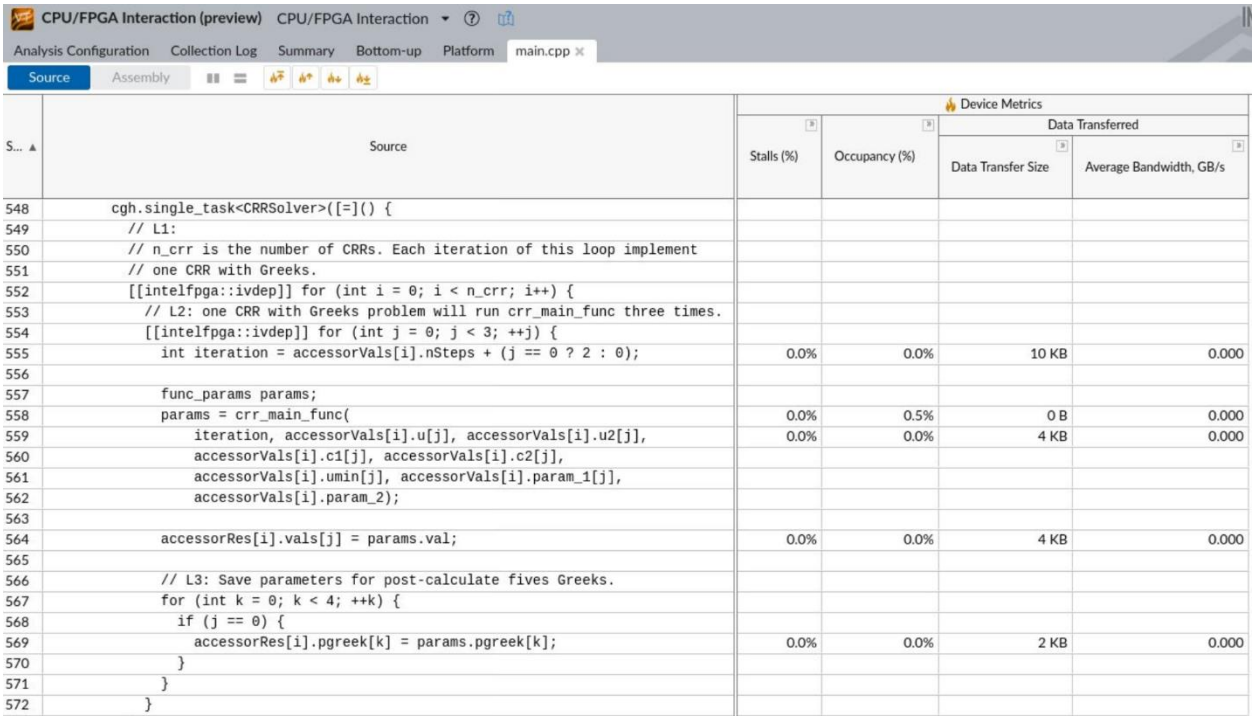
タイムライン・ビューを使用して、カーネル・インスタンスに関する次の情報を確認します。

- 開始時間/終了時間
- 全体のストール
- 占有率
- 帯域幅メトリック



[Bottom-up (ボトムアップ)] ウィンドウで、カーネルを右クリックしてコンテキスト・メニューから **[View Source (ソースを表示)]** を選択します。

[Source (ソース)] ビューが開いて、特定のカーネルソース行のメトリックを確認できます。



関連情報

- 設定レシピ
- CPU/FPGA 相互作用解析 (英語)

インテルのサンプリング・ドライバーを使用しないハードウェア・プロファイル

この一連のレシピは、インテル® VTune™ プロファイラーでドライバーを使用しない Linux* perf ベースのパフォーマンス・プロファイルを設定して、その利点と制限に対する回避策を理解するのに役立ちます。

コンテンツ・エキスパート: Alexey Budankov (英語)、Alexey Bayduraev (英語)

インテル® プロセッサーは、コードがハードウェア・リソースをどの程度有効に利用しているか解析するのに使用できるパフォーマンス・モニタリング・ユニット (PMU) イベントを提供します。インテル® VTune™ プロファイラーは、HPC パフォーマンス特性、メモリーアクセス、マイクロアーキテクチャー全般など、マイクロアーキテクチャー解析タイプで PMU を収集して解析できます。また、解析に必要なサンプリング間隔が短い場合など、必要に応じて、hotspot とスレッド解析タイプでも、デフォルトのユーザーモード・サンプリングの代わりに PMU イベントベース・サンプリングを使用するように設定できます。

PMU イベントベース解析では、インテル® VTune™ プロファイラーはインテルのサンプリング・ドライバー (英語) を使用します。このドライバーをターゲットシステムにインストールするには、管理者権限が必要です。管理者権限がない場合、またはサードパーティーのドライバーのインストールが許可されていない環境の場合、インテル® VTune™ プロファイラーはインテルのサンプリング・ドライバーを介して PMU イベントにアクセスし、ハードウェア・パフォーマンスのボトルネックを特定することができません。そのような場合、インテル® VTune™ プロファイラーは、Linux* の perf パフォーマンス・モニタリング・システムを介してハードウェア・パフォーマンス・モニタリング機能を提供します。

インテル® VTune™ プロファイラーは、次の場合にドライバーを使用しない perf によるハードウェア・イベントベース・サンプリング解析を有効にします。

- インテルのサンプリング・ドライバーがインストールできない (例: root 権限なしでインストールした場合など)
- 非ゼロのスタックサイズでスタックを含む収集が選択され、ドライバーを使用しない収集の要件が満たされている
- ドライバーを使用しない収集を使用するオプションが有効になっており、ドライバーを使用しない収集の要件が満たされている

インテル® VTune™ プロファイラーの最新バージョンは、Linux* perf (ドライバーを使用しない) サポートが拡張されており、インテルのサンプリング・ドライバーベースのソリューションに匹敵するプロファイル機能、収集オーバーヘッド、およびトレースサイズを提供します。しかし、ドライバーを使用しないモードでのインテル® VTune™ プロファイラーの機能は Linux* OS の設定に依存し、以下のレシピに示すようないくつかの制限があります。

注

- インテルのサンプリング・ドライバーで提供されるハードウェア・プロファイル機能と同等のドライバーを使用しない perf 収集を有効にするには、管理者権限で次のシステムオプションを設定する必要があります。

- 解析に使用されたコレクターの種類 (perf またはインテルのサンプリング・ドライバー (SEP)) は、**[Summary (サマリー)]** ウィンドウの **[Collection and Platform Info (収集とプラットフォーム情報)]** セクションで確認できます。

- **使用するもの:**

  インテル® VTune™ プロファイラー (またはインテル® VTune™ Amplifier 2019) は、次の要件を満たす場合にドライバーを使用しないモードを使用できます。

  - **コアとアンコアイベント。**インテル® VTune™ プロファイラーのすべてのハードウェア・イベントベース収集はコア PMU イベントを使用します。メモリーアクセスや IO など一部の解析タイプは、DRAM 帯域幅、QPI/UPI 帯域幅、PCI 帯域幅などのメトリックの収集を可能にするアンコアイベントへのアクセスが必要です。
  - **Linux* カーネル 2.6.32 以降の perf。**Linux* カーネルは、`/sys/bus/event_source/devices/cpu` および `/sys/bus/event_source/devices/uncore_*` ディレクトリーを利用して PMU イベントを認識します。ディレクトリーが空の場合、システム設定が PMU イベント収集をサポートしていない可能性があります。この場合、OS をアップデートするか、インテルのサンプリング・ドライバーをインストールしてください。
  - `/proc/sys/kernel/perf_event_paranoid` の値が 1 以下。

- **制限のレシピ:**
  - システム全体またはユーザープロセスのプロファイルを有効にする
  - コアとアンコアイベントの収集を有効にする
  - マルチプロセスのプロファイルを有効にする
  - マルチコアシステムで多数の PMU イベントをプロファイルする
  - スタックのサンプリングを有効にする
  - コンテキスト・スイッチを収集する
  - カーネル関数のシンボルを解決する
  - NMI ウォッチドックとのリソース競合を回避する
  - 収集オーバーヘッドを軽減する
  - 必要に応じてドライバーを使用しないモードを有効にする

注

インテル® VTune™ プロファイラーに付属の `vtune-self-checker.sh` スクリプトを実行して、解析システムの製品機能を検証できます。このスクリプトは、安定したベンチマークで解析タイプの代表的なセットを実行して、システムでインテル® VTune™ プロファイラーが直面した制限について通知します。診断の推奨事項は、ドライバーを使用しない perf 収集向けにシステムを適切に設定したり、システムを設定できない場合はインテルのサンプリング・ドライバーをインストールするのを支援します。スクリプトを実行するには、次のコマンドを使用します。

```
<vtune_install_dir>/bin64/vtune-self-checker.sh
```

システム全体またはユーザープロセスのプロファイルを有効にする

解析タイプ: すべて。

概念: システム全体の解析は、システムサービスなどを含む、システムで実行中のすべてのプロセスに関するパフォーマンス情報を収集します。

ドライバーを使用しないモードの制限: システム全体またはユーザープロセスのプロファイルを有効にするには、追加の設定が必要です。

ドライバーを使用しないモードでシステム全体の解析を有効にする

1. インテル® VTune™ プロファイラーのプロジェクトを設定します。**[WHAT (何を)]** ペインで **[Profile System (システムをプロファイル)]** ターゲットまたは **[Launch Application (アプリケーションを起動)]** ターゲットを選択して、**[Analyze system-wide (システム全体を解析)]** オプションを有効にします。
2. 次のコマンドを実行して、`/proc/sys/kernel/perf_event_paranoid` ファイルの値を確認します。

   ```
   cat /proc/sys/kernel/perf_event_paranoid
   ```

   値が 1 未満の場合、インテル® VTune™ プロファイラーはシステム全体の収集を続行できます。

3. `perf_event_paranoid` 値が 1 の場合 (収集はユーザープロセスのみに制限されます) または 1 よりも大きい場合 (インテル® VTune™ プロファイラーはドライバーを使用しない perf モードを使用できません)、システム全体の収集を行うため `perf_event_paranoid` 値を 0 に設定します。

   ```
   echo 0 > /proc/sys/kernel/perf_event_paranoid
   ```

注

一部の環境では、`perf_event_paranoid` はセキュリティー・ポリシーによって規制されます。Linux* perf のセキュリティー要件の詳細は、https://www.kernel.org/doc/html/latest/admin-guide/perf-security.html (英語) を参照してください。

インテルのサンプリング・ドライバーの制限: デフォルトでは、インテルのサンプリング・ドライバーはシステム全体の収集をサポートします。しかし、`--per-user` オプションを指定してビルドおよびロードされている場合、収集はユーザープロセスのみに制限されます。

コアとアンコアイベントの収集を有効にする

解析タイプ: メモリーアクセス、HPC パフォーマンス特性、その他のアンコア・イベント・ベースの解析タイプ

コアイベントは、システム全体とユーザープロセスごとのどちらでも収集できます。ドライバーを使用しない perf モードでアンコアイベントを収集するには、システム全体の解析を有効にします。

ドライバーを使用しないモードの制限:

- メモリーアクセス解析は、アンコアイベントにアクセスする必要があり、それらを収集することができない場合実行されません。HPC パフォーマンス特性など、その他の解析タイプは実行されますが、DRAM 帯域幅、OPA インターコネクト帯域幅、パケットレートなど、アンコア・イベント・ベースのメトリックは提供されません。
- ドライバーを使用しないモードでのアンコアイベントの収集は、Intel Atom® プロセッサーではサポートされていません。

ドライバーを使用しないモードでアンコアイベントを収集する

`perf_event_paranoid` 値を 0 に設定してシステム全体のパフォーマンス・モニタリングを有効にします。これは、アンコアイベント収集の必要条件です。

インテルのサンプリング・ドライバーの制限: なし。

マルチプロセスのプロファイルを有効にする

解析タイプ: すべて。

デフォルトでは、Linux* カーネルはパフォーマンス・データの収集に利用可能なメモリーサイズを 518KB に制限します。現在の値を確認するには、次のコマンドを使用します。

```
cat /proc/sys/kernel/perf_event_mlock_kb
```

ドライバーを使用しないモードの制限: マルチコアシステム (64 論理コアを超える) のユーザープロセス収集モードで一部の並列アプリケーション (例えば、ノードごとに複数のランクがある MPI アプリケーション) は、518KB 制限に達しデータ収集が利用できません。

マルチコアシステムでマルチプロセスの収集を有効にする

`perf_event_paranoid` 値を 0 に設定して、システム全体のパフォーマンス・モニタリングを有効にします。

インテルのサンプリング・ドライバーの制限: なし。デフォルトの設定で任意の数のプロセスをプロファイルできます。

マルチコアシステムで多数の PMU イベントをプロファイルする

解析タイプ: マイクロアーキテクチャー全般。

ドライバーを使用しないモードの制限: Linux* perf は、各 CPU で設定されている PMU イベントごとにファイル記述子を割り当てます。そのため、マルチコアシステムで多数のイベントを利用するマイクロアーキテクチャー全般などの解析を行う場合、制限に達しドライバーを使用しないモードでの収集が利用できません。

ドライバーを使用しないモードで多数の PMU イベントをプロファイルする

1. オープンファイルの上限を確認します。

   ```
   ulimit -n
   ```

2. 必要に応じて、`/etc/security/limits.conf` ファイルに次の行を追加するか、既存の行を変更して上限を増やします。
   - `soft nofile 65535`
   - `hard nofile 65535`

注

管理者権限が必要です。

インテルのサンプリング・ドライバーの制限: なし。

スタックのサンプリングを有効にする

解析タイプ: hotspot (ハードウェア・イベントベース・サンプリング・モード)、スレッド (ハードウェア・イベントベース・サンプリング・モードおよびスタック・スティッチ・モード)、HPC パフォーマンス特性 (**[Collect stacks (スタックを収集)]** オプションがオン)、GPU 計算/メディア hotspot (**[Collect stacks (スタックを収集)]** オプションがオン)。

ドライバーを使用しないモードの制限:

- 関数がスタックに多くのデータを割り当てる場合、デフォルトの 1024 バイトのスタックサイズではスタックを完全にアンワインドできない場合があります。この場合、収集データに `[Skipped stack frame(s)]` が表示されます。
- バージョン 3.7 よりも古い Linux* カーネルは、フレームポインター (FP) ベースのスタック・アンワインドのみをサポートしています。つまり、インテル® VTune™ プロファイラーは、フレームポインターを含めない (`-fomit-frame-pointer` コンパイラー・オプション) でビルドされたバイナリーのスタックを提供できません。また、glibc スタックもフレームポインターを含めないでビルドされているため提供できません。

ドライバーを使用しないモードでスタック・アンワインドの問題を回避する

スタックサイズを増やします。次に例を示します。

```
vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-
collection=true -knob stack-size=2048 <application>
```

そうでない場合、インテルのサンプリング・ドライバーに切り替えて、**[Stack size (スタックサイズ)]** オプションを 0 (無制限) に設定します。

注

インテルのサンプリング・ドライバーを使用したスタック・サンプリング収集はカーネル実装に依存するため、通常は新しいカーネルバージョンへの更新が必要であり、追加の製品保守コストが発生する可能性があります。このコストを軽減するため、インテル® VTune™ プロファイラー (およびインテル® VTune™ Amplifier 2019 Update 4 以降) では、スタック収集が有効なすべての解析タイプで、インテルのサンプリング・ドライバーがロードされている場合であっても、ドライバーを使用しないモードを使用します。スタックのサンプリング収集をインテルのサンプリング・ドライバーに切り替える必要がある場合は、カスタム解析タイプを作成して、**[Enable driverless collection (ドライバーを使用しない収集を有効にする)]** オプションを無効にするか、次に示す対応するコマンドライン設定を使用します。

```
vtune -collect-with runsa -knob enable-driverless-collection=false -knob
event-config=<event-list> <application>
```

インテルのサンプリング・ドライバーの制限: インテルのサンプリング・ドライバーは、コールスタック収集に異なるアルゴリズムを使用するため、スタック・アンワインドの制限はありません。インテル® VTune™ プロファイラーでサポートされる最新のカーネルバージョンよりも新しいカーネルバージョンを使用している場合、ドライバーのアップデートが必要になることがあります。

コンテキスト・スイッチを収集する

解析タイプ: スレッド。

概念: コンテキスト・スイッチの収集は、同期またはスレッド・プリエンプションによるスレッドのインアクティブ待機時間に基づくメトリックの取得に役立ちます。

ドライバーを使用しないモードの制限: Linux* perf は、カーネル 4.3 以降でコンテキスト・スイッチを収集します。コンテキスト・スイッチの原因 (同期またはプリエンプション) の識別は、カーネル 4.17 以降で利用できます。古いカーネルバージョンでは、インテル® VTune™ プロファイラーは、システムで利用可能な場合インテルのサンプリング・ドライバーに切り替えます。

インテルのサンプリング・ドライバーの制限: なし。

カーネル関数のシンボルを解決する

解析タイプ: すべて。

ドライバーを使用しないモードの制限: 追加で `kptr_restrict` ファイルを手動で設定する必要があります。

パフォーマンス・データとカーネル関数名を関連付ける

管理者権限で `kptr_restrict` 設定ファイルの値を 0 に設定します。

```
echo 0 > /proc/sys/kernel/kptr_restrict
```

値を 1 に設定すると、ファイル名の解決がユーザーレベルのモジュールに制限されます。

インテルのサンプリング・ドライバーの制限: なし。`/boot/System.map-<kernel_version>` ファイルが読み取り可能な場合、または `/proc/sys/kernel/kptr_restrict` が 0 に設定されている場合、インテルのサンプリング・ドライバーはカーネルシンボルを解決します。

NMI ウォッチドックとのリソース競合を回避する

解析タイプ: すべて。

ドライバーを使用しないモードの制限: NMI ウォッチドッグ (ハード・ロックアップ検出器) は、1 つの CPU パフォーマンス・カウンター・レジスターを利用し、このレジスターは Linux* perf で利用できなくなります。これにより多重化グループの数が増えて、統計サンプリング・データの精度に影響します。

ドライバーを使用しないモードで多数のイベントを収集する解析の実行の精度を向上する

管理者権限で NMI ウォッチドッグを無効にします。

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

ドライバーを使用しない perf 収集が完了したら NMI ウォッチドッグを (管理者権限で) 再度有効にできます。

```
echo 1 > /proc/sys/kernel/nmi_watchdog
```

インテルのサンプリング・ドライバーの制限: なし。インテルのサンプリング・ドライバーは、このデータ精度の問題を回避するため、収集時に自動で NMI ウォッチドッグを停止します。

収集オーバーヘッドを軽減する

解析タイプ: すべて。

ドライバーを使用しないモードの制限: CPU 負荷の高いアプリケーションではすべての CPU に完全な負荷がかかるため、Linux* perf 収集でオーバーヘッドが発生する可能性があります。

ドライバーを使用しないモードで収集のオーバーヘッドを軽減する

- 次のように、より積極的なアフィニティーの最適化を設定します。

  ```
  vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-
  collection=true -run-pass-thru=--perf-affinity=cpu <application>
  ```

- スタック・サンプリング収集のトレースサイズを軽減するため、インテル® VTune™ プロファイラーが使用する Linux* perf トレース圧縮は、追加のオーバーヘッドを発生する可能性があります。これを回避するには、`-run-pass-thru` オプションを指定してトレース圧縮を無効にします。

  ```
  vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-
  collection=true -run-pass-thru=--perf-compression=0 -run-pass-thru=--
  perf-aio=0 <application>
  ```

これにより、コレクターのオーバーヘッドを軽減できることがありますが、トレースサイズが大幅に増加します。

- Linux perf コレクターによって消費される CPU 時間の上限を設定します。例えば、10% に制限する場合、管理者権限で次のコマンドを実行します。

  ```
  cat 10 > /proc/sys/kernel/perf_cpu_time_max_percent
  ```

  これにより、サンプリング周波数と統計精度が低下する可能性があります。

インテルのサンプリング・ドライバーの制限: なし。

必要に応じてドライバーを使用しないモードを有効にする

インテル® VTune™ プロファイラーは、スタック・サンプリング収集を除くすべてのケースで、インテルのサンプリング・ドライバーがロードされている場合はそれを使用します。インテル® VTune™ プロファイラーがスタックを含まないサンプリングにドライバーを使用しない perf モードを使用するように切り替えるには、カスタム解析タイプを作成して、GUI で **[Enable driverless collection (ドライバーを使用しない収集を有効にする)]** オプションを選択するか、次のようにコマンドライン `knob` 値に `enable-driverless-collection=true` を設定します。

```
vtune -collect-with runsa -knob enable-driverless-collection=true -knob
event-config=<event-list> <application>
```

このオプションは、インテル® VTune™ Amplifier 2019 Update 4 以降で利用できます。

関連情報

- サンプリング・ドライバー (英語)
- スタックを含むハードウェア・イベントベースのサンプリング収集 (英語)
- Linux* カーネル解析を有効にする (英語)

MPI アプリケーションのプロファイル

このレシピは、インテル® VTune™ Amplifier を使用して MPI アプリケーションのインバランスと通信の問題を特定し、アプリケーション・パフォーマンスを向上します。

コンテンツ・エキスパート: Carlos Rosales-fernandez (英語)

- 使用するもの
- 手順:
    1. アプリケーションをビルドする
    2. 全体的なパフォーマンス特性を確立する
    3. HPC パフォーマンス特性解析を設定して実行する
    4. インテル® VTune™ Amplifier GUI で結果を解析する
    5. (オプション) インテル® VTune™ Amplifier GUI でコマンドラインを生成する
    6. (オプション) コマンドライン・レポートを使用して結果を解析する
    7. (オプション) 選択したコード領域をプロファイルする

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** `heart_demo` サンプル・アプリケーション。GitHub* https://github.com/CardiacDemo/Cardiac_demo.git (英語) からダウンロードできます。
- **ツール:**
    - インテル® C++ コンパイラー
    - インテル® MPI ライブラリー 2019
    - インテル® VTune™ Amplifier2019
    - インテル® VTune™ Amplifier アプリケーション・パフォーマンス・スナップショット

注

    - インテル® MPI ライブラリーは、https://www.isus.jp/intel-mpi-library/ から利用できます。
    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
- **オペレーティング・システム:** Linux*
- **CPU**: インテル® Xeon® Platinum 8168 プロセッサー (開発コード名 Skylake)
- **ネットワーク・ファブリック**: インテル® Omni-Path アーキテクチャー (インテル® OPA)

アプリケーションをビルドする

インテル® VTune™ Amplifier がパフォーマンス・データをソースコードとアセンブリーに関連付けることができるように、デバッグシンボル付きでアプリケーションをビルドします。

1. GitHub* リポジトリーのアプリケーションをローカルシステムにクローンします。

   ```
   git clone https://github.com/CardiacDemo/Cardiac_demo.git
   ```

2. インテル® C++ コンパイラーとインテル® MPI ライブラリーの環境を設定します。

   ```
   source <compiler_install_dir>/bin/compilervars.sh intel64
   source <mpi_install_dir>/bin/mpivars.sh
   ```

3. サンプルパッケージのルートレベルに build ディレクトリーを作成し、作成したディレクトリーに移動します。

   ```
   mkdir build
   cd build
   ```

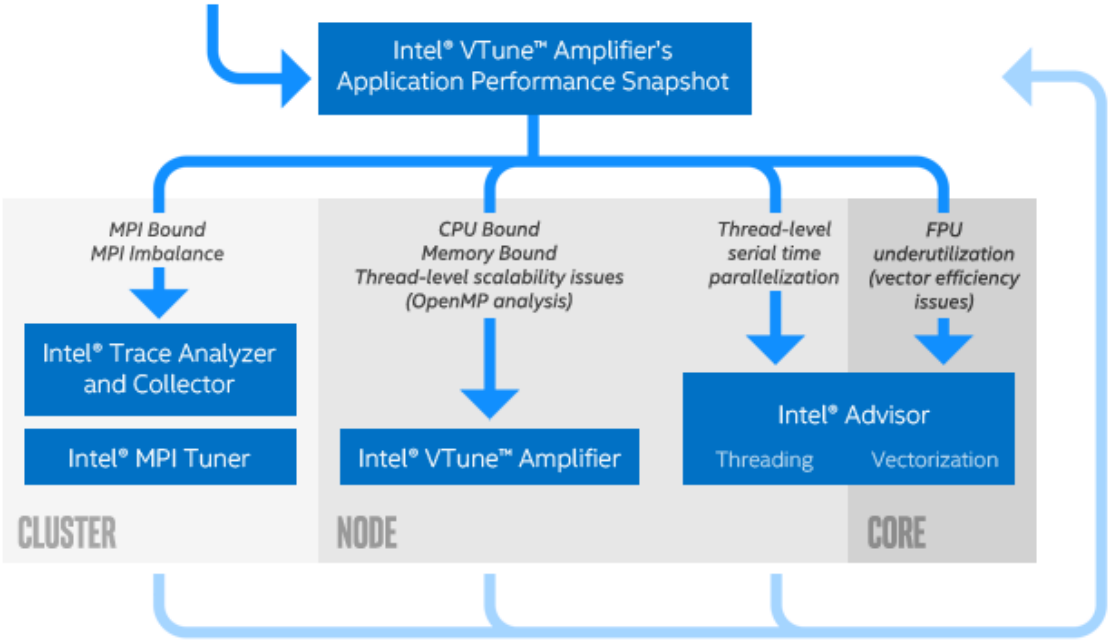
4. 次のコマンドを使用してアプリケーションをビルドします。

   ```
   mpiicpc ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -
   g -o heart_demo -O3 -std=c++11 -qopenmp -parallel-source-info=2
   ```

`heart_demo` 実行ファイルが現在のディレクトリーに作成されます。

全体的なパフォーマンス特性を確立する

インテル® ソフトウェア開発ツールを使用するアプリケーション・チューニングの推奨ワークフローでは、最初にアプリケーション・パフォーマンスのスナップショットを取得してから、最適なツールを使用して問題に注目します。インテル® VTune™ Amplifier のアプリケーション・パフォーマンス・スナップショットは、シンプルなインターフェイスと低オーバーヘッドの実装で、アプリケーションの全体的なパフォーマンス特性を提供します。特定の問題を詳しく調査する前に、アプリケーション・パフォーマンス・スナップショットを使用してアプリケーションの一般的な特性を理解します。



インテル® Xeon® スケーラブル・プロセッサー (開発コード名 Skylake) を使用して、デュアル・ソケット・ノードのセットのパフォーマンス・スナップショットを取得します。この例では、ソケットあたり 24 コアのインテル® Xeon® Platinum 8168 プロセッサーを使用して、ノードごとに 4 MPI ランク、ランクごとに 12 スレッドで実行するように設定します。ランクとスレッドの数は、使用しているシステムの要件に応じて変更してください。

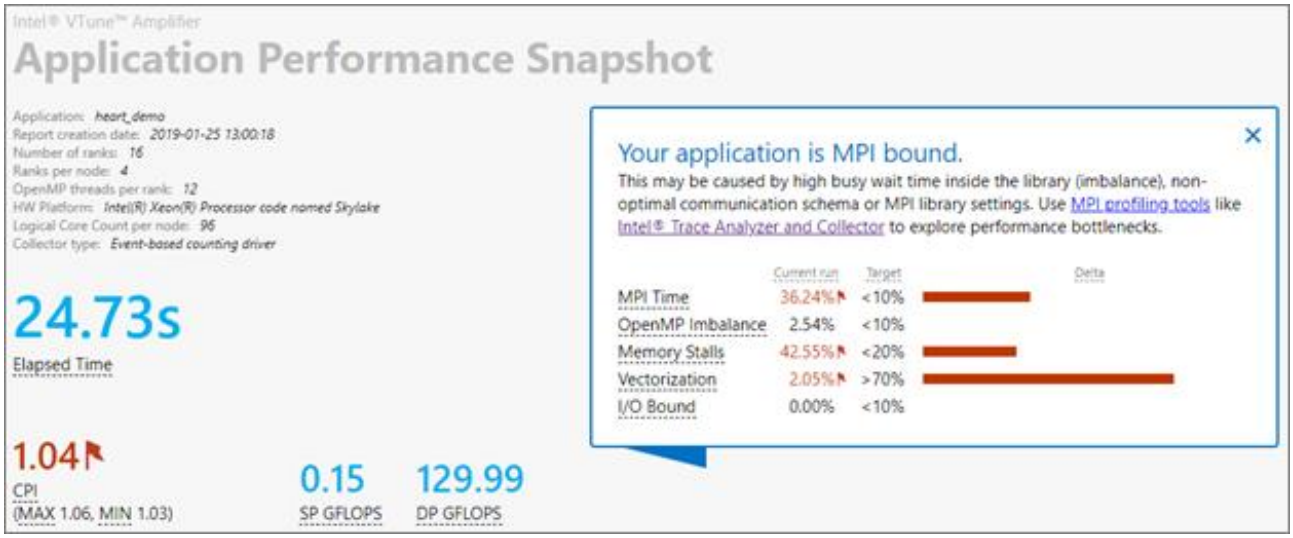
対話型セッションまたはバッチスクリプトで次のコマンドを実行して、4 ノードのパフォーマンス・スナップショットを取得します。

```
export OMP_NUM_THREADS=12
mpirun -np 16 -ppn 4 aps ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t
100
```

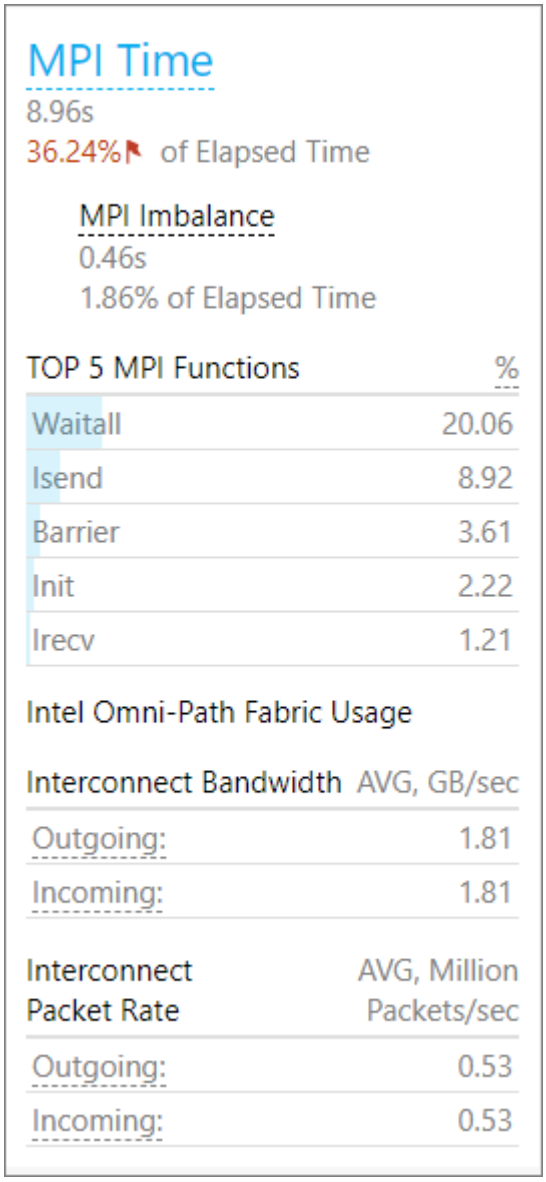
解析が完了すると、aps_result_YYYYMMDD という名前のディレクトリーとプロファイル・データが生成されます。ここで、YYYY は収集の年、MM は月、DD は日を表します。次のコマンドを実行して、結果の HTML スナップショットを生成します。

```
aps --report=./aps_result_20190125
```

作業ディレクトリーに aps_report_YYYYMMDD_*<stamp>*.html ファイルが作成されます。*<stamp>* 番号は、既存のレポートが上書きされるのを防ぐために追加されます。レポートには、MPI と OpenMP* のインバランス、メモリー・フットプリントと帯域幅の利用率、浮動小数点演算のスループットを含む、全体的なパフォーマンスに関する情報が含まれます。レポートの上部の説明は、アプリケーションの主な問題を示しています。



このアプリケーションは全体的に MPI 通信に依存していますが、メモリーとベクトル化の問題もあります。**[MPI Time (MPI 時間)]** セクションは、MPI インバランスや使用された上位の MPI 関数呼び出しなど、追加の情報を提供します。この情報から、コードは主にポイントツーポイント通信を使用しており、インバランスは中程度であることが分かります。



この結果は、コードに複雑な問題があることを示しています。パフォーマンスの問題をさらに詳しく調査し、問題を切り分けるため、インテル® VTune™ Amplifier の HPC パフォーマンス特性解析を使用します。

HPC パフォーマンス特性解析を設定して実行する

ほとんどのクラスターは、ログインノードと計算ノードで構成されています。通常、ユーザーはログインノードに接続し、スケジューラーを使用してジョブを計算ノードに送信し、ジョブが実行されます。クラスター環境でインテル® VTune™ Amplifier を使用して MPI アプリケーションをプロファイルする最も実用的な方法は、コマンドラインでデータを収集し、ジョブが完了したら GUI でパフォーマンス解析を行います。

MPI 関連のメトリックのレポートは、コマンドラインから簡単に取得できます。一般に、分散環境で実行する最も簡単な方法は、次のようにコマンドを作成することです。

```
<mpi launcher> [options] amplxe-cl [options] -r <results dir> -- <application> [arguments]
```

注

- コマンドは、対話型セッションで使用することも、バッチ送信スクリプトに含めることもできます。
- MPI アプリケーションでは、結果ディレクトリーを指定する必要があります。
- インテル® MPI ライブラリーを使用していない場合は、`-trace-mpi` を追加します。

次の手順に従って、コマンドラインから HPC パフォーマンス特性解析を実行します。

1. 関連するインテル® VTune™ Amplifier ファイルを source して環境を設定します。bash シェルを使用するデフォルトのインストールでは、次のコマンドを使用します。

   ```
   source /opt/intel/vtune_amplifier/amplxe-vars.sh
   ```

2. `hpc-performance` 解析を使用して `heart_demo` アプリケーションのデータを収集します。このアプリケーションは、OpenMP* と MPI の両方を使用して、前述の構成 (インテル® MPI ライブラリーを使用して 4 つの計算ノード、16 の MPI ランク) で実行されます。実行には、インテル® Xeon® Platinum 8168 プロセッサーと MPI ランクごとに 12 の OpenMP* スレッドを使用します。

   ```
   export OMP_NUM_THREADS=12
   mpirun -np 16 -ppn 4 amplxe-cl -collect hpc-performance -r vtune_mpi -
   - ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 100
   ```

   解析が開始され、次の命名規則に従って 4 つの出力ディレクトリーが生成されます: vtune_mpi.*<node host name>*。

注

ほかの MPI ランクを同時に実行しながら、特定の MPI ランクのみプロファイル・データを収集するように選択できます。詳細は、「選択した MPI ランクのプロファイル」 (英語) を参照してください。

インテル® VTune™ Amplifier GUI で結果を解析する

インテル® VTune™ Amplifier のグラフィカル・インターフェイスは、収集したパフォーマンス・データを解析するため、コマンドラインよりも豊富な機能とインタラクティブな体験を提供します。次のコマンドを実行して、結果をインテル® VTune™ プロファイラー GUI で開きます。

```
amplxe-gui ./vtune_mpi.node_1
```

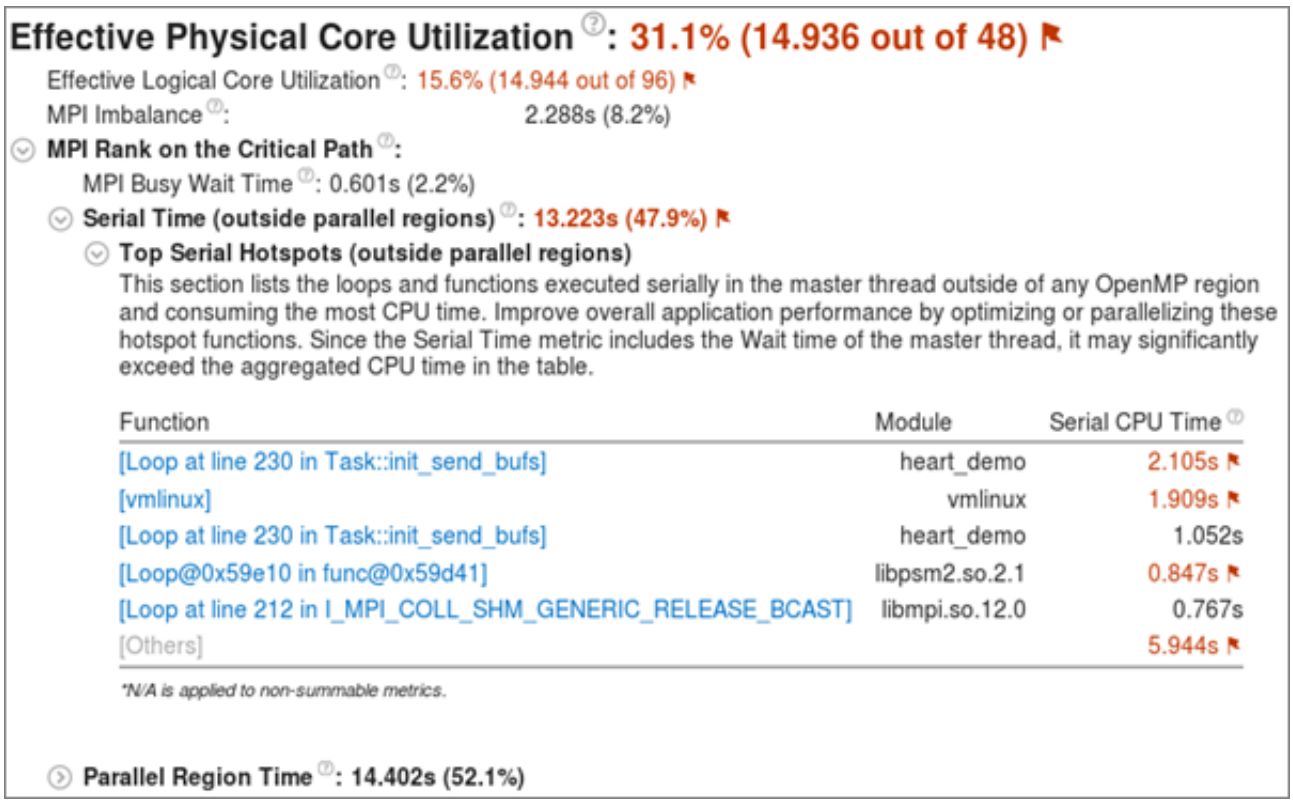
注

インテル® VTune™ Amplifier GUI を表示するには、ローカルシステムで実行されている X11 マネージャー、またはシステムに接続されている VNC セッションが必要です。システムはそれぞれ異なるため、推奨される方法についてはローカル管理者に相談してください。

結果がインテル® VTune™ Amplifer で開き、アプリケーション・パフォーマンスの概要を示す **[Summary (サマリー)]** ウィンドウが表示されます。`heart_demo` は MPI 並列アプリケーションであるため、**[Summary (サマリー)]** ウィンドウに通常のメトリックに加えて、[MPI Imbalance (MPI インバランス)] 情報と実行クリティカル・パスの MPI ランクに関する詳細が表示されます。

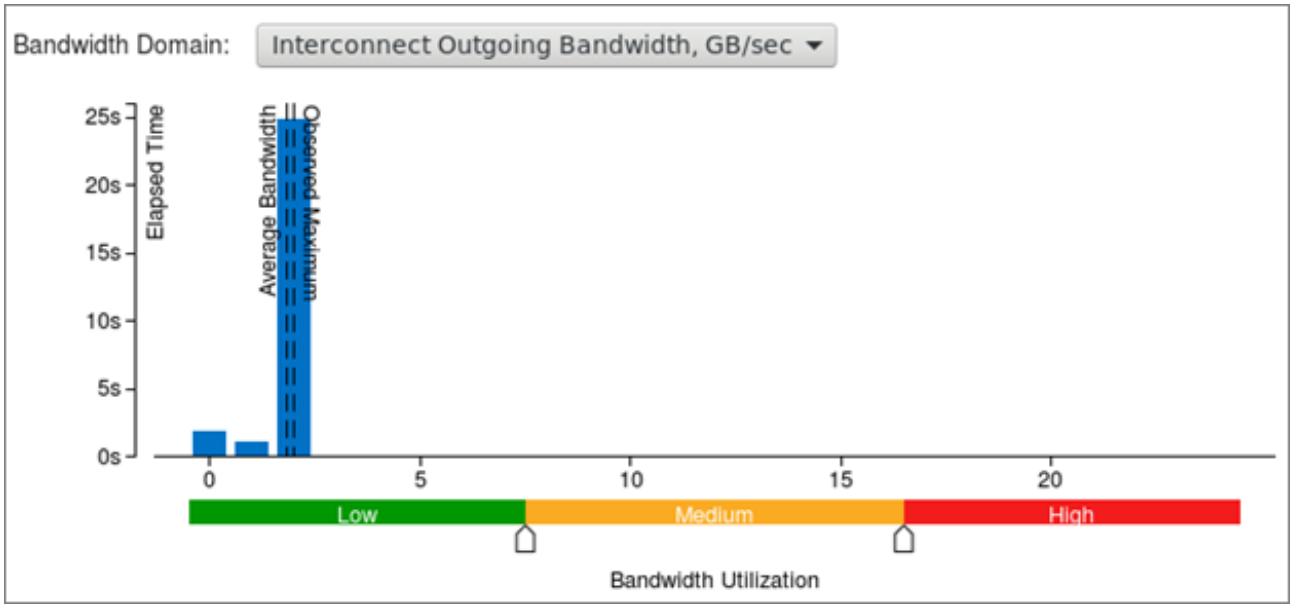
- **[MPI Imbalance (MPI インバランス)]** は、ノード上のすべてのランクの平均 MPI ビジー待機時間です。この値は、バランスが理想的であれば節約できる時間を示します。
- **[MPI Rank on the Critical Path (クリティカル・パス上のランク)]** は、最小ビジー待機時間のランクです。
- **[MPI Busy Wait Time (MPI ビジー待機時間)]** と **[Top Serial Hotspots (上位のシリアル・ホットスポット)]** は、クリティカル・パスのランクに対して表示されます。通常、インバランスやビジー待機メトリックの値が高い場合スケーラビリティーが制限されるため、これはスケーラビリティーの問題を特定する良い方法です。マルチノード実行でクリティカル・パス上のランクの **[MPI Busy Wait Time (MPI ビジー待機時間)]** が大きい場合、ほかのノード上に外れ値のランクがある可能性があります。

この例では、インバランスがあり、コードのシリアル領域で多くの時間が費やされています (以下の図には表示されていません)。

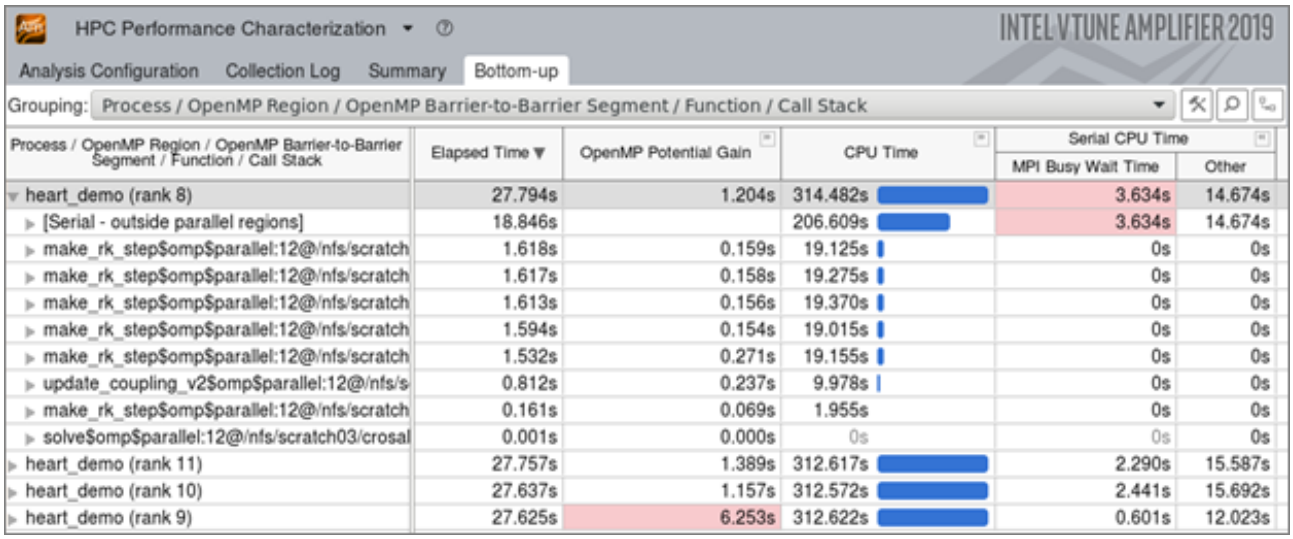


プロファイルはノード全体で収集されるかもしれませんが、すべての MPI データを確認するには、各ノードの結果を個別にロードする必要があります。詳細な MPI トレースには、インテル® Trace Analyzer & Collector を推奨します。

インテル® VTune™ Amplifer 2019 以降では、**[Summary (サマリー)]** ウィンドウにインテル® Omni-Path アーキテクチャー (インテル® OPA) ファブリックの使用分布図が含まれます。これらのメトリックは帯域幅とパケットレートを表示し、実行時間に占めるコードが高帯域幅使用率や高パケットレート使用率によって制限されていた割合 (%) を示します。`heart_demo` アプリケーションは、帯域幅やパケットレートによって制限されていませんが、分布図は平均に近い 1.8GB/秒の最大帯域幅使用率を示しています。これは、MPI 通信の継続的で非効率的な使用を示唆しています。



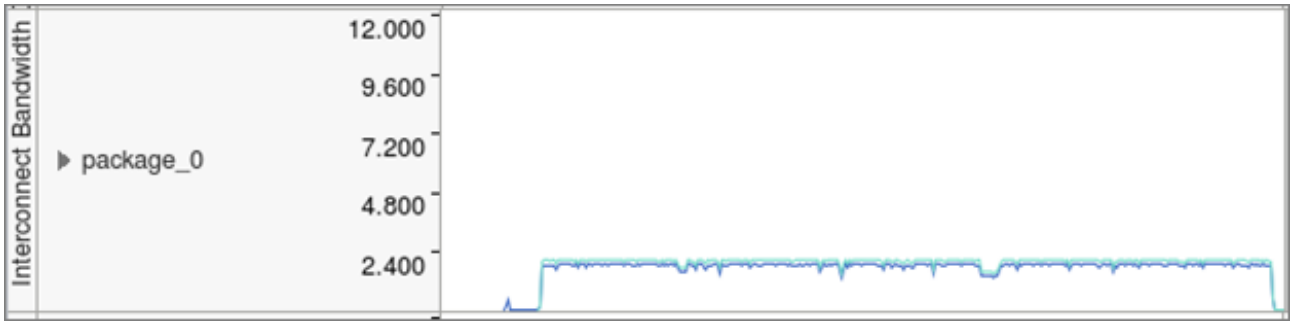
[Bottom-up (ボトムアップ)] タブに切り替えてさらに詳しく調査します。次のような表示になるように、**[Grouping (グループ化)]** で Process (プロセス) がトップレベルになるように設定します。



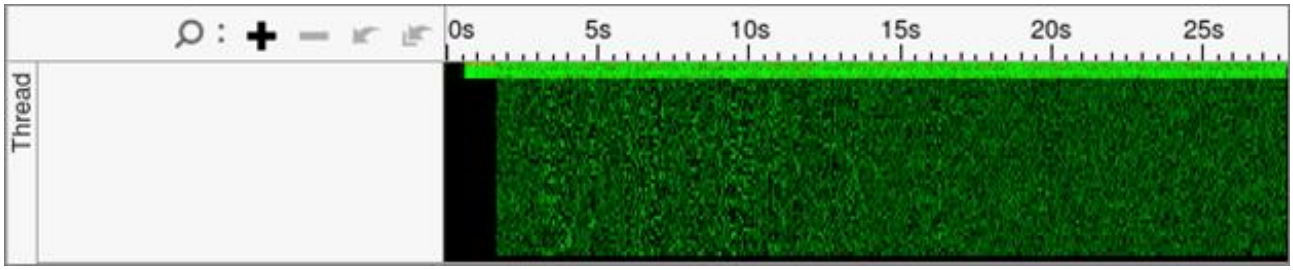
Process / OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time ▼	OpenMP Potential Gain	CPU Time	Serial CPU Time	
				MPI Busy Wait Time	Other
heart_demo (rank 8)	27.794s	1.204s	314.482s	3.634s	14.674s
[Serial - outside parallel regions]	18.846s		206.609s	3.634s	14.674s
make_rk_stepompparallel:12@/nfs/scratch	1.618s	0.159s	19.125s	0s	0s
make_rk_stepompparallel:12@/nfs/scratch	1.617s	0.158s	19.275s	0s	0s
make_rk_stepompparallel:12@/nfs/scratch	1.613s	0.156s	19.370s	0s	0s
make_rk_stepompparallel:12@/nfs/scratch	1.594s	0.154s	19.015s	0s	0s
make_rk_stepompparallel:12@/nfs/scratch	1.532s	0.271s	19.155s	0s	0s
update_coupling_v2ompparallel:12@/nfs/s	0.812s	0.237s	9.978s	0s	0s
make_rk_stepompparallel:12@/nfs/scratch	0.161s	0.069s	1.955s	0s	0s
solveompparallel:12@/nfs/scratch03/crosal	0.001s	0.000s	0s	0s	0s
heart_demo (rank 11)	27.757s	1.389s	312.617s	2.290s	15.587s
heart_demo (rank 10)	27.637s	1.157s	312.572s	2.441s	15.692s
heart_demo (rank 9)	27.625s	6.253s	312.622s	0.601s	12.023s

このコードは MPI と OpenMP* の両方を使用するため、[Bottom-up (ボトムアップ)] ウィンドウには通常の CPU とメモリーのデータに加えて、両方のランタイムに関連したメトリックが表示されます。この例では、クリティカル・パスから最も離れている MPI ランクの [MPI Busy Wait Time (MPI ビジー待機時間)] メトリックが赤色で示されています。また、最も大きな [OpenMP Potential Gain (OpenMP* 潜在的なゲイン)] も赤色で示されています。これは、スレッド化を改善することで、パフォーマンスが向上する可能性を示唆しています。

[Bottom-up (ボトムアップ)] ウィンドウの下部で DDR と MCDRAM 帯域幅、CPU 時間、インテル® OPA 使用率を含むいくつかのメトリックの実行タイムラインを確認します。このコードのインターコネクト帯域幅のタイムラインは、適度な帯域幅 (GB/秒単位) で継続的な使用率を示しています。これは、分散コンピューティングで一般的な間違いの 1 つである、小さなメッセージの通常の MPI 交換が原因です。



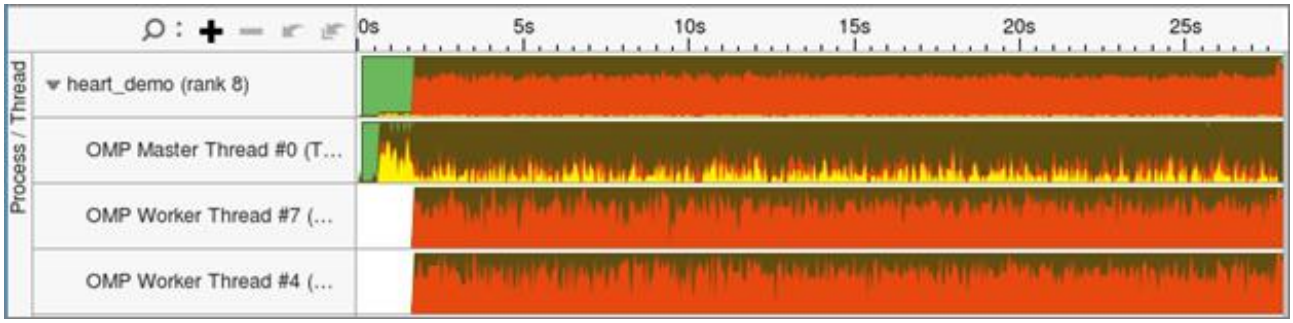
より興味深いことは、スレッドごとの詳細な実行時間と [Effective Time (有効時間)]、[Spin and Overhead Time (スピンとオーバーヘッド時間)]、および [MPI Busy Wait Time (MPI ビジー待機時間)] の内訳です。デフォルトのビューは [Super Tiny (最小)] 設定で、パフォーマンスのビジュアルマップにすべてのプロセスとスレッドをまとめて表示します。



このケースでは、ほとんどのスレッドで有効時間がわずかしかなく (緑色)、MPI オーバーヘッドの量もわずかです (黄色)。これは、スレッド化の実装に潜在的な問題があることを示しています。

さらに詳しく調査するため、グラフの左側の灰色の領域を右クリックして、**[Rich (最大)]** ビューを選択します。次に、各 MPI ランクと各スレッドの役割がより明確になるように、グラフの右側で **[Process/Thread (プロセス/スレッド)]** を選択して結果をグループ化します。このグループ化を使用することで、各プロセスの最上部のバーはすべての子スレッドの平均結果を示し、その下に各スレッドがスレッド番号とプロセス ID とともにリストされます。

この例では、マスタースレッドが各 MPI ランクのすべての MPI 通信を明確に処理しています。これはハイブリッド・アプリケーションでは一般的です。実行の最初の 10 秒間は、MPI 通信 (黄色) に多くの時間が費やされています。ここでは、問題のセットアップとデータの分配が行われていると考えられます。その後は通常の MPI 通信となり、[Bandwidth Utilization (帯域幅使用率)] タイムラインと [Summary (サマリー)] レポートの結果と一致しています。

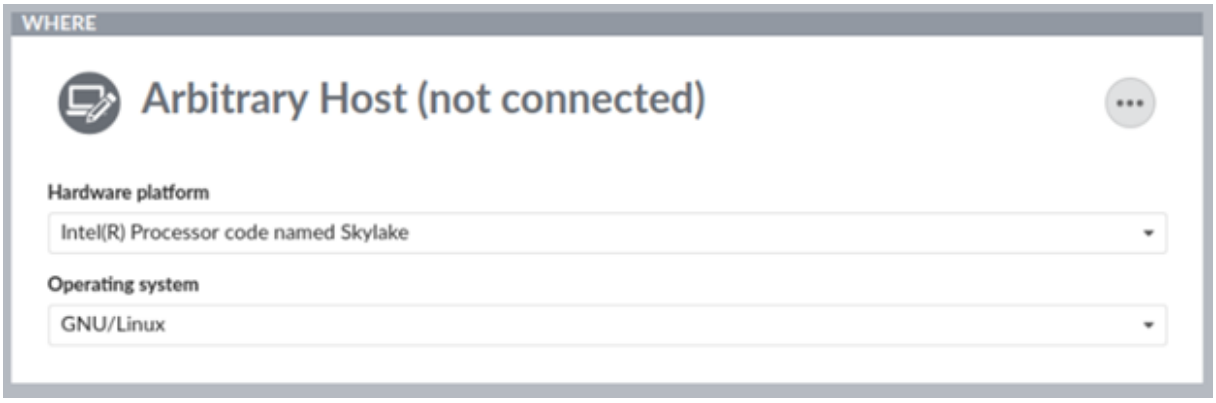


スピンとオーバーヘッド (デフォルトでは赤色で表示) が高いことが分かります。これは、アプリケーションのスレッド化の実装に問題があることを示しています。**[Bottom-up (ボトムアップ)]** ウィンドウの上部で、**[OpenMP Region / Thread / Function / Call Stack (OpenMP* 領域/スレッド/関数/コールスタック)]** を選択してデータをグループ化し、ウィンドウ下部のフィルターを適用して **[Functions only (関数のみ)]** を表示します。ツリーを展開すると、*init_send_bufs* 関数はスレッド 0 以外から呼び出されていないことが分かります。これが、パフォーマンス低下の原因です。この行をダブルクリックするとソースコード・ビューが開きます。コードを調査すると、関数の外側のループを並列化し、この問題を修正する簡単な方法があることが分かります。

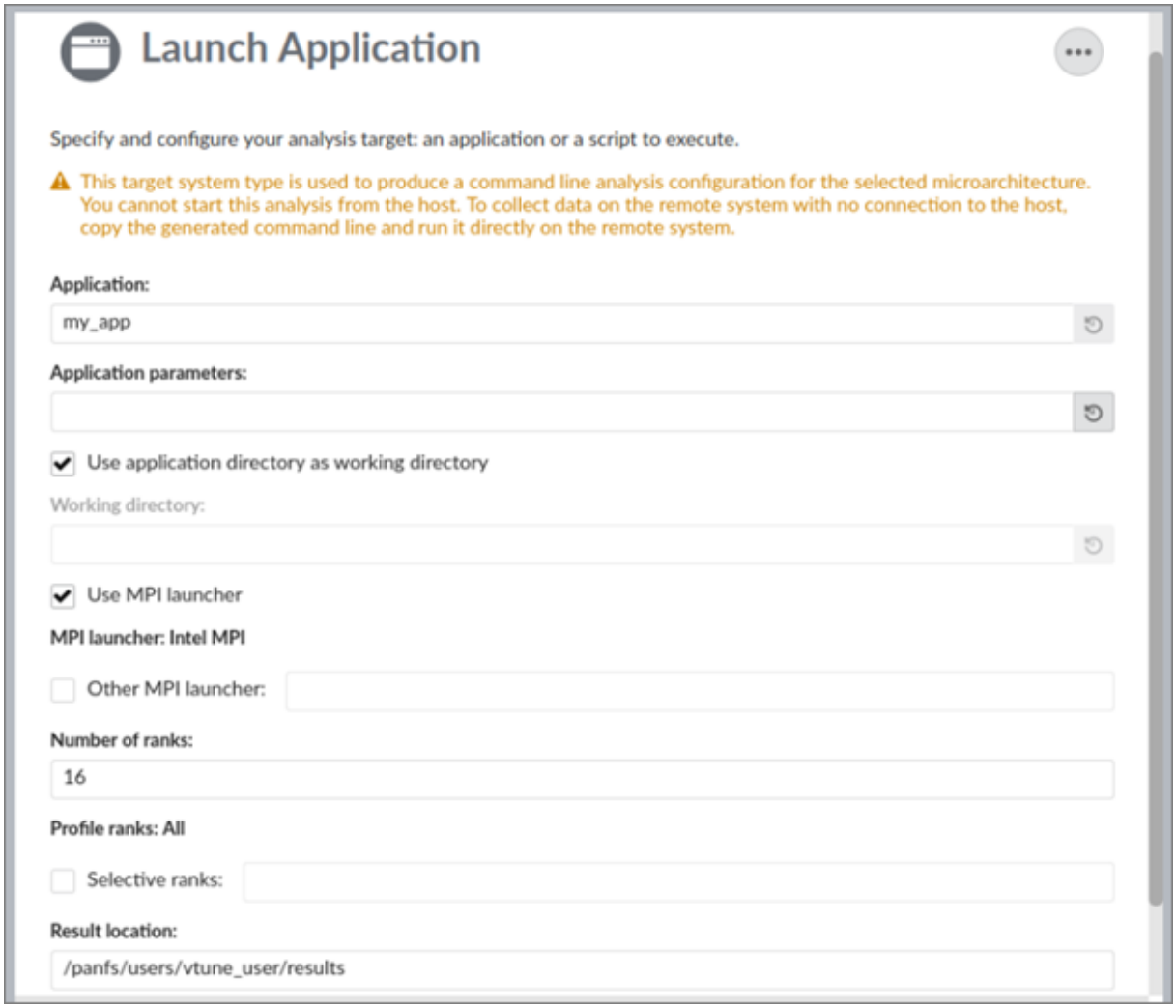
(オプション) インテル® VTune™ Amplifier GUI でコマンドラインを生成する

インテル® VTune™ Amplifier のあまり知られていない便利な機能は、GUI を使用して解析を設定し、対応するコマンドラインを保存して、コマンドラインから直接使用できます。これは、高度にカスタマイズされたプロファイルや複雑なコマンドを素早く作成するのに便利です。

1. インテル® VTune™ Amplifier を起動して、**[New Project (新規プロジェクト)]** をクリックするか、既存のプロジェクトを開きます。
2. **[Configure Analysis (解析の設定)]** をクリックします。
3. **[WHERE (どこを)]** ペインで **[Arbitrary Host (not connected) (任意のホスト (未接続))]** を選択し、ハードウェア・プラットフォームを指定します。

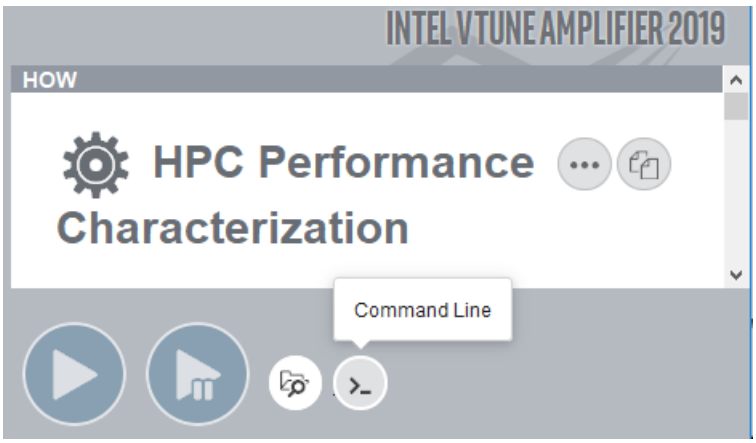


4. **[WHAT (何を)]** ペインで次の操作を行います。



a. アプリケーションを指定して、引数と作業ディレクトリーを設定します。
b. **[Use MPI launcher (MPI ランチャーを使用)]** チェックボックスをオンにして、MPI 実行に関連した情報を指定します。
c. (オプション) プロファイルする特定のランクを選択します。

5. **[HOW (どのように)]** ペインでデフォルトの **[Hotspots (ホットスポット)]** 解析から **[HPC Performance Characterization (HPC パフォーマンス特性)]** 解析に変更し、利用可能なオプションをカスタマイズします。



6. ウィンドウの下部にある **[Command Line (コマンドライン)]** ボタン  をクリックします。ポップアップ・ウィンドウが開き、GUI で設定したカスタム解析に対応するコマンドラインが表示されます。必要に応じて、コマンドにその他の MPI オプションを追加できます。

注

インテル® MPI ライブラリーの場合、コマンドラインは `-gtool` オプションで生成されます。これにより、選択したランクのプロファイル構文が非常に簡潔になります。

(オプション) コマンドライン・レポートを使用して結果を解析する

インテル® VTune™ Amplifier は、有益なコマンドライン・テキスト・レポートを提供します。例えば、サマリーレポートを取得するには、次のコマンドを実行します。

```
amplxe-cl -report summary -r ./results_dir
```

結果のサマリーが画面に出力されます。結果を直接ファイルに保存したり、別のファイル形式 (csv、xml、html) で保存するオプションなど、その他のオプションも利用できます。コマンドライン・オプションの詳細は、**amplxe-cl -help** コマンドで確認するか、ユーザーガイド (英語) を参照してください。

(オプション) 選択したコード領域をプロファイルする

デフォルトでは、インテル® VTune™ Amplifier はアプリケーション全体のパフォーマンス統計を収集しますが、バージョン 2019 Update 3 以降では MPI アプリケーションのデータ収集を制御する機能が追加されました。この機能を使用することで、高速に処理可能な小さな結果ファイルを生成して、対象コード領域に注目できます。

領域の選択は、`MPI_Pcontrol` 関数を使用して行います。`MPI_Pcontrol(0)` 呼び出しでデータ収集を一時停止し、`MPI_Pcontrol(1)` 呼び出しで再開します。API と `-start-paused` コマンドライン・オプションを一緒に使用して、アプリケーションの初期化フェーズを除外できます。この場合、初期化の直後に `MPI_Pcontrol(1)` を呼び出してデータ収集を再開します。この方法では、スタティック ITT API ライブラリーへのリンクが必要な ITT API 呼び出しを使用する場合とは異なり、アプリケーションのビルドプロセスの変更が不要です。

関連情報

- インテル® VTune™ Amplifier インストール・ガイド - Linux* (英語)
- MPI とインテル® Advisor およびインテル® VTune™ Amplifier の使用 (英語)
- Cray* XC システムでのインテル® VTune™ Amplifier の使用 (英語)
- チュートリアル: MPI/OpenMP* ハイブリッド・アプリケーションの解析 (英語)

注

このレシピの情報は、デベロッパー・フォーラム (英語) を参照してください。

最適化に関する注意事項

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

注意事項の改訂 #20110804

Node.js* の JavaScript* コードのプロファイル

このレシピは、`Node.js*` をリビルドし、インテル® VTune™ Amplifier を使用して、JavaScript* フレームとネイティブフレーム (ネイティブコード、例えば、JavaScript* コードから呼び出されたシステム・ライブラリーやネイティブ・ライブラリー) から成る混在モードのコールスタックを含む JavaScript* コードのパフォーマンスを解析するための設定手順を説明します。

コンテンツ・エキスパート: Denis Pravdin (英語)

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** `sample.js`。このアプリケーションはデモ用であり、ダウンロードすることはできません。
- **JavaScript* 環境:** `Node.js*` 8.0.0、Chrome* V8 5.8.283.41
- **パフォーマンス解析ツール:** インテル® VTune™ Amplifier 2018: 高度な hotspot 解析

注

    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Microsoft* Windows* 10

Node.js* でインテル® VTune™ Amplifier のサポートを有効にする

1. `Node.js*` のソース (nightly build) をダウンロードします。
2. ルートの `node-v8.0.0` フォルダーから `vcbuild.bat` スクリプトを実行します。

    ```
    echo vcbuild.bat enable-vtune
    ```

    このスクリプトは、インテル® VTune™ プロファイラーが JavaScript* コードのプロファイルをサポートするように `Node.js*` をビルドします。

注

- Linux* システムでは、`enable-vtune` フラグと `fully-static` 設定フラグを同時に使用しないでください。この組み合わせは互換性がなく、`Node.js*` 環境がクラッシュします。
- Microsoft* Visual Studio* 2015 以降の IDE を使用する場合は、`node-v8.0.0-win\deps\v8\src\third_party\vtune\vtune-jit.cc` ファイルに `#define _SILENCE_STDEXT_HASH_DEPRECATION_WARNINGS` を追加します。

```
#include <string.h>

#ifdef WIN32
#define _SILENCE_STDEXT_HASH_DEPRECATION_WARNINGS
#include <hash_map>
using namespace std;
#else
...
```

Node.js* で動作している JavaScript* コードをプロファイルする

このレシピはサンプル JavaScript* アプリケーションを使用します。

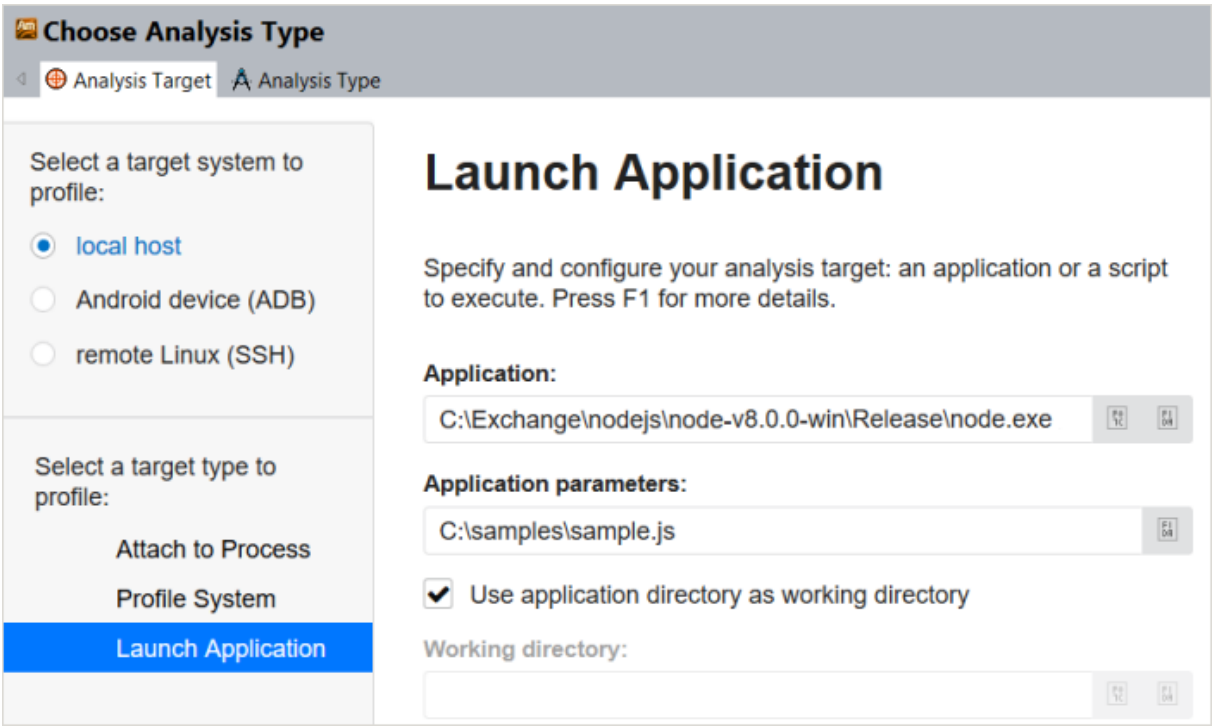
```
function say(word) {
console.log("Calculating ...");
var res = 0;
for (var i = 0; i < 20000; i++) {
  for (var j = 0; j < 20000; j++) {
    res = i * j / 2;
  }
 }
console.log("Done.");
console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello from Node.js!");
```

インテル® VTune™ Amplifier を使用してこのアプリケーションをプロファイルするには、次の操作を行います。

1. インテル® VTune™ Amplifier を起動します。

   ```
   amplxe-gui.exe
   ```

2. ツールバーの [New Project (新規プロジェクト)]  アイコンをクリックして新規プロジェクトを作成します。
3. **[Analysis Target (解析ターゲット)]** タブで、**[Application (アプリケーション)]** フィールドに `node.exe`、**[Application parameters (アプリケーションのパラメーター)]** フィールドに `sample.js` を指定します。

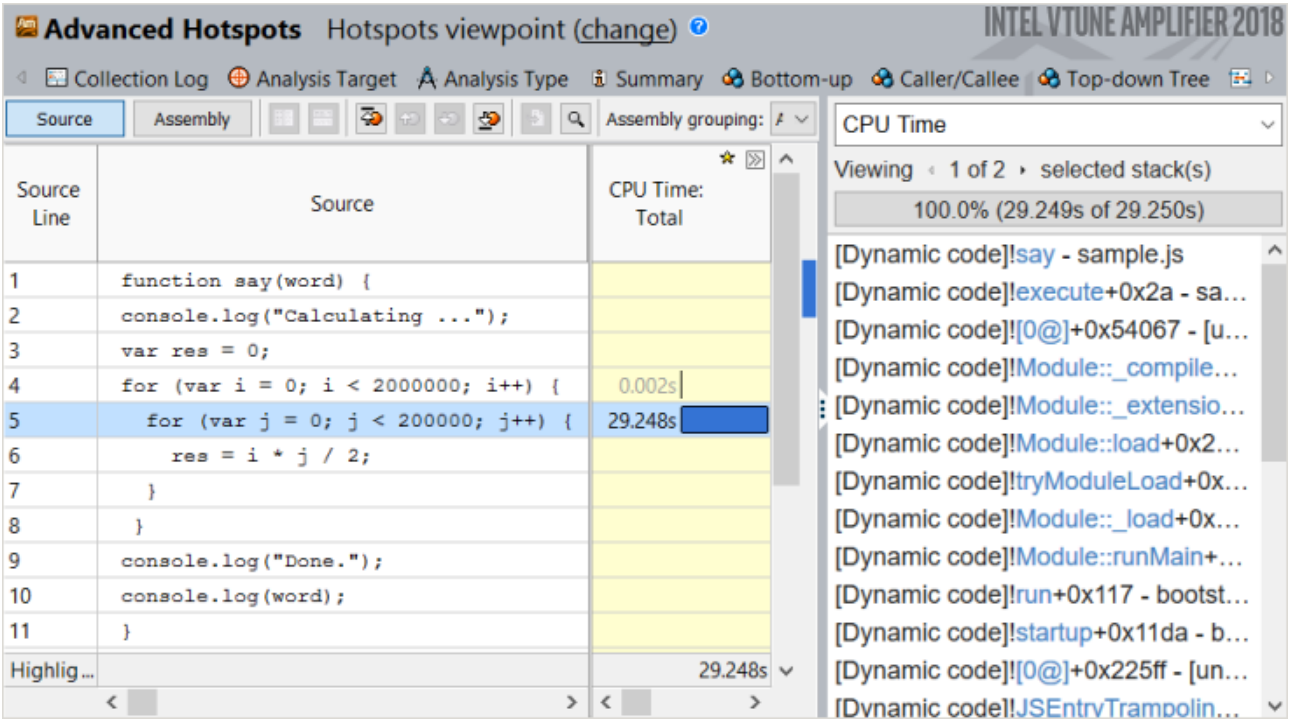


4. **[Analysis Type (解析タイプ)]** タブに切り替えて、左ペインから **[Advanced Hotspots (高度な hotspot)]** 解析タイプを選択し、**[Start (開始)]** をクリックして解析を実行します。

注

高度な hotspot 解析は、インテル® VTune™ Amplifier 2019 で汎用の hotspot 解析 (英語) に統合されました。ハードウェア・イベントベース・サンプリング収集モードで利用できます。

解析が完了すると、インテル® VTune™ Amplifier はデフォルトの [Hotspots (Hotspot)] ビューポイントに結果を表示します。**[Bottom-up (ボトムアップ)]** ウィンドウを使用して、サンプルが JavaScript* 関数にどのように分散されているか調べます。最も CPU 時間がかかっている関数をダブルクリックしてソースコードを表示し、最もホットなコード行を特定します。



Docker* コンテナーでのプロファイル

このレシピは、インテル® VTune™ Amplifier の解析向けに Docker* コンテナーを構成して、独立したコンテナー環境で動作しているアプリケーションの hotspot を特定します。

コンテンツ・エキスパート: Denis Pravdin (英語)

- 使用するもの
- 手順:
    1. Docker* コンテナーをインストールして設定する
    2. アタッチモードで高度な hotspot 解析を実行する
    3. コンテナーで収集したデータを解析する

注

高度な hotspot 解析は、インテル® VTune™ Amplifier 2019 で汎用の hotspot 解析 (英語) に統合されました。ハードウェア・イベントベース・サンプリング収集モードで利用できます。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** `MatrixMultiplication`。この Java* アプリケーションはデモ用であり、ダウンロードすることはできません。
- **ツール:** インテル® VTune™ Amplifier 2018: 高度な hotspot 解析

注

    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **Linux* コンテナーランタイム:** docker.io
- **オペレーティング・システム:** Ubuntu* 17.04
- **CPU:** インテル® プロセッサー (開発コード名 Skylake)、8 論理 CPU

Docker* コンテナーをインストールして設定する

1. [オプション] 必要に応じて、ホストシステムから以前の Docker* バージョンを削除します。

```
host> sudo apt-get remove docker
```

2. Docker* をインストールします。

```
host> sudo apt-get update
host> sudo apt-get install docker.io
```

注

- Docker* コンテナーランタイムのバージョンはオペレーティング・システムのバージョンに依存します。`apt-cache search "container runtime"` と入力すると、正しいバージョンが表示されます。
- パッケージをインストールできない場合、Docker* の `systemd` サービスファイルでプロキシーサーバーが設定されていることを確認してください。詳細は、https://docs.docker.com/engine/admin/systemd/#httphttps-proxy (英語) を参照してください。ステップ 1 から 6 に従い、例のプロキシー名を使用するプロキシー名に変更します。
- インストールの詳細は、https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/#install-docker-ce (英語) を参照してください。

3. Docker* イメージを作成します。

```
host> cd /tmp
host> touch Dockerfile
 host> echo FROM ubuntu:latest > ./Dockerfile
 host> docker build -t myimage .
Sending build context to Docker daemon 6.295 MB
Step 1: FROM ubuntu:latest
e0a742c2abfd: Pull complete
486cb8339a27: Pull complete
dc6f0d824617: Pull complete
4f7a5649a30e: Pull complete
672363445ad2: Pull complete
Digest:
sha256:84c334414e2bfdae99509a6add166bbb4fa4041dc3fa6af08046a66fed3005f
Status: Downloaded newer image for ubuntu:latest
---> 14f60031763d
Successfully built 14f60031763d
```

イメージ `myimage` が作成されたことを確認します。

```
host> docker images
REPOSITORY       TAG           IMAGE ID         CREATED        SIZE
myimage          latest        14f60031763d     2 weeks ago    119.5 MB
```

注

`docker load -i` *`image.tar`* コマンドを使用して、ファイルから圧縮されたリポジトリーをロードすることもできます。

4. `-t` および `-d` オプションを使用してコンテナーを実行します。

```
host> docker run -td myimage
```

注

インテル® VTune™ Amplifier のアルゴリズム解析 (基本 hotspot、並行性、ロックと待機) 向けに Docker* コンテナーを実行するには、次のように `ptrace` サポートを有効にします。

```
host> docker run --cap-add=SYS_PTRACE -td myimage
```

または、privileged モードでコンテナーを起動します。

```
host> docker run --privileged -td myimage
```

5. コンテナー ID を確認します。

```
host> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
98fec14f0c08        myimage             "/bin/bash"         10 seconds ago
Up 9 seconds                            sharp_thompson
```

6. コンテナー ID を使用してバックグラウンド・モードで (bash シェルを起動して) コンテナーに入ります。

```
host> docker exec -it 98fec14f0c08  /bin/bash
```

7. Java* アプリケーションと JVM を動作中の Docker* インスタンスにコピーします。次に例を示します。

```
host> docker cp /home/samples/jdk1.8.tar 98fec14f0c08:/var/local
host> docker cp /home/samples/matrix.tar 98fec14f0c08:/var/local
```

8. `matrix.tar` および `jdk` アーカイブを展開します。

アタッチモードで高度な hotspot 解析を実行する

1. ホストでインテル® VTune™ Amplifier を起動します。

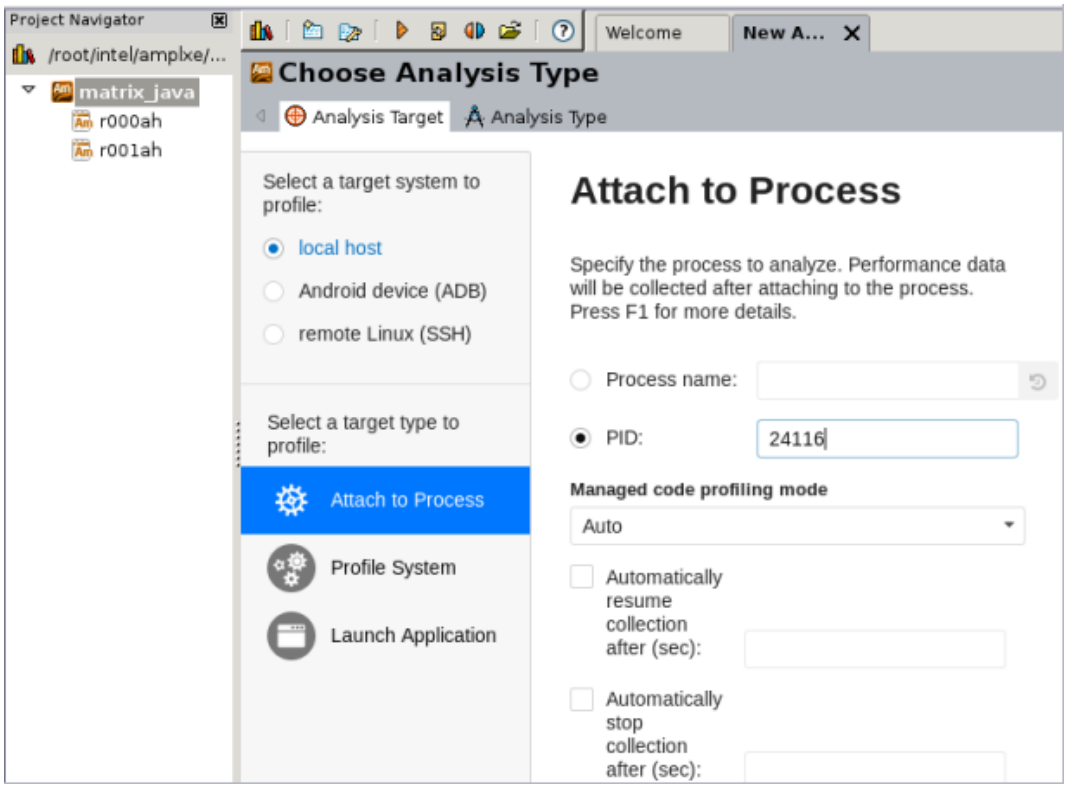
```
host> cd /opt/intel/vtune_amplifier_2018.0.2.522558
host> source ./amplxe-vars.sh
host> amplxe-gui
```

2. プロジェクト (例: `matrix_java`) を作成します。
3. コンテナー内で Java* アプリケーションを実行します。

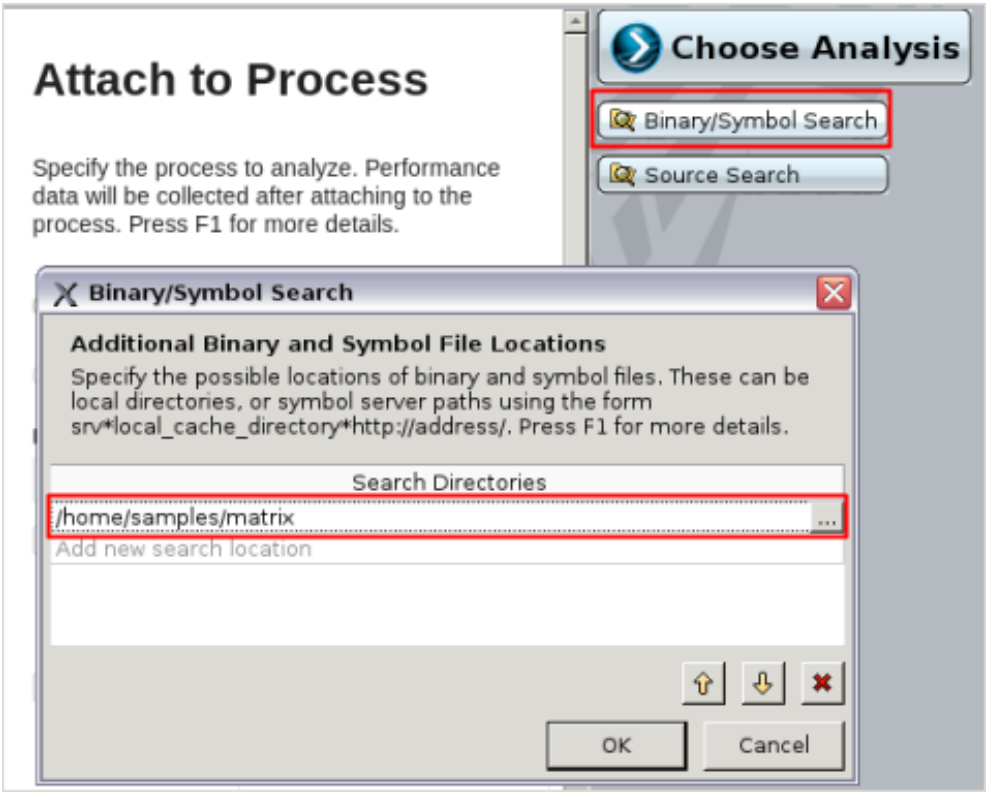
```
container> cd /var/local/matrix
container> /var/local/jdk1.8.0_72-x64/bin/java -cp .
MatrixMultiplication 2000 2000 2000 2000
```

4. `top` コマンドを実行して `java` プロセスの PID を取得します。
5. **[Analysis Target (解析ターゲット)]** タブで次の項目を設定します。
    - **[local host (ローカルホスト)]** ターゲット・システム・タイプ
    - **[Attach To Process (プロセスにアタッチ)]** ターゲットタイプ

- `java` の **PID** (プロセス ID) (例: 24116)

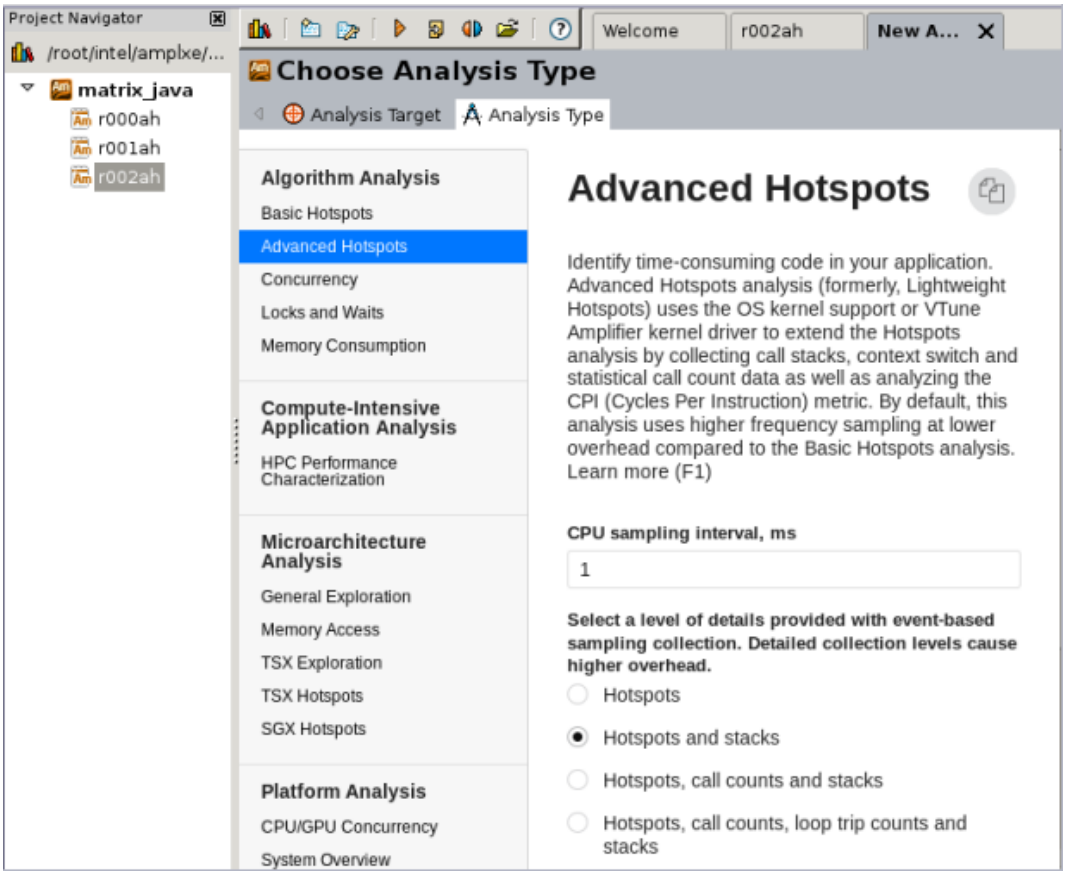


- 右の **[Binary/Source Search (バイナリー/ソース検索)]** ボタンをクリックして、ホスト上のソースが配置されている場所を指定します。



インテル® VTune™ Amplifier は、このパスを検索して、hotspot の原因であるソースコードとマシン命令を関連付けます。

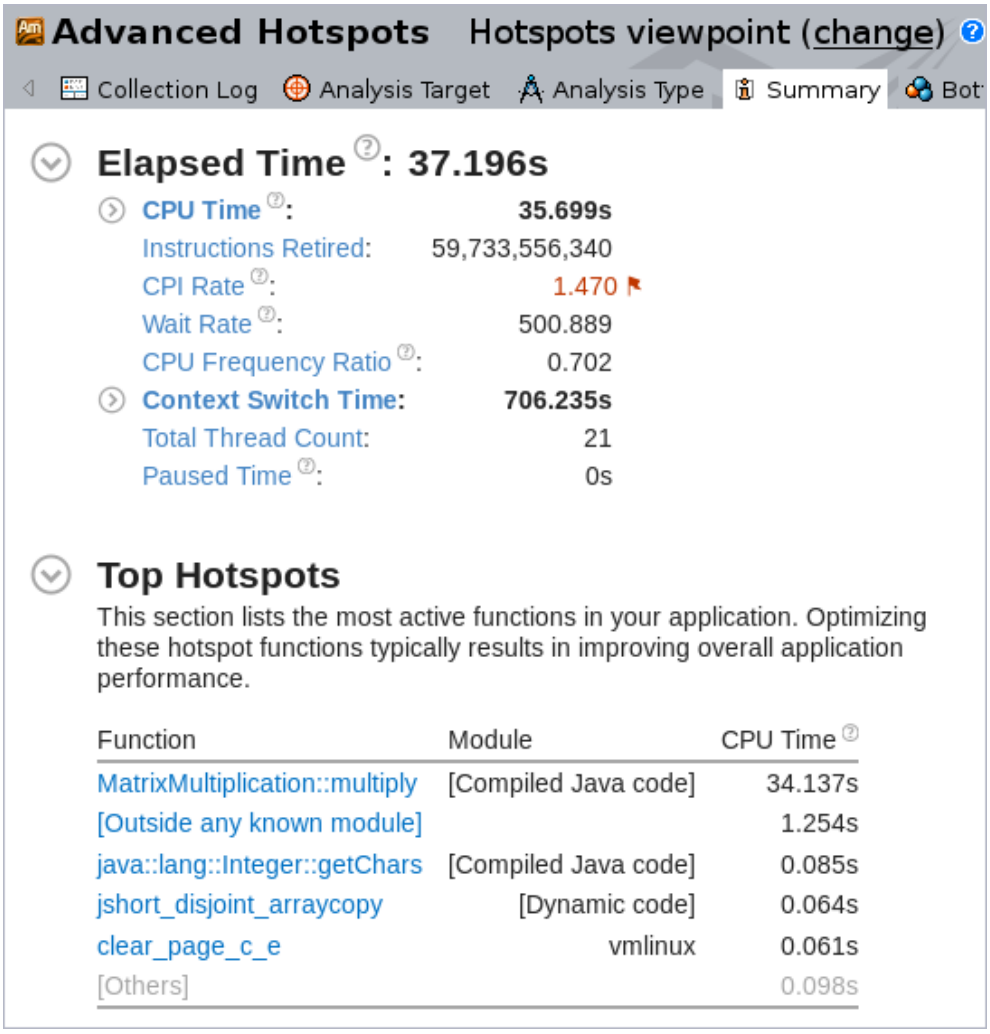
6. **[Analysis Type (解析タイプ)]** タブに切り替えて、**[Advanced Hotspots (高度な hotspot)]** 解析タイプを選択し、**[Hotspots and stacks (hotspot とスタック)]** オプションを選択します。



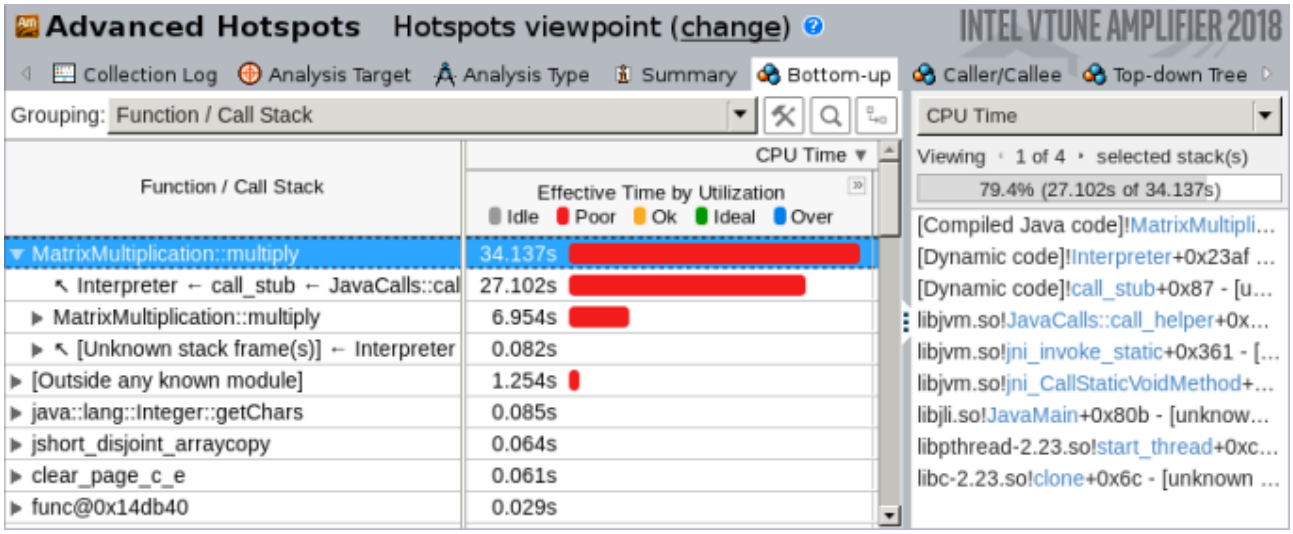
7. **[Start (開始)]** をクリックして解析を開始します。

コンテナーで収集したデータを解析する

データ収集が完了すると、インテル® VTune™ Amplifier はデフォルトの [Hotspots (hotspot)] ビューポイントに結果を表示します。

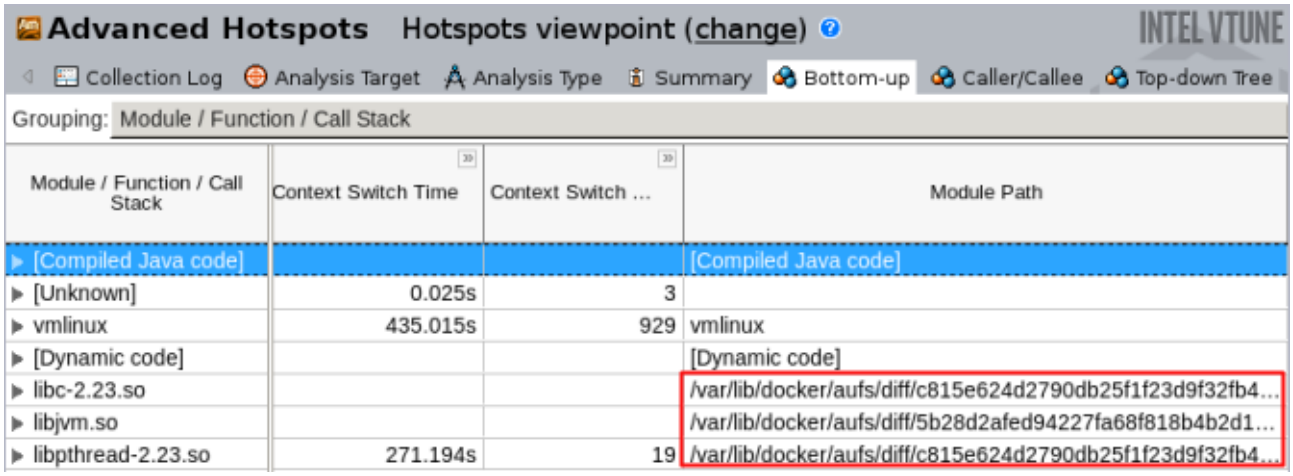


[Summary (サマリー)] ビューの **[Top Hotspots (上位の hotspot)]** セクションは、ターゲット・アプリケーションの `multiply` 関数に最も CPU 時間がかかっていることを示しています。リストの関数をクリックして **[Bottom-up (ボトムアップ)]** タブに切り替え、この hotspot 関数のスタックフローを確認します。



詳細に解析するため、hotspot 関数をダブルクリックして関数の hotspot ソース行を特定し、この行で収集されたメトリックデータを解析します。

Docker* コンテナーモジュールの統計を取得するには、**[Module/Function/Call Stack (モジュール/関数/コールスタック)]** でデータをグループ化し、モジュールパスの `docker` エントリーでコンテナーモジュールを識別します。



注

このレシピの情報は、デベロッパー・フォーラム (英語) を参照してください。

関連情報

- Java* コード解析 (英語)
- コンテナーターゲットのプロファイル (英語)

プロキシーサーバーを介したリモートターゲットのプロファイル

このレシピは、プロキシーサーバーを介してインテル® VTune™ プロファイラーを実行し、リモートターゲットをプロファイルする方法を説明します。

コンテンツ・エキスパート: Kirill Uhanov (英語)

リモート・ターゲット・システムをプロファイルする必要がある場合、このレシピに従って、プロキシーサーバーを介してインテル® VTune™ プロファイラーを実行します。このレシピは、Windows*、Linux*、および macOS* システムのホスト設定について説明します。

使用するもの

以下は、パフォーマンス解析シナリオで使用するソフトウェアのリストです。

- **オペレーティング・システム:** Windows*、Linux*、または macOS* システム。
- **ツール:** インテル® VTune™ プロファイラー。

  注

    - バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
    - インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
    - 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
        - インテル® VTune™ プロファイラー製品ページ
        - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)

Windows* ホスト設定

必要条件:

パスワードを使用しない SSH 接続用の RSA プライベート/パブリックキーがない場合は、以下の手順に従って、インテル® VTune™ プロファイラーの内部ジェネレーターを使用して生成します。

1. 次のコードを含む `generator.py` という名前の Python* スクリプトを作成します。スクリプト中の `USER` は実際のユーザー名、`HOSTNAME` はターゲットマシン名にそれぞれ置き換えます。

```
import sys
import pythonhelpers1.genhelpers as genhelpers
if len(sys.argv) < 2:
    print("Usage: amplxe-python generator.py USER@HOSTNAME")
    sys.exit(1)
private_key = "id_rsa_vtune_" + str(sys.argv[1])
genhelpers.ssh_keygen(private_key, private_key + '.pub')
```

2. コマンドウィンドウで次のコマンドを実行します。

```
VTUNE_INSTALL_DIR\bin64\amplxe-python generator.py USER@HOSTNAME
```

現在のディレクトリーにプライベート・キーとパブリックキーが生成されます。

プロキシーサーバーを介して Windows* ホストでインテル® VTune™ プロファイラーを実行する

1. 以下のキーをホストシステムの `%USERPROFILE%\.ssh` ディレクトリーにコピーします。
   - `id_rsa_vtune_`**`USER@HOSTNAME`**
   - `id_rsa_vtune_`**`USER@HOSTNAME`**`.pub`
2. パブリックキーの内容をターゲットシステムの `~/.ssh/authorized_keys` と `~` (**USER** ホーム・ディレクトリー) に追加します。
3. プロキシーサーバーを介したリモート接続のために、Windows* ホストにサードパーティー製の `Ncat` ユーティリティーをインストールします。これは https://nmap.org/ncat (英語) からダウンロードできます。
4. 次の行を含む `%USERPROFILE%\.ssh\config` を作成します。

```
Host HOSTNAME

    ProxyCommand “PATH\TO\NCAT\ncat.exe” --proxy-type <TYPE> --proxy
<PROXYADDR[:PORT]> %h %p
```

   説明:

   - `TYPE` は、プロキシーサーバーのタイプ。
   - `PROXYADDR` は、プロキシーサーバーのアドレス。
   - `PORT` は、ポート番号。
5. config とプライベート・キーの権限を変更します。

```
icacls %USERPROFILE%\.ssh\id_rsa_vtune_USER@HOSTNAME /inheritance:r
icacls %USERPROFILE%\.ssh\id_rsa_vtune_USER@HOSTNAME /grant:r
"%USERNAME%":"(R)"
icacls %USERPROFILE%\.ssh\config /inheritance:r
icacls %USERPROFILE%\.ssh\config /grant:r "%USERNAME%":"(R)"
```

6. `uname` コマンドで接続を確認します。

```
VTUNE_INSTALL_DIR\bin64\ssh.exe -i
"%USERPROFILE%\.ssh\id_rsa_vtune_USER@HOSTNAME" USER@HOSTNAME uname
```

これで、Windows* ホストでインテル® VTune™ プロファイラーを実行してリモートターゲットをプロファイルできます。

Linux*/macOS* ホスト設定

必要条件: パスワードを使用しない SSH 接続用の RSA プライベート/パブリックキーがない場合は、空のパスフレーズでこれらのキーを生成します。

```
host> ssh-keygen -t rsa
```

プロキシーサーバーを介して Linux*/macOS* ホストでインテル® VTune™ プロファイラーを実行する

1. パブリックキーの内容をターゲットシステムの `~/.ssh/authorized_keys` と `~` (**USER** ホーム・ディレクトリー) に追加します。
2. まだホストシステムにインストールされていない場合は、プロキシーを介したリモート接続のために、サードパーティー製の Ncat または Netcat ユーティリティーをダウンロードしてホストにインストールします。これは以下のサイトからダウンロードできます。
   - https://nc110.sourceforge.io/ (英語)
   - https://nmap.org/ncat (英語)
3. 次の行を含む `~/.ssh/config` を作成します。`nc` オプションは使用するバージョンに依存します。

   ```
   Host HOSTNAME
       ProxyCommand nc -X <TYPE> -x <PROXYADDR[:PORT]> %h %p
   ```

   または

   ```
   Host HOSTNAME
       ProxyCommand nc --proxy-type <TYPE> --proxy <PROXYADDR[:PORT]> %h %p
   ```

   説明:

   - `HOSTNAME` は、ターゲットマシンの名前。
   - `TYPE` は、プロキシーサーバーのタイプ。
   - `PROXYADDR` は、プロキシーサーバーのアドレス。
   - `PORT` は、ポート番号。
4. `uname` コマンドで接続を確認します。

   ```
   ssh USER@HOSTNAME uname
   ```

これで、Linux* または macOS* ホストでインテル® VTune™ プロファイラーを実行してリモートターゲットをプロファイルできます。

関連情報

- リモート解析向けに Linux* システムを設定 (英語)
- リモート収集向けに SSH アクセスを設定 (英語)

インテル® VTune™ プロファイラー・サーバーと Visual Studio* Code およびインテル® DevCloud for oneAPI の併用

このレシピでは、インテル® VTune™ プロファイラーをウェブサーバーとして使用し、リモートの開発マシンでパフォーマンスのチューニングを行う方法を紹介します。例として、リモートマシンにインテル® DevCloud for oneAPI のコンピュート・ノードを使用します。

コンテンツ・エキスパート: Stas Neverov (英語)、Jennifer DiMatteo (英語)

使用するもの

以下は、このレシピで使用するハードウェアとソフトウェアのリストです。

- インテル® DevCloud for oneAPI (英語) へのアクセス
- Visual Studio* (VS) Code (英語)
- インテル® VTune™ プロファイラー (インテル® DevCloud for oneAPI (英語) で利用可能)

注:

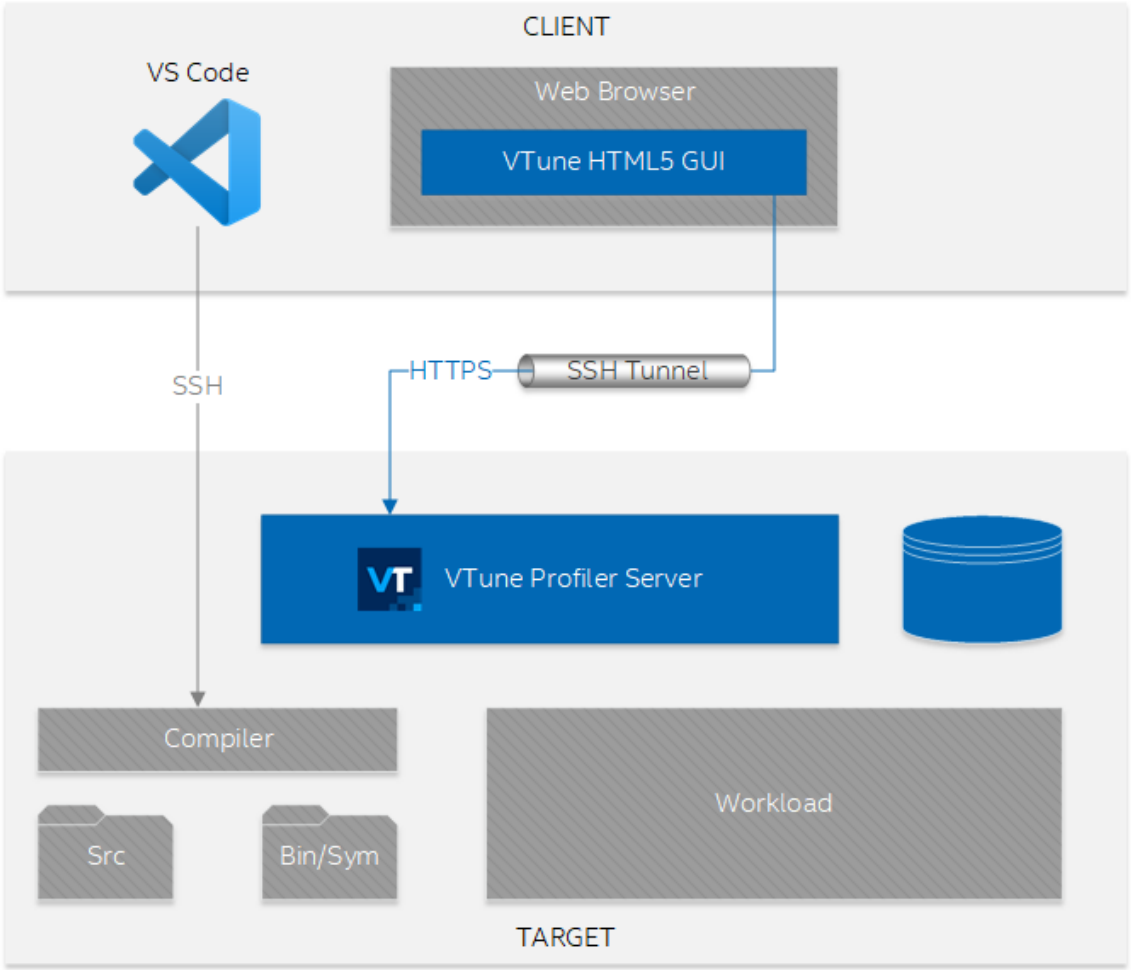
- バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
- インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
- 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
  - インテル® VTune™ プロファイラー製品ページ
  - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)

設定の概要

バージョン 2021.1.1 以降では、インテル® VTune™ プロファイラーをサーバーとして実行し、ウェブブラウザーを使用してリモートアクセスできます。この設定は、リモートシステムでアプリケーションを開発する場合に便利です。

- 開発システムでインテル® VTune™ プロファイラーを実行することで、バイナリー、デバッグ情報、ソースファイルに直接アクセスできます。
- また、インテル® VTune™ プロファイラーは、収集したトレースと処理データを同じシステムに保存するので、解析のためこれらのデータをクライアント・システムに転送する必要はありません。
- クライアント・システムに何もインストールする必要はありません。ウェブブラウザーだけあれば、インテル® VTune™ プロファイラーの GUI にアクセスできます。

以下は、この設定を図解したものです。



オプション 1: リモート開発にインテル® VTune™ プロファイラー・サーバーと Visual Studio* Code を併用する

1. インテル® DevCloud for oneAPI (英語) にログインします。
2. VS Code の接続を設定します (英語)。この手順を完了すると、ローカルの VS Code がインテル® DevCloud のコンピュート・ノードに接続されます。
3. VS Code のターミナルからコンピュート・ノード上のインテル® VTune™ プロファイラー・サーバーを実行します。

```
vtune-backend --enable-server-profiling
```

4. インテル® VTune™ プロファイラーの GUI を実行します。Ctrl キーを押しながら VS Code のターミナルでインテル® VTune™ プロファイラー・サーバーによって表示された URL をクリックして、ウェブブラウザーでインテル® VTune™ プロファイラーの GUI を開きます。
5. 設定を完了します。

オプション 2: SSH ターミナルを介してリモートシステムでインテル® VTune™ プロファイラー・サーバーを使用する

このケースでは、SSH トンネルを手動で設定する必要があります。この手順を簡略化するため、インテル® VTune™ プロファイラーを特定のポート (この例では `55001`) で実行します。`55001` がビジーの場合は、別のポートを選択できます。

1. インテル® DevCloud for oneAPI (英語) にログインします。
2. Windows* (英語) または Linux*/macOS* (英語) システム用の手順に従って、インテル® DevCloud への SSH 接続を設定します。
3. インテル® DevCloud のログインノードにログインします。

   ```
   ssh devcloud
   ```

4. インテル® DevCloud のコンピュート・ノードを予約します。

   ```
   qsub -I
   ```

   注:
   この手順の後、ターミナルを閉じないでください。ターミナルを閉じると、コンピュート・ノードが解放されます。

5. 新しいターミナルを開きます。
6. SSH ポートフォワードを有効にして、再度インテル® DevCloud のノードにログインします。

   ```
   ssh -L 127.0.0.1:55001:127.0.0.1:55001 devcloud
   ```

7. もう 1 つの SSH トンネルでログインノードからコンピュート・ノードへの SSH 接続を確立します。

   ```
   ssh -L 127.0.0.1:55001:127.0.0.1:55001 s000-n000
   ```

   `s000-n000` をお使いのコンピュート・ノード名に置き換えます。

8. コンピュート・ノード上でインテル® VTune™ プロファイラーを起動します。

   ```
   vtune-backend --web-port=55001 --enable-server-profiling
   ```

9. インテル® VTune™ プロファイラーの GUI を開きます。ウェブブラウザーでインテル® VTune™ プロファイラー・サーバーによって表示された URL を開きます。
10. 設定を完了します。

設定の完了

オプション 1 とオプション 2 のどちらを選択した場合も、以下の手順で設定を完了します。

1. インテル® VTune™ プロファイラー・サーバーの証明書を承認します。

   インテル® VTune™ プロファイラーの GUI を開いたときに、ウェブブラウザーがインテル® VTune™ プロファイラー・サーバーの自己署名証明書に関する確認メッセージを表示する場合があります。SSH トンネルは中間者攻撃 (MitM) から保護されているため、証明書をインストールしなくても安全に作業を進めることができます。トランスポート・セキュリティーの詳細は、「トランスポート・セキュリティー」(英語) を参照してください。

2. パスフレーズを設定します。

   インテル® VTune™ プロファイラー・サーバーを初めて起動したときに表示される URL には、ワンタイムトークンが含まれています。ブラウザーでこの URL を開くと、インテル® VTune™ プロファイラー・サーバーによってパスフレーズの設定を求められます。パスフレーズがないと、ほかのユーザーはインテル® VTune™ プロファイラー・サーバーにアクセスできません。パスフレーズのハッシュはサーバーに保存されます。また、ブラウザーには安全な HTTP クッキーが保存されるので、インテル® VTune™ プロファイラーの GUI を開くたびにパスフレーズを入力する必要はありません。パスフレーズを設定すると、インテル® VTune™ プロファイラーのようこそ画面が表示されます。

3. プロジェクトを作成します。
4. 解析を設定します。`--enable-server-profiling` オプションを指定してインテル® VTune™ プロファイラー・サーバーを起動したため、デフォルトではリモートマシンがターゲットシステムとして選択されます。
5. ターゲット・アプリケーションのパスとコマンドライン引数を設定します。詳細は、「解析ターゲットの設定」(英語) を参照してください。
6. 解析を実行します。

使用上の注意点

- オプション 1 の設定では、VS Code Remote - SSH 拡張 (英語) を利用して、VS Code のターミナルから起動されたプロセスで使用されるポート番号を監視します。Remote - SSH 拡張は、これらのポートを自動的に SSH トンネルに転送します。この動作は、デフォルトで有効な **remote.autoForwardPorts** 設定によって制御されます。
- `--enable-remote-profiling` コマンドライン・オプションを使用すると、インテル® VTune™ プロファイラー・サーバーをホストしているシステムをパフォーマンス・プロファイリング・ターゲットにすることができます。インテル® VTune™ プロファイラー解析の実行では、任意のコマンドラインでターゲット・アプリケーションを起動するため、セキュリティーの観点からこのオプションはデフォルトで無効になっています。複数のユーザーがインテル® VTune™ プロファイラー・サーバーの 1 つのインスタンスにアクセスすると、インテル® VTune™ プロファイラー・サーバーを実行するユーザーアカウントに代わって任意のコードを実行できるようになります。`--enable-remote-profiling` は、インテル® VTune™ プロファイラー・サーバーが単一のユーザーを対象としており、サーバーへのアクセスに使用するパスフレーズを共有していない場合にのみ有効にします。
- `--web-port=PORT` コマンドライン・オプションを使用して、インテル® VTune™ プロファイラー・サーバーを特定のポートで実行します。このオプションを使用しない場合、インテル® VTune™ プロファイラーはシステムで利用可能な任意のポートで実行されます。

- インテル® VTune™ プロファイラー・サーバーは、以下の警告を出力します。

  ```
  warn: Server access is limited to localhost only. To enable remote
  access restart with --allow-remote-ui.
  ```

  この手順では SSH ポートフォワードを使用するため、`--allow-remote-ui` を有効にする必要はありません。インテル® VTune™ プロファイラー・サーバーへの着信接続は SSH サーバーからの localhost 接続です。`--allow-remote-ui` を有効にすると、インテル® VTune™ プロファイラー・サーバーは実際のネットワーク・カードの IP アドレスまたは FQDN 名を含む URL を構築しますが、クライアント・マシンからはアクセスできない場合があります。

- デフォルトでは、インテル® VTune™ プロファイラー・サーバーはプロファイル結果をホーム・ディレクトリーに保存します。`--data-directory` コマンドライン引数を使用して、別のデータ・ディレクトリーを指定できます。また、この引数を使用して、インテル® VTune™ プロファイラー・サーバーで事前に収集されたインテル® VTune™ プロファイラーの結果を開くこともできます。インテル® VTune™ プロファイラーは、任意の子フォルダーに配置された結果を見つけることができます。

関連情報

インテル® VTune™ プロファイラーのウェブ・サーバー・インターフェイス (英語)
インテル® VTune™ プロファイラー・サーバーの使用モデル (英語)
インテル® DevCloud (英語)
インテル® VTune™ プロファイラーを使用したプロファイル解析の実行 (英語)

Singularity* コンテナーでのプロファイル

このレシピは、インテル® VTune™ Amplifier の解析向けに Singularity* コンテナーを構成して、独立したコンテナー環境で動作しているアプリケーションの hotspot を特定します。

コンテンツ・エキスパート: Denis Pravdin (英語)

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** `MatrixMultiplication`。

  この Java* アプリケーションはデモ用であり、ダウンロードすることはできません。

- **ツール:** インテル® VTune™ Amplifier 2018

注

  - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
  - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
  - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **Linux* コンテナーランタイム:** `singularity`
- **オペレーティング・システム:** Ubuntu* 16.04
- **CPU:** インテル® プロセッサー (開発コード名 Skylake)、8 論理 CPU

Singularity* コンテナーをインストールして設定する

1. Singularity* ソフトウェア (例えばバージョン 2.4.5) をインストールします。

```
host> VERSION=2.4.5
host> wget
https://github.com/singularityware/singularity/releases/download/$VERSI
ON/singularity-$VERSION.tar.gz
host> tar xvf singularity-$VERSION.tar.gz
host> cd singularity-$VERSION
host>./configure --prefix=/usr/local
```

```
host> make
host> sudo make install
```

注

詳しいインストール手順は、https://singularity.lbl.gov/install-linux (英語) を参照してください。

2. 新しい Singularity* コンテナーを、例えば Docker* Hub を使用して作成します。

```
host> singularity build ubuntu.img docker://ubuntu:latest
Docker image path: index.docker.io/library/ubuntu:latest
Cache folder set to /root/.singularity/docker
Importing: base Singularity environment
Importing:
/root/.singularity/docker/sha256:d3938036b19cfa369e1081a6776b07b54be961
2bc4c8fed7f139370c8142b79f.tar.gz
Importing:
/root/.singularity/docker/sha256:a9b30c108bda615dc10e402f62d712f413214e
a92c7ec4354cd1cc0f3450bc58.tar.gz
Importing:
/root/.singularity/docker/sha256:67de21feec183fcd009a5eddc4de8c346ee0f4
369a20047f1a302a90716fc741.tar.gz
Importing:
/root/.singularity/docker/sha256:817da545be2ba4bac8f6b4da584bca0fb48449
38ecc462b9feab1001b5df7405.tar.gz
Importing:
/root/.singularity/docker/sha256:d967c497ce230b63996a7b1fc6ec95b741aea9
348118d3328c676f13be789fa7.tar.gz
Importing:
/root/.singularity/metadata/sha256:c6a9ef4b9995d615851d7786fbc2fe72f723
21bee1a87d66919b881a0336525a.tar.gz
Building Singularity image...
Singularity container built: ubuntu.img
Cleaning up...
```

注

`Ubuntu.img` ファイルが現在のディレクトリーに作成されることを確認してください。

3. コンテナーを実行します。

Singularity* は、ホストシステム上のディレクトリーをコンテナー内にマップできます。これにより、簡単にホストシステム上のデータを読み書きできます。例えば、インテル® VTune™ Amplifier と Java* アプリケーションでホストフォルダー `/tmp/vtune` を使用する場合、コンテナーを実行して `/tmp/vtune` をコンテナー内の `/local/vtune` へマップする必要があります。

```
host> singularity shell --bind /tmp/vtune:/local/vtune ./ubuntu.img
Singularity: Invoking an interactive shell within container...
Singularity ubuntu.img:~>
```

コンテナー内でパフォーマンス解析を実行する

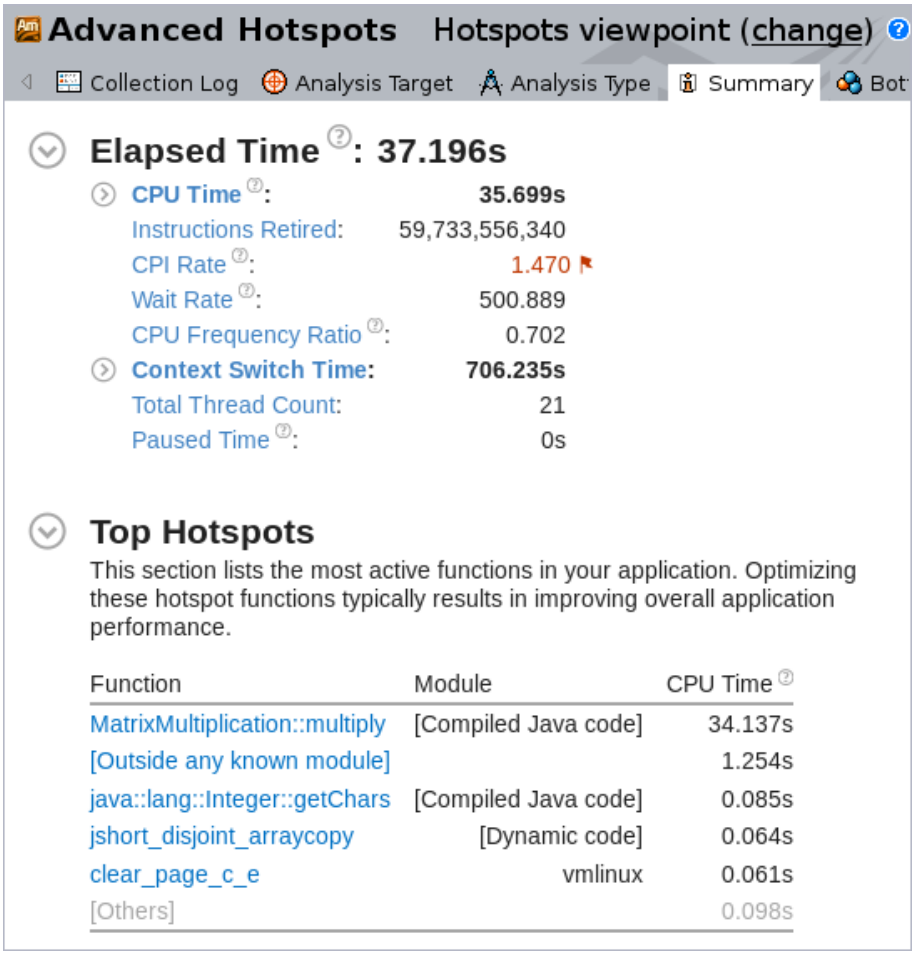
Singularity* コンテナーからインテル® VTune™ Amplifier のコマンドライン・インターフェイス `amplxe-cl` を起動して、Java* アプリケーションの解析を実行します。例えば、`MatrixMultiplication` アプリケーションの高度な hotspot 解析を実行するには、次のコマンドを入力します。

```
Singularity ubuntu.img:~> cd /local/vtune/matrix/
Singularity ubuntu.img:/local/vtune/matrix> /local/vtune/bin64/vtune -collect
advanced-hotspots -- /local/vtune/jdk9.0.1-x64/bin/java -cp .
MatrixMultiplication 2000 2000 2000 2000
```

注

- Singularity* コンテナーで動作しているターゲット・アプリケーションをプロファイルするには、同じコンテナーからインテル® VTune™ Amplifier を起動します。Singularity* プロファイルでは、コンテナー外からのインテル® VTune™ Amplifier の実行はサポートされていません。
- 高度な hotspot 解析は、インテル® VTune™ Amplifier 2019 で汎用の hotspot 解析 (英語) に統合されました。ハードウェア・イベントベース・サンプリング収集モードで利用できます。

収集後、ホストシステムにインストールされているインテル® VTune™ Amplifier の GUI から解析結果がを開き、従来の解析フローに従ってアプリケーション・パフォーマンスの概要を提供する **[Summary (サマリー)]** ウィンドウから開始します。



注

Singularity* コンテナー外 (例えば、ホストシステムにインストールされているインテル® VTune™ Amplifier の GUI) で解析結果を再ファイナライズする必要がある場合、モジュールに必要なすべてのバイナリーとソースファイルにコンテナー外からアクセスできることを確認してください。

関連情報

- Java* コード解析 (英語)

Linux*、Android*、および QNX* のシステムブート時のプロファイル

このレシピは、インテル® VTune™ Amplifier のパフォーマンス解析を Linux*、Android*、および QNX* オペレーティング・システムのブートフローと統合する方法を示します。この解析は、OS ブート時に CPU コアで予想外に長く実行されるアクティビティーを識別するのに役立ちます。これにより、ブート順序のさらに詳しい調査が可能になります。

コンテンツ・エキスパート: Vitaly Slobodskoy (英語)、Kirill Uhanov (英語)、Dmitry Obrezchikov (英語)、Artem Shcherbak (英語)

ブート時のプロファイルでは、インテル® VTune™ Amplifier のパフォーマンス・データ収集コマンドを (初期化スクリプトで設定するか、特定のサービスを使用して) OS ブートの初期段階に挿入します。最良の結果を得るため、次の要件を満たしていることを確認してください。

- インテル® VTune™ Amplifier データコレクターのバイナリーファイルを利用可能な最も早くロードされるファイル・ディレクトリーに配置します。
- Linux* と Android* では、データコレクターはファイルシステムに書き込むため、出力ファイル名は利用可能な最も早くロードされる書き込み可能ディレクトリーを使用する必要があります。
- Linux* と Android* では、インテル® VTune™ Amplifier のデータ収集コマンドはファイルシステムの可用性に依存すべきです。QNX* では、コマンドはネットワークの可用性に依存すべきです。

注

このアプローチは、OS ブート時のいくつかの問題に対処するのに適していますが、ブートプロセス全体をカバーすることはできません。例えば、カーネルの展開とファイルシステムのマウントはカバーされません。

使用するもの

以下は、パフォーマンス解析シナリオで使用するソフトウェアのリストです。

- **オペレーティング・システム:**
    - `systemd` によりシステムを初期化する Linux* (root アクセスが有効)
    - QNX*
- **ツール:**
    - QNX* Momentics* ツールスイート

- QNX* 7.0 SDK
- インテル® VTune™ Amplifier 2019 以降

注

- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
- ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。

Linux* ターゲットシステムのブート時をプロファイルする

必要条件:

- ターゲット Linux* システムにインテル® VTune™ Amplifier をインストール (英語) します。
- システムの初期化方法をチェックします。システムが `systemd` を使用していることを確認するには、次のコマンドを実行します。

```
systemctl | grep "\-\.mount"
```

`systemd` が使用されている場合、次の出力になります。

```
-.mount  loaded  active  mounted  /
```

Linux* システムのブート時をプロファイルする

1. `/boot_profile` ファイルを作成して、インテル® VTune™ Amplifier のハードウェア解析 (hotspot、I/O、ほか) を実行するように設定します。

   例えば、高精度のデータが得られるように短いサンプリング間隔でシステム全体の hotspot 解析を 30 秒間実行するには、次のコマンドを使用します。

```
#!/bin/bash
/opt/intel/vtune_amplifier/bin64/amplxe-cl -c hotspots -knob sampling-mode=hw -knob sampling-interval=0.1 -d 30 -finalization-mode=none -r /tmp/boot_profile &
```

注

- ファイル内のパスがインテル® VTune™ Amplifier のインストール・ディレクトリーを正しく指定していることを確認してください。デフォルトでは、Linux* のインストール・ディレクトリーは、`/opt/intel/vtune_amplifier` です。
- `boot_profile` スクリプトのパスは、`/tmp` など、初期ブート段階で利用可能な任意のローカルパスにすることができます。

2. データ収集起動スクリプトの権限を変更します。

```
chmod 755 /boot_profile
```

3. 次の内容で `/etc/systemd/system/vtune_boot.service` ファイルを作成します。

```
[Unit]
Description=VTune Amplifier boot profile service

[Service]
Type=forking
ExecStart=/boot_profile

[Install]
WantedBy=multi-user.target
```

4. サービスを有効にします。

```
systemctl enable vtune_boot
```

5. システムを再起動して、OS ブートプロセス中にインテル® VTune™ Amplifier のデータ収集を開始します。

データ収集が完了すると、`/tmp/boot_profile` に結果ディレクトリーが生成されます。このディレクトリーは `root` ユーザー以下にあります。通常のユーザーで結果を開く必要がある場合、フォルダーのアクセス権限を変更してください。

```
sudo chmod -R a+w /tmp/boot_profile
```

追加のコマンド

- サービスを無効にします。

```
systemctl disable vtune_boot
```

- 障害が発生した場合、OS ブートプロセス中にインテル® VTune™ Amplifier コレクター の出力を解析します。

```
sudo journalctl -u vtune_boot
```

Android*ターゲットシステムのブート時をプロファイルする

必要条件:

1. ホストシステムにインテル® VTune™ Amplifier をインストールします。
2. Android* ターゲットシステムで `lsmod` コマンドを実行して、インテルのサンプリング・ドライバーが利用できることを確認します。

   ドライバー (`pax.ko`、`sep5.ko`、`socperf3.ko`) が見つからない場合、以降のステップをスキップしてドライバーを使用しないモードで続行するか、以降のステップに従ってドライバーをビルドして署名します。

a. ホストシステムで次のコマンドを実行します。

```
<vtune-install-dir>/target/<android-version-arch>/sepdk/build-driver
```

b. 入力を求められたら、ターゲットシステムのビルドに使用する GCC* コンパイラーと Android* カーネル・ソース・ディレクトリーのパスを指定します。

例えば、カーネル・ソース・ディレクトリーは `<android-source-dir>/out/target/product/<name>/obj/kernel` で、コンパイラー・ディレクトリーは `<android-source-dir>/prebuilts/gcc/linux-x86/x86/x86_64-linux-android-<version>/bin/x86_64-linux-android-gcc` です。

ビルドしたドライバーは次のディレクトリーにあります。

- `<vtune-install-dir>/target/<android-version-arch>/sepdk/pax/pax.ko`
- `<vtune-install-dir>/target/<android-version-arch>/sepdk/sep5.ko`
- `<vtune-install-dir>/target/<android-version-arch>/sepdk/src/socperf/src/socperf.ko`

c. 次のように、ドライバーに署名します。

```
$KERNEL_DIR/scripts/sign-file $(CONFIG_MODULE_SIG_HASH)
$KERNEL_DIR/$(CONFIG_MODULE_SIG_KEY)
$KERNEL_DIR/certs/signing_key.x509 <driver_file_name.ko>
```

ここで、*<driver_file_name.ko>* は署名するドライバーの名前です。各ドライバーは個別に署名する必要があります。

`KERNEL_DIR` にあるカーネルの `config` ファイルを使用して、`CONFIG_MODULE_SIG_HASH` と `CONFIG_MODULE_SIG_KEY` パラメーターの値を取得します。

Android* システムのブート時をプロファイルする

1. インテル® VTune™ Amplifier のターゲットコレクターをインストールします。
   a. 通常の方法でターゲットシステムをブートします。
   b. インテル® VTune™ Amplifier GUI を実行して新しいプロジェクトを作成します。
   c. 新しい解析を設定します。[WHERE (どこを)] ペインで、**[Android Device (ADB) (Android* デバイス (ADB))]** 接続タイプを選択します。[ADB destination (ADB の対象)] フィールドでターゲットデバイスを選択します。これ以降、インテル® VTune™ Amplifier は自動的にターゲットコレクターをターゲットシステムへアップロードします。
2. ターゲットコレクターを利用可能な最も早くロードされるファイルシステムの場所 (例: `/vendor`) へコピーします。

```
adb shell cp -rf /data/data/com.intel.vtune/perfrun /vendor/vtune
```

3. インテルのサンプリング・ドライバーが利用可能な場合、`pax.ko`、`sep5.ko`、および `socperf3.ko` ドライバーを `/vendor/vtune` へコピーします。

4. 収集トレースの出力先として、利用可能な最も早くロードされる書き込み可能な場所を選択します。例えば、/data/vtune を選択して、次のインテルのサンプリング・ドライバー・モードまたはドライバーを使用しないモードのいずれかで実行スクリプト (/vendor/vtune/vtune.sh) を作成します。

インテルのサンプリング・ドライバー・モード	ドライバーを使用しないモード
#!/bin/sh rm -rf /data/vtune mkdir /data/vtune 0777 /system/bin/insmod /vendor/vtune/pax.ko /system/bin/insmod /vendor/vtune/socperf3.ko /system/bin/insmod /vendor/vtune/sep5.ko LD_LIBRARY_PATH=/vendor/vtune/perfrun/lib64 SEP_BASE_DIR=/vendor/vtune/perfrun/lib64/ vendor/vtune/perfrun/bin64/sep -start -d 10 -out /data/vtune/android_boot.tb7	#!/bin/sh rm -rf /data/vtune mkdir /data/vtune 0777 echo 0 > /proc/sys/kernel/perf_event_paranoid echo 0 > /proc/sys/kernel/kptr_restrict /vendor/vtune/perfrun/bin64/amplxe-perf record -a -o /data/vtune/android_boot.data -- sleep 10

5. このスクリプトは、hotspot 収集を開始して 10 秒間実行します。
6. このスクリプトをターゲットの init.rc に追加します。実際のブートフローに応じて、post-fs やほかのトリガーの使用を検討してください。

```
on fs
          start vtune
service vtune /vendor/vtune/vtune.sh
          user root
          group root
          seclabel u:r:init:s0
          oneshot
          disabled
```

注

読み取り専用のファイルシステムでは、ホストでこれらのファイルを変更して、ソースコートから Android* システムをビルドすることを検討してください。

7. オプションで、OS の設定に応じて、次の行を /system/sepolicy/private/file_contexts ファイルに追加します。

```
/system/bin/toolbox      u:object_r:toolbox_exec:s0
+ /system/bin/insmod     u:object_r:toolbox_exec:s0
+ /system/bin/sep        u:object_r:toolbox_exec:s0
+ /system/bin/sh         u:object_r:toolbox_exec:s0
```

注

Android* デバイスが permissive モードでブートされていることを確認します。

8. Android* ターゲットシステムを再起動して、データが収集されるのを待ちます。
9. 詳しく解析するため、/data/vtune/android_boot.tb7 ファイルをホストシステムへコピーします。

QNX* システムのブート時をプロファイルする

必要条件:

- QNX* Momentics* Tool Suite をホストにインストールします。
- QNX* 7.0 SDK をインストールします。
- **[File]** > **[Import]** > **[QNX]** > **[QNX Source Package and BSP]** を選択して BSP を QNX* Momentics* ワークスペースにインポートします。
- ホストシステムにインテル® VTune™ Amplifier をインストール (英語) します。

QNX* システムのブート時をプロファイルする

1. `<vtune-install-dir>/target/qnx_x86_64` から `<qnx-sdk-path>\qnx700\target\qnx7\x86_64\usr\bin` へターゲット・プロファイル・エージェント (`sep` バイナリー) をコピーします。
2. QNX* イメージの `*.build` ファイルを変更します。
   a. 文字列 `/usr/bin/gzip=gzip` を検索して、その後に `/usr/bin/sep=sep` を追加します。
   b. 起動スクリプトセクションに `sep -p1 &` を追加します。

```
[+script] startup-script = {
...
    # NOTE: Temporary enable for UART devices on OCP bridge
    # will be able to removed once ABL is fixed
    ocp_init -d 0:24:0 0x200=0xffff04b5 0x204=7
    ocp_init -d 0:24:1 0x200=0xffff04b5 0x204=7
    ocp_init -d 0:24:2 0x200=0xffff04b5 0x204=7 # console
    ocp_init -d 0:24:3 0x200=0xffff04b5 0x204=7
    # the sep run before this could move system to unstable
    # state and crash it
    sep -p1 -d 10 &
```

`sep` ターゲット・プロファイル・エージェントのオプションは、次のとおりです。

- `-p<mode>` は、収集モードを設定します。

  `0` は、通常のデフォルトのモードです。プロファイル・エージェントは、ホストからの TCP/IP 接続を待機します。

  `1` は、エージェントがスタックを含まない事前定義済みの収集を開始します。収集サンプルはターゲットのメモリーに格納されます。ホストへデータを転送するには、TCP/IP 接続が必要です。

  `2` は、エージェントがサンプル・コールスタックを含む事前定義済みの収集を開始します。収集サンプルはターゲットのメモリーに格納されます。ホストへデータを転送するには、TCP/IP 接続が必要です。

- `-d <sec>` は、収集の最大期間 (秒) を設定します。収集は、指定した時間が経過した後、またはメモリーバッファーが一杯になると停止します。
- `-s <sec>` は、指定した時間が経過した後に収集を開始します。

        - `-b <size_ratio>` は、収集バッファーサイズを設定します (1 ^ size_ratio バイト)。例えば、単一の CPU コアの場合、8MB バッファーサイズは `-b 23` と指定します。エージェントは、CPU コアごとのターゲットメモリーの使用量が 16MB になるように、バッファーの切り替えにダブルバッファー・スキーマを使用します。4 CPU コアの場合、エージェントの合計メモリー使用量は 64MB になります。デフォルト値は 19 (0.5MB) です。
3. QNX* イメージをリビルドしてフラッシュします。
4. システムを再起動して、OS ブートプロセス中にインテル® VTune™ Amplifier のデータ収集を開始します。
5. 収集結果をホストに転送します。

   事前定義済みの収集モード (`-p1` または `-p2`) では、ターゲット・エージェントはメモリーバッファー内のワークロードを指定された期間プロファイルして、リスニングモードに切り替わり、次のようなメッセージをコンソールに出力します: *'sep5_0: Waiting for control connection from host on port XXXX...'* (sep5_0: ポート XXXX でホストからの制御接続を待機しています...)。このメッセージが出力されたら、ホストで `sep` ユーティリティーを起動して、ネットワーク経由でターゲットから収集データをコピーできます。使用するホストのコマンドライン・オプションがターゲット・エージェントのオプションと対応していることを確認してください。例えば、`-p1` モードの場合、ホストコマンドは次のようになります。

   ```
   <vtune-install-dir>/bin64/sep -start -target-ip <target-system-ip-address> -target-port 9321 -out /tmp/qnx_boot.tb7
   ```

   `-p2` の場合は次のようになります。

   ```
   <vtune-install-dir>/bin64/sep -start -target-ip <target-system-ip-address> -target-port 9321 -lbr call_stack -out /tmp/qnx_boot.tb7
   ```

結果をインテル® VTune™ Amplifier プロジェクトへインポートする

1. ホストシステムでインテル® VTune™ Amplifier スタンドアロン GUI を起動して、結果のファイナライズ中に適切なバイナリーファイルを選択します。

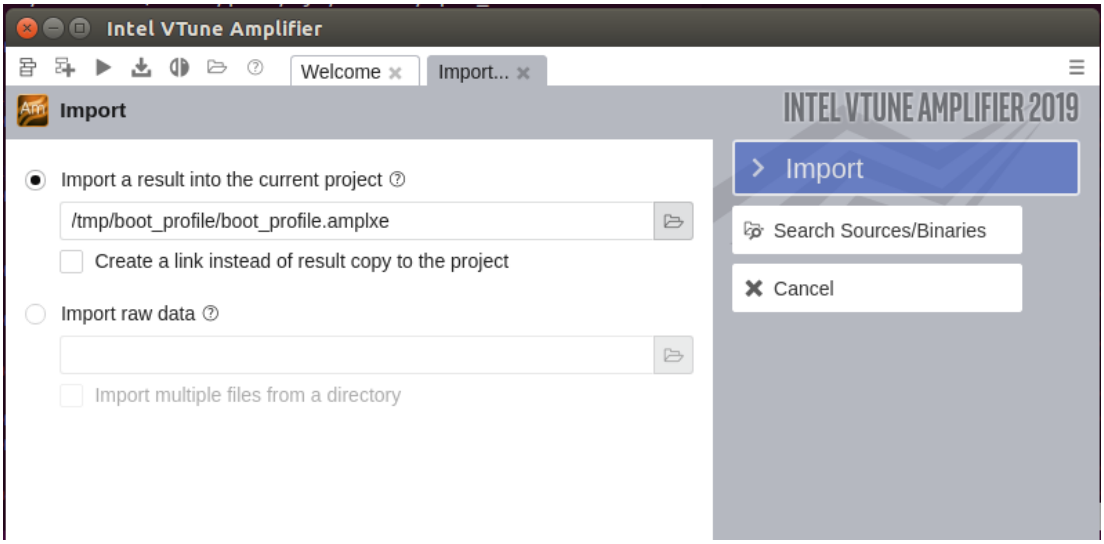
   例えば、Windows* システムでインテル® VTune™ Amplifier を起動するには、次のコマンドを使用します。

   ```
   <vtune-install-dir>/bin64/amplxe-gui.exe
   ```

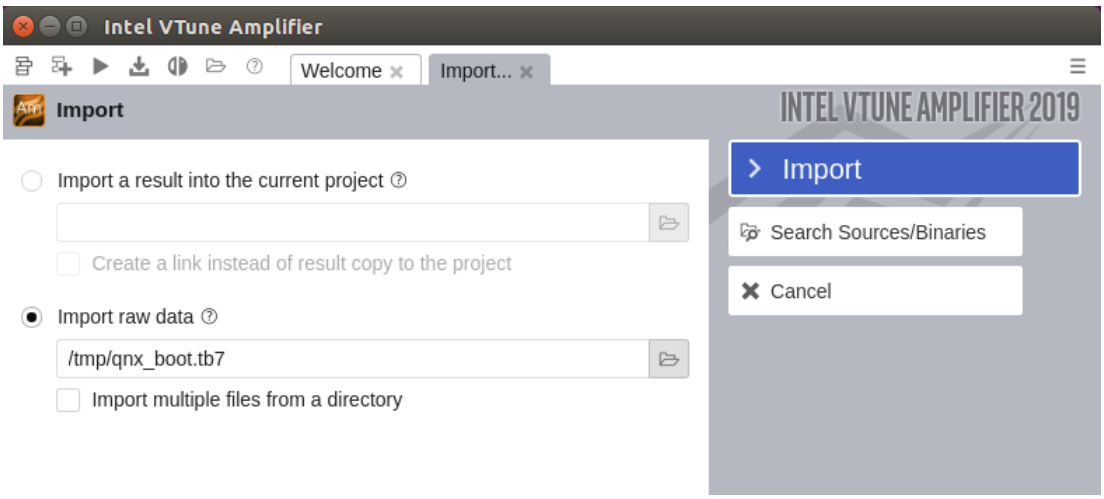
2. 新しいインテル® VTune™ Amplifier プロジェクトを作成して (英語)、バイナリー/シンボル検索ディレクトリー (英語) にカーネルやドライバーのデバッグファイルのパスを設定します。

   Linux* ホストでは、kptr_restrict value (英語) を 0 に変更してカーネル関数名の解決を有効にできます。

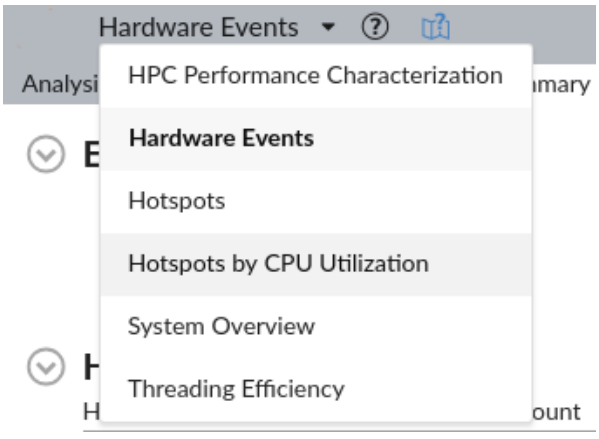
3. 結果をプロジェクトへインポートする
   - Linux* の結果をインポートするには、**[Import a result into the current project (現在のプロジェクトに結果をインポート)]** オプションを使用します。次に例を示します。



- Android* または QNX* から結果をインポートするには、**[Import raw data (生のトレースデータをインポート)]** オプションを使用して、参照ボタンをクリックして必要な `*.tb7` ファイルを選択します。

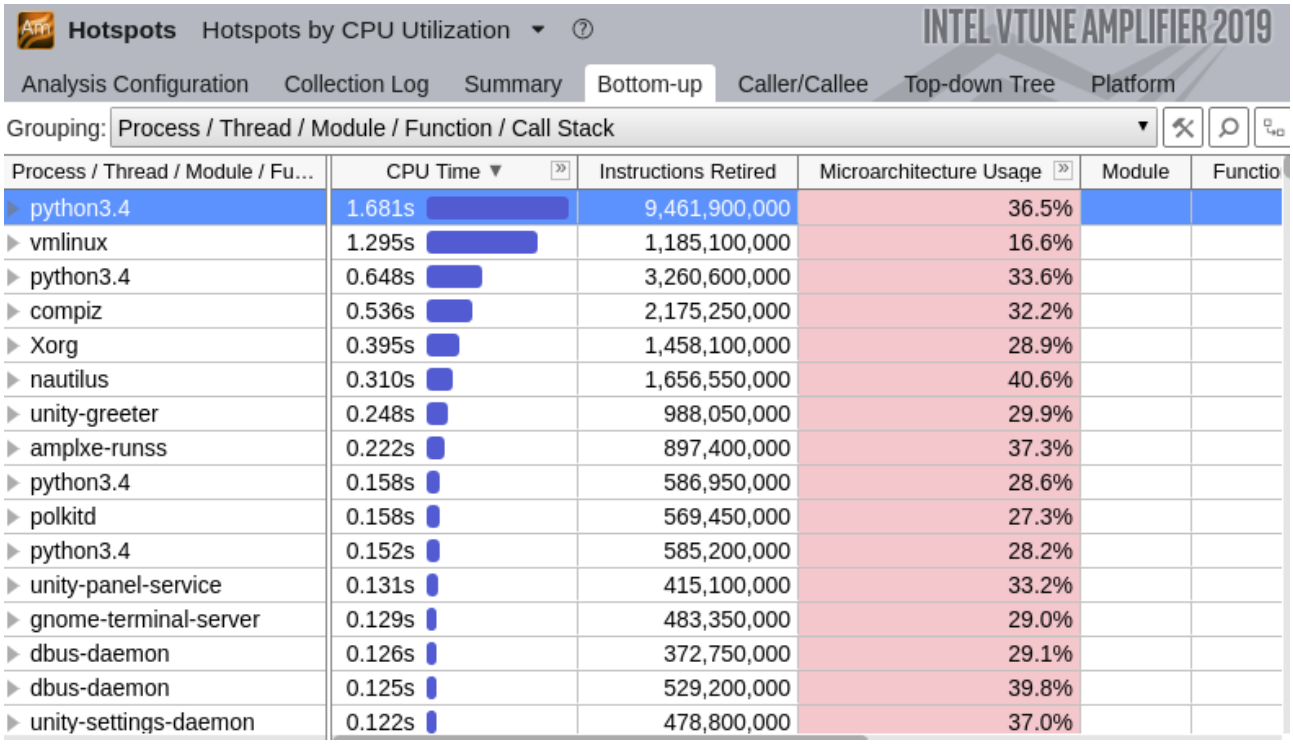


`*.tb7` ファイルがインポートされ結果がファイナライズされたら、**[Hotspots by CPU Utilization (CPU 利用率によるホットスポット)]** ビューポイントに切り替えます。



プロセスの実行を解析する

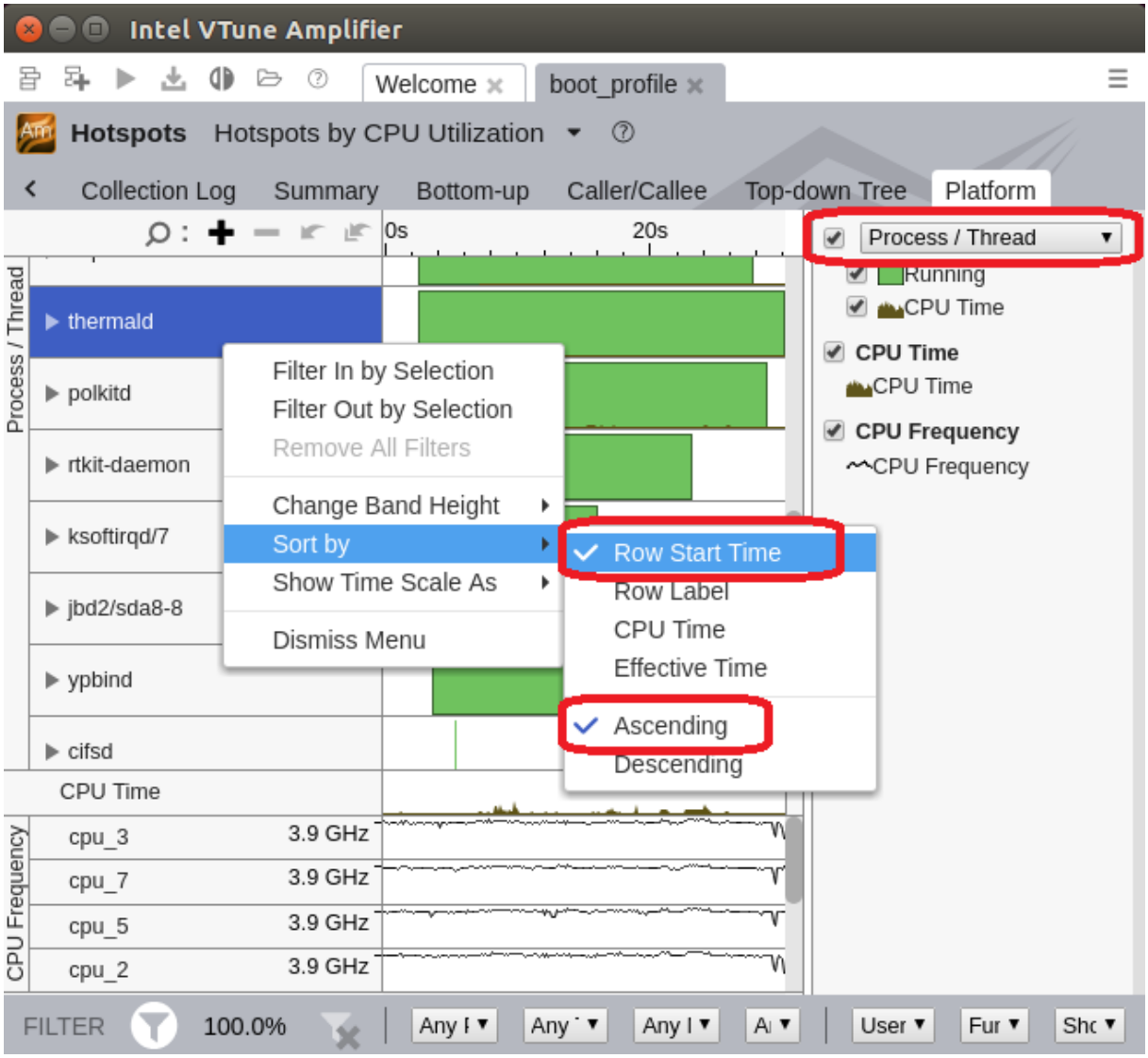
収集結果を開いて [Bottom-up (ボトムアップ)] タブ (英語) をクリックし、最も多くの CPU リソースを占有していたプロセスを特定します。



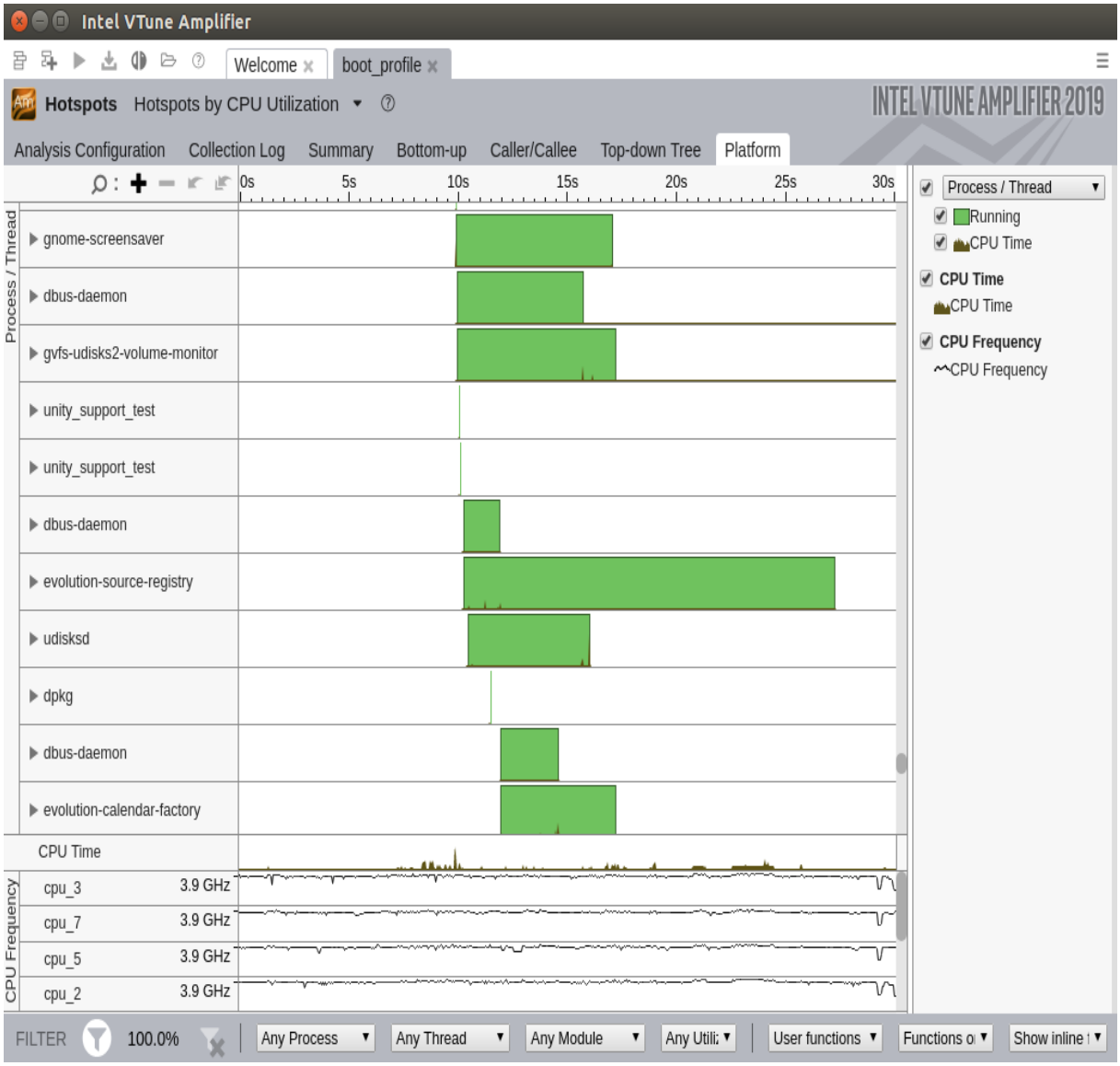
Process / Thread / Module / Fu...	CPU Time ▼	Instructions Retired	Microarchitecture Usage	Module	Functio
python3.4	1.681s	9,461,900,000	36.5%		
vmlinux	1.295s	1,185,100,000	16.6%		
python3.4	0.648s	3,260,600,000	33.6%		
compiz	0.536s	2,175,250,000	32.2%		
Xorg	0.395s	1,458,100,000	28.9%		
nautilus	0.310s	1,656,550,000	40.6%		
unity-greeter	0.248s	988,050,000	29.9%		
amplxe-runss	0.222s	897,400,000	37.3%		
python3.4	0.158s	586,950,000	28.6%		
polkitd	0.158s	569,450,000	27.3%		
python3.4	0.152s	585,200,000	28.2%		
unity-panel-service	0.131s	415,100,000	33.2%		
gnome-terminal-server	0.129s	483,350,000	29.0%		
dbus-daemon	0.126s	372,750,000	29.1%		
dbus-daemon	0.125s	529,200,000	39.8%		
unity-settings-daemon	0.122s	478,800,000	37.0%		

[Platform (プラットフォーム)] タブに切り替えて、プロセス/サービスの実行シーケンスを解析します。次の操作を行います。

1. タイムラインのグループ化を **[Process/Thread (プロセス/スレッド)]** に変更します。
2. 右クリックしてコンテキスト・メニューを開き、[Sort by (ソート)] から **[Row Start Time (開始時間行)]** と **[Ascending (昇順)]** を選択して行をソートします。


Intel VTune Amplifier
Welcome
boot_profile
Hotspots
Hotspots by CPU Utilization
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
0s
20s
Process / Thread
Running
CPU Time
CPU Time
CPU Time
CPU Frequency
CPU Frequency
Process / Thread
thermald
polkitd
rtkit-daemon
ksoftirqd/7
jbd2/sda8-8
ypbind
cifsd
Filter In by Selection
Filter Out by Selection
Remove All Filters
Change Band Height
Sort by
Show Time Scale As
Dismiss Menu
Row Start Time
Row Label
CPU Time
Effective Time
Ascending
Descending
CPU Time
CPU Frequency
cpu_3 3.9 GHz
cpu_7 3.9 GHz
cpu_5 3.9 GHz
cpu_2 3.9 GHz
FILTER
100.0%
User

3. プロセスの実行順序を解析します。



関連情報

- QNX* ターゲット (英語)
- 結果とトレースをインテル® VTune™ Profiler GUI へインポートする (英語)

システム・アナライザーによるリアルタイム・モニタリング

このレシピは、システム・アナライザーの概要を紹介し、ターゲットシステムをリアルタイムにモニタリングして、CPU、GPU、メモリー、ディスク、ネットワークによる制限を特定します。

コンテンツ・エキスパート: Jeffrey K. Reinemann (英語)

システム・アナライザーは、継続的なリアルタイム・モニタリングに加えて、基本パフォーマンスとリソースの使用状況に関するデータのレポートを可能にします。これは、インテル® VTune™ Amplifier 2019 Update 2 で追加されたプレビュー機能です。ユーザーからのフィードバックに応じて、将来のアップデートやリリースで利便性やユーザー体験が改善されたり、あるいは削除される可能性があります。

注

ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。

データが収集される「ターゲットシステム」には、Linux* または Windows* システムを利用できます。選択したデータを GUI で表示する「ホストシステム」には、Linux*、macOS*、または Windows* システムを利用できます。システム・アナライザーは、ほぼリアルタイムでターゲットシステムをモニタリングし、その高度な結果に基づいて、詳細なプロファイル向けに適切なインテル® VTune™ Amplifier の解析を設定できます。

システム・アナライザーは、ターゲット・データ・コレクターと、パフォーマンス・メトリックのタイムライン・グラフを表示するグラフィカル・ユーザー・インターフェイス (GUI) で構成されています。次のパフォーマンス・メトリックと表示オプションを利用できます。

- CPU 使用率と周波数
- メモリー使用率 (Linux* のみ)
- ネットワーク・スループット (Linux* のみ)
- GPU 使用率とその他のグラフィックス・メトリック (インテル® グラフィックス・ドライバーがインストールされている場合)

システム・アナライザーは、インテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA) パッケージにも含まれます。グラフィックス・レンダリングのワークロードとボトルネックを解析するには、システム・アナライザーとインテル® GAP パッケージに含まれるほかのツールを使用します。

- 使用するもの
- 手順:
    1. システム・アナライザーを起動する
    2. [System View] を設定する
    3. 詳細な解析向けに異常を特定する
    4. 異常を検出するようにインテル® VTune™ Amplifier を設定する
    5. システム・アナライザーとインテル® VTune™ Amplifier を実行する

使用するもの

以下は、パフォーマンス解析シナリオで使用するソフトウェアのリストです。

- **ツール**: インテル® VTune™ Amplifier 2019 (システム・アナライザー・パッケージを含む)

  ホスト上のシステム・アナライザー・パッケージのインストール・フォルダーは、`<install-dir>/system_analyzer` です。`<install-dir>` は、インテル® VTune™ Amplifier のバージョンにより異なります。

  - インテル® oneAPI ベース・ツールキットに含まれるインテル® VTune™ プロファイラー: `[Program Files]\inteloneapi\vtune\<version>` (Windows*)、`/opt/intel/inteloneapi/vtune/<version>` (Linux*)
  - スタンドアロン: `[Program Files]\IntelSWTools\VTune Profiler <version>` (Windows*)、`/opt/intel/vtune_profiler_version` (Linux* および Android*)
  - インテル® System Studio: `[Program Files]\IntelSWTools\system_studio_<version>\VTune Profiler` (Windows*)、Linux* は次のいずれか:
    - スーパーユーザーの場合: `/opt/intel/system_studio_version/vtune_profiler`
    - 通常のユーザーの場合: `$HOME/intel/system_studio_version/vtune_profiler`
  - Apple* macOS* システム: `/Applications/Intel VTune Profiler <version>.app`

  プロファイルされるターゲットシステムが以前にインテル® VTune™ Amplifier によってプロファイルされたことがある場合、インテル® VTune™ Amplifier ファイルはホストから設定されたディレクトリーにあります。例えば、`/opt/intel/vtune_amplifier_version` (root ユーザーのデフォルト) または `/tmp/vtune_amplifier_version` (非 root ユーザーのデフォルト)。ターゲット・システム・アナライザー・ファイルは、`system_analyzer/target/` サブディレクトリーにあります。

注

- すべてのケースで、ターゲット・システム・コレクター・ファイルは `target` サブディレクトリーにあり、ホスト GUI は `host` サブディレクトリーにあります。
- システム・アナライザーでサポートされるホスト OS とターゲット OS のリストは、「サポートされるプラットフォームとアプリケーション」(英語) を参照してください。
- インテル® VTune™ Amplifier 評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。

システム・アナライザーを起動する

必要条件:

- ターゲットシステムがホストシステムと異なる場合、ターゲット・データ・コレクター・ファイルをターゲットシステムにコピーします。

- Linux* ターゲットシステムで非ルートユーザーとして `gpa_router` を実行するには、次のコマンドを実行して `/proc/sys/dev/i915/perf_stream_paranoid` を 0 に設定します。

  ```
  echo 0 > /proc/sys/dev/i915/perf_stream_paranoid
  ```

1. ターゲットシステムでシステム・アナライザー・コレクターをセットアップします。
   - Windows* ターゲットシステムでは、システム・アナライザー・コレクター・ディレクトリーに移動して `gpa_router` を起動します。オプションで、ホストの IPv4 アドレスを指定して、システム・アナライザー GUI を実行し、このインスタンスへの接続を許可することができます。

     ```
     target> gpa_router.exe --ip-whitelist 10.7.158.142
     ```

   - Linux* ターゲットシステムでは、システム・アナライザーのターゲット・コレクター・ディレクトリーに移動して `gpa_router` を起動します。オプションで、ホストの IPv4 アドレスを指定して、システム・アナライザー GUI を実行し、このインスタンスへの接続を許可することができます。

     ```
     target> ./gpa_router --ip-whitelist 10.0.0.2
     ```

     システム・アナライザー GUI がターゲットシステムに接続すると、`gpa_router` は着信接続をレポートします。次に例を示します。

     ```
     target> ./gpa_router --ip-whitelist 10.0.0.2

     Start listening for new connections from port #27072
     New incoming connection established with 10.0.0.2.
     ```

注

使用する TCP ポートを指定するための構文を含む、`gpa_router` コマンドライン・オプションの詳細を確認するには、`gpa_router --help` と入力します。

2. システム・アナライザーを起動します。

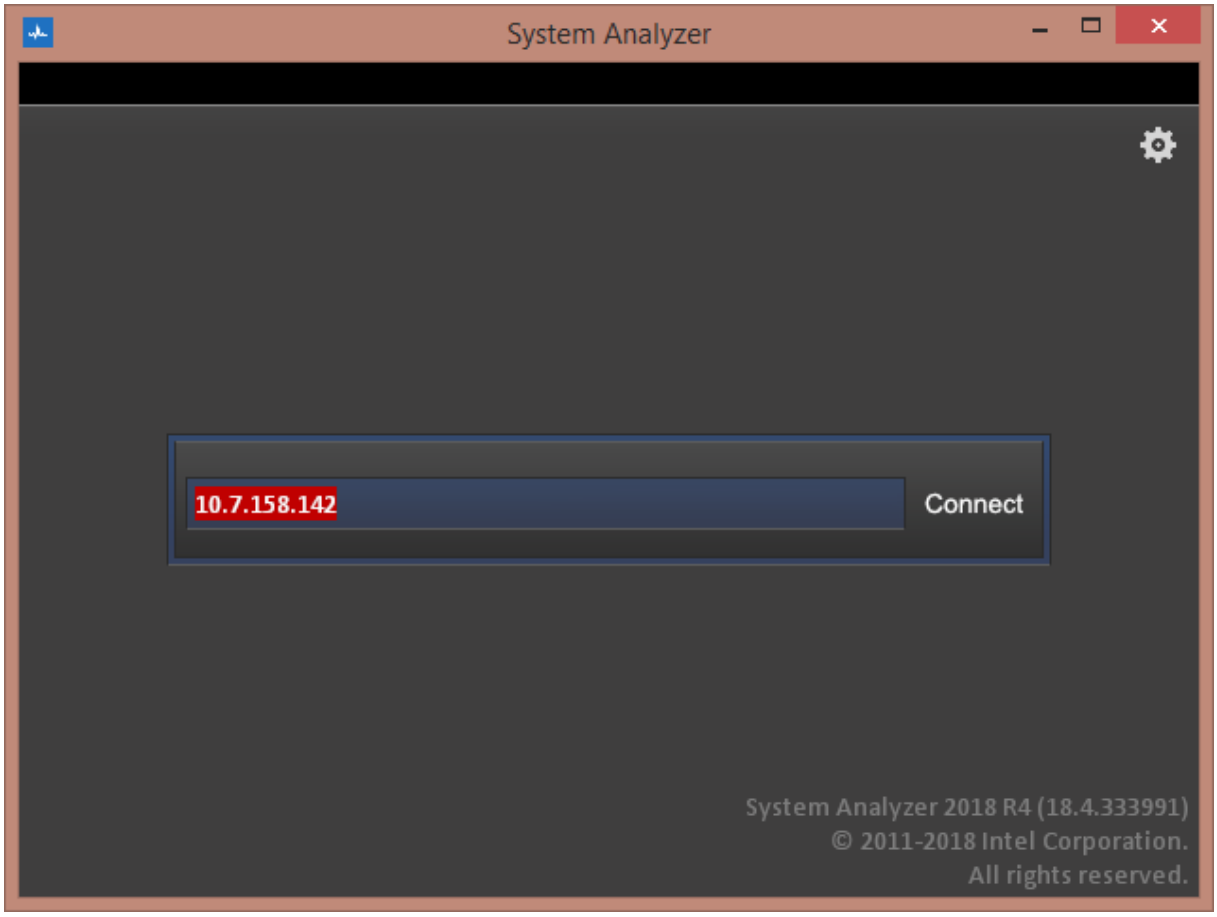
   例えば、Windows* ホストシステムでは、次のコマンドを実行します。

   ```
   host> SystemAnalyzer.exe
   ```

注

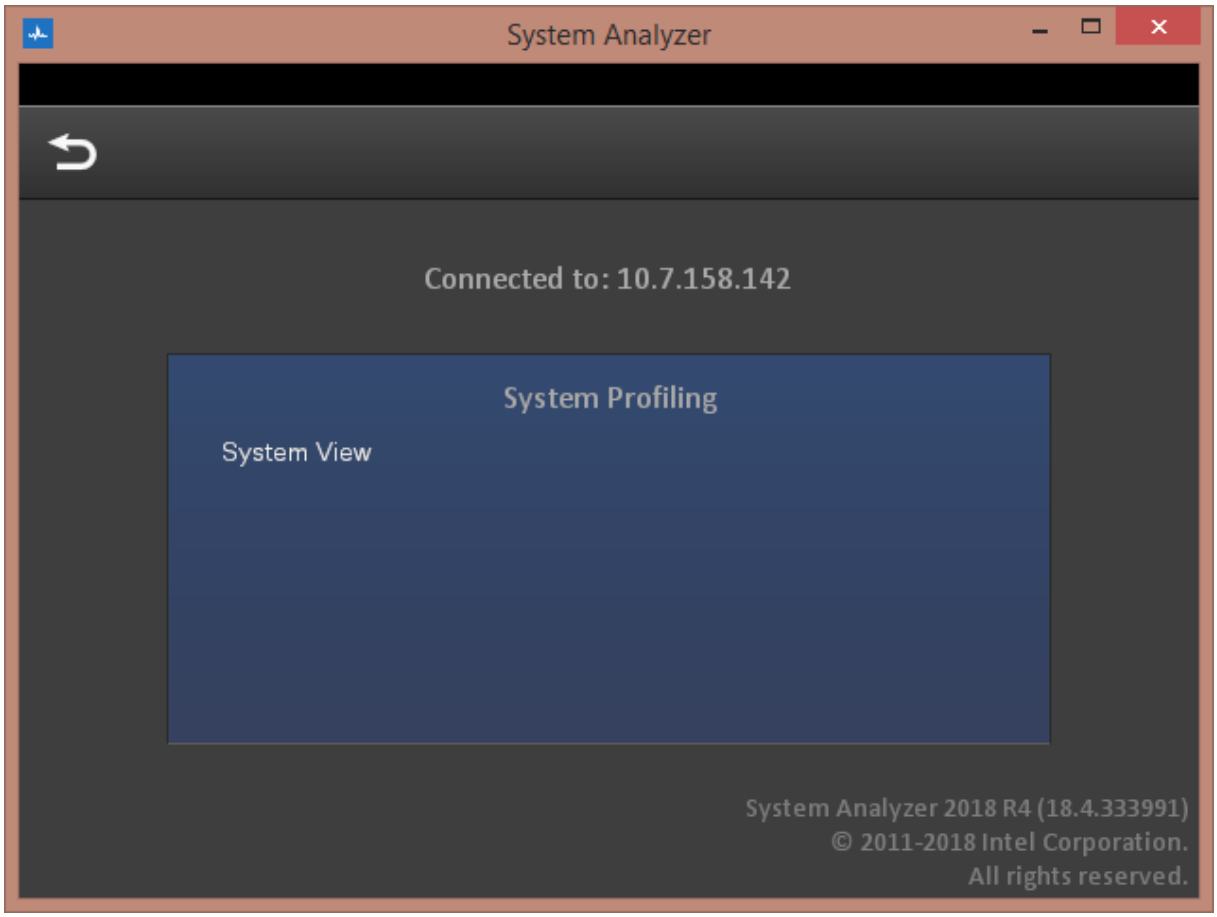
ターゲットシステムで実行中の `gpa_router` が「*Incoming connection from host <IPv4> was rejected since address is not in ip-whitelist.* (ホスト <IPv4> からの着信接続は、ip-whitelist にないため拒否されました。)」エラーを報告した場合、`--ip-whitelist` オプションでホストシステムの IPv4 アドレスを指定して `gpa_router` を再起動します。

3. `localhost` を指定するか、ターゲットシステムの名前または IPv4 アドレスを指定して、**[Connect]** ボタンをクリックします。



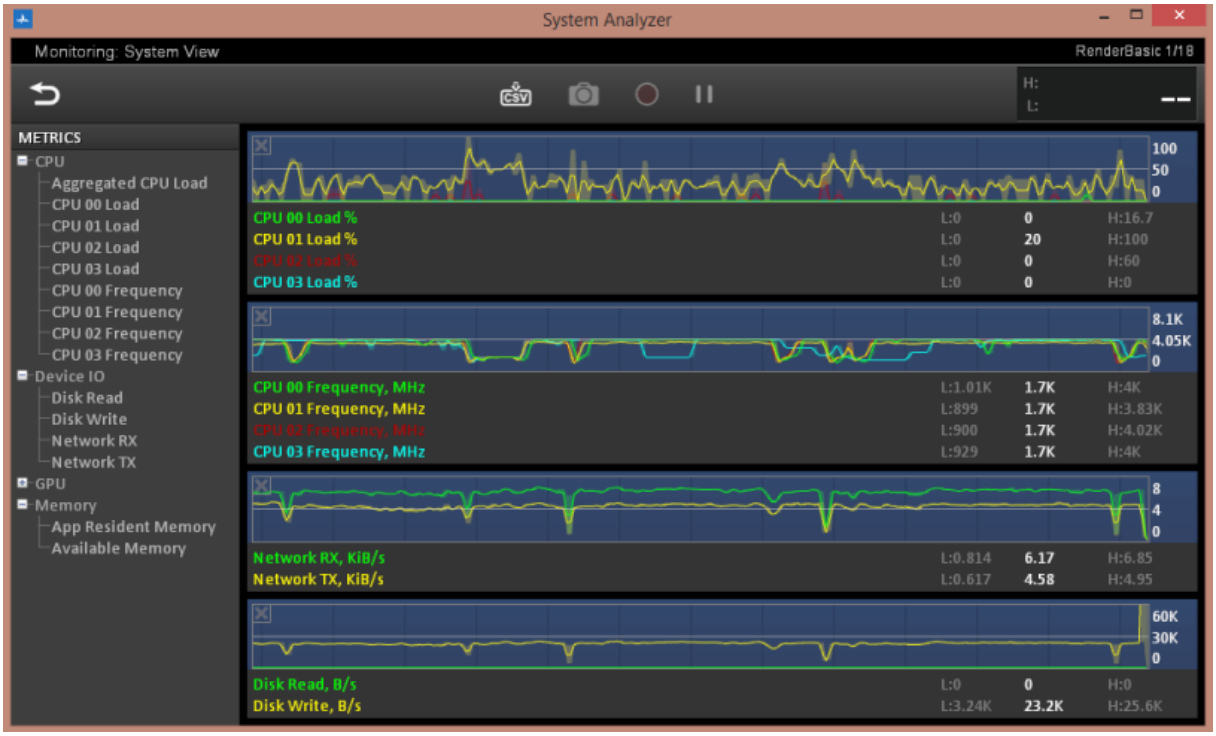
[System View] を設定する

1. システム・アナライザーがターゲットシステムに接続したら、**[System Profiling]** の下にある **[System View]** オプションを選択します。



2. **[System View]** では、マウスでグラフの表示領域へメトリックをドラッグアンドドロップして、モニタリングするメトリックを指定します。カウンターを追加する場合は、Ctrl キーを押しながらグラフの表示領域へカウンターをドラッグします。タイムラインを追加する場合は、メトリックを下部のタイムライン・グラフへドラッグします。

例えば、次に示すような CPU 使用率と周波数、ネットワーク I/O、ディスク I/O のタイムライン・グラフを設定します。



注

グラフに表示するため現在選択されているデータをカンマ区切りのファイルへエクスポートするには、システム・アナライザー GUI ウィンドウの上部にある **[CSV]** アイコンをクリックします。システム・アナライザー GUI は、エクスポート先のディレクトリーと `*.csv` ファイルの名前をタイトルバーに表示します。

詳細な解析向けに異常を特定する

システム・アナライザーが準ランダムで周期的な異常を示す場合 (例えば、ターゲットシステムのネットワーク負荷がときどき突然ゼロに近くなることがある場合)、インテル® VTune™ Amplifier の入力と出力解析やマイクロアーキテクチャー全般解析で詳しく調査し、原因を特定する必要があります。

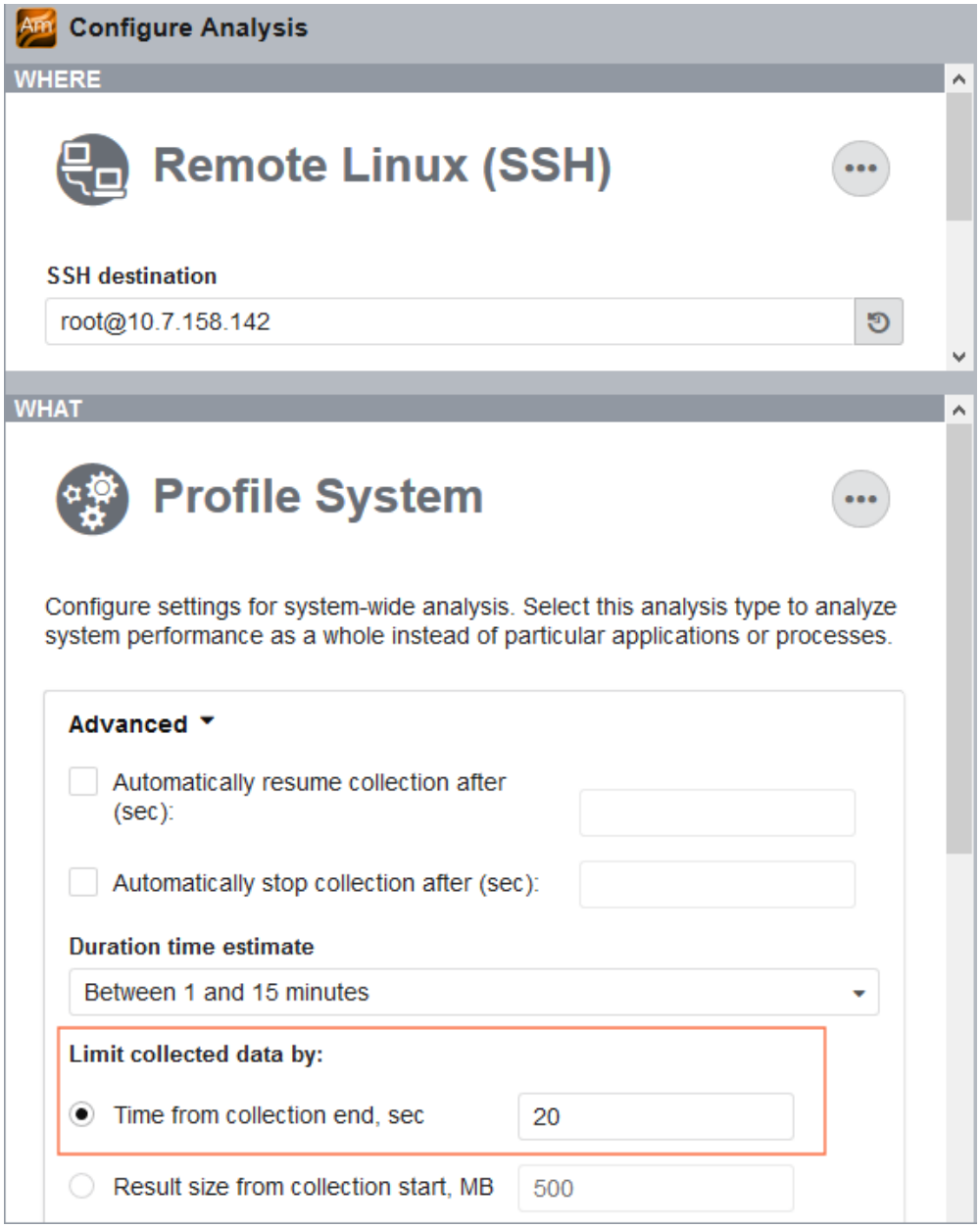
この例では、グラフのいくつかの個所で、ディスクへの書き込みとネットワーク・トラフィックが同時にやや低下しています。そして、ある個所ではディスクへの書き込みが通常の倍以上に急増し、同時にネットワーク・トラフィックがゼロに近くなっています。

この場合、ディスクへの書き込みが急増する直前にインテル® VTune™ Amplifier の詳細なデータ収集を開始し、数秒後に停止すべきです。しかしこのケースでは、ディスクへの書き込みが急増するタイミングを正確に事前予測することは困難です。そこで、リングバッファーを使用して、ディスクへの書き込みが急増しネットワーク・トラフィックが低下した周辺データの最後の数秒間のみを保存するようにデータ収集を設定すべきです。

異常を検出するようにインテル® VTune™ Amplifier を設定する

最後の 1 ~ 20 秒間のみが保存されディスクへ書き込まれるリングバッファーを使用して、システムの詳細なプロファイルを行うようにインテル® VTune™ Amplifier を設定できます。

インテル® VTune™ Amplifier GUI では、**[WHAT (何を)]** ペインの [Attach to Process (プロセスにアタッチ)]、[Profile System (システムをプロファイル)]、または [Launch Application (アプリケーションを起動)] で、**[Limit collected data by: (収集データを制限)]** にある [Time from collection end (収集終了からの時間)] オプションを使用してリングバッファーを設定できます。



コマンドラインからリングバッファーを設定し解析を実行するには、`--ring-buffer` オプションを使用します。

システム・アナライザーとインテル® VTune™ Amplifier を実行する

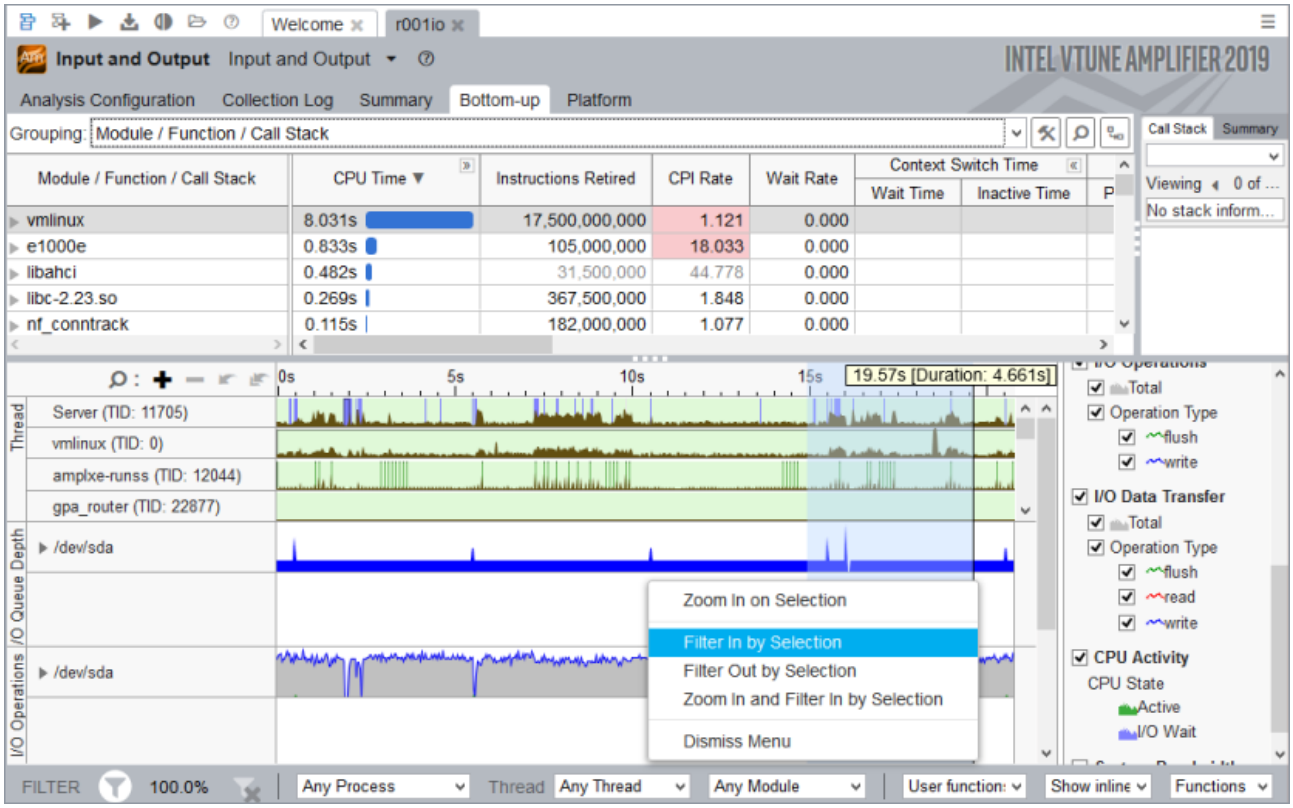
インテル® VTune™ Amplifier とシステム・アナライザーを同時に実行して、インテル® VTune™ Amplifier の解析により詳細なパフォーマンス・メトリックを取得し、システム・アナライザーによりメトリックの表示をモニタリングしてインテル® VTune™ Amplifier のデータ収集を停止するタイミングを特定します。これにより、想定外の動作が発生する直前、最中、直後のシステムを解析できるだけでなく、収集期間の短縮によりディスクへの書き込みとストレージ要件が軽減され、収集のオーバーヘッドを最小化できます。

このレシピでシステム・アナライザーによって検出されたディスク I/O 問題の場合、入力と出力解析を **[System Disk IO API (システムディスク I/O API)]** モード（デフォルト）で実行します。

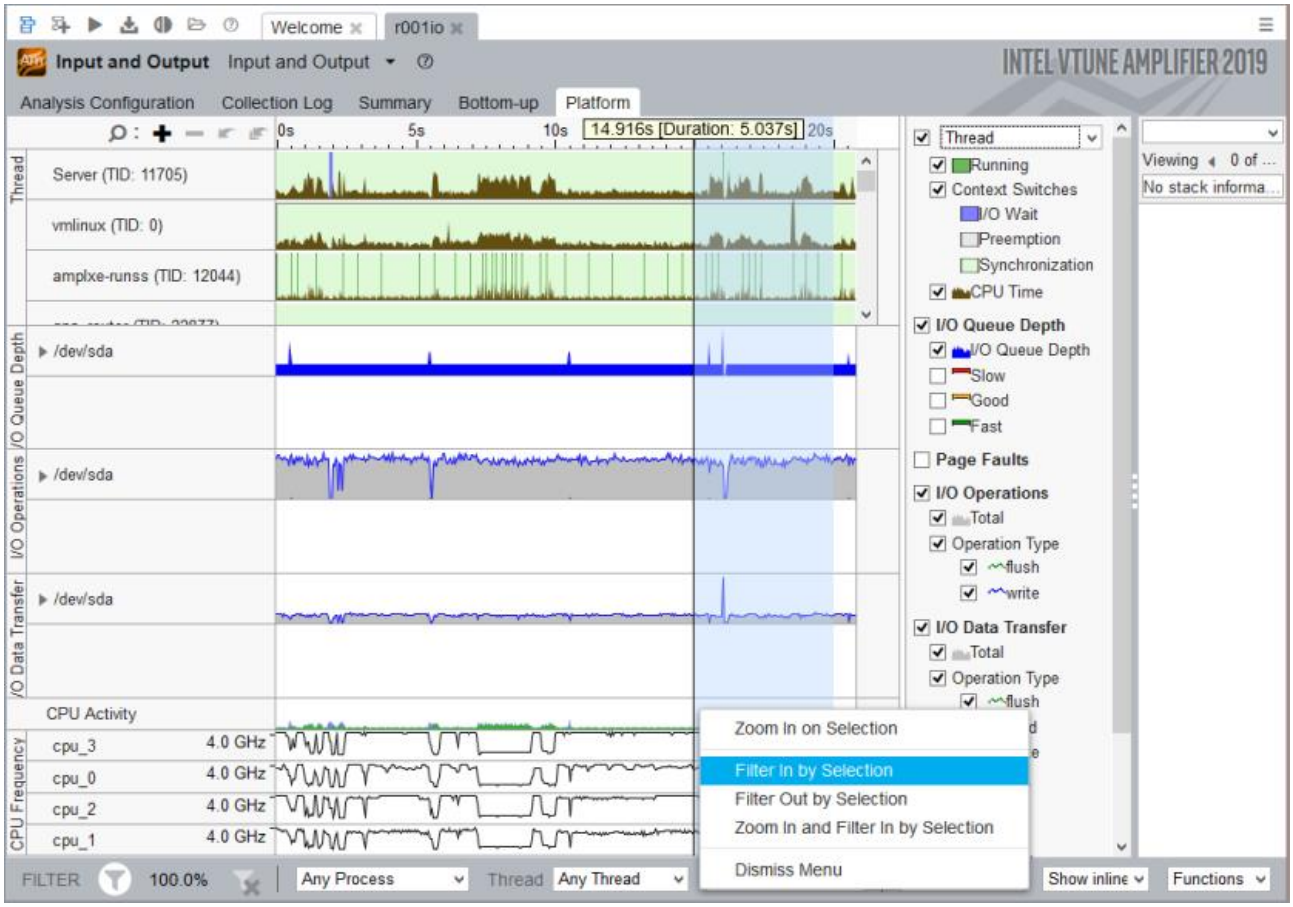
コマンドラインからインテル® VTune™ Amplifier の収集を実行するには、次のコマンドを入力します。

```
amplxe-cl -collect io -knob collect-memory-bandwidth=true -ring-buffer=20 --
duration unlimited
```

収集したデータを入力と出力ビューポイントで確認します。フィルターを使用したり、タイムラインのディスクへの書き込みが急増しネットワーク・トラフィックが低下している部分へズームインすることで、調査対象を絞り込むことができます。例えば、[Bottom-up (ボトムアップ)] ウィンドウでは、次のように表示されます。



[Platform (プラットフォーム)] ウィンドウでは、次のように表示されます。



関連情報

- 入力と出力解析 (英語)
- インテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA) for Windows* ホスト (英語)
- インテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA) for Ubuntu* ホスト (英語)

チューニング・レシピ

インテル® VTune™ プロファイラーと従来のインテル® VTune™ Amplifier で検出可能な最も一般的なパフォーマンスの問題を調査し、パフォーマンスを最適化するためのステップを提供します。

- セグメント化されたキャッシュ環境におけるキャッシュ関連のレイテンシー問題
  このレシピは、キャッシュ・アロケーション・テクノロジー (CAT) を使用して、コア間のキャッシュを分割する際にキャッシュ関連のレイテンシー問題 (キャッシュミス) を制御する方法を示します。
- フォルス・シェアリング
  このレシピは、インテル® VTune™ Amplifier の全般解析とメモリーアクセス解析を使用してメモリー依存の `linear_regression` アプリケーションをプロファイルします。
- 頻繁な DRAM アクセス
  このレシピは、インテル® VTune™ Amplifier のマイクロアーキテクチャー全般解析とメモリーアクセス解析を使用してメモリー依存の `matrix` アプリケーションをプロファイルし、頻繁な DRAM アクセスの原因を理解します。
- 低いポート使用率
  このレシピは、インテル® VTune™ Amplifier のマイクロアーキテクチャー全般解析を使用してコア依存の `matrix` アプリケーションをプロファイルし、低いポート使用率の原因を理解します。また、インテル® Advisor を使用してコンパイラーがベクトル化を行うようにします。
- ページフォールト
  このレシピは、インテル® VTune™ プロファイラーのマイクロアーキテクチャー全般、システム概要、メモリー消費解析を使用して、ページフォールトがターゲット・アプリケーションのパフォーマンスに与える影響を特定して測定する方法を説明します。
- 命令キャッシュミス
  このレシピは、インテル® VTune™ Amplifier の全般解析を使用してフロントエンド依存のアプリケーションをプロファイルし、PGO オプションを指定して ICache ミスを減らします。
- 非効率な同期
  このレシピは、スタック収集を有効にしてインテル® VTune™ Amplifier の高度な hotspot 解析を実行し、コードの非効率な同期を特定する方法を説明します。
- 非効率な TCP/IP 同期
  このレシピは、タスク収集を有効にしてインテル® VTune™ Amplifier のロックと待機解析を実行し、コードの非効率な TCP/IP 同期を特定する方法を説明します。
- I/O 問題: 高いレイテンシーと低い PCIe* 帯域幅
  このレシピは、I/O 依存のサンプル・アプリケーションに対してインテル® VTune™ Amplifier のディスク I/O 解析を実行します。そして、PCIe* デバイス向けにアフィニティーを変更して、読み取りアクセスの帯域幅が向上するように最適化します。
- OS スレッド・マイグレーション
  このレシピは、インテル® VTune™ Amplifier の高度な hotspot 解析を使用して NUMA アーキテクチャーの OS スレッド・マイグレーションを特定する手順を説明します。
- OpenMP* インバランスとスケジュール・オーバーヘッド
  このレシピは、バリアやスケジュール・オーバーヘッドのインバランスなど、OpenMP* プログラムでよくある並列ボトルネックを検出して修正する方法を説明します。
- 低いプロセッサー・コア利用率: OpenMP* シリアル時間
  このレシピは、OpenMP* で並列化されたアプリケーションのシリアル実行部分を特定し、追加の並列化の機会を見つけ、アプリケーションのスケーラビリティーを向上します。

- インテル® TBB アプリケーションのスケジュール・オーバーヘッド
  このレシピは、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) アプリケーションのスケジュール・オーバーヘッドを検出して修正する方法を説明します。
- PMDK アプリケーションのオーバーヘッド
  このレシピは、PMDK ベースのアプリケーションのメモリーアクセスのオーバーヘッドを検出して修正する方法を説明します。

セグメント化されたキャッシュ環境におけるキャッシュ関連のレイテンシー問題

このレシピは、キャッシュ・アロケーション・テクノロジー (CAT) を使用して、コア間のキャッシュを分割する際にキャッシュ関連のレイテンシー問題 (キャッシュミス) を制御する方法を示します。

コンテンツ・エキスパート: Kirill Uhanov (英語)

- 使用するもの
- 手順:
    1. メモリーアクセス解析を実行する
    2. キャッシュミスを特定する
    3. CAT を使用してコア間のキャッシュセグメントを再構成する

使用するもの

以下は、このパフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:**

  ラストレベルキャッシュ (LLC) に収まるバッファーが割り当てられたリアルタイム・アプリケーション (RTA)。RTA は、このバッファーから継続的に読み取りを行います。

  RTA は、ユーザーが即時または現在として指定した時間内で機能するプログラムです。リアルタイム・プログラムは、指定された時間内 (「デッドライン」とも呼ばれます) での応答を保証する必要があります。デッドラインをミスする理由はいくつかあります。

    - プリエンプション
    - 割り込み
    - クリティカル・コード実行中の予期しないレイテンシー

  リアルタイム・オペレーティング・システム (RTOS) は、アプリケーションを分離してプリエンプションや割り込みを回避する効率的なソリューションを提供します。しかし、CPU キャッシュミスのペナルティーなど、CPU マイクロアーキテクチャーの問題によりレイテンシーが生じる場合があります。次に RTA の例を示します。

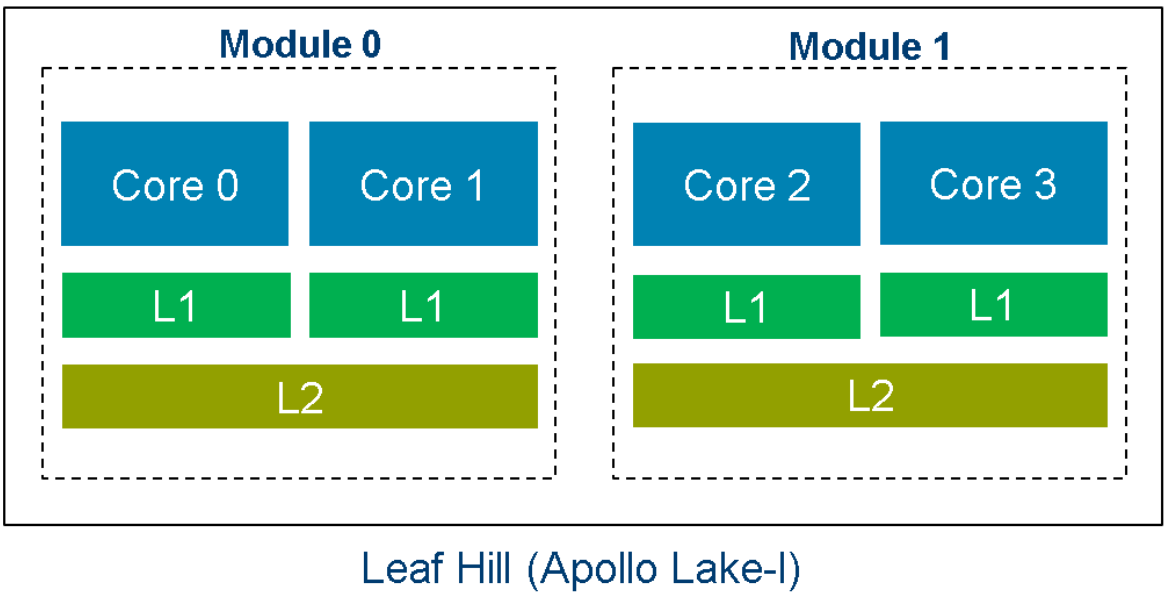
```
struct timespec sleep_timeout =
     (struct timespec) { .tv_sec = 0, .tv_nsec = 10000000 };
...
buffer=malloc(128*1024);
...
run_workload(buffer,128*1024);
void run_workload(void *start_addr,size_t size) {
    unsigned long long i,j;
    for (i=0;i<1000;i++)
    {
        nanosleep(&sleep_timeout,NULL);
```

```
            for (j=0;j<size;j+=32)
            {
                asm volatile("mov(%0,%1,1),%%eax"
                :
                :"r" (start_addr),"r"(i)
                :"%eax","memory");
            }
        }
    }
```

- **「ノイズ」アプリケーション:** stress-ng は、キャッシュをロードしてストレスをかけます。
- **ツール:** インテル® VTune™ プロファイラー - メモリーアクセス解析。`AMPLXE_EXPERIMENTAL=cat` を設定して、**Cache Availability (preview) (キャッシュ利用率 (プレビュー))** 機能を有効にします。

  注

  - バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
  - インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
  - 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
    - インテル® VTune™ プロファイラー製品ページ
    - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)
- **オペレーティング・システム:** Linux*。
- **ハードウェア:** Intel Atom® プロセッサー E3900 シリーズ (開発コード名 Apollo Lake) Leaf Hill、L2 CAT 機能有効。



Leaf Hill (Apollo Lake-I)

メモリーアクセス解析を実行する

「ノイズ」環境で RTA を実行してパフォーマンスの低下に気付いたときは、メモリーアクセス解析を実行して CPU キャッシュミスのペナルティーなどのマイクロアーキテクチャーの問題を詳しく調べます。

次の例では、RTA はコア 3 にピニングされ、`stress-ng` はコア 2 にピニングされます。

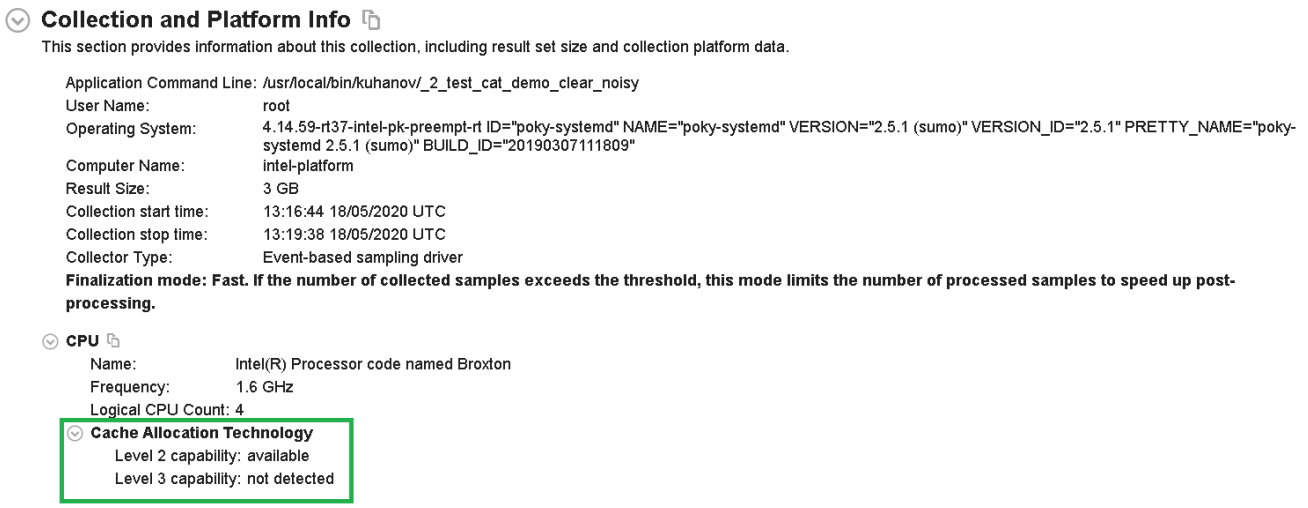
```
stress-ng -C 10 --cache-level 2 --taskset 2 --aggressive -v --metrics-brief
```

両方のコアはモジュール 1 に属していて、L2 LLC を共有します。

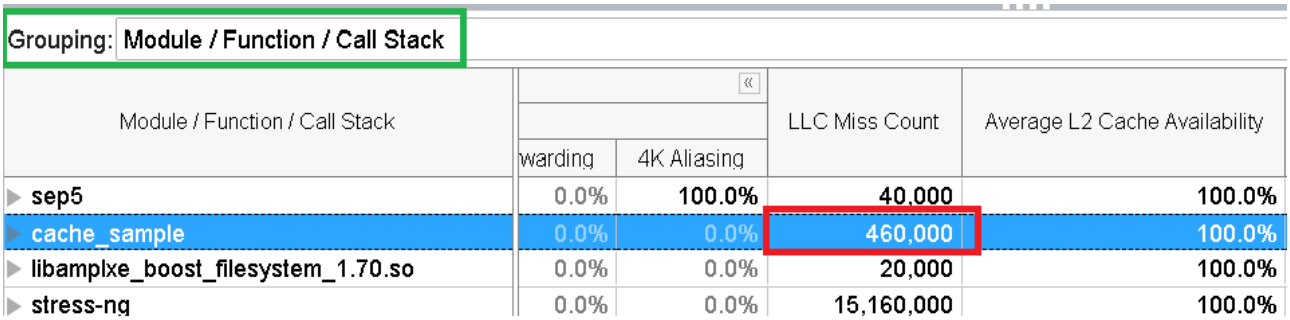
1. `AMPLXE_EXPERIMENTAL=cat` を設定して、**Cache Availability (preview) (キャッシュ利用率 (プレビュー))** 機能を有効にします。
2. インテル® VTune™ プロファイラー GUI を開きます。
3. 新しいプロジェクトを作成します。**[Create a Project (プロジェクトの作成)]** ダイアログボックスが表示されます。
4. プロジェクトの名前と場所を指定したら、**[Create Project (プロジェクトの作成)]** ボタンをクリックします。**[Configure Analysis (解析の設定)]** ダイアログボックスが開きます。
5. **[WHERE (どこを)]** ペインで、解析のターゲットシステムとして **[Remote Linux (SSH) (リモート Linux* (SSH))]** を選択します。
6. Linux ターゲットへのパスワードなしの SSH アクセスを有効にします (英語)。
7. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** を選択して、解析するターゲット・アプリケーションを指定します。
8. **[HOW (どのように)]** ペインで、解析ヘッダーをクリックして解析ツリーから **[Memory Access (メモリーアクセス)]** 解析を選択します。
9. **[Analyze cache allocation (キャッシュ割り当ての解析)]** を設定して、コアごとのキャッシュセグメントの使用状況を解析します。
10. **[Start (開始)]** ボタンをクリックして、解析を実行します。

キャッシュミスを特定する

インテル® VTune™ プロファイラーが解析を完了したら、**[Summary (サマリー)]** ペインの **[Collection and Platform Info (収集とプラットフォーム情報)]** セクションを確認します。このセクションで、キャッシュ・アロケーション・テクノロジー (CAT) の L2 機能と L3 機能の情報を確認します。この例では、ハードウェアは LLC (L2 キャッシュ) の分割操作をサポートしています。



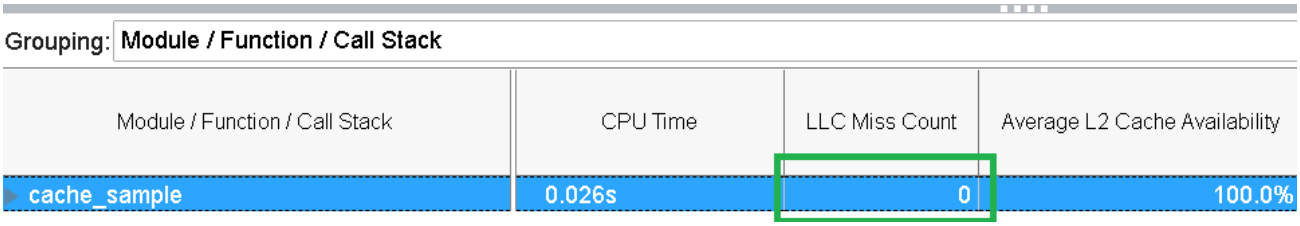
次に、**[Bottom-up (ボトムアップ)]** ペインに切り替えます。**[Module / Function / Call Stack (モジュール / 関数 / コールスタック)]** グループ化を選択して、`cache_sample` モジュールの LLC Miss Count (LLC ミスカウント) の値を確認します。



Grouping: Module / Function / Call Stack

Module / Function / Call Stack	warding	4K Aliasing	LLC Miss Count	Average L2 Cache Availability
sep5	0.0%	100.0%	40,000	100.0%
cache_sample	0.0%	0.0%	460,000	100.0%
libamplxe_boost_filesystem_1.70.so	0.0%	0.0%	20,000	100.0%
stress-ng	0.0%	0.0%	15,160,000	100.0%

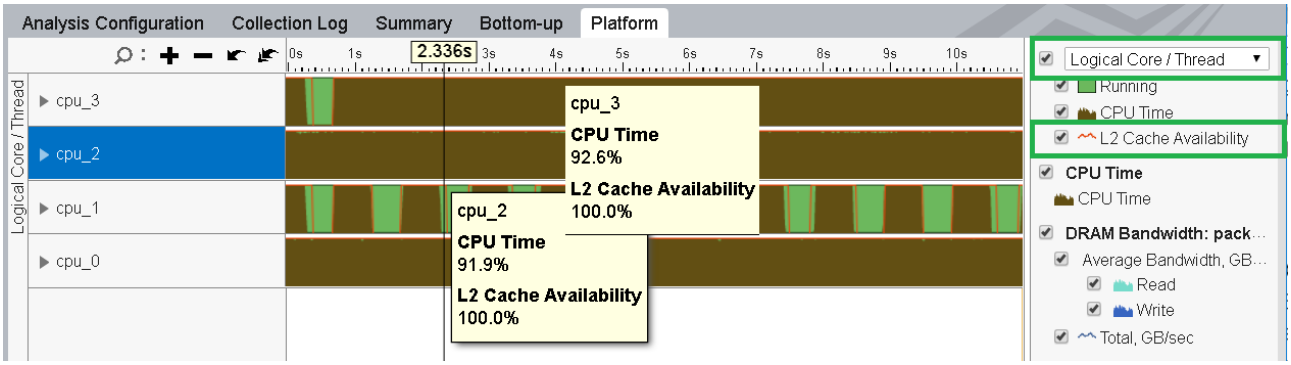
キャッシュミスの数が多くなっています。しかし、stress-ng がない場合、キャッシュミスはゼロです。



Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time	LLC Miss Count	Average L2 Cache Availability
cache_sample	0.026s	0	100.0%

次に、**[Platform (プラットフォーム)]** ペインを開きます。結果を **[Logical Core/Thread (論理コア/スレッド)]** でグループ化して、**[L2 Cache Availability (L2 キャッシュ利用率)]** チェックボックスをオンにします。



タイムラインは、L2 キャッシュ利用率が 100% であることを示しています。これは、**cpu_2** 上でターゲット・アプリケーションが存在する間、L2 キャッシュのすべてのセグメントが利用可能だったことを意味します。しかし、**cpu_3** の stress-ng も、存在する間 L2 キャッシュを共有していて、L2 キャッシュ利用率が 100% になっています。これは、`cache_sample` モジュール内で膨大な量のキャッシュミスが発生する原因となります。

そこで、CAT を使用してコア間のキャッシュセグメントを分割して、各コアがセグメントを排他的に使用できるようにします。

CAT を使用してコア間のキャッシュセグメントを再構成する

キャッシュセグメントを分割して、RTA と stress-ng に割り当てます。CAT を使用して、スレッド、アプリケーション、仮想マシン、コンテナーで使用されるキャッシュの量を制御するようにソフトウェアをプログラムできます。キャッシュセグメントを分離して、リソースの共有問題に対処できます。次の 2 つの方法があります。

- MSR を直接設定します。詳細は、『Intel® 64 and IA-32 Architectures Software Developer's Manual、Volume 3 (3A, 3B, 3C & 3D): System Programming Guide』(英語) のセクション 17.19 を参照してください。
- CPU リソース割り当てのカーネル・インターフェイス、Resource Control (resctrl) (英語) を使用します。

この例では、resctrl を使用して次のように割り当てます。

- 1 つのキャッシュセグメントを **cpu_3** に割り当てます。
- 7 つのキャッシュセグメントを **cpu_2** に割り当てます。

```
#set '00000001' Capacity Bit Mask for CORE 3
mkdir /sys/fs/resctrl/clos0
echo 8 > /sys/fs/resctrl/clos0/cpus
echo 'L2:1=1' > /sys/fs/resctrl/clos0/schemata
#set '11111110' CBM for rest CORE
echo 'L2:1=fe' > /sys/fs/resctrl/schemata
```

メモリーアクセス解析を再度実行する収集が完了したら、**[Bottom-up (ボトムアップ)]** ペインを確認します。

Grouping: **Module / Function / Call Stack**

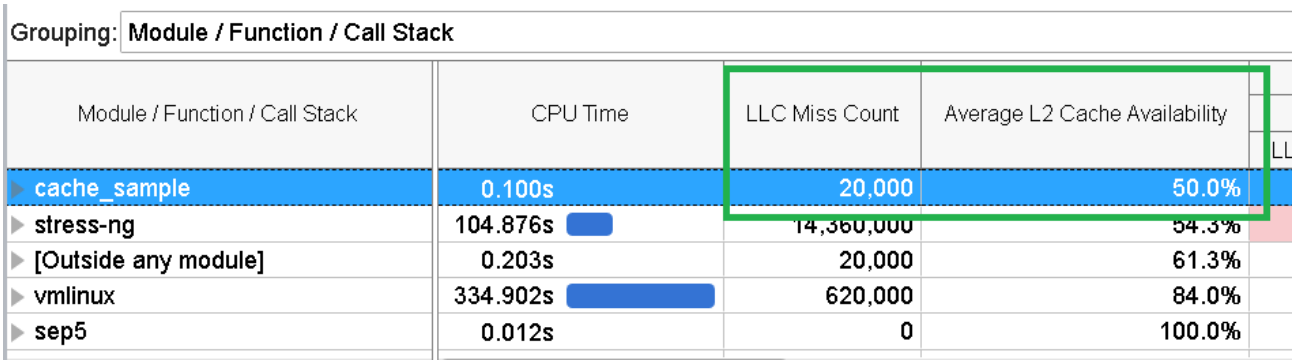
Module / Function / Call Stack	CPU Time	Average L2 Cache Availability	LLC Miss Count ▼
cache_sample	0.409s	12.5%	380,000
sep5	0.014s	86.5%	20,000
ld-2.27.so	0.016s	100.0%	0
socperf3	0.004s	100.0%	0
[vdso]	0.004s	100.0%	0
libsystemd-shared-237.so	0.011s	100.0%	0
sep	0.002s	100.0%	0

キャッシュの一部 (12.5%) が排他的に利用できるようになりました。しかし、キャッシュミスの数にはまだ改善の余地があります。

アプリケーションに割り当てるキャッシュを増やして、再度解析を実行してみましょう。キャッシュ割り当てを 50% または 4 セグメントに増やします。

```
#set '00001111' Capacity Bit Mask for CORE 3
mkdir /sys/fs/resctrl/clos0
echo 8 > /sys/fs/resctrl/clos0/cpus
echo 'L2:1=f' > /sys/fs/resctrl/clos0/schemata
#set '11110000' CBM for rest CORE
echo 'L2:1=f0' > /sys/fs/resctrl/schemata
```

[Bottom-up (ボトムアップ)] ペインを確認すると、キャッシュ割り当てを増やしたことでキャッシュミスの数が大幅に低下していることが分かります。しかし、この結果は、利用可能なキャッシュ全体の半分をアプリケーションに排他的に割り当てるという大きな代償を払うことにより得られたものです。



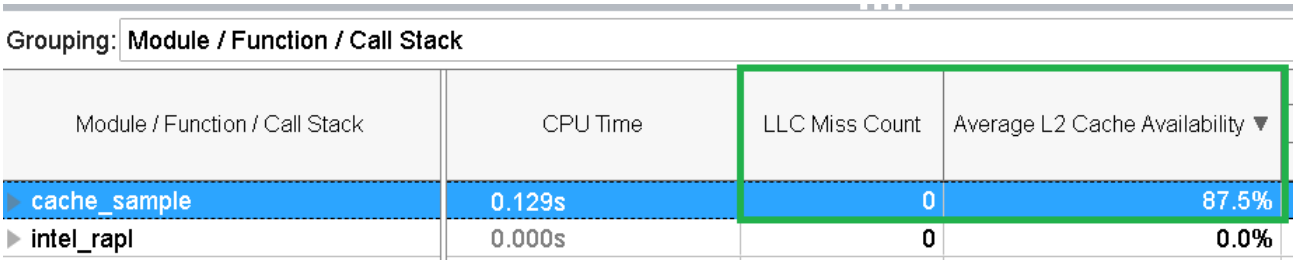
Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time	LLC Miss Count	Average L2 Cache Availability
cache_sample	0.100s	20,000	50.0%
stress-ng	104.876s	14,360,000	54.3%
[Outside any module]	0.203s	20,000	61.3%
vmlinux	334.902s	620,000	84.0%
sep5	0.012s	0	100.0%

別の解決策として、「擬似ロック」を使用する方法があります。擬似ロックは、同じキャッシュを使用しようとするほかのプロセスによるキャッシュデータの退避を防ぐのに役立ちます。RTA は、重要なデータをキャッシュの特別なセグメントに割り当てて、別のスレッド、プロセス、コアによる使用からデータを保護することができます。セグメントが非表示の間も、セグメントのデータにアクセスできます。

```
#Create the pseudo-locked region with 1 cache segment
mkdir /sys/fs/resctrl/demolock
echo pseudo-locksetup > /sys/fs/resctrl/demolock/mode
echo 'L2:1=0X1' > /sys/fs/resctrl/demolock/schemata
cat /sys/fs/resctrl/demolock/mode
pseudo-locked

struct timespec sleep_timeout =
     (struct timespec) { .tv_sec = 0, .tv_nsec = 10000000 };
...
/* buffer=malloc(128*1024); */
open("/dev/pseudo_lock/demolock",0_RDWR);
buffer=mmap(0,128*1024,PROT_READ|PROT_WRITE,MAP_SHARED,dev_fd,0);
...
run_workload(buffer,128*1024);
void run_workload(void *start_addr,size_t size) {
    unsigned long long i,j;
    for (i=0;i<1000;i++)
    {
        nanosleep(&sleep_timeout,NULL);
        for (j=0;j<size;j+=32)
        {
            asm volatile("mov(%0,%1,1),%%eax"
            :
            :"r" (start_addr),"r"(i)
            :"%eax","memory");
        }
    }
}
```

解析を再度実行します。**[Bottom-up (ボトムアップ)]** ペインを開いて結果を確認します。



システムの 1 つのセグメントをロックするだけで、「すべての」キャッシュミスを解決するのに役立ちました。

キャッシュ・アロケーション・テクノロジーは、サイズに関係なく、(メモリーアクセスにより引き起こされる) 小さなレイテンシーが重要なリアルタイム環境やワークロードで非常に役立ちます。このレシピで説明したように、キャッシュミスがゼロになるまで、段階的にキャッシュセグメントを割り当ててください。

注
このレシピの情報は、アナライザー・デベロッパー・フォーラム (英語) を参照してください。

関連情報

マイクロアーキテクチャー全般解析
メモリーアクセス解析
キャッシュ・アロケーション・テクノロジー (CAT) の概要 (英語)

フォルス・シェアリング

このレシピは、インテル® VTune™ Amplifier の全般解析とメモリーアクセス解析を使用してメモリー依存の `linear_regression` アプリケーションをプロファイルします。

コンテンツ・エキスパート: Dmitry Ryabtsev (英語)

注

全般解析は、インテル® VTune™ Amplifier 2019 でマイクロアーキテクチャー全般解析に改名されました。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** `linear_regression`。`linear_regression.tgz` サンプルパッケージは、製品の `<install-dir>/samples/en/C++` ディレクトリーに含まれています。https://github.com/kozyraki/phoenix/tree/master/sample_apps/linear_regression (英語) からダウンロードすることもできます。
- **パフォーマンス解析ツール:**
    - インテル® VTune™ Amplifier 2018: 全般解析、メモリーアクセス解析

注

    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Ubuntu* 16.04 64 ビット
- **CPU:** インテル® Core™ i7-6700K プロセッサー

全般解析を実行する

サンプル・アプリケーションの潜在的なパフォーマンス・ボトルネックを理解するため、まず、インテル® VTune™ Amplifier の全般解析を実行します。

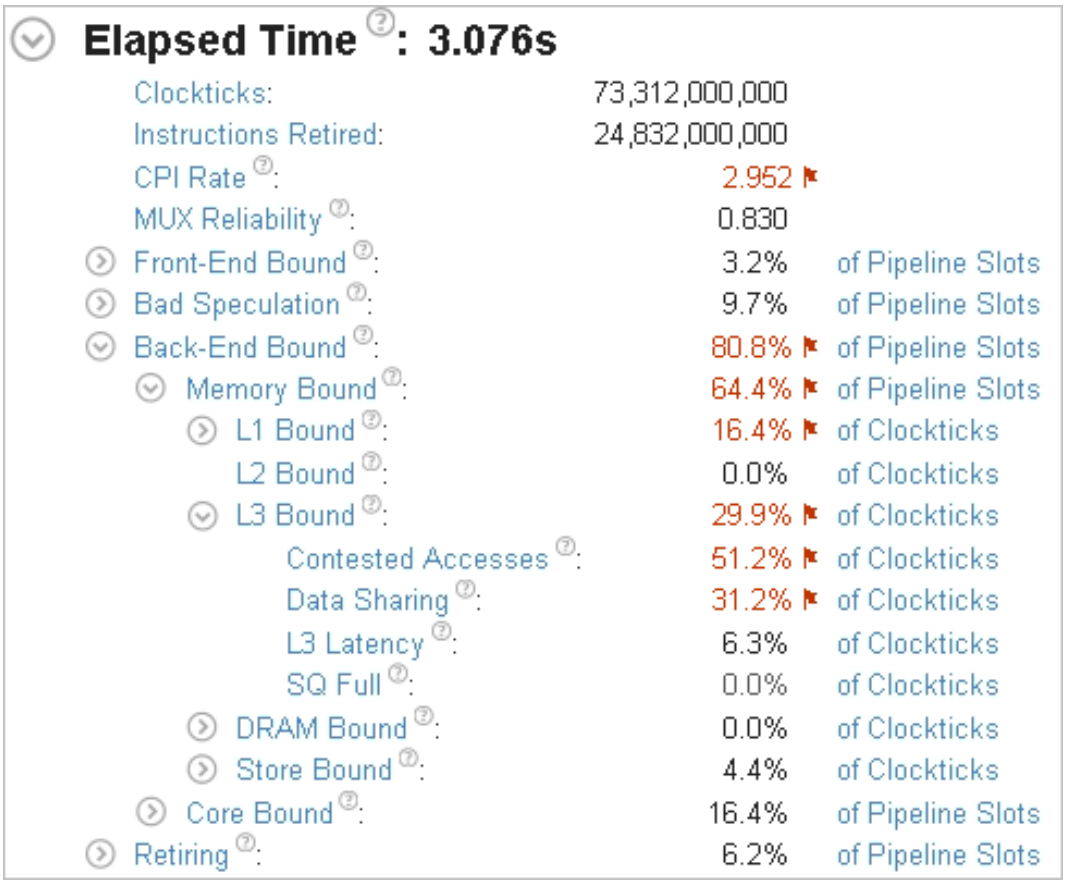
1. ツールバーの **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `linear_regression`) を指定します。
2. **[Analysis Target (解析ターゲット)]** ウィンドウで、ホストベースの解析として **[local host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、右ペインで解析するアプリケーションを指定します。
4. 右の **[Choose Analysis (解析の選択)]** ボタンをクリックし、**[Microarchitecture Analysis (マイクロアーキテクチャー解析)]** > **[General Exploration (全般)]** を選択して、**[Start (開始)]** をクリックします。

   インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

ボトルネックを特定する

ハードウェア・メトリックごとのアプリケーション・レベルの統計が表示される **[Summary (サマリー)]** ビューから始めます。

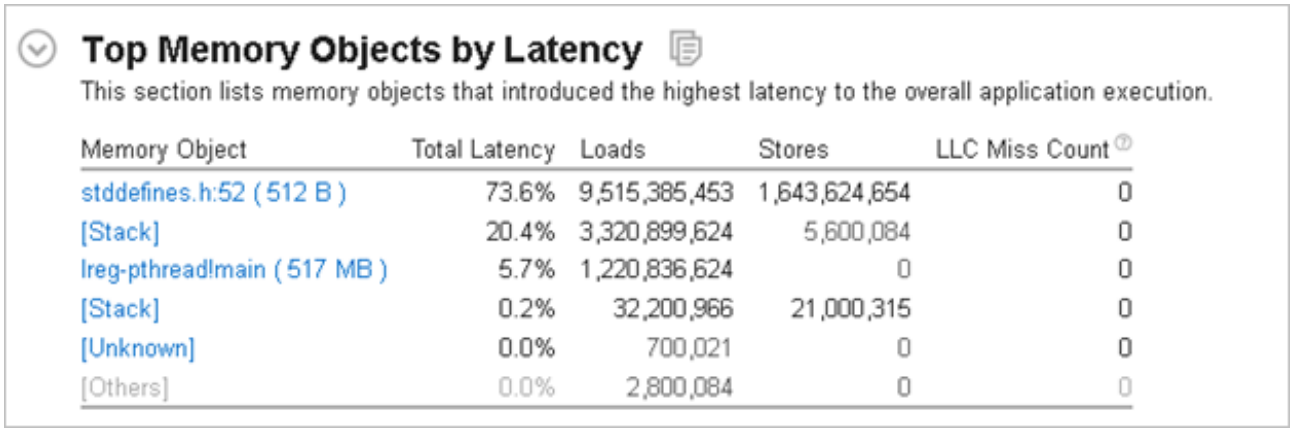
一般に、パフォーマンス解析では、**ベースライン**を作成して以降の最適化を測定することを推奨します。このケースでは、アプリケーションの [Elapsed Time (経過時間)] をベースラインとして使用します。



サマリーメトリックから、メモリーアクセスの競合によりアプリケーションのパフォーマンスが制限されていることが分かります。

競合するデータ構造を見つける

[Contested Accesses (アクセス競合)] メトリックの値が高い原因を調べるため、**[Analyze dynamic memory objects (動的メモリー・オブジェクトを解析)]** オプションを有効にしてメモリーアクセス解析を実行します。この解析は、競合問題の原因になっているデータ構造へのアクセスを見つけるのに役立ちます。

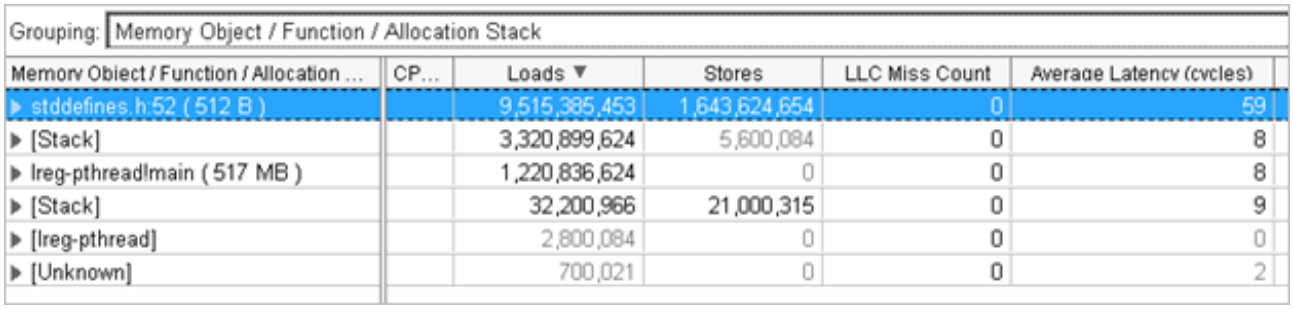


Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
stddefines.h:52 (512 B)	73.6%	9,515,385,453	1,643,624,654	0
[Stack]	20.4%	3,320,899,624	5,600,084	0
lreg-pthread!main (517 MB)	5.7%	1,220,836,624	0	0
[Stack]	0.2%	32,200,966	21,000,315	0
[Unknown]	0.0%	700,021	0	0
[Others]	0.0%	2,800,084	0	0

[Summary (サマリー)] ビューから、ファイル `stddefines.h` の行 52 のメモリー割り当てデータ・オブジェクトでアプリケーション実行のレイテンシーが高くなっていることが分かります。割り当てのサイズは 512 バイトと非常に小さいため、L1 キャッシュに完全に収まるはずです。詳細を確認するため、このオブジェクトをクリックして [Bottom-up (ボトムアップ)] ビューに切り替えます。

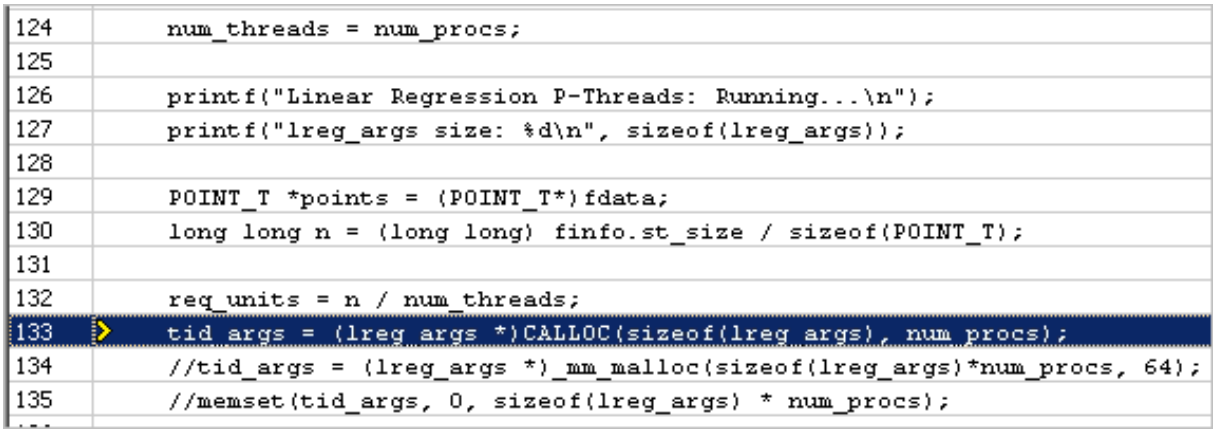


Grouping: Memory Object / Function / Allocation Stack

Memory Object / Function / Allocation ...	CP...	Loads ▼	Stores	LLC Miss Count	Average Latency (cycles)
stddefines.h:52 (512 B)		9,515,385,453	1,643,624,654	0	59
[Stack]		3,320,899,624	5,600,084	0	8
lreg-pthread!main (517 MB)		1,220,836,624	0	0	8
[Stack]		32,200,966	21,000,315	0	9
[lreg-pthread]		2,800,084	0	0	0
[Unknown]		700,021	0	0	2

このオブジェクトの平均アクセス・レイテンシーは 59 サイクルと、L1 キャッシュ上にあると予想されるメモリーサイズとしては非常に高い値になっています。これがアクセス競合パフォーマンス問題の原因になっている可能性があります。

グリッドの **stddefines.h:52 (512B)** メモリー・オブジェクトを展開して割り当てスタックを表示します。割り当てスタックをダブルクリックして [Source (ソース)] ビューを開きます。オブジェクトが割り当てられているコード行がハイライトされます。



```
124     num_threads = num_procs;
125
126     printf("Linear Regression P-Threads: Running...\n");
127     printf("lreg_args size: %d\n", sizeof(lreg_args));
128
129     POINT_T *points = (POINT_T*)fdata;
130     long long n = (long long) finfo.st_size / sizeof(POINT_T);
131
132     req_units = n / num_threads;
133     tid_args = (lreg_args *)CALLOC(sizeof(lreg_args), num_procs);
134     //tid_args = (lreg_args *)_mm_malloc(sizeof(lreg_args)*num_procs, 64);
135     //memset(tid_args, 0, sizeof(lreg_args) * num_procs);
```

`lreg_args` の内容を次に示します。

```
typedef struct
{
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

次のように、`lreg_args` 配列にアクセスしているコードをスレッド化します。

```
// ADD Up RESULTS
for (i = 0; i < args->num_elems; i++)
{
    //Compute SX, SY, SYY, SXX, SXY
    args->SX  += args->points[i].x;
    args->SXX += args->points[i].x*args->points[i].x;
    args->SY  += args->points[i].y;
    args->SYY += args->points[i].y*args->points[i].y;
    args->SXY += args->points[i].x*args->points[i].y;
}
```

各スレッドは別々に配列の要素にアクセスしているため、フォルス・シェアリング問題が考えられます。

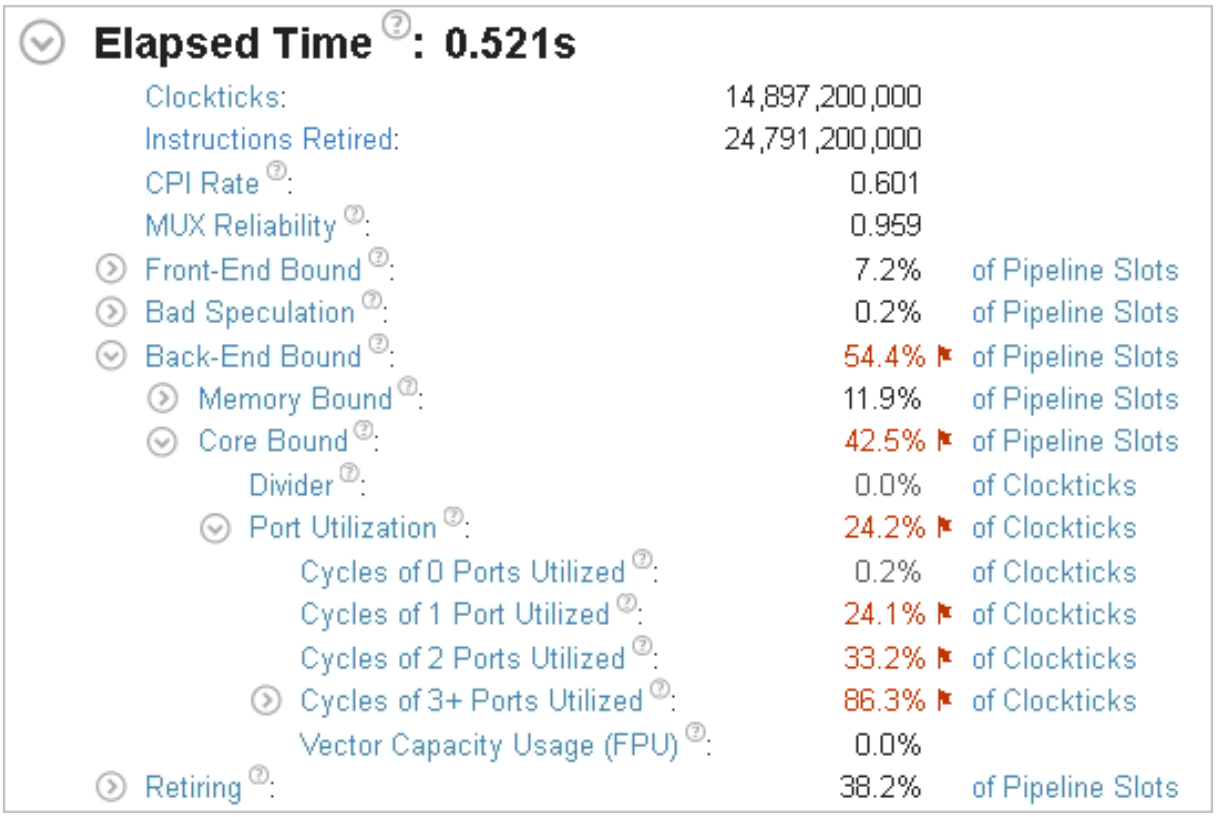
サンプルの `lreg_args` 構造のサイズは 64 バイトで、キャッシュラインのサイズと一致しています。しかし、これらの構造の配列を割り当てるときに、この配列が 64 バイトでアライメントされる保証はありません。その結果、配列要素がキャッシュライン境界を超えて、意図しない競合問題 (フォルス・シェアリング) が発生することがあります。

フォルス・シェアリング問題を修正する

このフォルス・シェアリング問題を修正するため、メモリーを 64 バイト・アライメントで割り当てる `_mm_malloc` 関数に変更します。

```
50   inline void * CALLOC(size_t num, size_t size)
51   {
52       void * temp =  mm_malloc(num*size, 64);
53       assert(temp);
54       memset(temp, 0, num*size);
55       return temp;
56   }
```

再コンパイルしてインテル® VTune™ プロファイラーのアプリケーション解析を再度実行すると、結果は次のようになりました。



Elapsed Time (経過時間) は 0.5 秒になり、オリジナルの 3 秒からパフォーマンスが大幅に向上しました。メモリー依存のボトルネックが解消し、フォルス・シェアリング問題が修正されました。

注

このレシピの情報は、デベロッパー・フォーラム (英語) を参照してください。

関連情報

- マイクロアーキテクチャー全般解析 (英語)
- メモリーアクセス解析 (英語)
- トップダウン・マイクロアーキテクチャー解析法

頻繁な DRAM アクセス

このレシピは、インテル® VTune™ Amplifier のマイクロアーキテクチャー全般解析とメモリーアクセス解析を使用してメモリー依存の `matrix` アプリケーションをプロファイルし、頻繁な DRAM アクセスの原因を理解します。

コンテンツ・エキスパート: Dmitry Ryabtsev (英語)

- 使用するもの
- 手順:
    1. ベースラインを作成する
    2. マイクロアーキテクチャー全般解析を実行する
    3. ハードウェアの hotspot を特定する
    4. メモリーアクセス解析を実行する
    5. ホットなメモリーアクセスを特定する
    6. ループ交換の最適化を適用する

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** 2048x2048 サイズの 2 つの行列を乗算する行列乗算サンプル (要素は double 型)。`matrix_vtune_amp_axe.tgz` サンプルパッケージは、製品の `<install-dir>/samples/en/C++` ディレクトリーに含まれています。https://software.intel.com/en-us/product-code-samples (英語) からダウンロードすることもできます。
- **パフォーマンス解析ツール:**
    - インテル® VTune™ Amplifier 2019: マイクロアーキテクチャー全般解析 (旧: 全般解析)、メモリーアクセス解析

注

    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Ubuntu* 16.04 64 ビット
- **CPU:** インテル® Core™ i7-6700K プロセッサー

ベースラインを作成する

サンプルコードの初期バージョンは、次のコードにより、メインカーネルに単純な乗算アルゴリズムを実装しています。

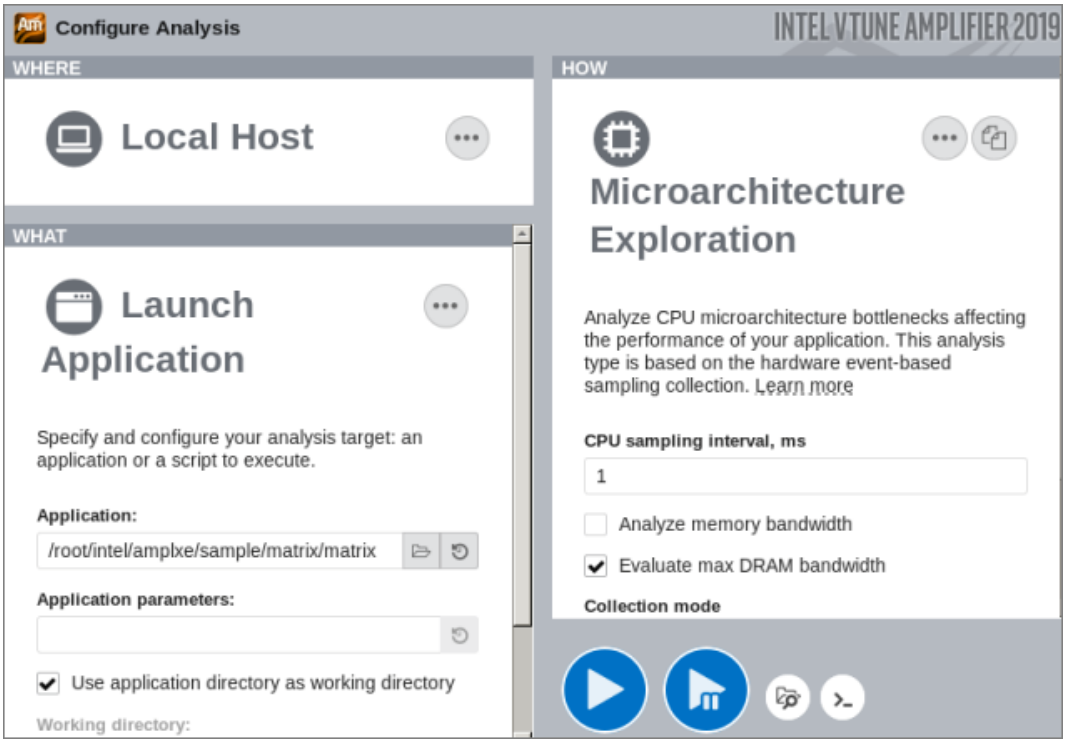
```
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE v[][NUM],
TYPE c[][NUM], TYPE t[][NUM])
{
    int i,j,k;

    // Naive implementation
    for(i=tidx; i<msize; i=i+numt) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

コンパイルしたアプリケーションの実行には約 22 秒かかります。これが、以降の最適化で使用するパフォーマンスの**ベースライン**となります。

マイクロアーキテクチャー全般解析を実行する

サンプル・アプリケーションの潜在的なパフォーマンス・ボトルネックを理解するため、まず、インテル® VTune™ プロファイラーのマイクロアーキテクチャー全般解析を実行します。

1. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `matrix`) を指定します。
2. **[Configure Analysis (解析の設定)]** ウィンドウの **[WHERE (どこを)]** ペインで、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、解析するアプリケーションを指定します。
4. **[HOW (どのように)]** ペインで、[...] ボタンをクリックして **[Microarchitecture (マイクロアーキテクチャー)]** グループから **[Microarchitecture Exploration (マイクロアーキテクチャー全般)]** 解析を選択します。

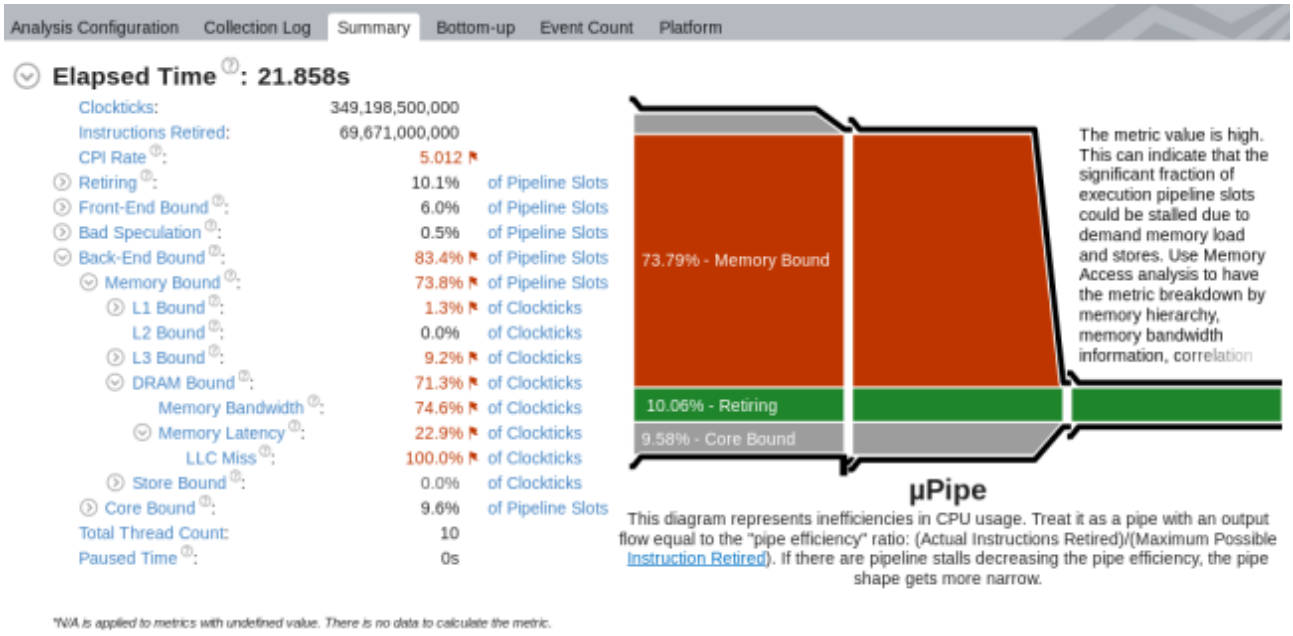


5. **[Start (開始)]** ボタンをクリックします。

   インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

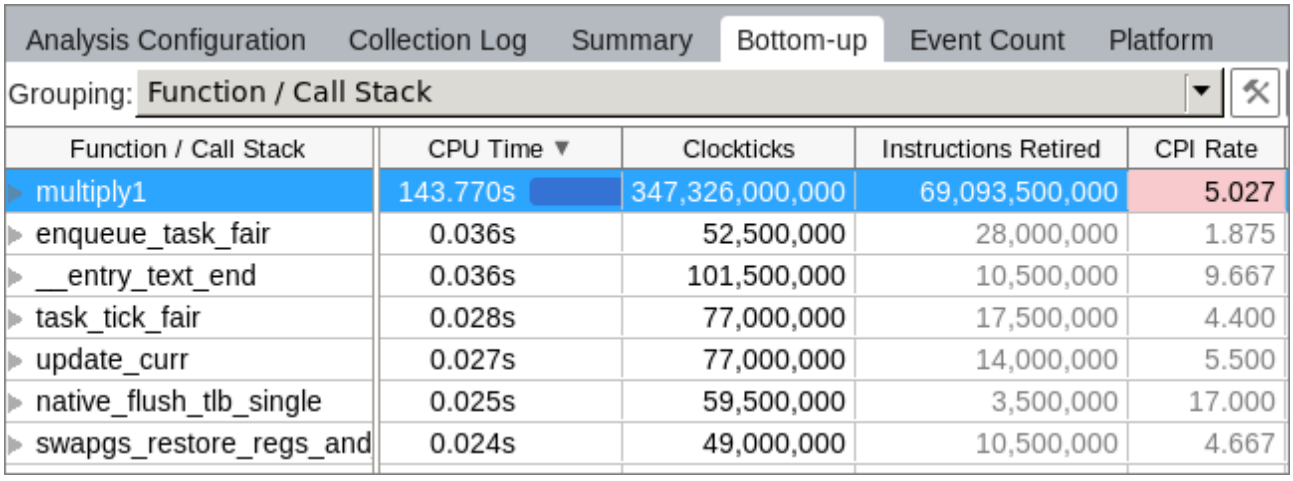
ハードウェアの hotspot を特定する

マイクロアーキテクチャー全般解析を実行すると、コードの主要なボトルネックを確認できます。解析したアプリケーションの CPU マイクロアーキテクチャーの効率と CPU パイプライン・ストールが表示される **[Summary (サマリー)]** ビューの **[µPipe (µ パイプ)]** から解析を始めます。以下の **[µPipe (µ パイプ)]** では、出力パイプのフローが非常に狭いため、アプリケーションのパフォーマンスを向上するには、**[Retiring (リタイア)]** メトリックの値を増やす必要があります。このパイプの主な問題は、**[Memory Bound (メモリー依存)]** メトリックの値です。

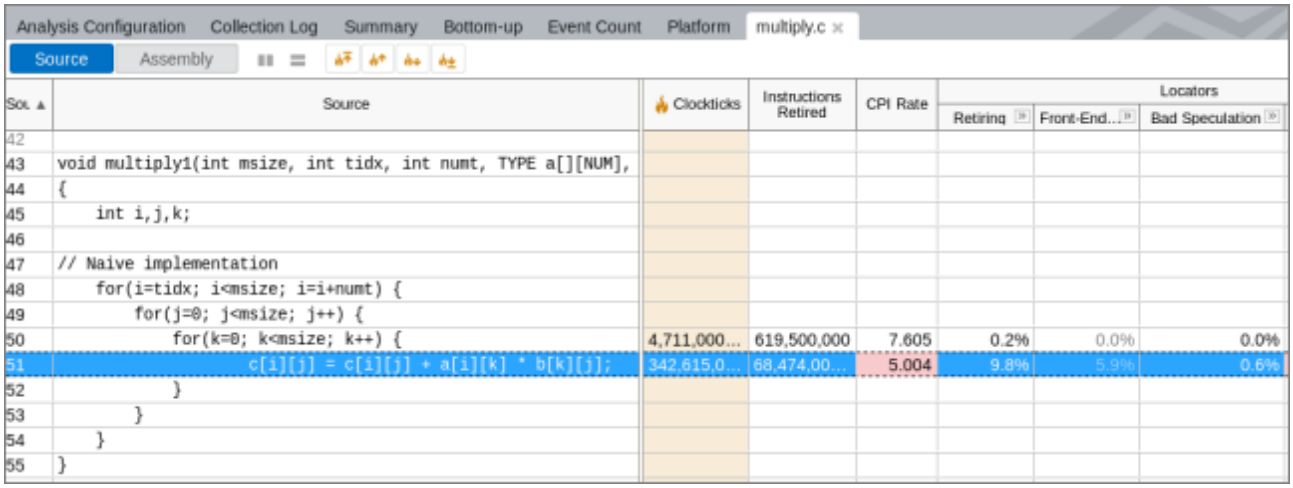


左側のメトリックツリーから、パフォーマンスは主に DRAM アクセスによって制限されていることが分かります。

[Bottom-up (ボトムアップ)] ビューに切り替えると、アプリケーションに 1 つの大きな hotspot 関数 `multiply1` があることが分かります。



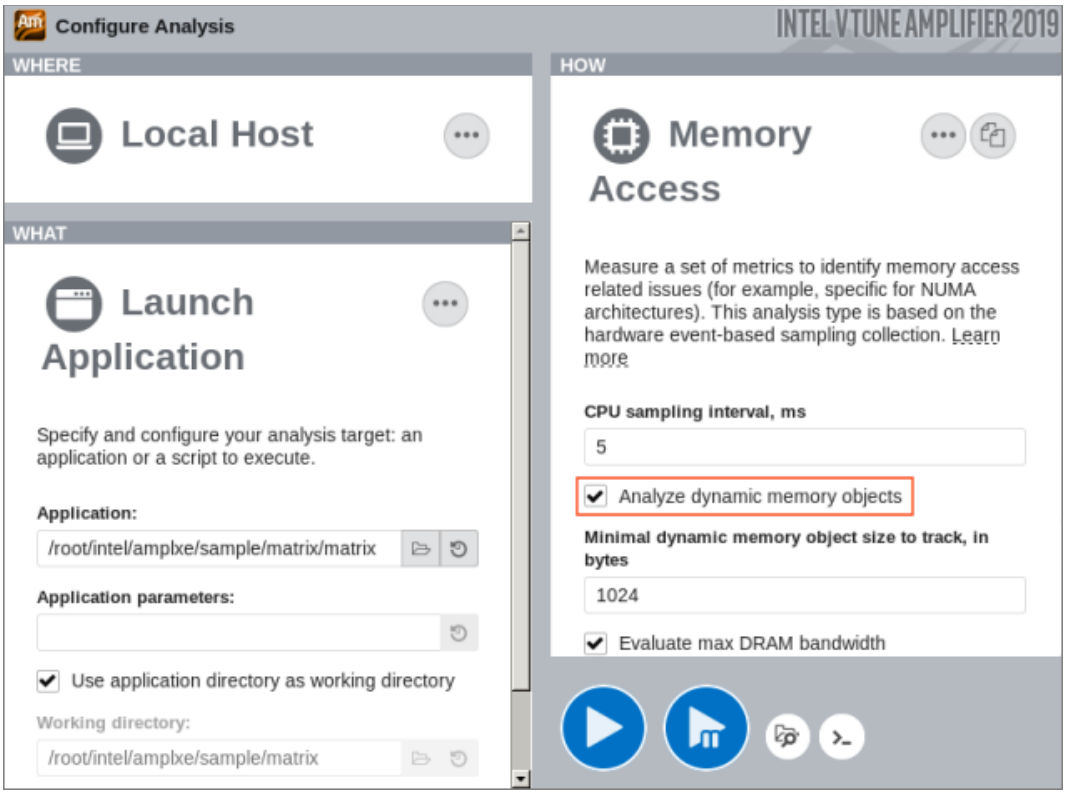
この関数をダブルクリックして [Source (ソース)] ビューを開きます。最もパフォーマンス・クリティカルなコード行がハイライトされます。



ほとんどの時間が、3 つの配列 (a、b、および c) を操作しているソース行 51 で費やされています。

メモリーアクセス解析を実行する

最も時間がかかった配列アクセスを調べるため、**[Analyze dynamic memory objects (動的メモリー・オブジェクトを解析)]** オプションを有効にしてメモリーアクセス解析を実行します。



ホットなメモリーアクセスを特定する

次のように、メモリーアクセス解析結果の **[Summary (サマリー)]** ウィンドウに、上位のメモリー・オブジェクトが表示されます。

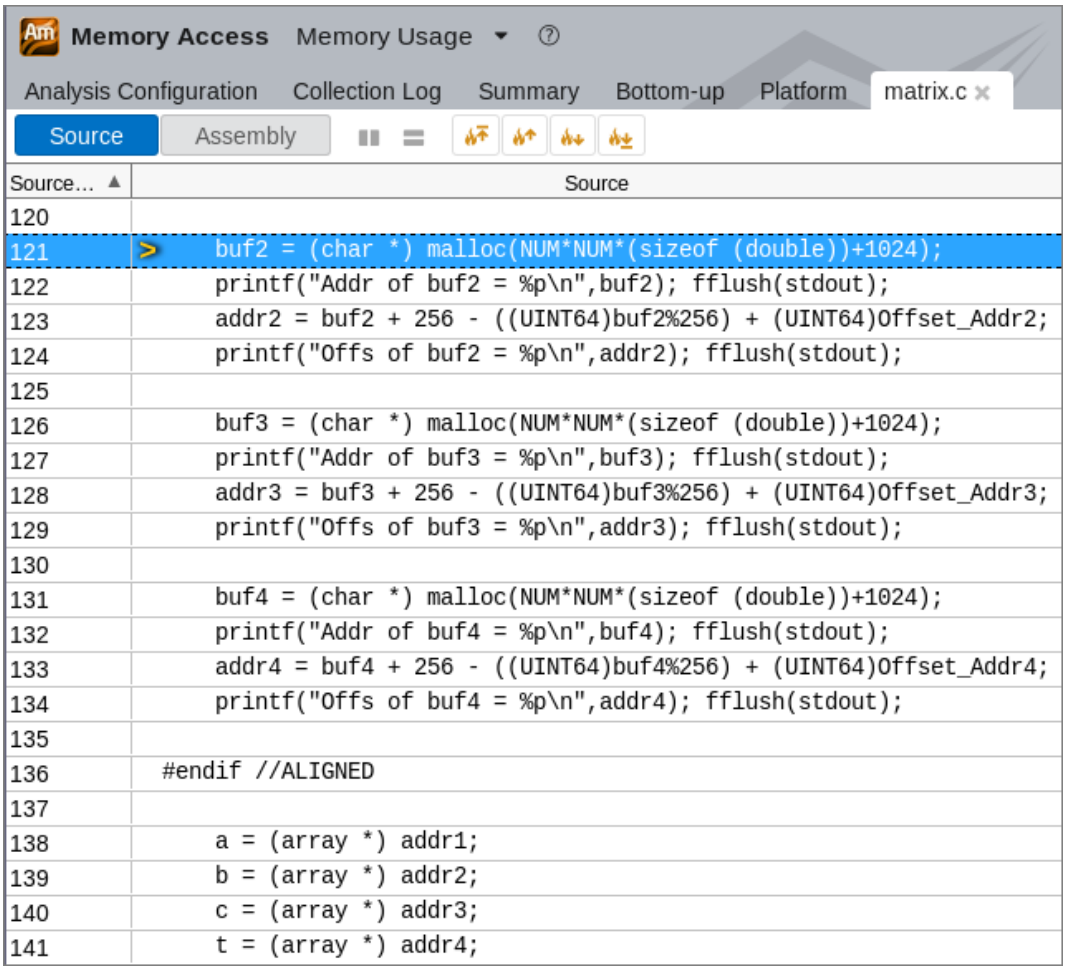
Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
matrix.c:121 (32 MB)	94.8%	14,125,223,744	0	7,208,432,480
matrix.c:116 (32 MB)	5.1%	3,061,691,848	0	4,800,288
[vmlinux]	0.1%	232,006,960	102,403,072	1,200,072
[sep5]	0.0%	8,000,240	0	0
matrix.c:126 (32 MB)	0.0%	800,024	8,688,260,640	400,024
[Others]	0.0%	13,600,408	2,400,072	0

*N/A is applied to non-summable metrics.

リストの最初の hotspot オブジェクト `matrix.c:121` をクリックして **[Bottom-up (ボトムアップ)]** ビューに切り替えた後、グリッドでハイライトされているこのオブジェクトをダブルクリックして **[Source (ソース)]** ビューを開き、このメモリー・オブジェクトの行を確認します。



`buf2` 変数が `addr2` に代入され、それが配列 `b` に代入されていることが分かります。つまり、問題のある配列は `b` と考えられます。ツールバーの  **[Open Source File Editor (ソース・ファイル・エディターを開く)]** ボタンをクリックして、コードを再度確認します。

```
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE v[][NUM],
TYPE c[][NUM], TYPE t[][NUM])
{
    int i,j,k;

    // Naive implementation
    for(i=tidx; i<msize; i=i+numt) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

問題の根本的な原因が分かりました。最内サイクルが非効率な方法で配列 `b` を反復しているため、各反復で大きなメモリーチャンクにジャンプしています。

ループ交換の最適化を適用する

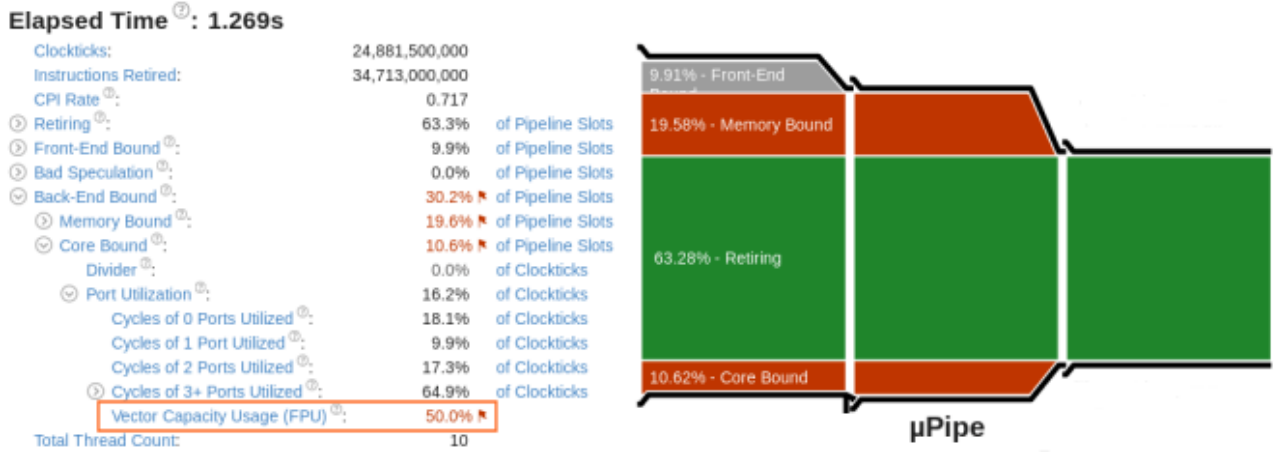
次のように、`j` と `k` にループ交換アルゴリズムを適用します。

```
for(i=tidx; i<msize; i=i+numt) {
                                    for(k=0; k<msize; k++) {
                                                                for(j=0;
j<msize; j++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

新しいコードをコンパイルして実行すると、実行時間は 1.3 秒になり、オリジナル (26 秒) の 20 倍にパフォーマンスが向上しました。

次のステップ

最適化したコードでマイクロアーキテクチャー全般解析を再度実行します。**[µPipe (µ パイプ)]** 図の **[Retiring (リタイア)]** メトリックの値が 10.06% から 63.28% へ大幅に増加しました。



その他のフラグの付いているメトリックに注目して、さらなるパフォーマンス向上の可能性 (例えば、低いポート使用率) を特定します。

関連情報

- ハードウェア問題のマイクロアーキテクチャー全般解析 (英語)
- メモリーアクセス解析 (英語)
- トップダウン・マイクロアーキテクチャー解析法

低いポート使用率

このレシピは、インテル® VTune™ Amplifier のマイクロアーキテクチャー全般解析を使用してコア依存の `matrix` アプリケーションをプロファイルし、低いポート使用率の原因を理解します。また、インテル® Advisor を使用してコンパイラーがベクトル化を行うようにします。

コンテンツ・エキスパート: Dmitry Ryabtsev (英語)

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** 2048x2048 サイズの 2 つの行列を乗算する行列乗算サンプル (要素は double 型)。`matrix_vtune_amp_axe.tgz` サンプルパッケージは、製品の `<install-dir>/samples/en/C++` ディレクトリーに含まれています。https://software.intel.com/en-us/product-code-samples (英語) からダウンロードすることもできます。
- **パフォーマンス解析ツール:**
    - インテル® VTune™ Amplifier 2019: マイクロアーキテクチャー全般解析 (英語)

注

        - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
        - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
        - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
    - インテル® Advisor: ベクトル化解析 (英語)
- **オペレーティング・システム:** Ubuntu* 16.04 64 ビット
- **CPU:** インテル® Core™ i7-6700K プロセッサー

ベースラインを作成する

単純な乗算アルゴリズムを実装した `matrix` コードの初期バージョンを最適化することにより (「頻繁な DRAM アクセス」レシピを参照)、実行時間は 26 秒から 1.3 秒になりました。これが、以降の最適化で使用する新しいパフォーマンスのベースラインとなります。

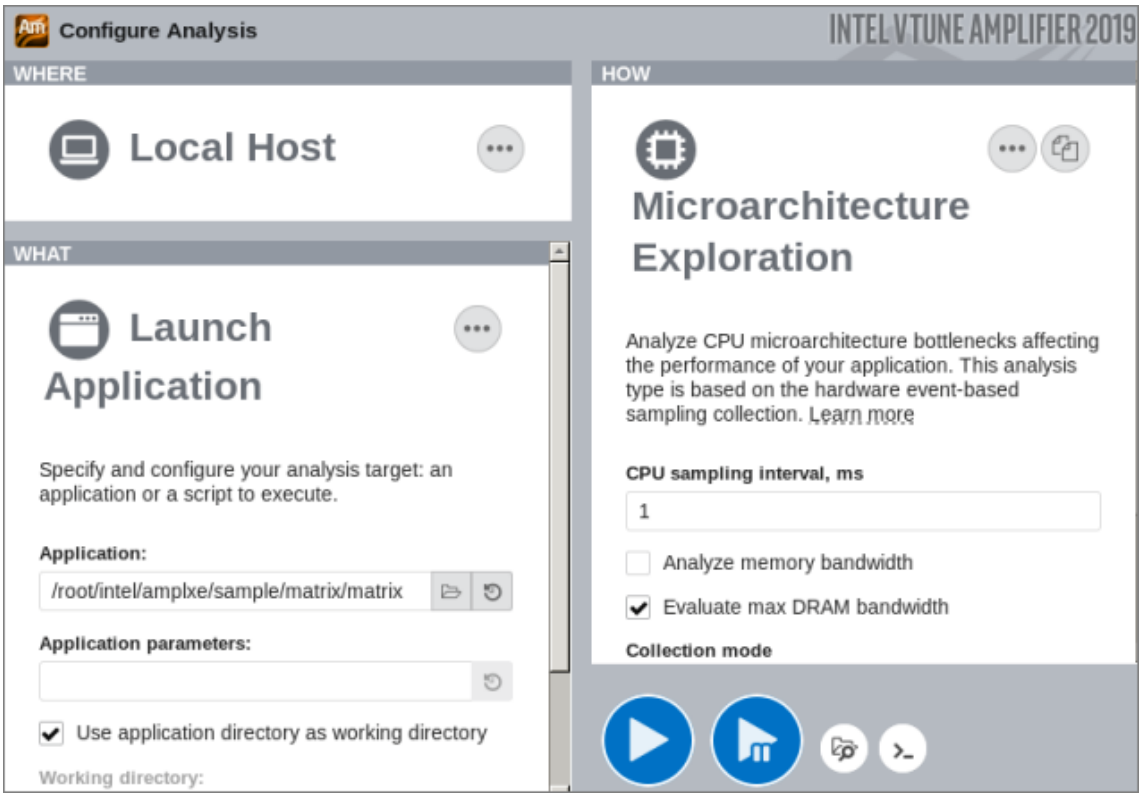
マイクロアーキテクチャー全般解析を実行する

サンプル・アプリケーションの潜在的なパフォーマンス・ボトルネックを理解するため、全般解析を実行します。

1. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `matrix`) を指定します。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

2. **[WHERE (どこを)]** ペインで、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、解析するアプリケーションを指定します。
4. **[HOW (どのように)]** ペインで、[...] ボタンをクリックして **[Microarchitecture (マイクロアーキテクチャー)] > [Microarchitecture Exploration (マイクロアーキテクチャー全般)]** を選択します。
5. オプションで、最適化した `matrix` アプリケーションのように小さなワークロードで、サンプリング間隔を 0.1 秒にして信頼性のあるメトリック値が得られるか確認します。
6. **[Start (開始)]** をクリックして解析を開始します。



インテル® VTune™ プロファイラーは、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

低いポート使用率の原因を特定する

ハードウェア・メトリックごとのアプリケーション・パフォーマンスの統計が表示される **[Summary (サマリー)]** ビューから始めます。

Front-End Bound:	14.7%	of Pipeline Slots
Bad Speculation:	0.3%	of Pipeline Slots
Back-End Bound:	57.1%	of Pipeline Slots
Memory Bound:	20.6%	of Pipeline Slots
L1 Bound:	10.6%	of Clockticks
DTLB Overhead:	14.3%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	2.1%	of Clockticks
FB Full:	0.2%	of Clockticks
L2 Bound:	1.8%	of Clockticks
L3 Bound:	0.1%	of Clockticks
DRAM Bound:	0.2%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	36.5%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	22.6%	of Clockticks
Cycles of 0 Ports Utilized:	0.4%	of Clockticks
Cycles of 1 Port Utilized:	22.2%	of Clockticks
Cycles of 2 Ports Utilized:	26.6%	of Clockticks
Cycles of 3+ Ports Utilized:	84.5%	of Clockticks
Vector Capacity Usage (FPU):	25.0%	
Retiring:	27.9%	of Pipeline Slots

主要なボトルネックが **[Core Bound (コア依存)]** > **[Port Utilization (ポート利用率)]** に移動し、ほとんどの時間で 3 つ以上の実行ポートが同時に使用されていることが分かります。**[Vector Capacity Usage (ベクトル能力の使用)]** メトリックもクリティカルな値としてフラグが付いていることに注意してください。これは、コードがベクトル化されていないか、ベクトル化の効率が悪いことを意味します。確認のため、次のように、カーネルの [Assembly (アセンブリー)] ビューに切り替えます。

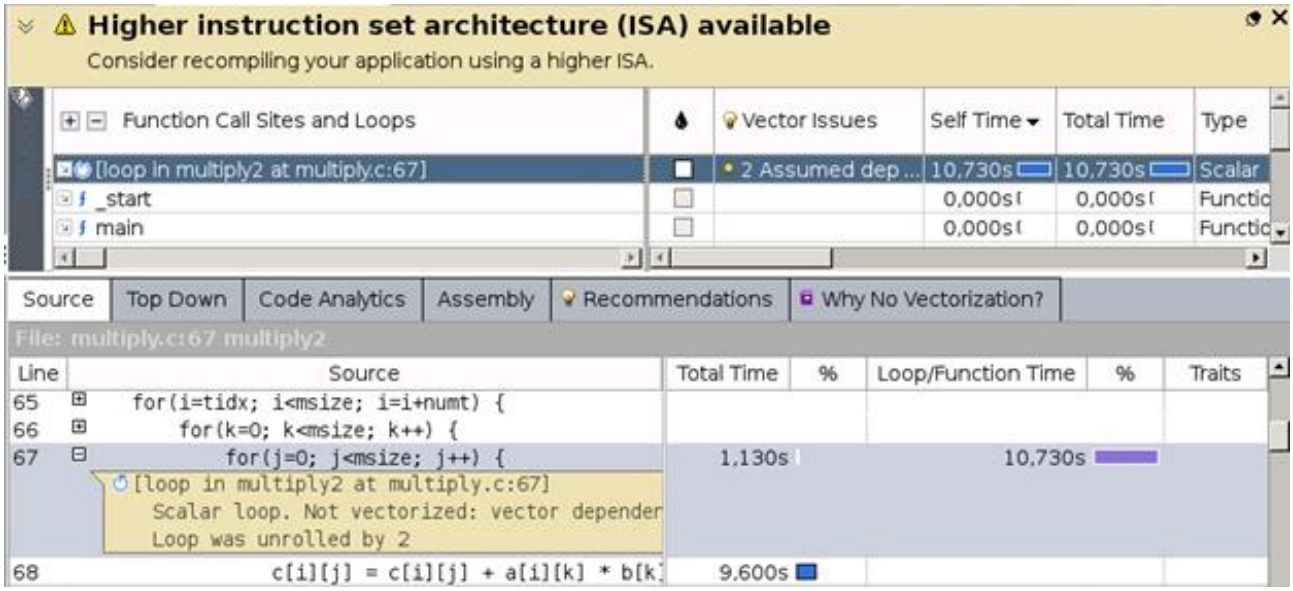
1. **[Vector Capacity Usage (FPU) (ベクトル能力の使用 (FPU))]** メトリックをクリックして、このメトリックでソートされた **[Bottom-up (ボトムアップ)]** ビューに切り替えます。
2. ホットな `multiply1` 関数をダブルクリックして **[Source (ソース)]** ビューを開きます。
3. ツールバーの **[Assembly (アセンブリー)]** ボタンをクリックして、逆アセンブルしたコードを表示します。

0x4016ca	68	movsd xmm0, qword ptr [r12]	1,200,000	0
0x4016d0	67	inc rbx	94,800,000	91,200,000
0x4016d3	68	mulsd xmm0, qword ptr [r11+rcx*1]	4,330,800,000	7,659,200,000
0x4016d9	68	addsd xmm0, qword ptr [r11+r15*1]	4,378,400,000	2,163,200,000
0x4016df	68	movsd qword ptr [r11+r15*1], xmm0	11,392,400, ...	4,576,800,000
0x4016e5	68	movsd xmm1, qword ptr [r12]	3,642,800,000	9,865,200,000
0x4016eb	68	mulsd xmm1, qword ptr [r11+rcx*1+0x8]	988,800,000	836,000,000
0x4016f2	68	addsd xmm1, qword ptr [r11+r15*1+0x8]	2,749,600,000	1,459,600,000
0x4016f9	68	movsd qword ptr [r11+r15*1+0x8], xmm1	9,280,000,000	6,056,800,000
0x401700	67	add r11, 0x10	4,434,000,000	18,955,600, ...
0x401704	67	cmp rbx, rdx	38,400,000	1,200,000
0x401707	67	jb 0x4016ca <Block 10>		
0x401709		**Block 11:**		
0x401709	68	lea ecx, ptr [rbx+rbx*1+0x1]		
0x40170d		**Block 12:**		
0x40170d	67	lea ebx, ptr [rcx-0x1]	1,200,000	2,000,000
0x401710	67	cmp ebx, edi	2,800,000	7,200,000
0x401712	67	jnb 0x401736 <Block 14>		
0x401714		**Block 13:**		
0x401714	68	movsxd rcx, ecx		
0x401717	68	movsd xmm0, qword ptr [r12]		
0x40171d	68	lea rbx, ptr [r8+rcx*8]		
0x401721	68	mulsd xmm0, qword ptr [rbx+r14*1-0x8]		
0x401728	68	addsd xmm0, qword ptr [r15+rcx*8-0x8]		
0x40172f	68	movsd qword ptr [r15+rcx*8-0x8], xmm0		

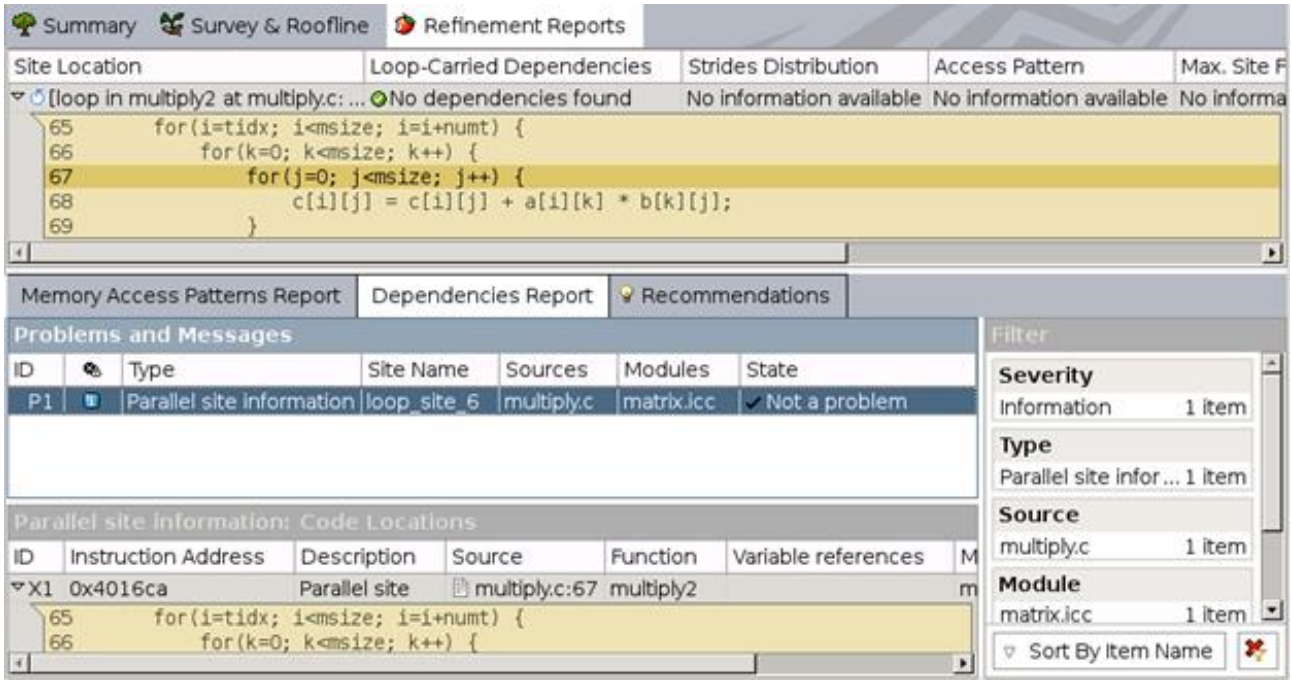
スカラー命令が使用されていることが分かります。コードはベクトル化されていません。

ベクトル化のオプションを調べる

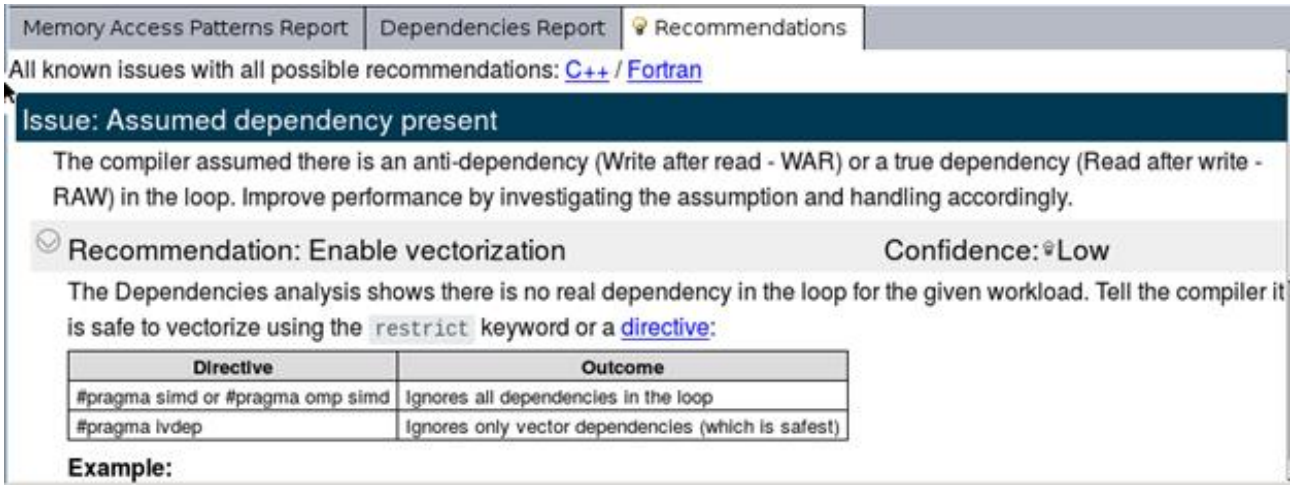
インテル® Advisor のベクトル化アドバイザー・ツールを使用して、コードのベクトル化を妨げている原因を調べます。



インテル® Advisor は、依存関係が仮定されたためにループがベクトル化されなかったと報告しました。詳細に確認するため、ループをマークしてインテル® Advisor の依存関係解析を実行します。



レポートによれば、依存関係は存在していません。インテル® Advisor は、コンパイラーが仮定された依存関係を無視するように `#pragma` を使用することを推奨しています。



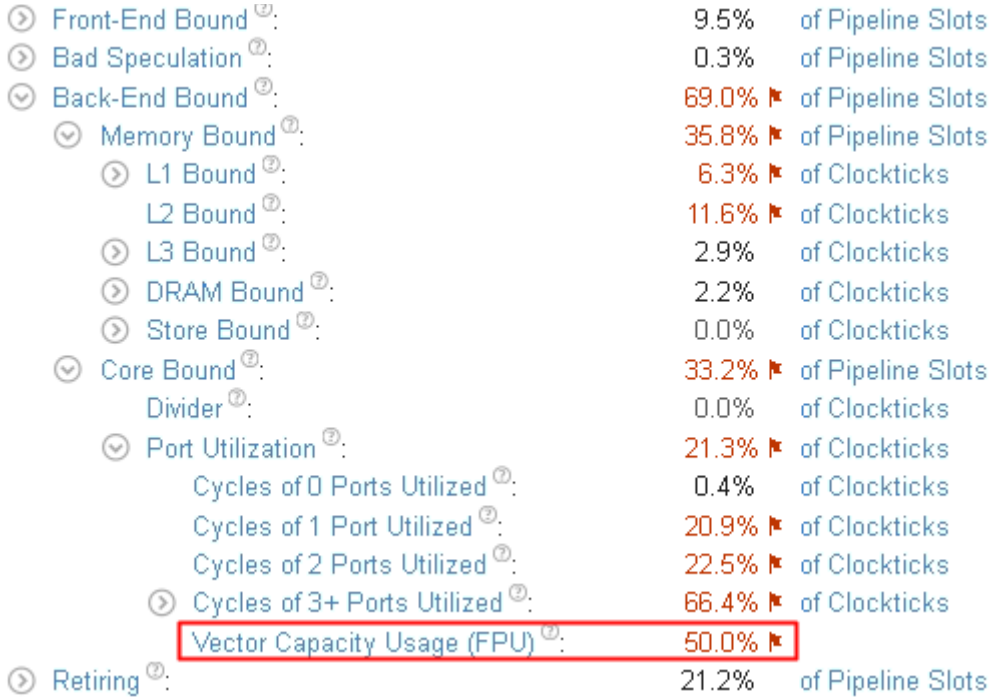
次の ように、matrix コードに #pragma を追加します。

```
void multiply2_vec(inte msize, int tidx, int numt, TYPE a[][NUM],
    TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM]
{
    int i,j,k;

    for(i=tidx; i<msize; i=i+numt) {
        for(k=0; k<msize; k++) {
#pragma ivdep
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][j] * b[i][j];
            }
        }
    }
}
```

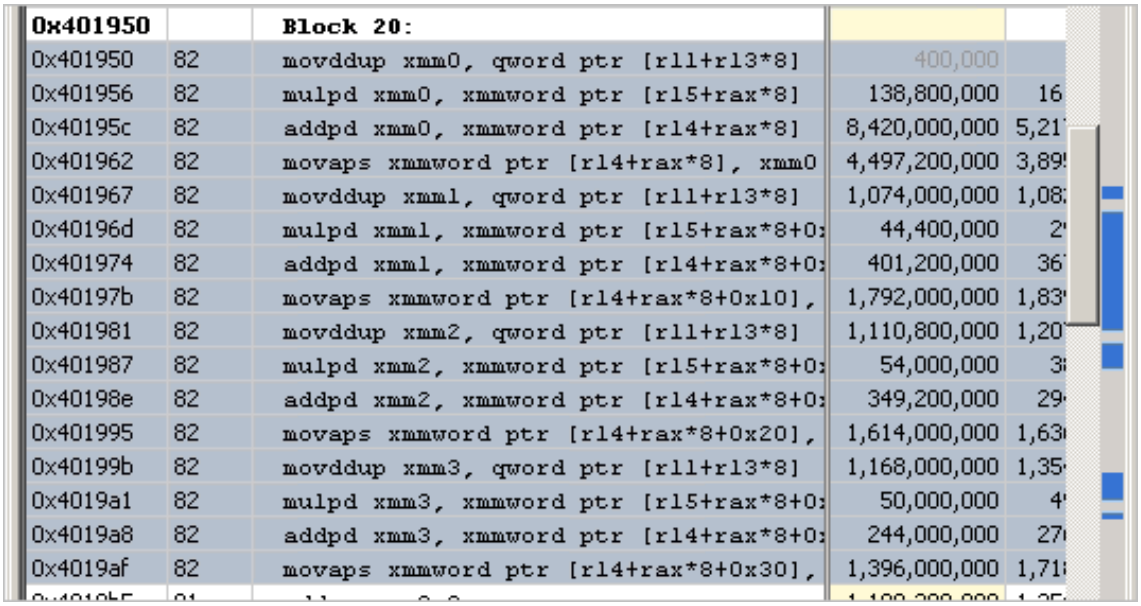
更新したコードをコンパイルして実行すると、実行時間は 0.7 秒になりました。

最新の命令セットを使用してコンパイルする

最新バージョンのコードでインテル® VTune™ プロファイラーのマイクロアーキテクチャー全般解析を再度実行すると、結果は次のようになりました。



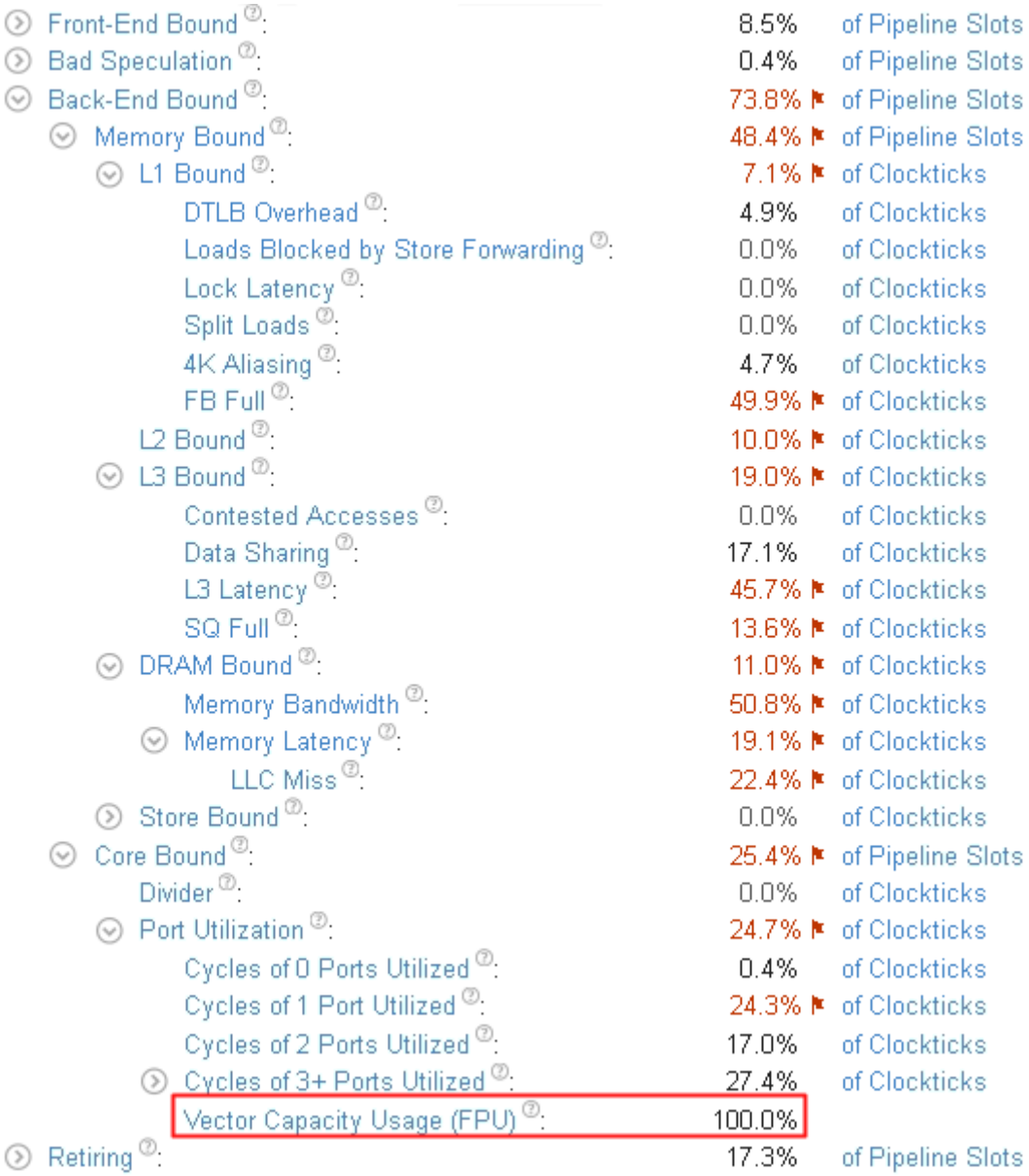
[Vector Capacity Usage (ベクトル能力の使用)] は 50% まで向上していますが、まだパフォーマンス・クリティカルのフラグが付いたままです。詳細な情報を得るため、[Assembly (アセンブリー)] ビューを再度調べます。



0x401950		Block 20:		
0x401950	82	movddup xmm0, qword ptr [r11+r13*8]	400,000	
0x401956	82	mulpd xmm0, xmmword ptr [r15+rax*8]	138,800,000	16
0x40195c	82	addpd xmm0, xmmword ptr [r14+rax*8]	8,420,000,000	5,21
0x401962	82	movaps xmmword ptr [r14+rax*8], xmm0	4,497,200,000	3,89
0x401967	82	movddup xmm1, qword ptr [r11+r13*8]	1,074,000,000	1,08
0x40196d	82	mulpd xmm1, xmmword ptr [r15+rax*8+0	44,400,000	2
0x401974	82	addpd xmm1, xmmword ptr [r14+rax*8+0	401,200,000	36
0x40197b	82	movaps xmmword ptr [r14+rax*8+0x10],	1,792,000,000	1,83
0x401981	82	movddup xmm2, qword ptr [r11+r13*8]	1,110,800,000	1,20
0x401987	82	mulpd xmm2, xmmword ptr [r15+rax*8+0	54,000,000	3
0x40198e	82	addpd xmm2, xmmword ptr [r14+rax*8+0	349,200,000	29
0x401995	82	movaps xmmword ptr [r14+rax*8+0x20],	1,614,000,000	1,63
0x40199b	82	movddup xmm3, qword ptr [r11+r13*8]	1,168,000,000	1,35
0x4019a1	82	mulpd xmm3, xmmword ptr [r15+rax*8+0	50,000,000	4
0x4019a8	82	addpd xmm3, xmmword ptr [r14+rax*8+0	244,000,000	27
0x4019af	82	movaps xmmword ptr [r14+rax*8+0x30],	1,396,000,000	1,71

[Assembly (アセンブリー)] ビューから、ここで使用している CPU はインテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2) 命令セットをサポートしているにも関わらず、コードはインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) を使用していることが分かります。新しい命令セットをサポートするため、`-xCORE-AVX2` オプションを指定してコードを再コンパイルし、全般解析を再度実行します。

再コンパイルしたコードでは、実行時間は 0.6 秒になりました。マイクロアーキテクチャー全般解析を再度実行して最適化を確認します。**[Vector Capacity Usage (ベクトル能力の使用)]** メトリックの値は 100% になりました。



関連情報

- ハードウェア問題のマイクロアーキテクチャー全般解析 (英語)
- トップダウン・マイクロアーキテクチャー解析法

ページフォールト

このレシピは、インテル® VTune™ プロファイラーのマイクロアーキテクチャー全般、システム概要、メモリー消費解析を使用して、ページフォールトがターゲット・アプリケーションのパフォーマンスに与える影響を特定して測定する方法を説明します。

コンテンツ・エキスパート: Vitaly Slobodskoy (英語)

「ページフォールト」は、実行中のプログラムがプロセスの仮想アドレス空間にマップされていないメモリーページにアクセスしたときに発生します。マッピングは、メモリー位置へのアクセスにかかる時間を短縮するため、トランスレーション・ルックアサイド・バッファー (TLB) をキャッシュとして使用して、メモリー管理ユニット (MMU) により処理されます。TLB ミスが発生すると、ページが実際にマップされていないプロセスにアクセスすることがあります。または、ページフォールト例外を発行して、ストレージデバイスからページコンテンツをロードする必要があります。ページフォールトは仮想メモリーを処理するための一般的なメカニズムですが、ページサイズが大きくなるためにアプリケーションのパフォーマンスに大きな影響を与えることがあります。

- 使用するもの
- 手順:
    1. マイクロアーキテクチャー全般解析を使用して TLB ミスを特定する
    2. システム概要解析を使用してカーネル・アクティビティーをトレースする
    3. メモリー消費解析を使用して割り当てメモリーの量を計算する
    4. ヒュージページを使用してページフォールトを軽減する

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション**: `matrix` アプリケーション。製品の `<install-dir>/samples/en/C++` ディレクトリーに含まれています。このレシピでは、`src/multiply.h` の NUM 値を 2048 から 8192 に変更して行列のサイズを変更し、`/linux` ディレクトリーから `make` を実行して `matrix` アプリケーションをリビルドします。
- **パフォーマンス解析ツール**: インテル® oneAPI ベース・ツールキット > インテル® VTune™ プロファイラー > マイクロアーキテクチャー全般、システム概要、メモリー消費解析タイプ

  注

    - バージョン 2020 から、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。
    - インテル® VTune™ プロファイラー・パフォーマンス解析クックブックのほとんどのレシピは、異なるバージョンのインテル® VTune™ プロファイラーにも適用できます。バージョンにより、わずかな調整が必要になる場合があります。
    - 最新バージョンのインテル® VTune™ プロファイラーは以下から入手できます。
        - インテル® VTune™ プロファイラー製品ページ
        - インテル® oneAPI スタンドアロン・コンポーネント・ページ (英語)
- **オペレーティング・システム**: Ubuntu* 18.04.1 LTS 64 ビット

- **CPU**: インテル® Core™ i7-6700K プロセッサー

マイクロアーキテクチャー全般解析を使用して TLB ミスを特定する

アプリケーションのハードウェア・リソースの使用状況について詳細な情報を取得するには、マイクロアーキテクチャー全般解析を実行します。

1. インテル® VTune™ プロファイラーを起動します。

   デフォルトでは、インテル® VTune™ プロファイラーは `sample (matrix)` プロジェクトを開きます。

   注
   このプロジェクトが NUM=8192 で `matrix` アプリケーションを起動するように設定されていることを確認してください。設定されていない場合、新しいプロジェクトを作成してください。

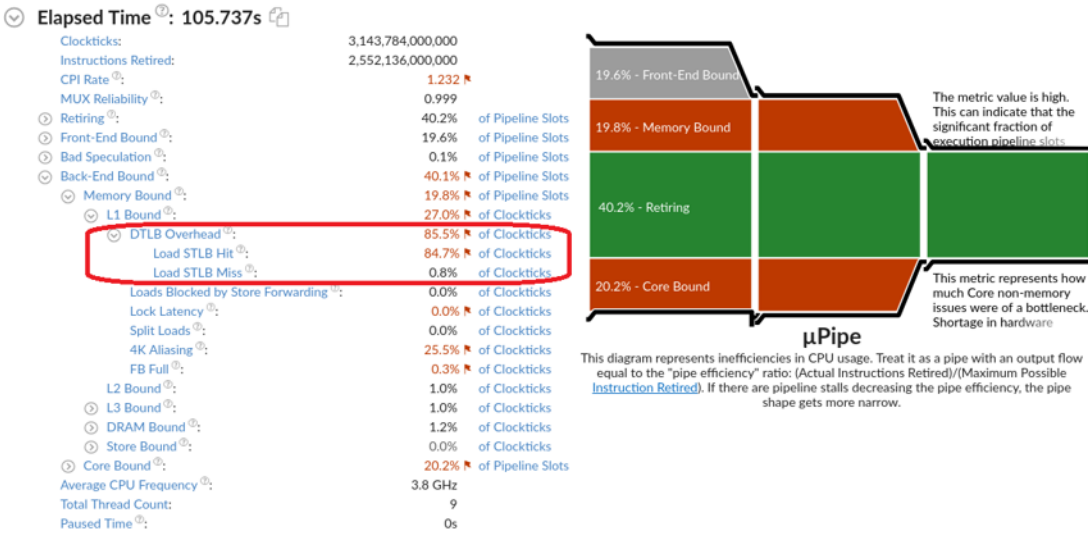
2. [ようこそ] ページの **[Configure Analysis... (解析の設定...)]** ボタンをクリックします。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

3. **[HOW (どのように)]** ペインで、下矢印ボタンをクリックして **[Microarchitecture (マイクロアーキテクチャー)]** 解析グループから **[Microarchitecture Exploration (マイクロアーキテクチャー全般)]** を選択します。
4. [Start (開始)] ボタンをクリックして、解析を実行します。

   インテル® VTune™ プロファイラーがデータを収集し、アプリケーション・レベルの統計を含む **[Summary (サマリー)]** ウィンドウを開きます。

TLB ミスに起因するバックエンド依存問題を調べます。

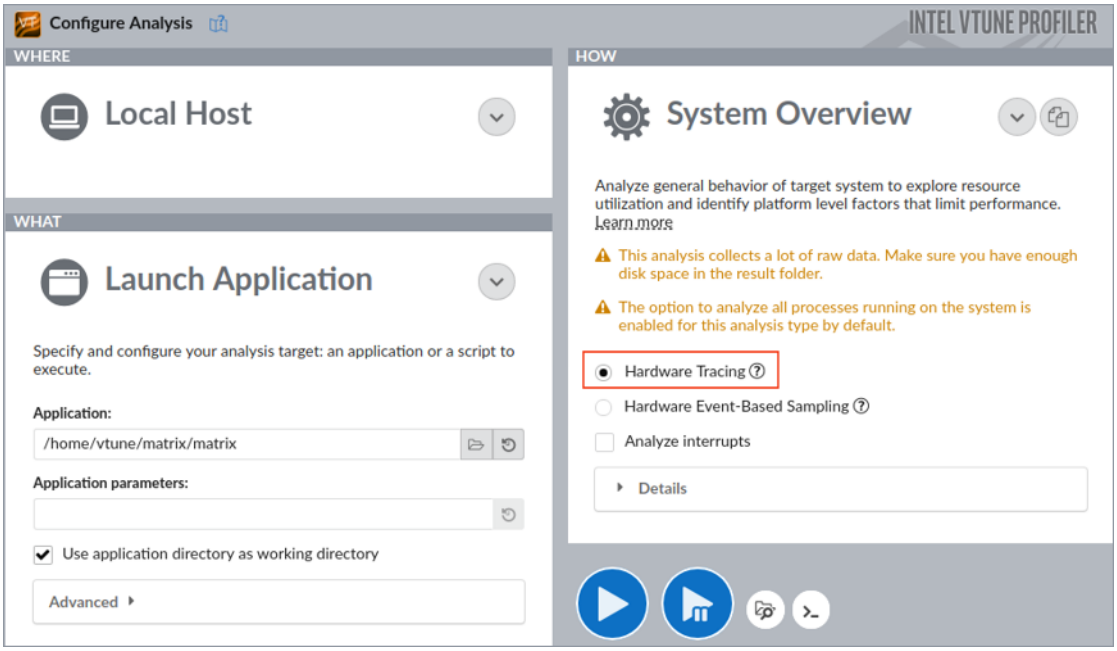


[DTLB Overhead (DTLB オーバーヘッド)] メトリックは TLB ミスのパフォーマンス・ペナルティーを推定します。ほとんどのオーバーヘッドは、セカンドレベル TLB (STLB) にヒットしたファーストレベル (DTLB) のミスをカウントする **[Load STLB Hit (ロード STLB ヒット)]** メトリックによるものです。ハードウェア・ページウォー

クを実行するサイクルの一部を表す **[Load STLB Miss (ロード STLB ミス)]** メトリックの値は大きくありません。これらのメトリックは、ページフォールト例外内で費やされた時間を考慮していないことに注意してください。そのため、マイクロアーキテクチャー全般解析は TLB 関連の問題の診断には役立ちますが、アプリケーションの経過時間に対するページフォールト例外の影響は推定できません。

システム概要解析を使用してカーネル・アクティビティーをトレースする

ページフォールトは、Linux* カーネルによりキャッチされた割り込みがトリガーとなります。Linux* カーネル内で費やされた正確な CPU 時間を測定するには、より詳細な解析が必要です。**[Hardware Tracing (ハードウェア・トレース)]** モードのシステム概要解析は、インテル® Processor Trace テクノロジーを使用して、CPU コアのすべてのリタイアした分岐命令をキャプチャーします。特に、この解析により、割り込みを含むすべてのカーネル・アクティビティーの正確なトレースが可能になります。



[Launch Application (アプリケーションを起動)] ターゲット設定を使用しても、この解析はシステム全体のデータ収集を実行します。

大量の分岐命令のため、この解析は多くのローデータを収集します。コマンドラインから解析を起動して、データ収集範囲を最初の 3 秒間に制限できます。

```
vtune -collect system-overview -knob collecting-mode=hw-tracing -d 3 -r
matrix-so ./matrix
```

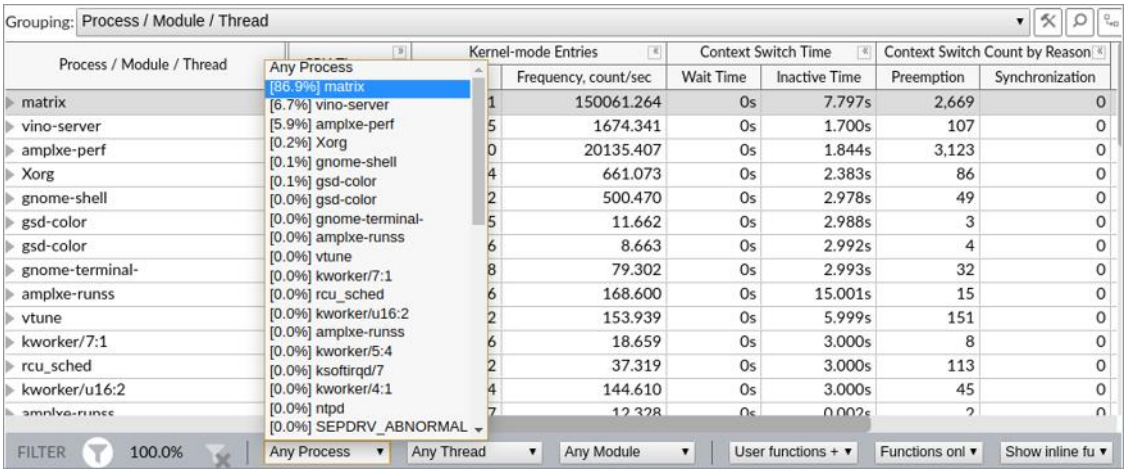
注

`vtune` コマンドライン・インターフェイスを起動する前に、製品のインストール・ディレクトリーから `source env/vars.sh` コマンドを実行して環境変数を設定してください。

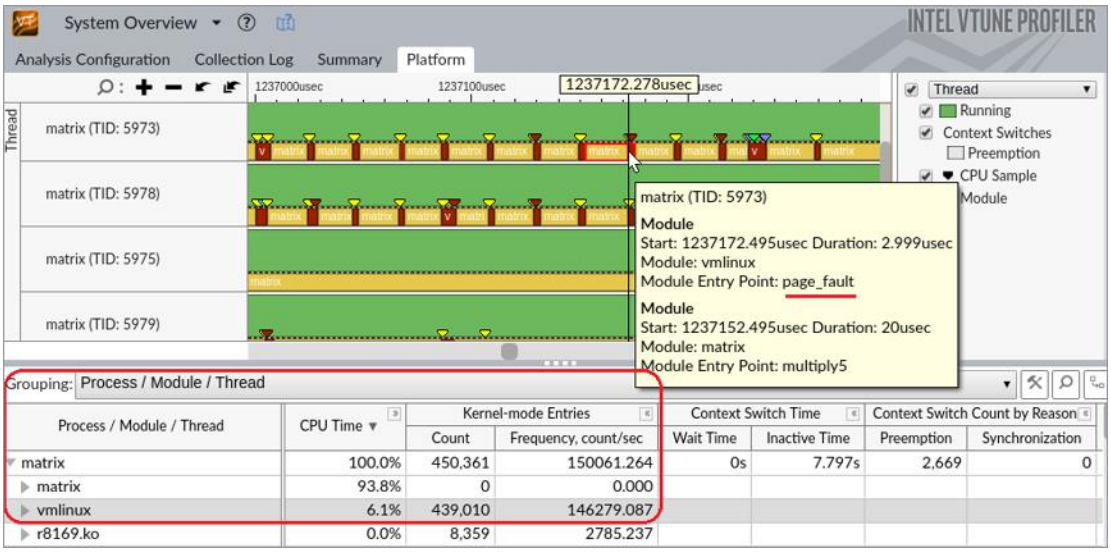
インテル® VTune™ プロファイラー GUI で結果を開きます。

```
vtune-gui ./matrix-so
```

結果を開いたら、**[Platform (プラットフォーム)]** タブに切り替え、ドロップダウン・メニューを使用して `matrix` プロセスで収集したデータをフィルタリングします。



オーバータイム・ビューを提供する [Timeline (タイムライン)] ペインから、ほとんどの CPU 時間が `multiply` 関数を実行する `matrix` モジュール内で費やされていることが分かります。この関数は継続的に実行されず、通常、数ミリ秒で中断され、最大の割り込みがページフォールトにより発生しています。



グリッドビューから、サンプル・アプリケーションが Linux* カーネル内で費やした時間は 6.1% であり、439K のカーネルエントリーがアプリケーション実行の最初の 3 秒以内に発生していることが分かります。この問題を解決するため、ヒュージページを使用することを検討します。

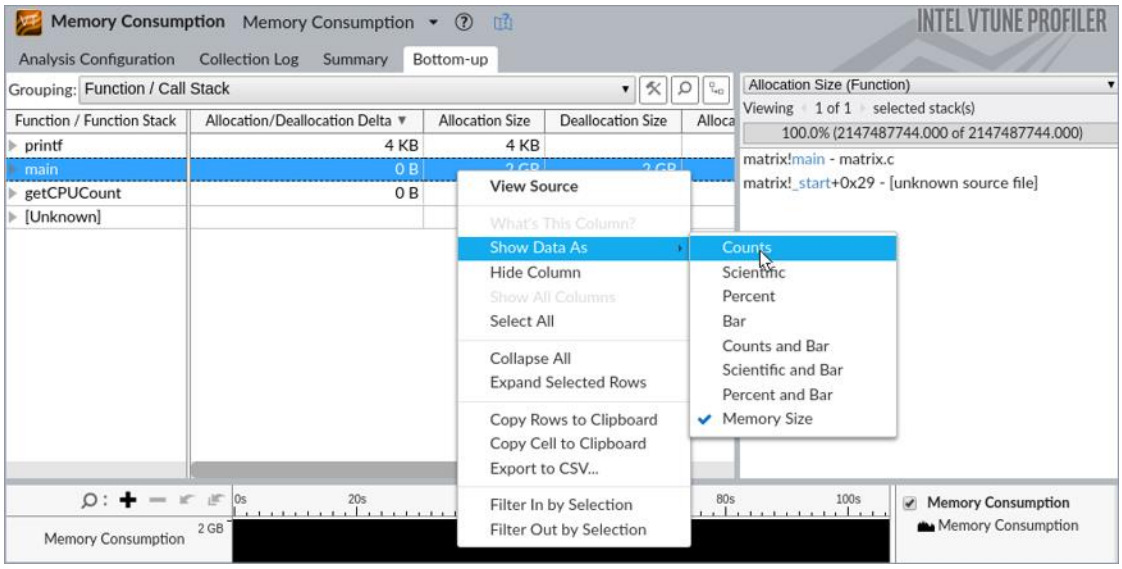
メモリー消費解析を使用して割り当てメモリーの量を計算する

ヒュージページに切り替えるには、必要なページを定義します。このために、アプリケーションが割り当てるメモリーの量を計算します。`matrix` のような単純なアプリケーションでは、ソースコードを調べるだけでかまいません。より複雑なアプリケーションでは、**[Memory Consumption (メモリー消費)]** 解析を使用することを検討してください。この解析は、正確な割り当てメモリーのサイズを提供し、ヒュージページを使用する必要があるオブジェクトを識別します。

1. **[Configure Analysis (解析の設定)]** をクリックして `matrix` プロジェクトの設定を開きます。
2. **[HOW (どのように)]** ペインで、下矢印ボタンをクリックして **[Hotspots (ホットスポット)]** 解析グループから **[Memory Consumption (メモリー消費)]** を選択します。
3. **[Minimal dynamic memory object size to track (追跡する最小動的メモリー・オブジェクト・サイズ)]** オプションの値を 1 に変更します。
4. [Start (開始)] ボタンをクリックして、解析を実行します。

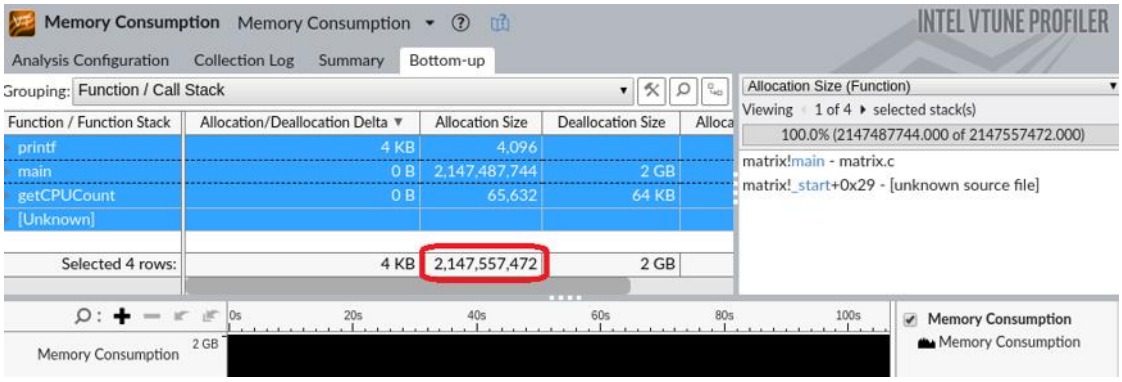
   インテル® VTune™ プロファイラーがデータを収集し、アプリケーション・レベルの統計を含む **[Summary (サマリー)]** ウィンドウを開きます。

5. **[Bottom-up (ボトムアップ)]** タブをクリックします。**[Allocation Size (割り当てサイズ)]** カラムを右クリックして、**[Show Data As (データを表示)]** > **[Counts (カウント)]** をバイト表現に選択します。



6. グリッドを再度右クリックして **[Select All (すべて選択)]** を選択して (または **Ctrl + A** キーを押して)、合計割り当てサイズを確認します。

   アプリケーションは 2147557472 バイト割り当てています。



ヒュージページを使用してページフォールトを軽減する

デフォルトでは、ページサイズは 4KB です。ヒュージページでは、デフォルトのページサイズは 2MB で、1GB まで増やすことができます。libhugetlbfs (英語) を使用すると、ヒュージページに簡単に切り替えることができます。

まず、必要な 2MB ページの数を計算する必要があります。サンプルの `matrix` アプリケーションは 2147557472 バイト割り当てています。つまり、2147557472 / 2097152 = 1025 の 2MB ページが必要になります (切り上げ)。

ヒュージページに切り替えるには、次の操作を行います。

1. ページの数を設定します。

   ```
   sudo hugeadm --pool-pages-min 2Mb:1025
   ```

2. 次の内容で `matrix.sh` スクリプトを作成します。

   ```
   #!/bin/bash

   LD_PRELOAD=libhugetlbfs.so HUGETLB_MORECORE=yes ./matrix
   ```

3. スクリプトを実行可能にします。

   ```
   chmod u+x ./matrix.sh
   ```

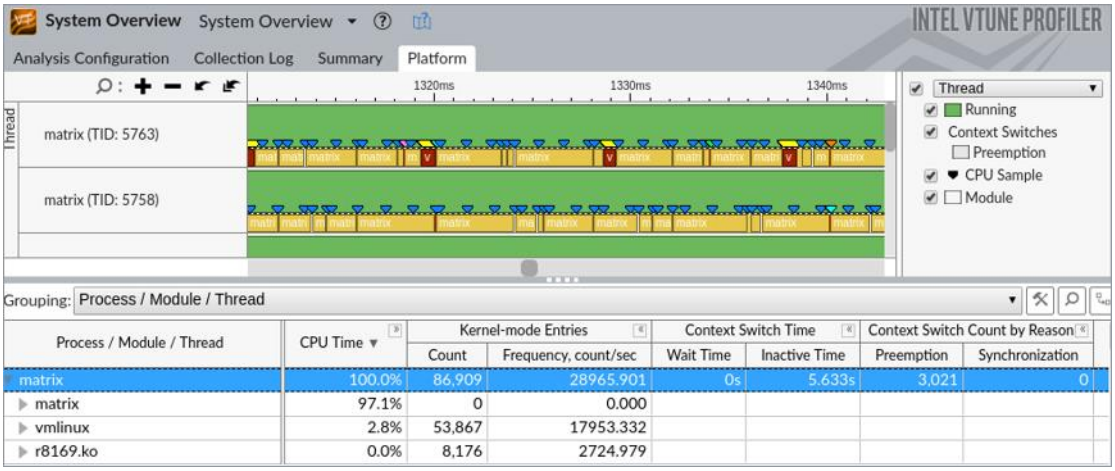
4. システム概要解析を再度実行します。

   ```
   vtune -collect system-overview -knob collecting-mode=hw-tracing -d 3 -r matrix-so-hp ./matrix.sh
   ```

5. インテル® VTune™ プロファイラー GUI で結果を開きます。

   ```
   vtune-gui ./matrix-so-hp
   ```

[Platform (プラットフォーム)] ビューを確認します。カーネル CPU 時間が 3.3% 短縮され、カーネルモードのエントリーが 8.1 倍削減されました。



ヒュージページに切り替えることにより、コードを変更することなく、`matrix` アプリケーションの所要時間が 106.4 秒から 100.5 秒に (約 5%) 短縮されました。

関連情報

ハードウェア問題のマイクロアーキテクチャー全般解析
システム概要解析
メモリー消費解析

命令キャッシュミス

このレシピは、インテル® VTune™ Amplifier の全般解析を使用してフロントエンド依存のアプリケーションをプロファイルし、PGO オプションを指定して ICache ミスを減らします。

コンテンツ・エキスパート: Dmitry Ryabtsev (英語)

- 使用するもの
- 手順:
    1. 全般解析を実行する
    2. ハードウェアの hotspot を特定する
    3. PGO オプションを指定してコードを再コンパイルする
    4. 最適化を確認する

注

全般解析は、インテル® VTune™ Amplifier 2019 でマイクロアーキテクチャー全般解析に改名されました。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** `sqlite` データベース・ベースのテストサンプル。このアプリケーションはデモ用であり、ダウンロードすることはできません。
- **ツール:**
    - インテル® VTune™ Amplifier 2018: 全般解析

注

        - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
        - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
        - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
    - インテル® C++ コンパイラー
- **オペレーティング・システム:** Microsoft* Windows* 7
- **CPU:** インテル® プロセッサー (開発コード名 Skylake)

全般解析を実行する

サンプル・アプリケーションの潜在的なパフォーマンス・ボトルネックを理解するため、まず、インテル® VTune™ Amplifier の全般解析を実行します。

1. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `sqlite`) を指定します。
2. **[Analysis Target (解析ターゲット)]** ウィンドウで、ホストベースの解析として **[local host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、右ペインで解析するアプリケーションを指定します。
4. 右の **[Choose Analysis (解析の選択)]** ボタンをクリックし、**[Microarchitecture Analysis (マイクロアーキテクチャー解析)] > [General Exploration (全般)]** を選択して、**[Start (開始)]** をクリックします。

   インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

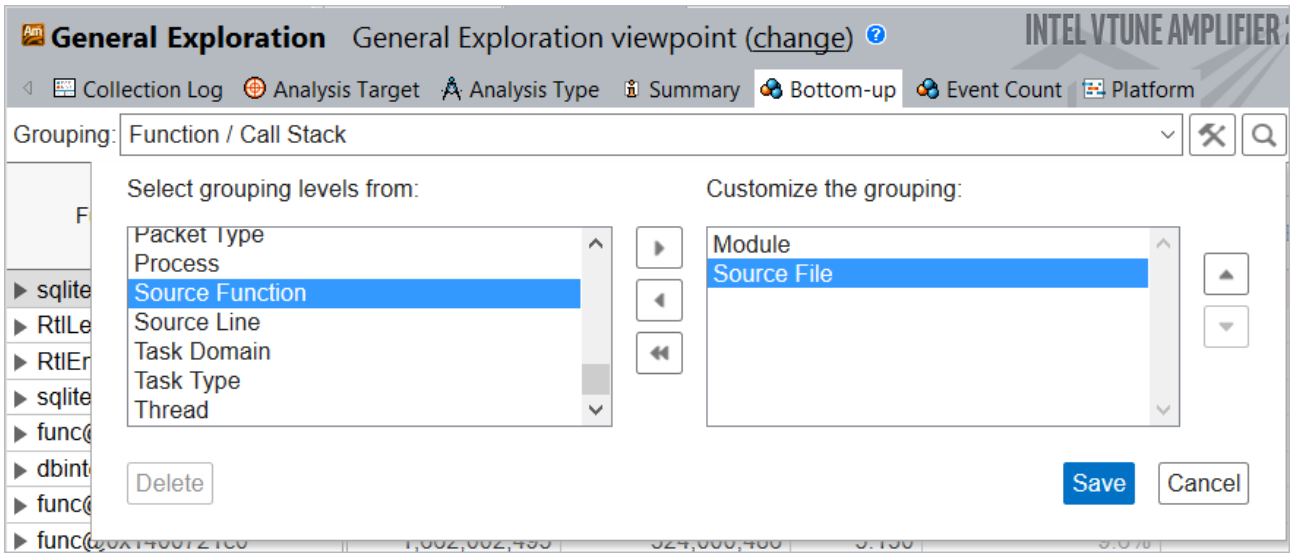
ハードウェアの hotspot を特定する

全般解析を実行すると、コードの主要なボトルネックを確認できます。ハードウェア・メトリックごとのアプリケーション・レベルの統計が表示される [Summary (サマリー)] ビューから解析を始めます。フラグの付いているパフォーマンス問題に注目します。

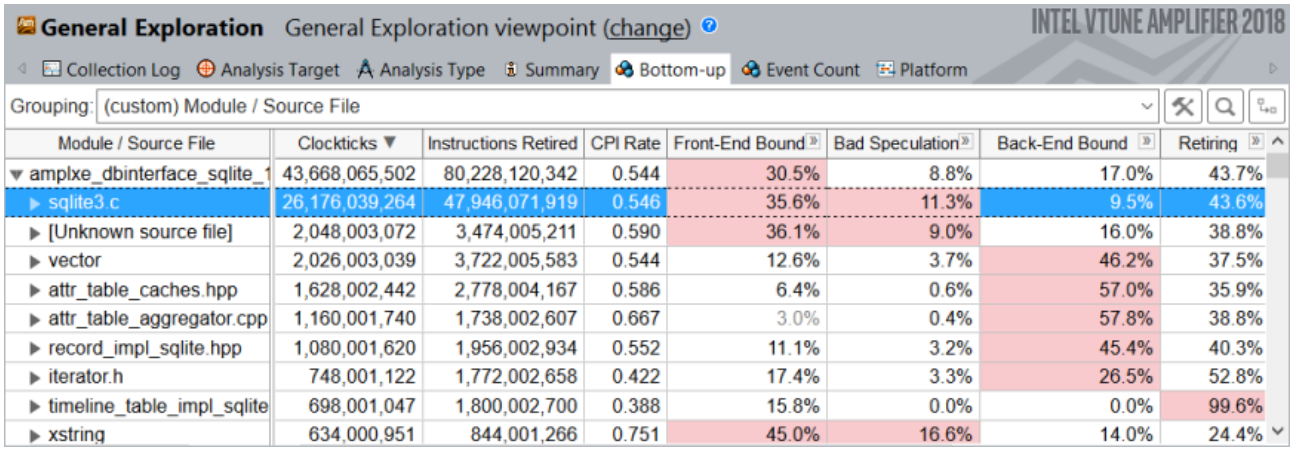
Elapsed Time: 31.539s		
Clockticks:	85,288,127,932	
Instructions Retired:	131,378,197,067	
CPI Rate:	0.649	
MUX Reliability:	0.957	
Front-End Bound:	29.3%	of Pipeline Slots
Front-End Latency:	20.1%	of Pipeline Slots
ICache Misses:	7.1%	of Clockticks
ITLB Overhead:	3.1%	of Clockticks
Branch Resteers:	4.8%	of Clockticks
DSB Switches:	2.6%	of Clockticks
Length Changing Prefixes:	0.1%	of Clockticks
MS Switches:	3.1%	of Clockticks
Front-End Bandwidth:	9.3%	of Pipeline Slots
Bad Speculation:	8.2%	of Pipeline Slots
Back-End Bound:	21.5%	of Pipeline Slots
Retiring:	40.9%	of Pipeline Slots
Total Thread Count:	6	
Paused Time:	0s	

サンプル・アプリケーションは、フロントエンド依存 (パイプライン・スロットの 29.3%) で、命令キャッシュミスが主要なボトルネック (クロック数の 7.1%) です。

[Bottom-up (ボトムアップ)] タブに切り替えてコードの問題を調べます。**[Grouping (グループ)]** ツールバーの横の  **[Customize Grouping (グループのカスタマイズ)]** ボタンをクリックして、新しいカスタムグループ **[Module/Source File (モジュール/ソースファイル)]** を作成します。



新しいグループを収集した結果に適用すると、`sqlite3.c` ファイルがほとんどの CPU サイクルを費やしているメインの hotspot であると表示されます。



Module / Source File	Clockticks ▼	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound	Retiring
▼ amplxe_dbinterface_sqlite_1	43,668,065,502	80,228,120,342	0.544	30.5%	8.8%	17.0%	43.7%
▶ sqlite3.c	26,176,039,264	47,946,071,919	0.546	35.6%	11.3%	9.5%	43.6%
▶ [Unknown source file]	2,048,003,072	3,474,005,211	0.590	36.1%	9.0%	16.0%	38.8%
▶ vector	2,026,003,039	3,722,005,583	0.544	12.6%	3.7%	46.2%	37.5%
▶ attr_table_caches.hpp	1,628,002,442	2,778,004,167	0.586	6.4%	0.6%	57.0%	35.9%
▶ attr_table_aggregator.cpp	1,160,001,740	1,738,002,607	0.667	3.0%	0.4%	57.8%	38.8%
▶ record_impl_sqlite.hpp	1,080,001,620	1,956,002,934	0.552	11.1%	3.2%	45.4%	40.3%
▶ iterator.h	748,001,122	1,772,002,658	0.422	17.4%	3.3%	26.5%	52.8%
▶ timeline_table_impl_sqlite	698,001,047	1,800,002,700	0.388	15.8%	0.0%	0.0%	99.6%
▶ xstring	634,000,951	844,001,266	0.751	45.0%	16.6%	14.0%	24.4%

ICache Misses (ICache ミス) メトリックに移動すると、`sqlite3.c` ファイルの値も高いことが分かります。

Module / Source File	Front-End Bound						
	Front-End Latency						Front-End Bandwidth
	ICache Misses	ITLB Overhead	Branch Resteers	DSB Switches	Length Changing...	MS Switches	
▼ amplxe_dbinterface_sqlite	7.3%	2.8%	4.2%	3.3%	0.0%	2.0%	11.1%
▶ sqlite3.c	9.3%	3.2%	4.0%	3.4%	0.0%	2.3%	13.1%
▶ [Unknown source file]	9.8%	2.1%	5.9%	2.9%	0.0%	0.0%	19.5%
▶ vector	5.9%	0.8%	3.0%	1.0%	0.0%	2.0%	5.7%
▶ attr_table_caches.hpp	2.5%	1.8%	1.2%	2.5%	0.0%	2.5%	1.5%
▶ attr_table_aggregator.cp	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	3.0%
▶ record_impl_sqlite.hpp	0.0%	1.9%	1.9%	1.9%	0.0%	0.0%	3.7%

PGO オプションを指定してコードを再コンパイルする

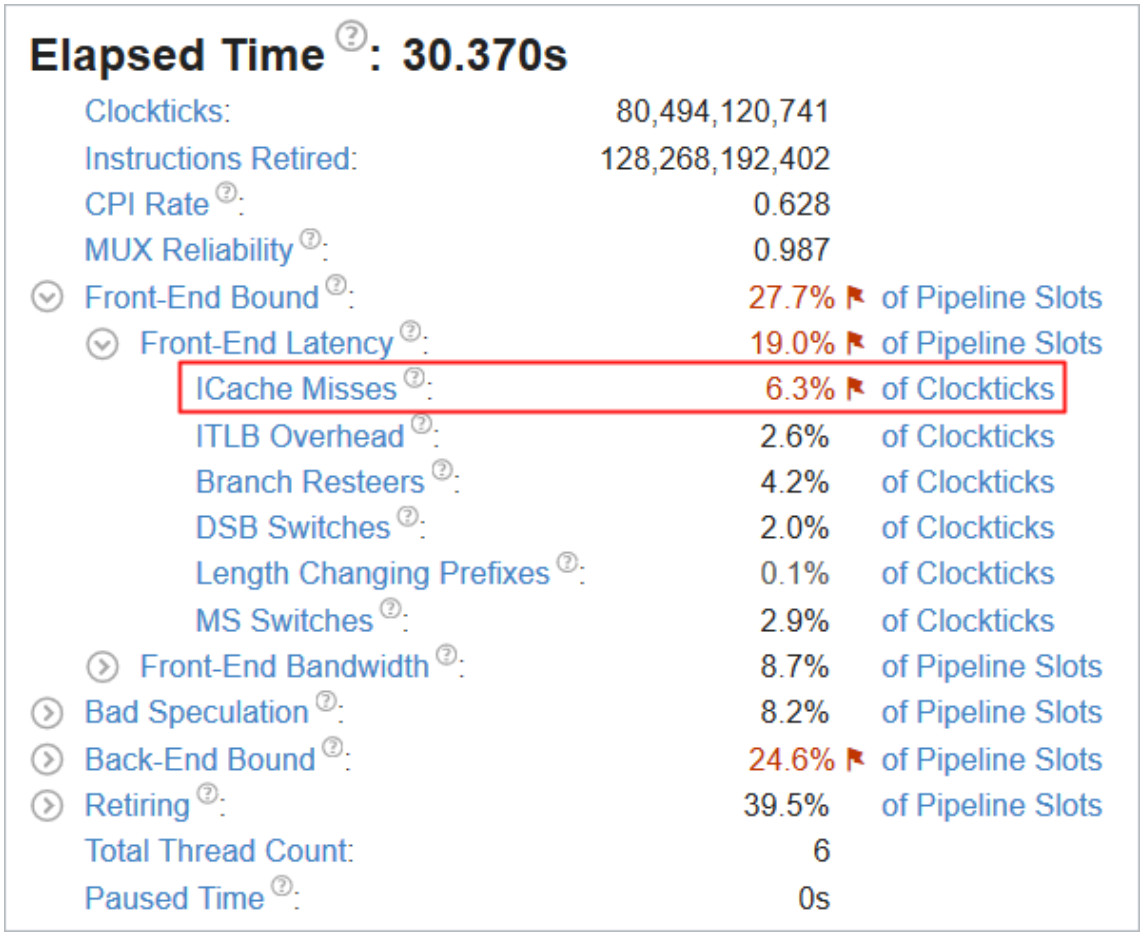
インテル® C++ コンパイラーを使用して、プロファイルに基づく最適化 (PGO) を `sqlite` ライブラリーに適用します。

1. `/Qprof-gen` オプションを指定してコードを再コンパイルします。
2. ベンチマークを実行します。
3. `/Qprof-use` オプションを指定してコードを再コンパイルします。

詳細は、「プロファイルに基づく最適化の概要」(英語) を参照してください。

最適化を確認する

最適化したコードで全般解析を再度実行します。新しい結果では、Elapsed Time (経過時間) は 30.3 秒になり、オリジナルの 31.5 秒からパフォーマンスが約 4% 向上しました。



Elapsed Time: 30.370s

Clockticks:	80,494,120,741	
Instructions Retired:	128,268,192,402	
CPI Rate:	0.628	
MUX Reliability:	0.987	
Front-End Bound:	27.7%	of Pipeline Slots
Front-End Latency:	19.0%	of Pipeline Slots
ICache Misses:	6.3%	of Clockticks
ITLB Overhead:	2.6%	of Clockticks
Branch Resteers:	4.2%	of Clockticks
DSB Switches:	2.0%	of Clockticks
Length Changing Prefixes:	0.1%	of Clockticks
MS Switches:	2.9%	of Clockticks
Front-End Bandwidth:	8.7%	of Pipeline Slots
Bad Speculation:	8.2%	of Pipeline Slots
Back-End Bound:	24.6%	of Pipeline Slots
Retiring:	39.5%	of Pipeline Slots
Total Thread Count:	6	
Paused Time:	0s	

`sqlite` ライブラリーで ICache ミスによりストールしていたクロック数は 9.3% から 6.4% に減りました。

Module / Source File	Front-End Bound						
	Front-End Latency						Front-End Bandwidth
	ICache Misses	ITLB Overhead	Branch Resteers	DSB Switches	Length Changing...	MS Switches	
amplxe_dbinterface_sqlite	5.5%	1.9%	4.0%	2.6%	0.0%	0.9%	9.7%
sqlite3.c	6.4%	2.1%	4.9%	2.7%	0.0%	0.9%	8.3%
vector	3.0%	1.5%	4.0%	4.0%	0.0%	0.0%	5.6%
[Unknown source file]	5.7%	2.3%	2.3%	1.1%	0.0%	0.0%	12.2%
attr_table_caches.hpp	1.2%	0.5%	1.2%	1.2%	0.0%	2.5%	6.8%
attr_table_aggregator.cp	0.0%	0.2%	0.0%	0.0%	0.0%	0.0%	2.1%
record_impl_sqlite.hpp	5.2%	1.0%	1.7%	0.0%	0.0%	0.0%	2.6%

関連情報

-
- ハードウェア問題のマイクロアーキテクチャー全般解析 (英語)
- トップダウン・マイクロアーキテクチャー解析法

非効率な同期

このレシピは、スタック収集を有効にしてインテル® VTune™ Amplifier の高度な hotspot 解析を実行し、コードの非効率な同期を特定する方法を説明します。

- 使用するもの
- 手順:
    1. スタック収集を有効にして高度な hotspot 解析を実行する
    2. タイムラインで同期を見つける
    3. 平均待機メトリックを解析する
    4. 同期コンテキスト・スイッチを解析する

注

高度な hotspot 解析は、インテル® VTune™ Amplifier 2019 で汎用の hotspot 解析 (英語) に統合されました。ハードウェア・イベントベース・サンプリング収集モードで利用できます。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- アプリケーション: `sample.exe` (OpenMP* ランタイムを使用)。このアプリケーションはデモ用であり、ダウンロードすることはできません。
- **パフォーマンス解析ツール:** インテル® VTune™ Amplifier 2017: 高度な hotspot 解析

注

    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Microsoft* Windows* 8
- **CPU:** インテル® プロセッサー (開発コード名 Skylake)

スタック収集を有効にして高度な hotspot 解析を実行する

インテル® VTune™ Amplifier を起動して解析するプロジェクトを設定します。

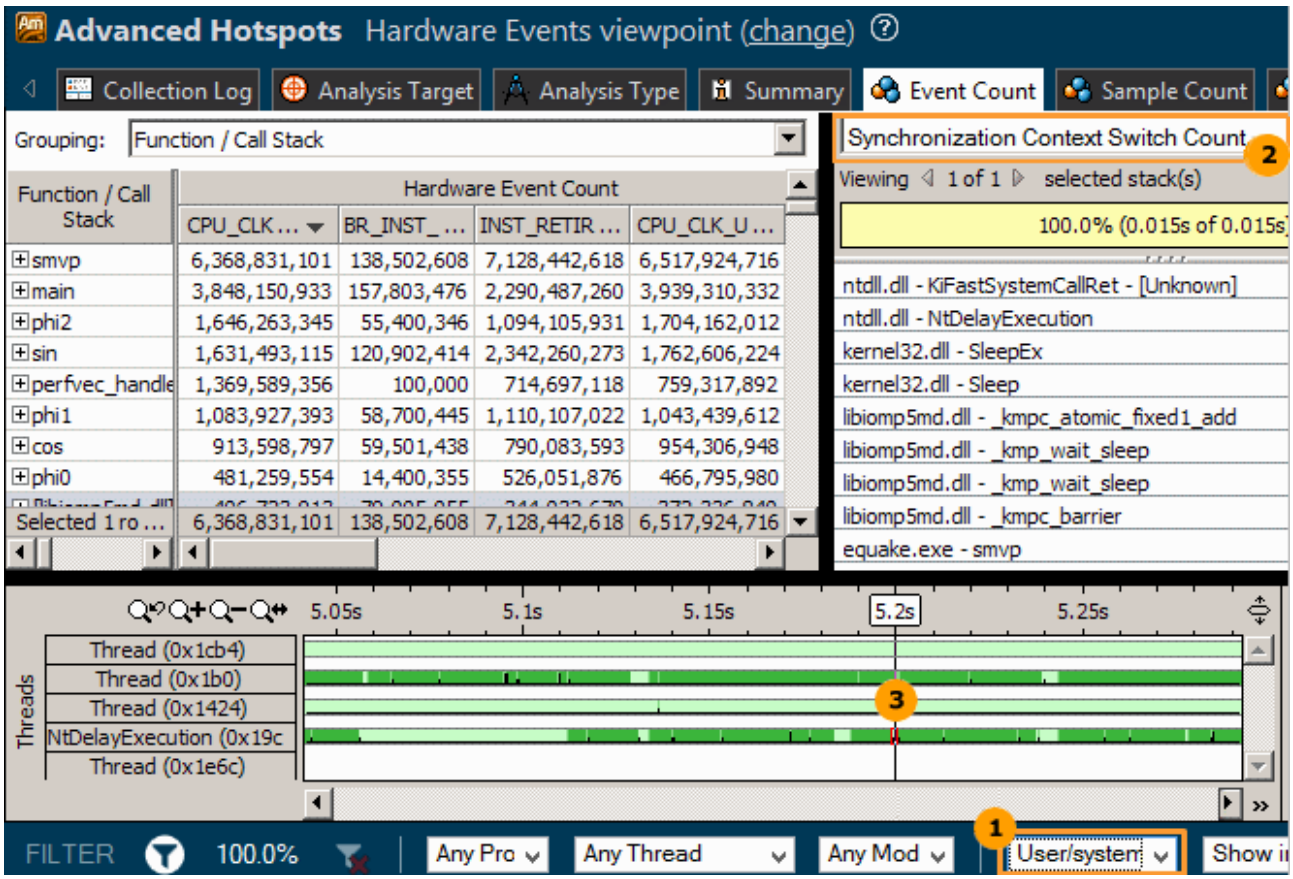
1. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `sqlite`) を指定します。

2. **[Analysis Target (解析ターゲット)]** ウィンドウで、ホストベースの解析として **[local host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、右ペインで解析するアプリケーションを指定します。
4. 右の **[Choose Analysis (解析の選択)]** ボタンをクリックし、**[Algorithm Analysis (アルゴリズム解析)] > [Advanced Hotspots (高度な hotspot)]** を選択して、**[Hotspots and stacks (hotspot とスタック)]** オプションを選択します。
5. **[Start (開始)]** をクリックします。

   インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

タイムラインの同期を見つける

[Hardware Event (ハードウェア・イベント)] ビューポイントで、解析中に収集されたデータを開きます。



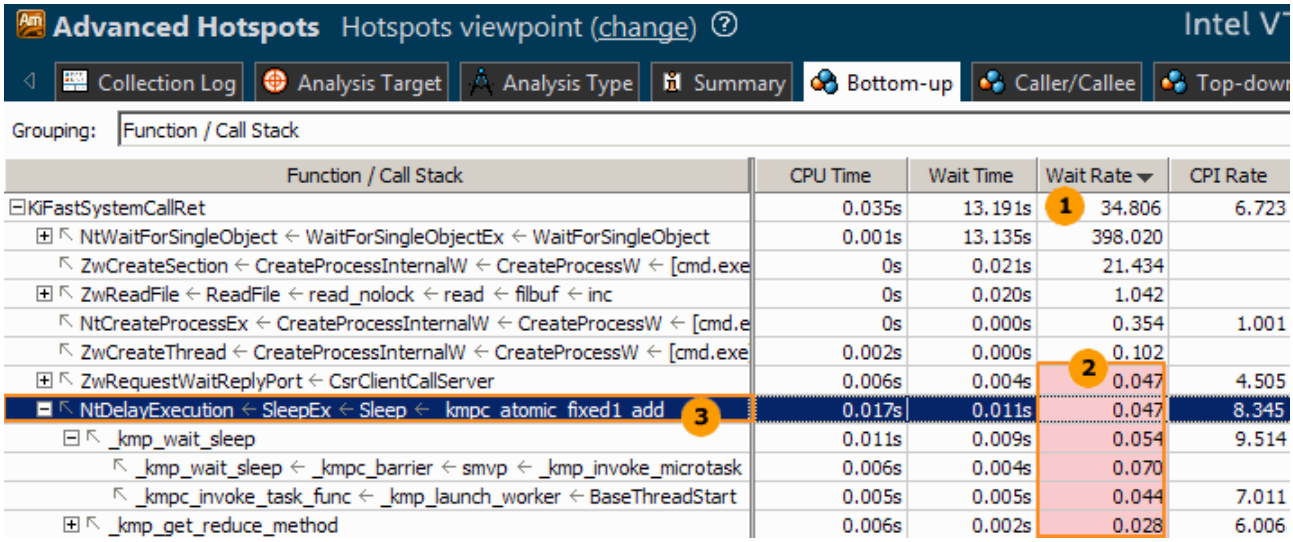
1 **[User/system functions (ユーザー/システム関数)]** コールスタック・モードを選択して、**[Call Stack (コールスタック)]** ペインにユーザー関数とシステム関数の両方を表示します。

2 **[Call Stack (コールスタック)]** ペインで、ドロップダウン・メニューから **[Synchronization Context Switch Count (同期コンテキスト・スイッチ・カウント)]** タイプを選択して、**[Timeline (タイムライン)]** ペインで選択した同期コンテキスト・スイッチのコールスタックを確認します。

3 タイムラインの頻繁な同期を見つけて、コンテキスト・スイッチの上にカーソルを移動します。ツールヒントに詳細が表示されます。例えば、上記の高度な hotspot 解析の結果では、`NtDelayExecution` スレッドに同期が原因の大量のコンテキスト・スイッチが含まれています。タイムラインのコンテキスト・スイッチを選択すると、**[Call Stack (コールスタック)]** ペインが更新され、スレッド実行が中断された呼び出しシーケンスが表示されます。

平均待機メトリックを解析する

(change) リンクをクリックして **[Hotspots (hotspot)]** ビューポイントを開きます。



1 同期コンテキスト・スイッチあたりの平均待機時間 (秒単位) である、[Wait Rate (待機レート)] メトリックデータを解析します。このメトリックは、非効率で頻繁な同期および消費電力問題を特定するのに役立ちます。

2 インテル® VTune™ プロファイラーは、低い待機レートメトリック値 (1 ミリ秒未満) をパフォーマンス問題として判断し、ピンクでハイライトします。これらの値は、スレッド間の競合およびシステム API の非効率な使用を示すことがあるためです。

3 待機時間が短く CPU 時間が長い (すべてのシステムコール時間の約半分の) 同期スタックを特定し、ダブルクリックして hotspot 関数のソースコードを調べます。

同期コンテキスト・スイッチを解析する

(change) リンクをクリックして **[Hardware Event (ハードウェア・イベント)]** ビューポイントを開きます。デフォルトでは、**[Event Count (イベントカウント)]** グリッドはクロック数イベントでソートされます。最も CPU 時間 (クロック数) がかかっていて、最も頻繁に同期を行っているホットな関数を特定します。

Function / Call Stack	Event Type	Context Switch Co
	CPU_CLK_UNHALTE...	Synchronization
InterpolatorKic<unsigned char,float>::InterpolateN	585,912,744	23
InterpolatorKic<unsigned char,unsigned char>::Do ← _kmp_invoke_microtask ← _kmpc_invoke_	585,912,744	23
_kmpc_invoke_task_func ← _kmp_launch_worker ← BaseThreadInitThunk ← RtlUserThreadSt	523,897,299	23
_kmp_fork_call ← _kmpc_fork_call ← InterpolatorKic<unsigned char,unsigned char>::Do ← C	62,015,445	0
KImage::getData	181,995,760	8
NtWaitForSingleObject	173,576,490	17,315
WaitForSingleObjectEx ← _kmp_launch_monitor ← BaseThreadInitThunk ← RtlUserThreadStart	173,576,490	17,315

このサンプル OpenMP* アプリケーションでは、インテル® VTune™ Amplifier は `InterpolateN` 関数を OpenMP* 領域から呼び出された計算 hotspot として表示しています。OpenMP* ランタイム内部の `WaitForSingleObject` で競合が発生し、約 30% のパフォーマンス損失 (待機関数のクロック数/hotspot 関数のクロック数) が発生していることも分かります。

`InterpolateN` 関数をダブルクリックしてソースコードを表示し、非効率な同期の原因を特定します。

```
for(i = 0; i < block_no; i++)
{
    #pragma omp parallel for
    for(j = 0; j < lines_in_block; j++)
    {
        /// do processing
    } /// implicit barrier causing contention and overhead
}
```

サンプル・アプリケーションのコード解析により、ブロック行でピクチャーを処理して各ブロックを個別に並列化するために過度の OpenMP* バリアが追加されていることが判明しました。この問題を解決するには、`nowait` 節を使用するか、ピクチャー全体に `parallel_for` を適用して、動的ワーク・スケジューリングを使用します。

Function / Call Stack	Event Type	Context Switch Count
	CPU_CLK_UNHALTE...	Synchronization
NtDelayExecution	286,509,847	26,997
SleepEx ← _kmp_invoke_microtask ← _kmp_wait_yield_4 ← _kmp_acquire_lock	286,509,847	26,997
NtWaitForSingleObject	248,806,404	7,565
WaitForSingleObjectEx	241,934,524	7,541
_kmp_launch_worker	206,421,435	5,982
_kmp_wait_sleep ← _kmpc_invoke_task_func ← _kmp_launch_worker ←	85,531,355	3,023
_kmp_get_reduce_method ← _kmpc_invoke_task_func	120,890,080	2,959
_kmp_launch_monitor ← BaseThreadInitThunk ← RtlUserThreadStart	22,055,633	1,496
QThreadStorageData::finish(void * ptr64 * ptr64) ← QMutex::lock(void)	13,457,456	63
RtlDeNormalizeProcessParams ← RtlDeNormalizeProcessParams	6,871,880	24
InterpolatorPixel<unsigned char>::Do	15,302,558,732	273

最適化した結果では、`Sleep()` の競合の相対的なコストが低くなりました (26,997)。

1 つの `parallel_for` と動的ワーク・スケジューリングを `WaitForSingleObject` 関数に使用することで、競合とパフォーマンスへの悪影響を 1% 未満に減らすことができました。

2 つ目の最適化した結果では、`Sleep()` 関数でも多くの競合が発生していることが分かります (同期コンテキスト・スイッチ・メトリックが 26,997)。しかし、その実行時間を確認すると、実行時間は最上位の hotspot (表示

されていません) の 2% 以内であり、それほど重要ではありません。ただし、多くのプロセッサー上でアプリケーションを実行した場合、この関数が問題になる可能性があります。

注

最初の (最適化前の) サンプルデータ収集セッションは一定の時間間隔で行われたものです。最適化バージョンでは、アプリケーションが制限なしで実行されています。

関連情報

- スタックを含むハードウェア・イベントベースのサンプリング収集 (英語)

非効率な TCP/IP 同期

このレシピは、タスク収集を有効にしてインテル® VTune™ Amplifier のロックと待機解析を実行し、コードの非効率な TCP/IP 同期を特定する方法を説明します。

コンテンツ・エキスパート: Kirill Uhanov (英語)

- 使用するもの
- 手順:
    1. ロックと待機解析を実行する
    2. タイムラインで同期の遅延を見つける
    3. ITT API カウンターを使用して `send`/`receive` バッファーサイズを検出する
    4. 非効率な TCP/IP 同期の原因を特定する

注

ロックと待機解析は、インテル® VTune™ Amplifier 2019 でスレッド解析に改名されました。

使用するもの

- **アプリケーション**: TCP ソケット通信を使用するクライアント/サーバー・アプリケーション
- **パフォーマンス解析ツール:** インテル® VTune™ Amplifier 2018: ロックと待機解析
- **サーバー・オペレーティング・システム:** Microsoft* Windows Server* 2016
- **クライアント・オペレーティング・システム**: Linux*

ロックと待機解析を実行する

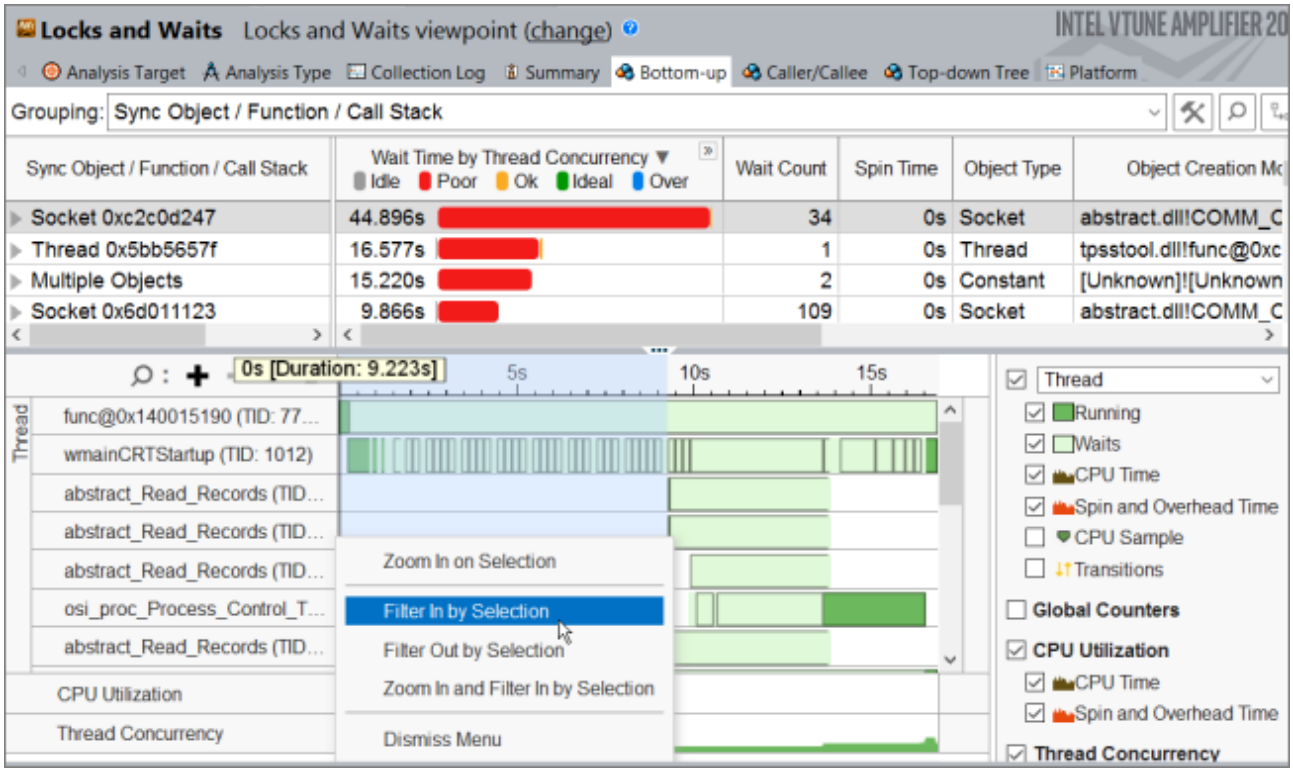
クライアント・アプリケーションのウォームアップに時間がかかる場合、ロックと待機解析を実行して、同期オブジェクトごとの待機統計を調査することを検討してください。

1. ツールバーの **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクト (例: `tcpip_delays`) を作成します。
2. **[Analysis Target (解析ターゲット)]** ウィンドウで、ホストベースの解析として **[local host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、右ペインで解析するアプリケーションを指定します。
4. 右の **[Choose Analysis (解析の選択)]** ボタンをクリックし、**[Algorithm Analysis (アルゴリズム解析)] > [Locks and Waits (ロックと待機)]** を選択します。
5. **[Start (開始)]** をクリックします。

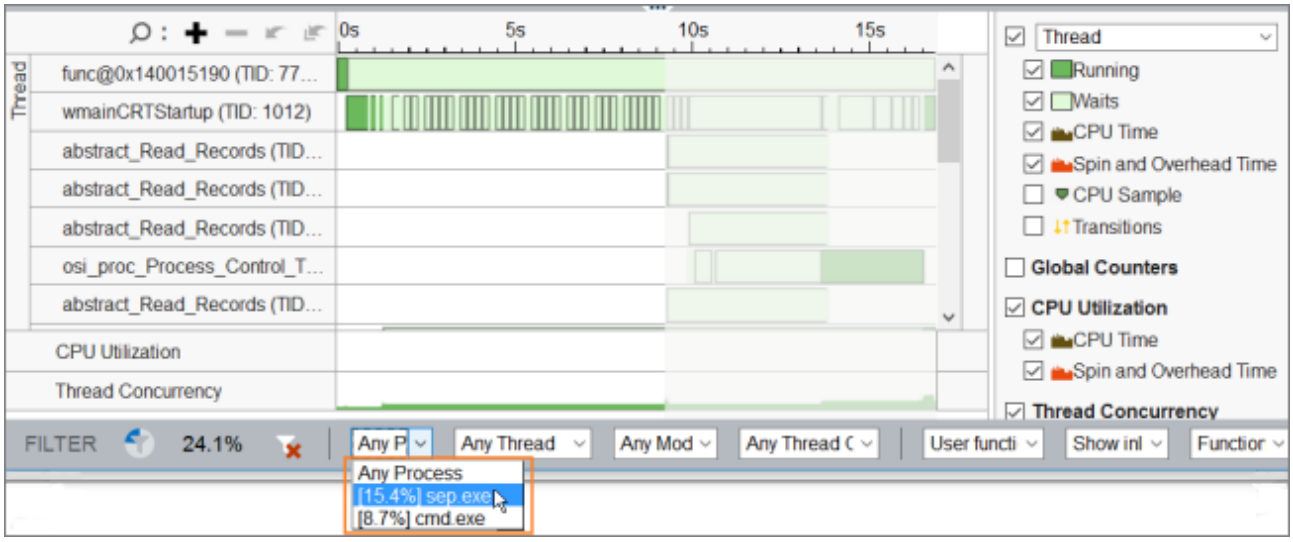
インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

タイムラインで同期の遅延を見つける

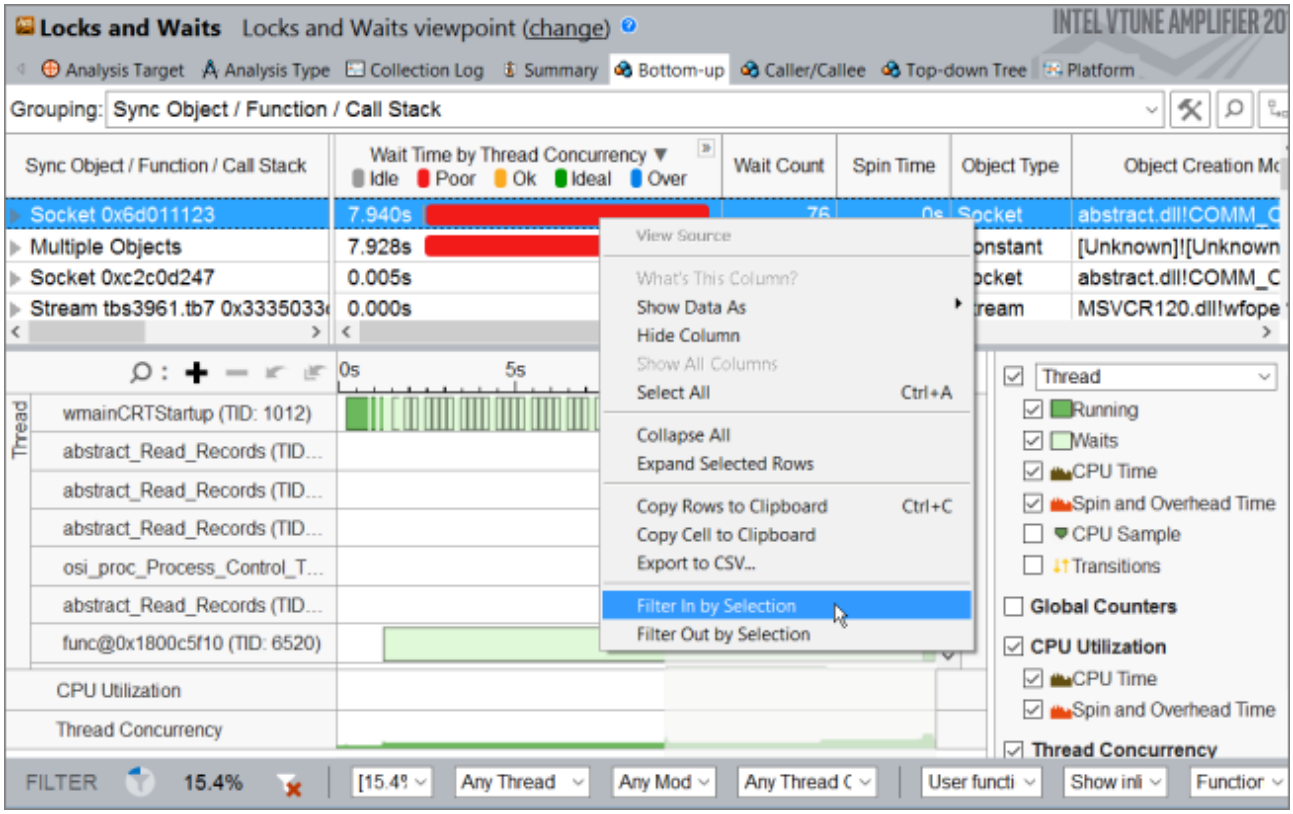
収集結果を開いて **[Bottom-up (ボトムアップ)]** タブをクリックし、同期オブジェクトごとのパフォーマンスの詳細を表示します。**[Timeline (タイムライン)]** ペインでは、テスト・アプリケーションが実行を開始すると、複数の同期の遅延が表示されます。これらの起動時の遅延を引き起こす同期オブジェクトを特定するには、ドラッグアンドドロップで最初の 9 秒間を選択し、コンテキスト・メニューから **[Filter In by Selection (選択してフィルターイン)]** を使用します。



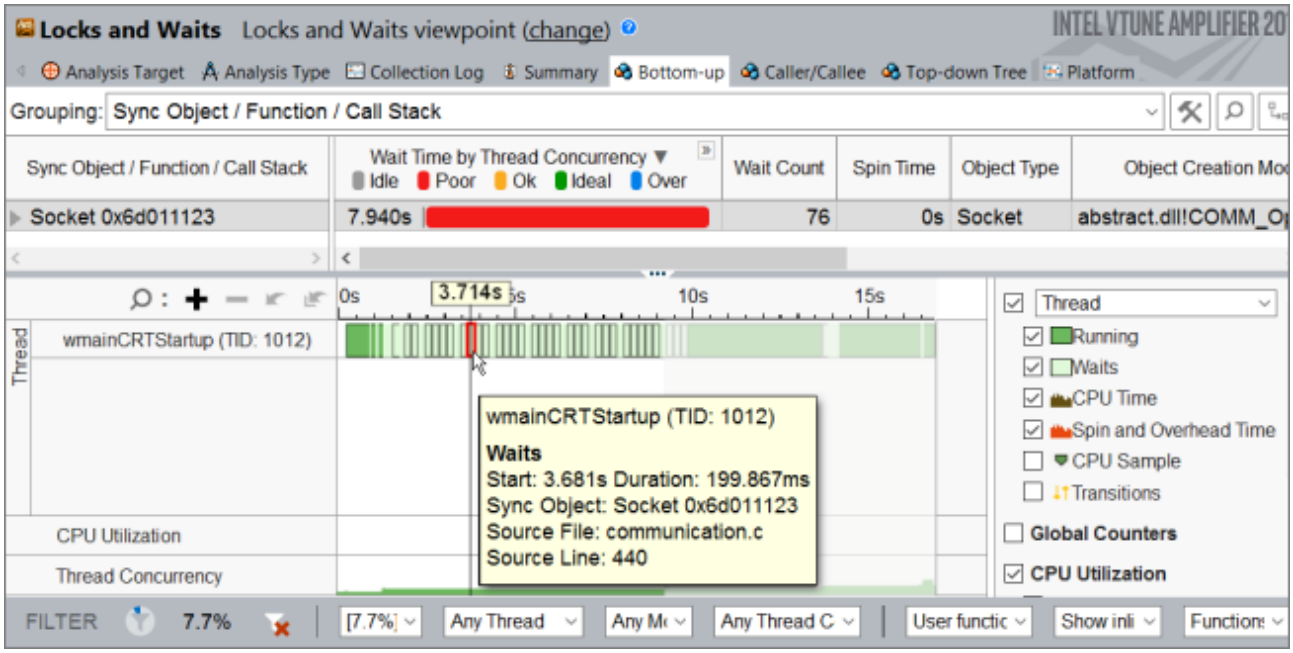
フィルターバーの **[Process (プロセス)]** メニューで、ホストの通信プロセス別にフィルターインします。



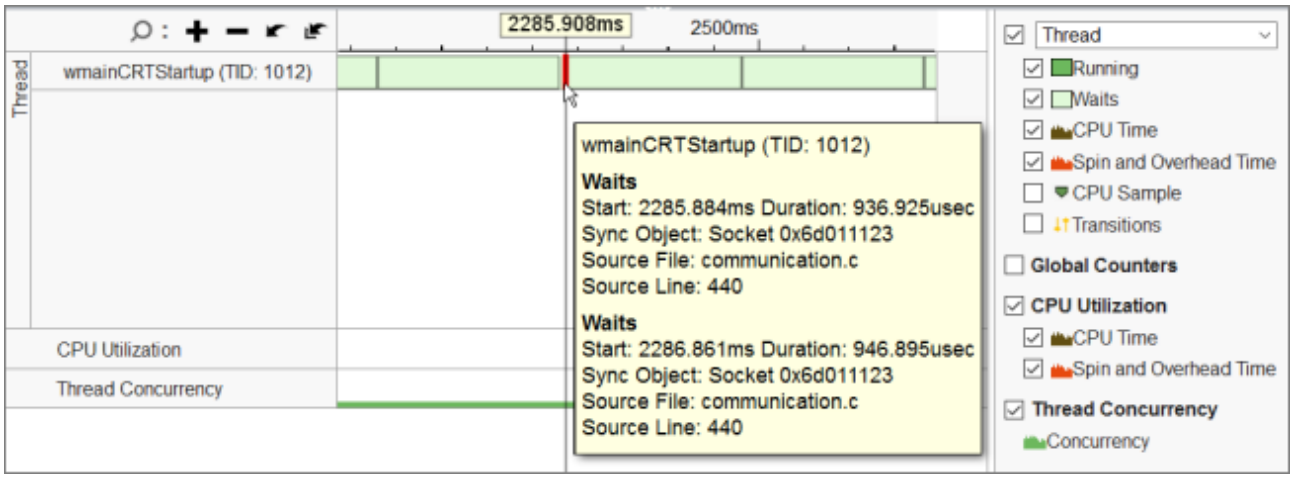
そして、選択した期間内で待機時間が最も大きい `Socket` 同期オブジェクトを選択し、**[Filter In by Selection (選択してフィルターイン)]** メニューを使用してデータをフィルターインします。



`Socket` 同期オブジェクトの待機時間を調査するには、タイムラインに注目します。



[Zoom In (ズームイン)] ボタンをクリックすると、高速と低速の 2 種類のソケット待機が表示されます。低速の (長い) ソケット待機は約 200 ミリ秒で、高速の (短い) ソケット待機は約 937 マイクロ秒です。



高速な待機と低速な待機の原因を理解するには、ITT カウンターですべての `send/receive` 呼び出しをラップして、`send/receive` バイトを計算します。

ITT API カウンターを使用して `send/receive` バッファーサイズを検出する

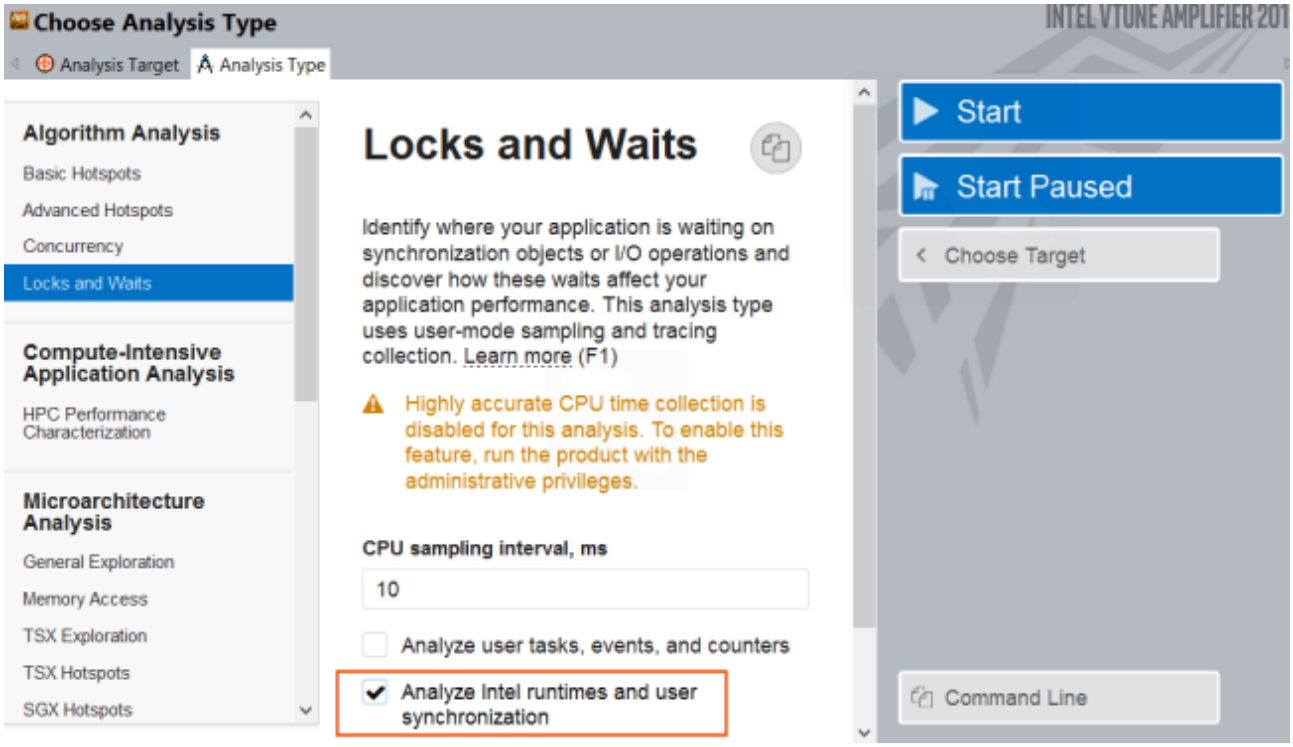
インストルメンテーションとトレース・テクノロジー (ITT) API を使用して `send/receive` 呼び出しをトレースするには、次の操作を行います。

1. API ヘッダーとライブラリーにアクセスできるようにシステムを設定 (英語) します。
2. ITT API ヘッダーをソースファイルにインクルードして、`<vtune-install-dir>\[lib64 or lib32]\libittnotify.lib` スタティック・ライブラリーをアプリケーションにリンクします。
3. ITT カウンターを使用して `send/receive` 呼び出しをラップします。

```
#include <ittnotify.h>

__itt_domain*                                                    g_domain
=__itt_domain_createA("com.intel.vtune.tests.userapi_counters");
__itt_counter g_sendCounter = __itt_counter_create_typedA("send_header",
g_domain->nameA, __itt_metadata_s32);
__itt_counter                        g_sendCounterArgs                    =
__itt_counter_create_typedA("send_args",               g_domain->nameA,
__itt_metadata_s32);
__itt_counter                         g_recieveCounter                    =
__itt_counter_create_typedA("recieve_header",          g_domain->nameA,
__itt_metadata_s32);
__itt_counter                      g_recieveCounterCtrl                   =
__itt_counter_create_typedA("recieve_ctrl",            g_domain->nameA,
__itt_metadata_s32);
__itt_counter g_incDecCounter = __itt_counter_createA("inc_dec_counter",
g_domain->nameA);

.....

    sent_bytes = send(...);
```

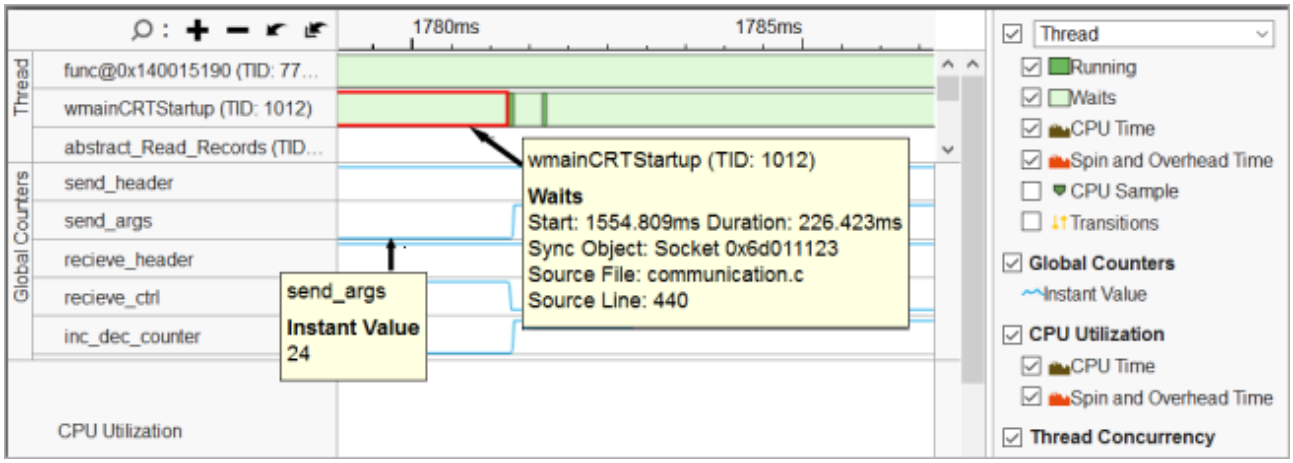
```
        __itt_counter_set_value(g_sendCounter, &sent_bytes);
.....
        sent_bytes = send(...);
        __itt_counter_set_value(g_sendCounterArgs, &sent_bytes);
.....
        while(data_transferred < header_size)) {
            if ((data_size = recv(...) < 0) {
                                    .....
        }
        __itt_counter_set_value(g_recieveCounter, &data_transferred);
.....

            while(data_transferred < data_size) {
                if ((data_size = recv(...) < 0) {
                                                    ....
                                                    }
            }
        }
        __itt_counter_set_value(g_recieveCounterCtrl, &data_transferred);
```

アプリケーションを再コンパイルして、**[Analyze user tasks, events, and counters (ユーザータスク、イベント、およびカウンターを解析)]** オプションを有効にしてロックと待機解析を再度実行します。

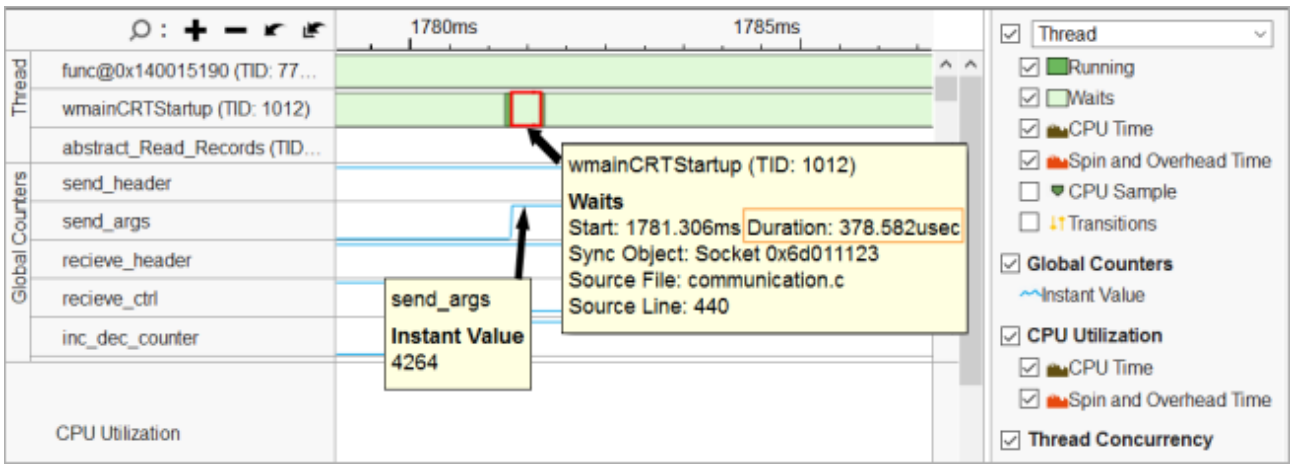


非効率な TCP/IP 同期の原因を特定する

新しい結果では、[Timeline (タイムライン)] ペインに ITT API を介して収集された `send`/`receive` 呼び出しの分布を表示する **[Global Counters (グローバルカウンター)]** が追加されます。マウスでスレッドの待機やカウンター値をポイントすると、対応するカウンターのインスタント値が表示されます。この値は、長い (低速な) 待機では小さくなります。



そして、短い (高速な) 待機では大きくなります。

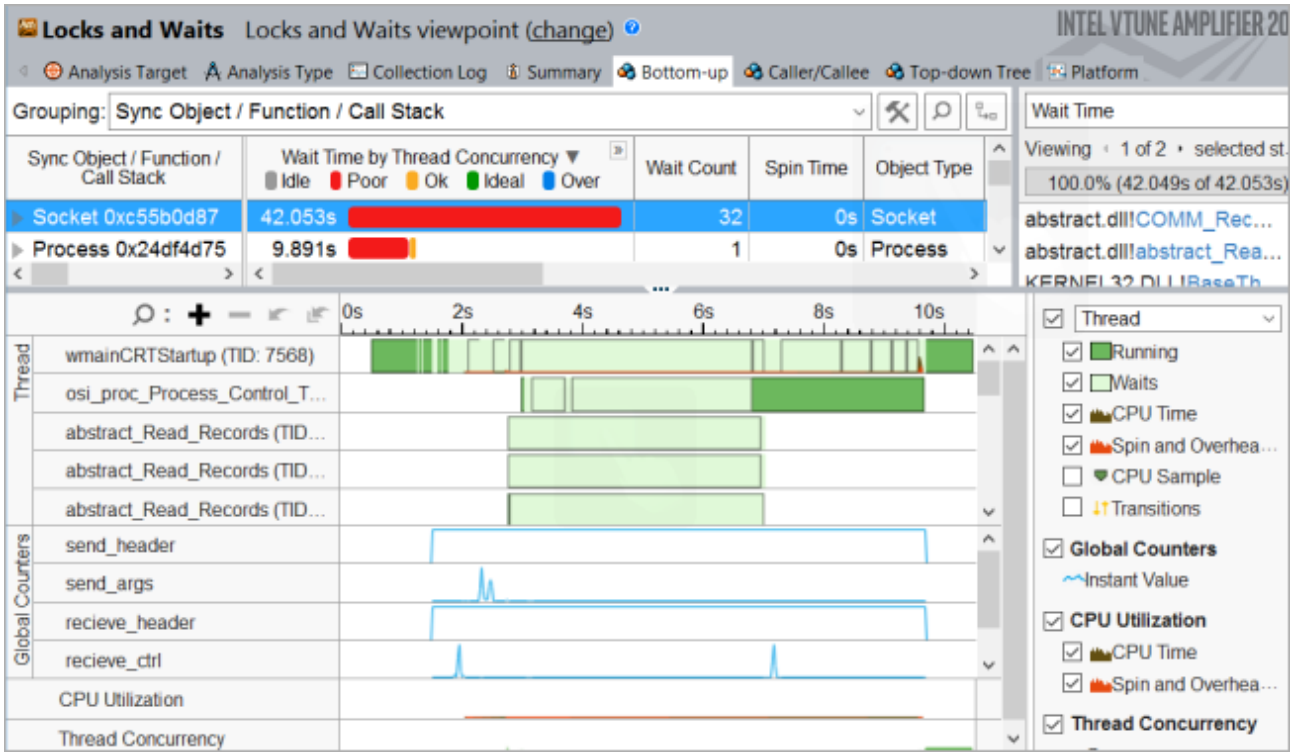


リモートターゲットの通信待機のプロファイルでは、対称的な結果となります。小さなサイズのバッファーでは待機時間が長くなり、十分なサイズのバッファーでは待機時間が短くなります。

このレシピでは、通信コマンドチャネルが分ります。ほとんどのコマンドはサイズが小さく、結果的に長い待機時間が発生します。

問題の原因は、小さなバッファーの待機時間を増やす `tcp ack` 遅延メカニズムにあります。

サーバー側の入力 (`setsockopt` (…, `SO_RCVBUF`, ..,)) バッファーを小さくすると、起動時間が 5 倍以上 (数十秒から数秒へ) 高速になります。



関連情報

- インストルメンテーションとトレース・テクノロジー API (英語)

I/O 問題: 高いレイテンシーと低い PCIe* 帯域幅

このレシピは、I/O 依存のサンプル・アプリケーションに対してインテル® VTune™ Amplifier のディスク I/O 解析を実行します。そして、PCIe* デバイス向けにアフィニティーを変更して、読み取りアクセスの帯域幅が向上するように最適化します。

コンテンツ・エキスパート: Roman Sudarikov (英語)

注

ディスク I/O 解析は、インテル® VTune™ Amplifier 2019 で入力と出力解析に改名されました。

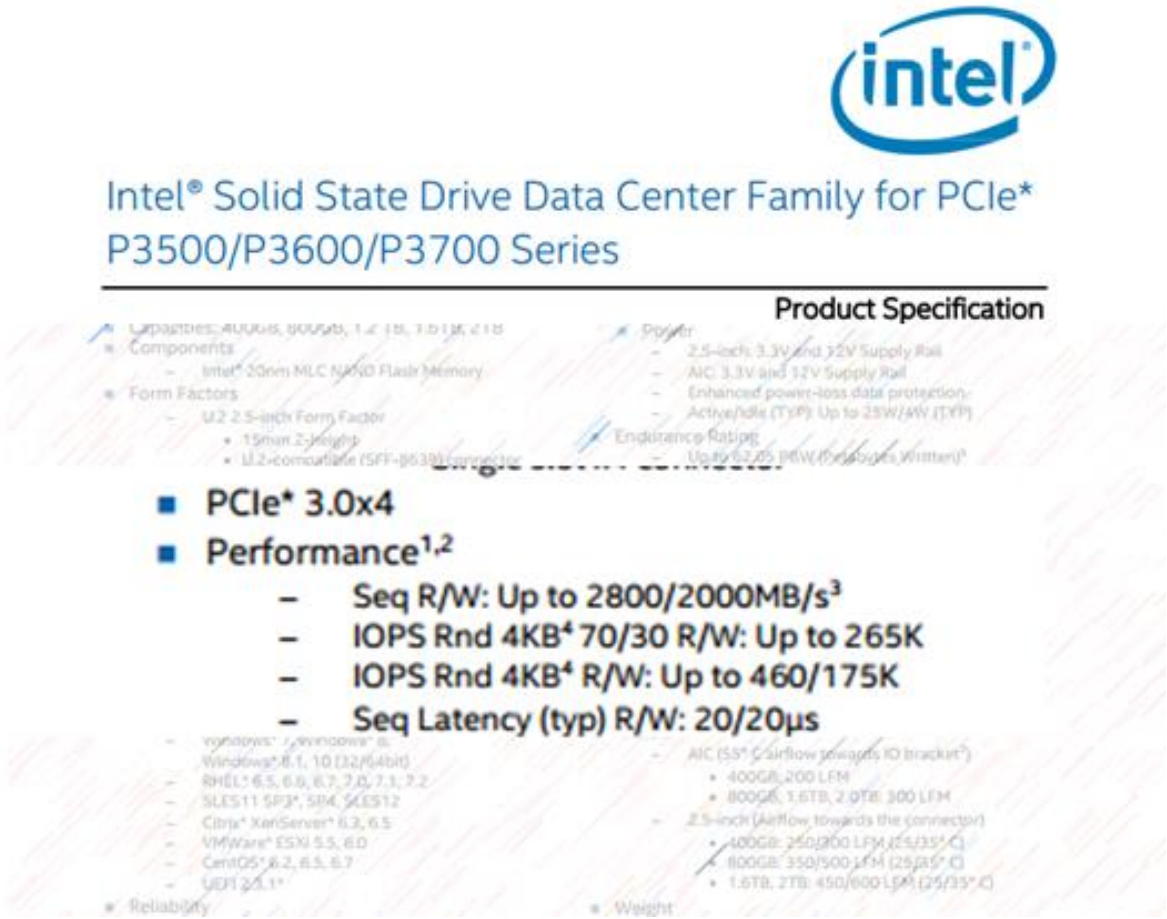
使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション**: 3 秒間に連続する 128K 読み取りアクセスを実行する `hdparm`。https://sourceforge.net/projects/hdparm (英語) から入手できます。
- **パフォーマンス解析ツール:**
    - インテル® VTune™ Amplifier 2018: ディスク I/O 解析

注

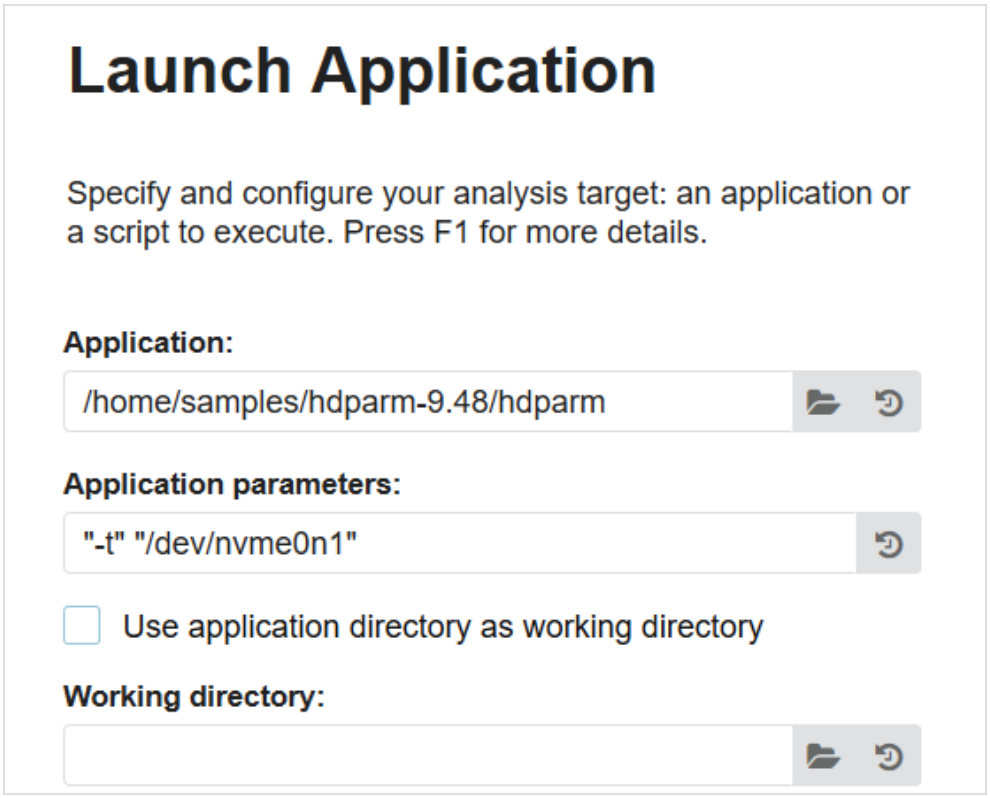
    - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Red Hat* Enterprise Linux* Server 7.2
- **CPU:** インテル® プロセッサー (開発コード名 Skylake)
- **I/O デバイス**: インテル® SSD DC P3500/P3600/P3700 シリーズ



ディスク I/O 解析を実行する

I/O 依存のアプリケーションでは、ディスク I/O 解析から始めることを推奨します。

1. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `hdparm`) を指定します。
2. **[Analysis Target (解析ターゲット)]** ウィンドウで、ホストベースの解析として **[local host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、右ペインで解析するアプリケーションを指定します。



4. 右の **[Choose Analysis (解析の選択)]** ボタンをクリックし、**[Platform Analysis (プラットフォーム解析)] > [Disk Input and Output (ディスク I/O)]** を選択して、**[Start (開始)]** をクリックします。

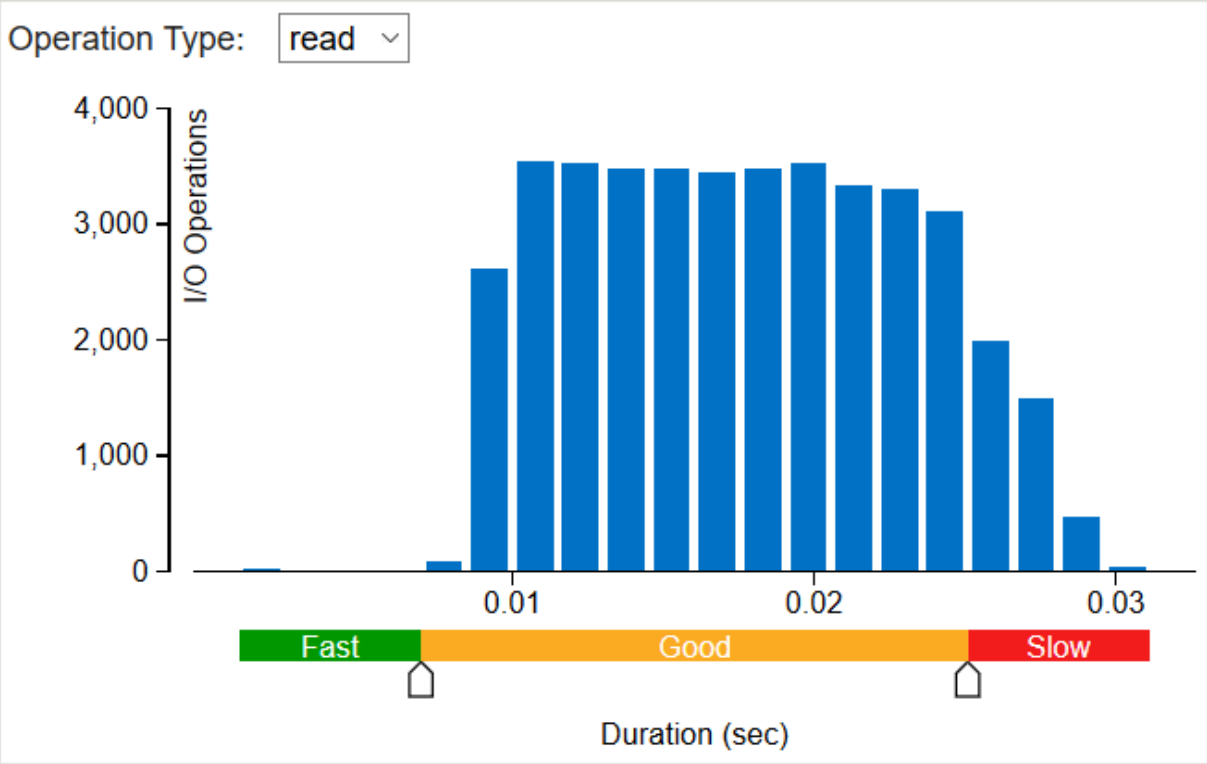
   インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

帯域幅とレイテンシーのメトリックを解析する

アプリケーション実行の統計が表示される **[Summary (サマリー)]** ビューから解析を始めます。I/O 効率の主要な指標である **[I/O Wait Time (I/O 待機時間)]** メトリックに注目します。

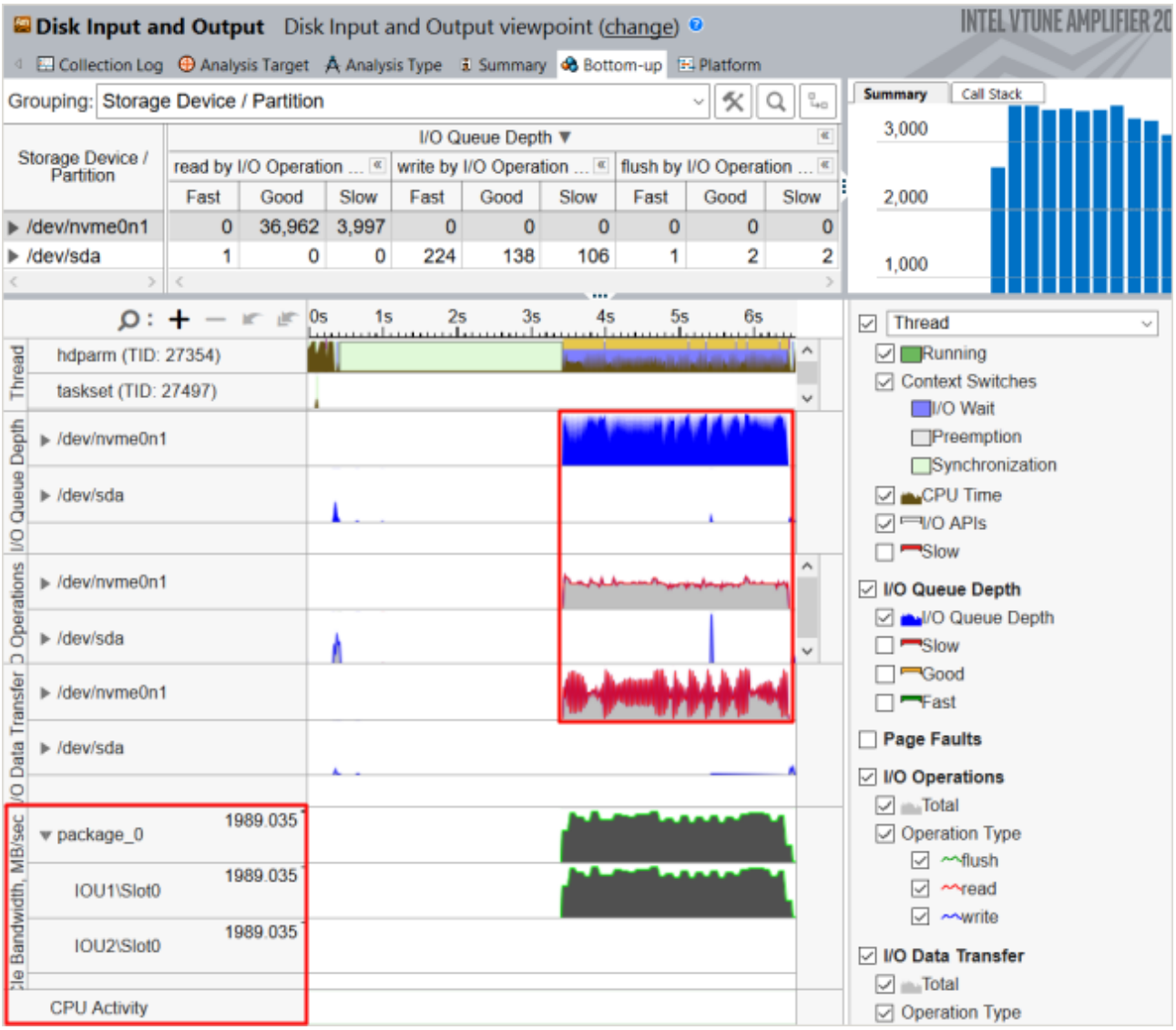
I/O 待機時間メトリックは、`hdparm` アプリケーションが経過時間の約 30% を I/O 待機に費やしていることを示しています。

ヒストグラムで `read` ディスク I/O 操作タイプを選択して、読み取りアクセスの時間分布を解析します。



不規則なフローは、通常、パフォーマンスの低下を示します。これは、デバイス仕様で宣言されている値 (20 マイクロ秒) よりも 3 桁大きい読み取りアクセスの値からも確認できます。

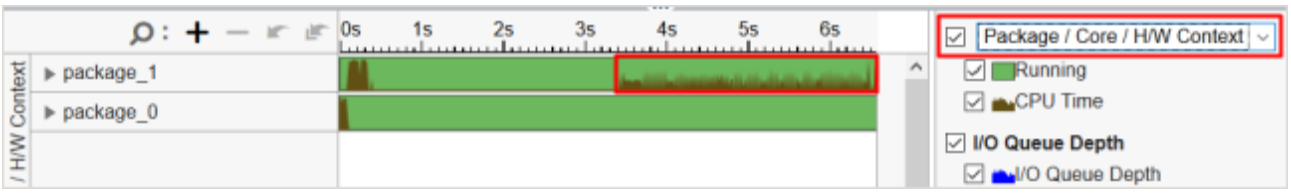
[Bottom-up (ボトムアップ)] ウィンドウに切り替えて **[Storage Device/Partition (ストレージデバイス/パーティション)]** グループレベルを適用します。タイムライン・データに注目します。



タイムライン・ビューの **[I/O Operations (I/O 操作)]** と **[Data Transfers (データ転送)]** セクションは、高い I/O 待機と不規則なデータフローを示しています。

[PCIe Bandwidth (PCIe* 帯域幅)] セクションは、デバイス (`package_0`) の読み取り帯域幅が、デバイス仕様で宣言されている値の約 65% しかないことを示しています。

タイムラインのグループレベルを **[Package / Core / H/W Context (パッケージ/コア/ハードウェア・コンテキスト)]** に変更して、アプリケーションのアフィニティーを調査します。

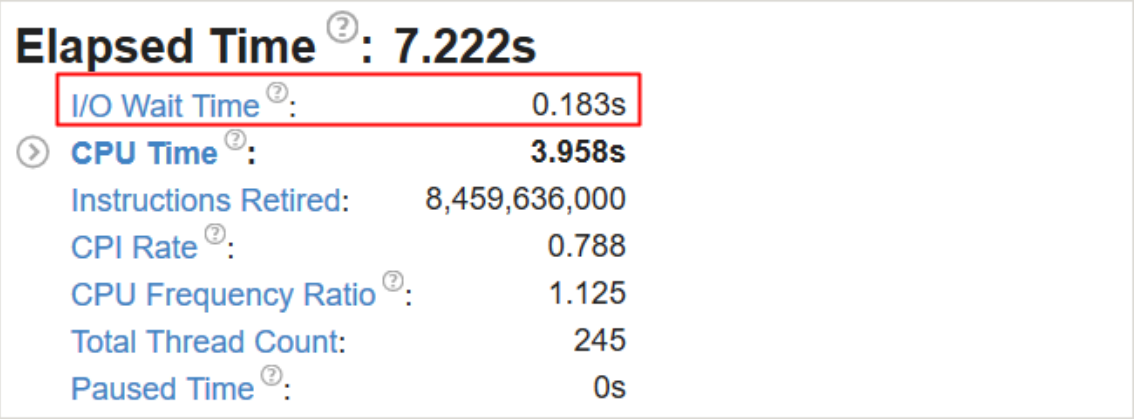


デバイスは `package_0` ですが、アプリケーションは `package_1` で実行していることが分かります。これが、高いレイテンシーと低い帯域幅の原因の可能性があります。

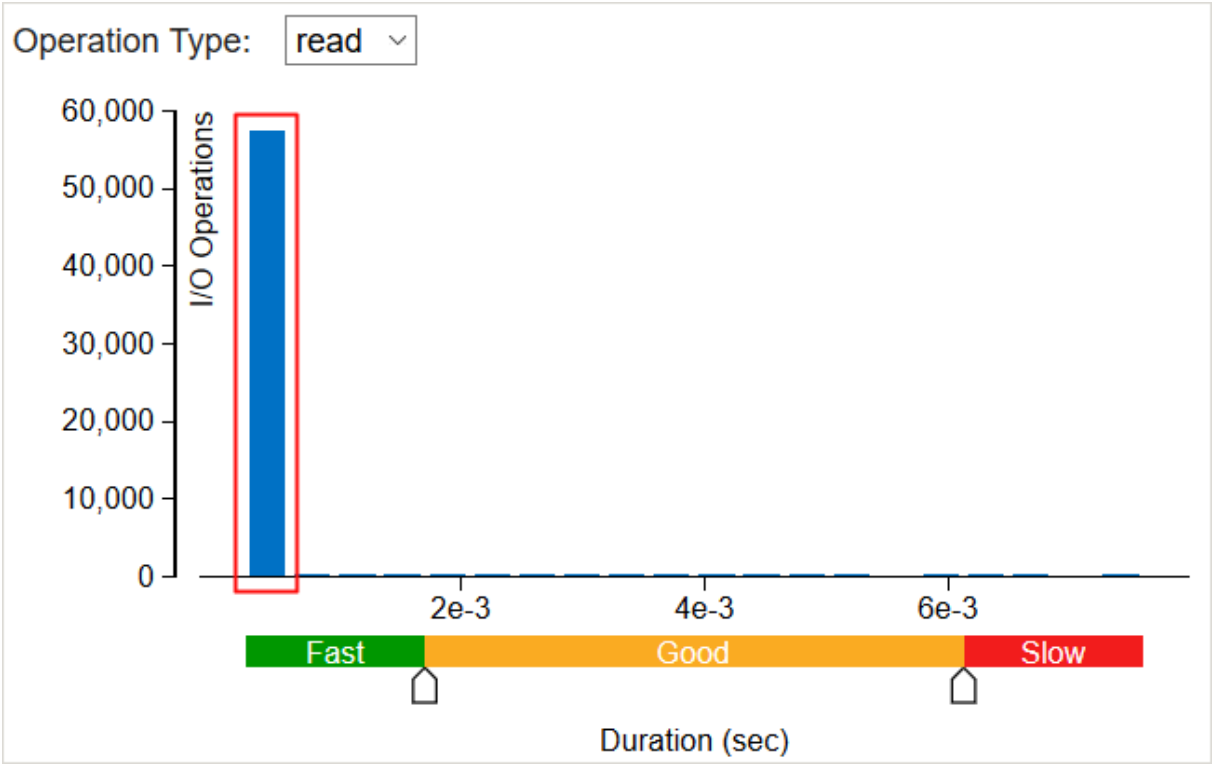
アプリケーションのアフィニティーを変更して解析を再度実行する

ワークロードとデバイスの設定を維持したまま検出された I/O 問題を解決するため、アプリケーションのアフィニティーを変更して、ディスク I/O 解析を再度実行します。

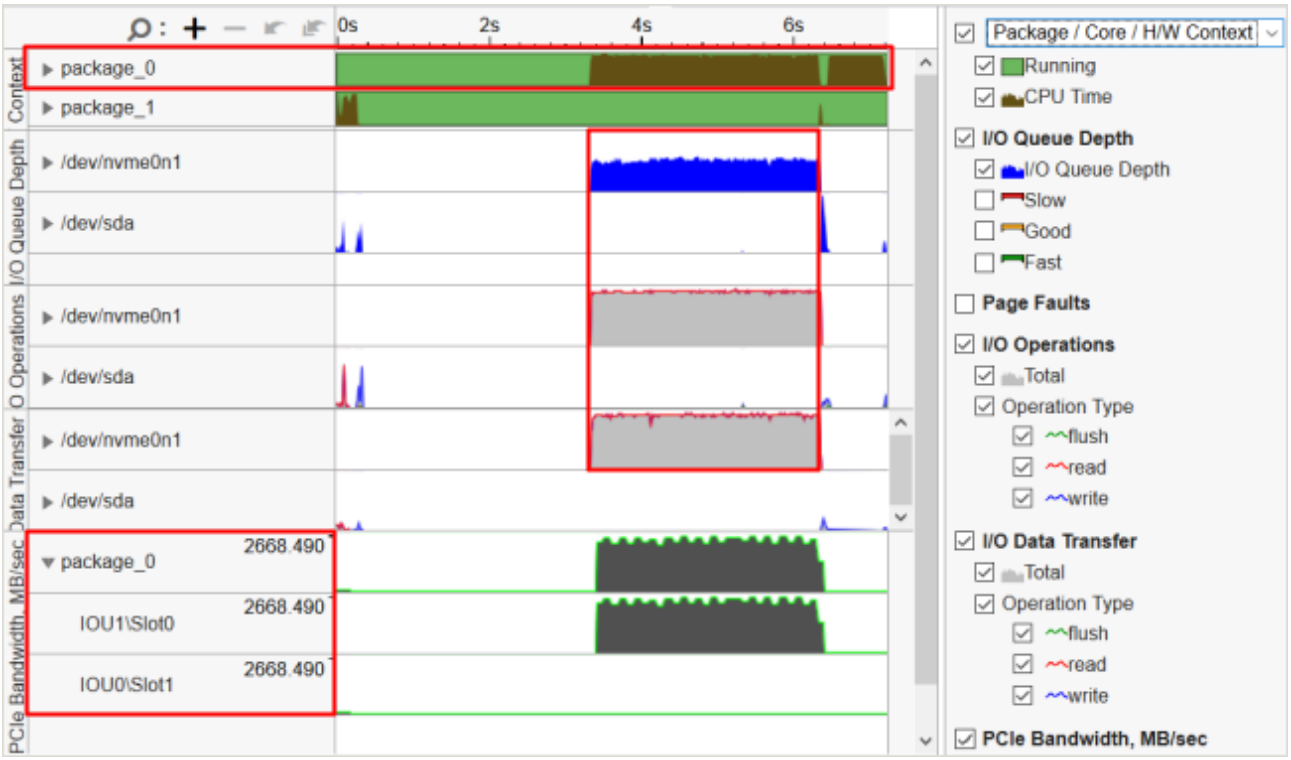
新しい結果は、アプリケーションの I/O 待機時間が経過時間の約 2% になったことを示しています。



ヒストグラムに読み取りアクセスの時間分布が表示されなくなりました。すべての I/O 操作がミリ秒未満で実行されています。



タイムライン・ビューでは、I/O 操作と I/O データ転送の規則的なデータフローが確認できます。これは、アフィニティーの最適化によりレイテンシーが軽減されたことを示しています。



さらに、この変更により、PCIe* 帯域幅が向上し、デバイス仕様で宣言されている値の約 93% になりました。

まとめ

PCIe* 帯域幅に依存するアプリケーションの I/O パフォーマンス解析に関して、次の点を説明しました。

- PCIe* デバイスの I/O ユニット (IOU) アフィニティーを決定する
- アプリケーションを I/O ユニットに適切に分配する
- デバイスの性能を理解する
- 合理的なパフォーマンス目標を設定する
- ディスク I/O 解析 (英語) を実行して低い帯域幅の I/O ソリューションをデバッグする

OS スレッド・マイグレーション

このレシピは、インテル® VTune™ Amplifier の高度な hotspot 解析を使用して NUMA アーキテクチャーの OS スレッド・マイグレーションを特定する手順を説明します。

注

高度な hotspot 解析は、インテル® VTune™ Amplifier 2019 で汎用の hotspot 解析 (英語) に統合されました。ハードウェア・イベントベース・サンプリング収集モードで利用できます。

現代の複雑なオペレーティング・システムは、スケジューラーを使用してアプリケーション・スレッド (ソフトウェア・スレッド) をプロセッサー・コアに割り当てます。スケジューラーは、システムステート、システムポリシーなどのさまざまな異なる要因に応じて、物理コア上のアプリケーション・スレッドの配置を選択します。ソフトウェア・スレッドは、スワップアウトされて待機状態になるまで、コアで一定時間実行されます。ソフトウェア・スレッドは、I/O によるブロックのようなさまざまな理由により待機します。利用可能な場合、別のソフトウェア・スレッドがこのコアで実行されます。オリジナルのソフトウェア・スレッドが再度実行可能になると、スケジューラーは、オリジナルのソフトウェア・スレッドが実行できるように別のソフトウェア・スレッドを別のコアに移動します。ソフトウェア・スレッドを移動すると、スレッドとすでにキャッシュにフェッチされたデータの関連付けが解消され、データアクセスのレイテンシーが大きくなるため、新しい計算アーキテクチャーでは問題が発生します。この問題は、各プロセッサーが個別のローカルメモリーを保持し、それらに直接アクセスする NUMA (Non Uniform Memory Access) アーキテクチャーではさらに大きくなります。NUMA アーキテクチャーでは、ソフトウェア・スレッドを別のコアに移動すると、以前のコアのローカルメモリーに格納されていたデータがリモートになり、メモリーアクセス時間が大幅に増加します。スレッド・マイグレーションはパフォーマンス低下の原因となるため、アプリケーションでスレッド・マイグレーションが発生しているかどうか確認することが重要です。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** OpenMP* テスト・アプリケーション。このアプリケーションはデモ用であり、ダウンロードすることはできません。
- **パフォーマンス解析ツール:** インテル® VTune™ Amplifier 2018: 高度な hotspot 解析

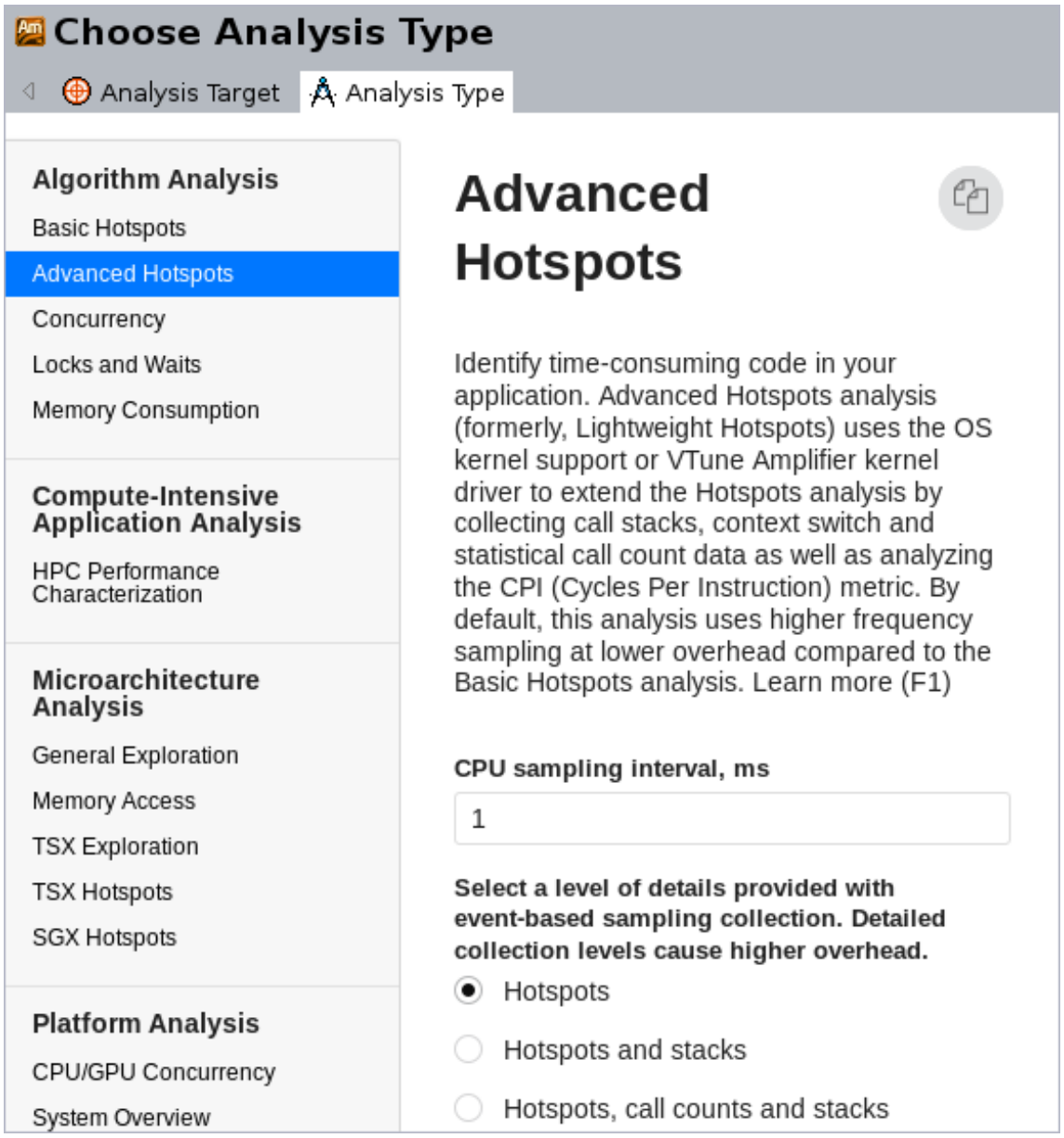
注

- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。

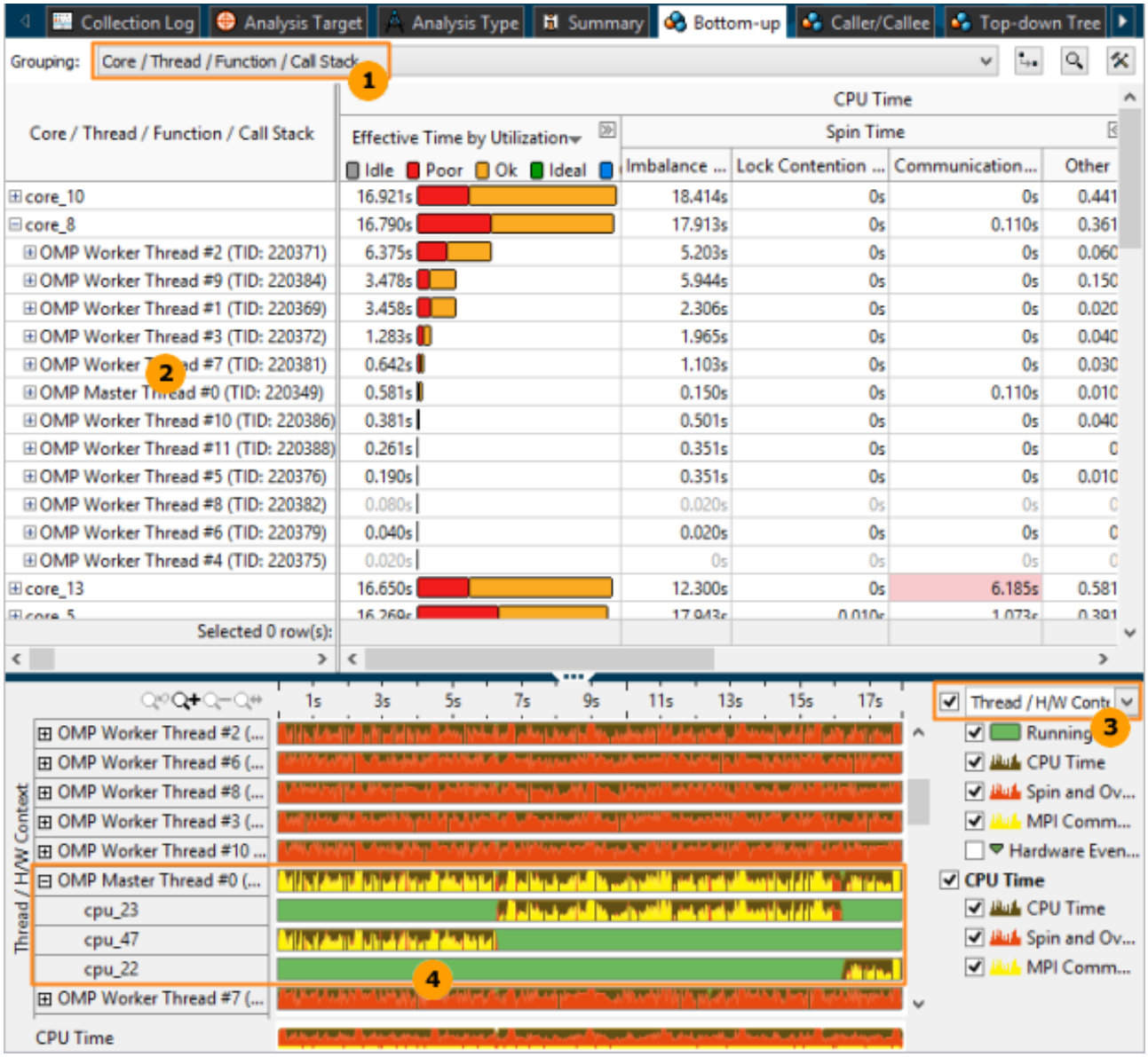
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Ubuntu* 16.04 64 ビット
- **CPU:** インテル® Core™ i7-6700K プロセッサー

高度な hotspot 解析を実行する

インテル® VTune™ Amplifier (GUI または `amplxe-cl`) を使用して、インテル® アーキテクチャー上で実行中のアプリケーションのソフトウェア・スレッド・マイグレーションを特定します。OS スレッド・マイグレーションを特定するには、アプリケーションで基本 hotspot 解析または高度な hotspot 解析を実行します。高度な hotspot 解析の例を次に示します。



スレッド・マイグレーションを特定する



1. GUI を使用してスレッド・マイグレーションを特定するには、**[Core/Thread/Function/Call Stack (コア/スレッド/関数/コールスタック)]** グループを選択します。

2. コアノードを展開してソフトウェア・スレッドの数を確認します。一般に、ソフトウェア・スレッドの数は、CPU でサポートしているハードウェア・スレッドの数以下にする必要があります。また、スレッドをコア間で均等に分散する必要もあります。いずれかのコアに予想よりも多くのソフトウェア・スレッドが表示されている場合、アプリケーションでスレッド・マイグレーションが発生しています。上記の例では、(インテル® Xeon® プロセッサーはインテル® ハイパースレッディング・テクノロジーをサポートしているため) 予想される 2 スレッドではなく 12 の OpenMP* ワーカースレッドが core_8 で実行されています。これはスレッド・マイグレーションを示しています。

3. **[Thread/H/W Context (スレッド/ハードウェア・コンテキスト)]** グループを選択して、**[Timeline (タイムライン)]** ペインでスレッド・マイグレーションを解析します。

4. スレッドのノードを展開して、このスレッドが実行された CPU の番号を確認し、経時的なスレッド実行を解析します。上記の例では、OpenMP* スレッド #0 は cpu_23 で実行された後、cpu_47 に移動しています。

次のように、コマンドラインから直接これらの結果を見ることもできます。

```
amplxe-cl -group-by thread,cpuid -report hotspots -r  /temp/test/omp -s "H/W
Context" -q | less
Thread                    H/W Context  CPU Time:Self
------------------------------  -----------  -------------
OMP Worker Thread #5 (0x3d86)    cpu_0                0.004
matmul-intel64 (0x3d52)          cpu_1                0.013
OMP Worker Thread #15 (0x3d90)   cpu_10               2.418
matmul-intel64 (0x3d52)          cpu_10               2.023
OMP Worker Thread #8 (0x3d89)    cpu_10               0.687
OMP Worker Thread #13 (0x3d8e)   cpu_10               0.097
OMP Worker Thread #6 (0x3d87)    cpu_10               0.065
OMP Worker Thread #4 (0x3d85)    cpu_10               0.059
OMP Worker Thread #1 (0x3d82)    cpu_10               0.048
OMP Worker Thread #9 (0x3d8a)    cpu_10               0.034
OMP Worker Thread #11 (0x3d8c)   cpu_10               0.009
```

多くの OpenMP* ワーカースレッドが cpu_10 で実行されていることも分かります。

スレッド・マイグレーションを訂正する

スレッド・マイグレーションはスレッド・アフィニティーを設定することで訂正できます。スレッド・アフィニティーは、特定のスレッドの実行をマルチプロセッサー・コンピューターの物理処理ユニットの一部に限定します。インテルのランタイム・ライブラリーには、OpenMP* スレッドを物理処理ユニットにバインドする機能があります。`OMP_PROC_BIND` および `OMP_PLACES`、またはインテルのランタイム固有の `KMP_AFFINITY` 環境変数を使用して OpenMP* アプリケーションのスレッド・アフィニティーを設定することもできます。

関連情報

- OpenMP* コード解析

OpenMP* インバランスとスケジュール・オーバーヘッド

このレシピは、バリアやスケジュール・オーバーヘッドのインバランスなど、OpenMP* プログラムでよくある並列ボトルネックを検出して修正する方法を説明します。

コンテンツ・エキスパート: Dmitry Prohorov (英語)

バリアは、スレッドチームのすべてのスレッドがバリアに到達した後に実行できる同期ポイントです。実行作業が不規則で、作業チャンクがワーカースレッドによって均等かつ静的に分散されている場合、バリアに到達したスレッドは、有効な作業を行う代わりにほかのスレッドを待機して時間を無駄にします。チーム内のスレッド数で正規化されたバリアでの合計待機時間は、インバランスを排除することでアプリケーションが軽減できる経過時間を示しています。

バリアのインバランスを排除する 1 つの方法は、動的スケジューリングを使用してスレッド間で動的に作業チャンクを分散することです。ただし、細粒度のチャンクでこれを行うと、スケジュール・オーバーヘッドにより状況がさらに悪化することがあります。このレシピを参考にして、インテル® VTune™ Amplifier を使用して OpenMP* ロード・インバランスとスケジュール・オーバーヘッドの問題に対応するワークフローを学びます。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** 特定の範囲の素数を計算するアプリケーション。メインループは、OpenMP* parallel `for` 構文で並列化されています。
- **コンパイラー**: インテル® コンパイラー 13 Update 5 以降。このレシピは、インテル® VTune™ Amplifier の解析で使用されるインテルの OpenMP* ランタイム・ライブラリー内のインストルメンテーションを実行するため、このコンパイラー・バージョンを必要とします。インテル® コンパイラーの `parallel-source-info=2` オプションを追加してコンパイルすることで、OpenMP* 領域名でソースファイル情報が提供され、ユーザーが識別しやすくなります。
- **パフォーマンス解析ツール:**
    - インテル® VTune™ Amplifier 2018: HPC パフォーマンス特性解析

注

- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。

    - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Ubuntu* 16.04 LTS
- **CPU:** インテル® Xeon® プロセッサー E5-2699 v4 @ 2.20GHz

ベースラインを作成する

サンプルコードの初期バージョンは、デフォルトの静的スケジューリングでループに OpenMP* parallel `for` プラグマを使用します (行 21)。

```
#include <stdio.h>
#include <omp.h>

#define NUM 100000000

int isprime( int x )
{
    for( int y = 2; y * y <= x; y++ )
    {
        if( x % y == 0 )
            return 0;
    }

    return 1;
}

int main( )
{
    int sum = 0;

#pragma omp parallel for reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }

    printf( "Number of primes numbers: %d", sum );

    return 0;
}
```

コンパイルしたアプリケーションの実行には約 3.9 秒かかります。これが、以降の最適化で使用するパフォーマンスの**ベースライン**となります。

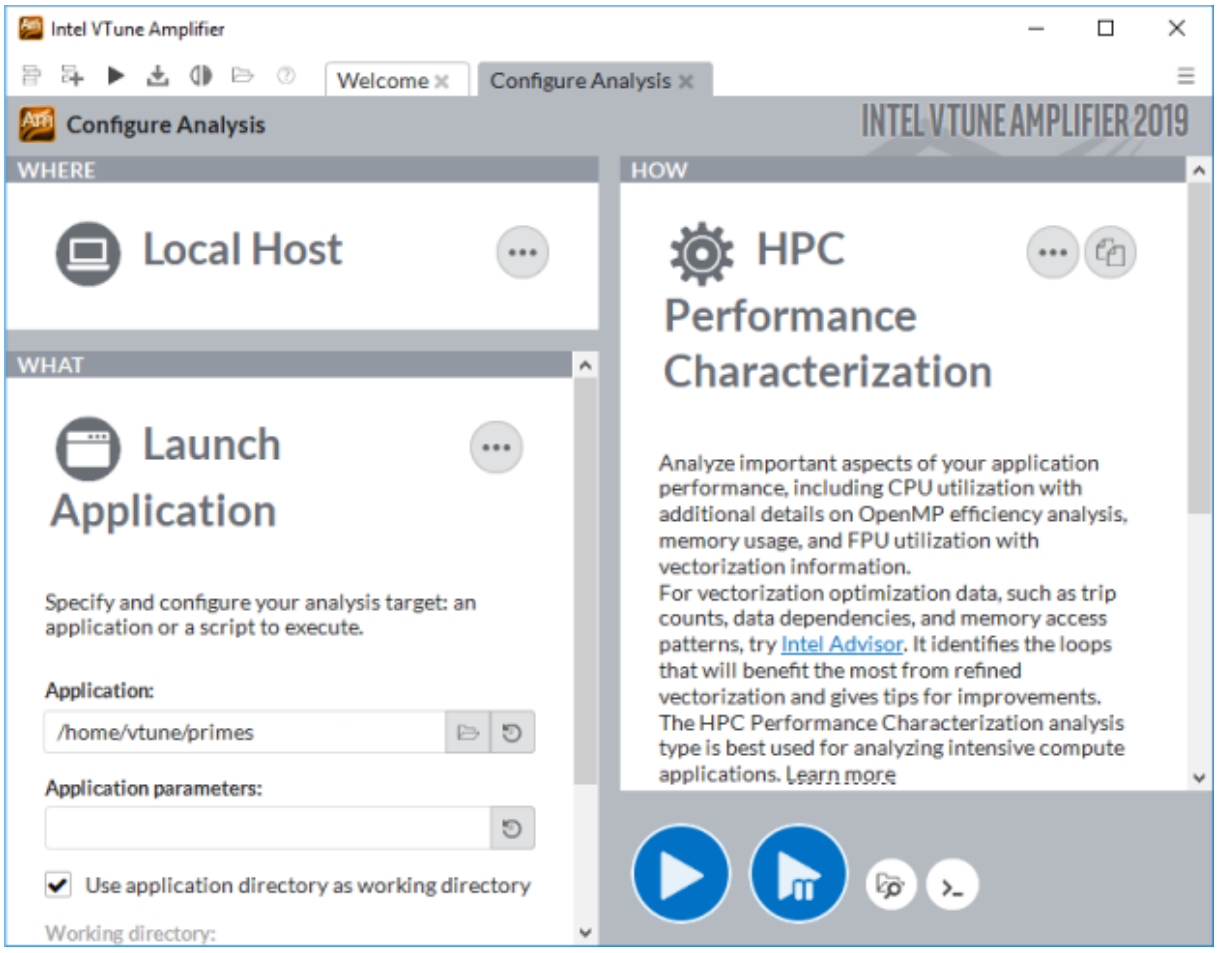
HPC パフォーマンス特性解析を実行する

サンプル・アプリケーションの潜在的なパフォーマンス・ボトルネックを理解するため、まず、インテル® VTune™ プロファイラーの HPC パフォーマンス特性解析を実行します。

1. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `primes`) を指定します。
2. **[Create Project (プロジェクトの作成)]** をクリックします。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

3. **[WHERE (どこを)]** ペインで、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
4. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、解析するアプリケーションを指定します。次に例を示します。



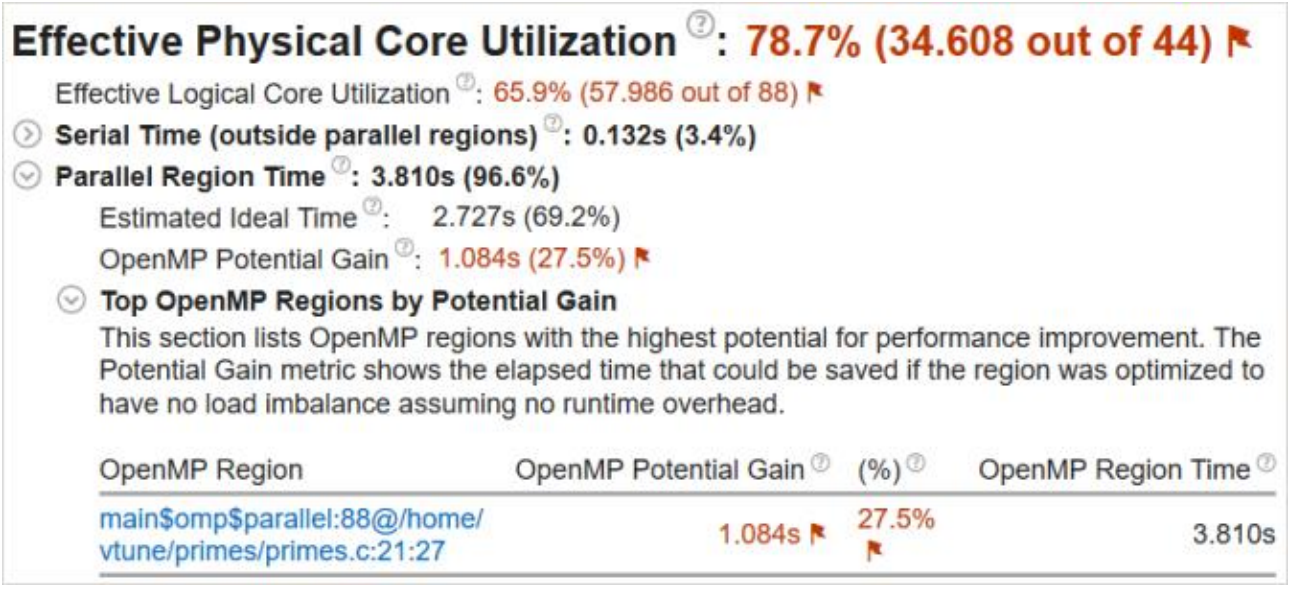
5. **[HOW (どのように)]** ペインで、[...] ボタンをクリックして **[Parallelism (並列処理)]** グループから **[HPC Performance Characterization (HPC パフォーマンス特性)]** 解析を選択します。
6.  **[Start (開始)]** ボタンをクリックします。

インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

OpenMP* のインバランスを特定する

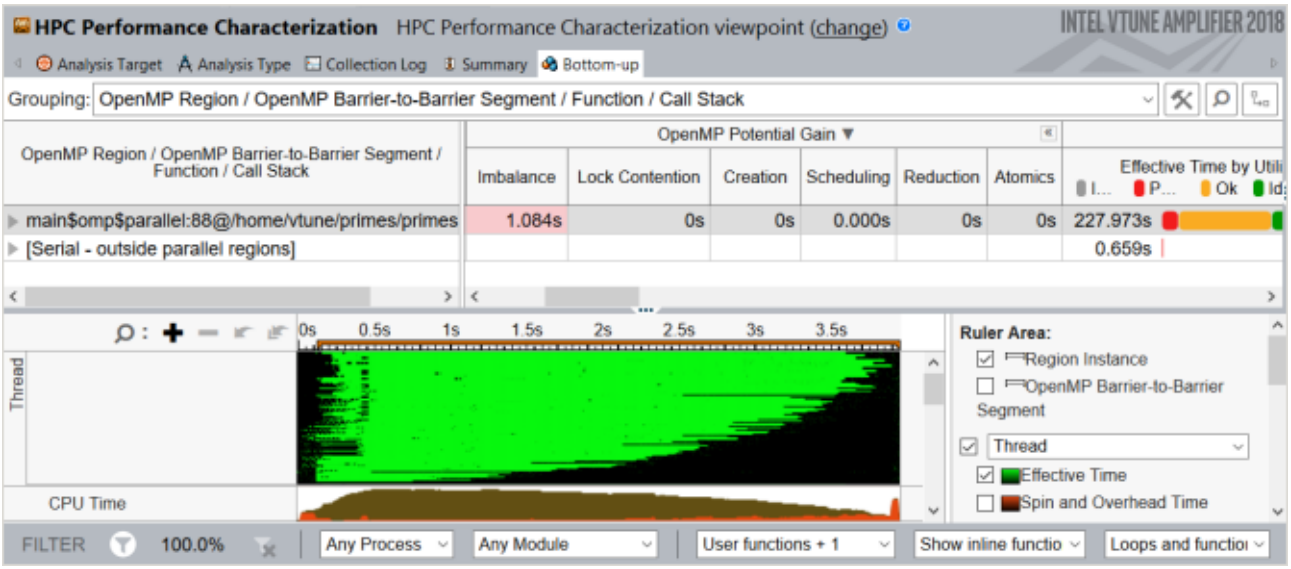
HPC パフォーマンス特性解析は、CPU 使用率 (並列性)、メモリーアクセス効率、ベクトル化などのパフォーマンス・ボトルネックの理解に役立つ重要な HPC メトリックを収集して表示します。このレシピのようにインテルの OpenMP* ランタイムを使用するアプリケーションでは、スレッド並列処理の問題の特定を支援する特別な OpenMP* 効率メトリックが役立ちます。

アプリケーション・レベルの統計が表示される [Summary (サマリー)] ビューから解析を始めます。フラグが付いた **[Effective Physical Core Utilization (効率的な物理コア利用率)]** メトリック (一部のシステムでは **[CPU Utilization (CPU 使用率)]**) は、調査すべきパフォーマンスの問題を示しています。

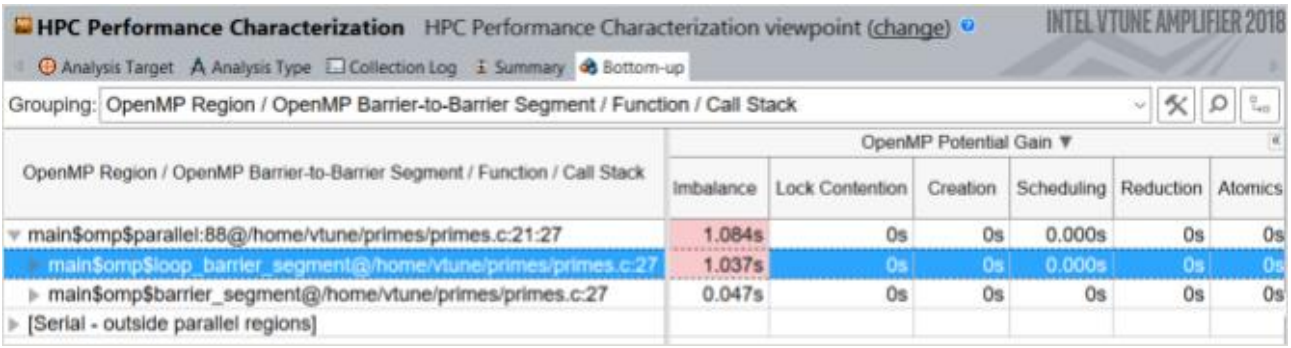


[Parallel Region Time (並列領域時間)] > **[OpenMP Potential Gain (OpenMP* 潜在的なゲイン)]** メトリックに移動して、非効率な並列処理を改善することで得られる最大ゲインを予測します。このサンプルでは、1.084 秒 (アプリケーションの実行時間の 27.5%) にフラグが付いているため、parallel 構文を詳しく調べる価値があります。

このサンプル・アプリケーションでは、**[Top OpenMP Regions by Potential Gain (潜在的なゲインによる上位 OpenMP* 領域)]** セクションに 1 つの parallel 構文があります。テーブルで領域名をクリックして、[Bottom-up (ボトムアップ)] ビューで詳細を確認します。**[Bottom-up (ボトムアップ)]** グリッドの **[OpenMP Potential Gain (OpenMP* 潜在的なゲイン)]** カラムを展開して非効率の詳細を表示します。このデータは、アプリケーションではなく OpenMP* で CPU 時間が費やされている原因と、それによる経過時間への影響を理解するのに役立ちます。



グリッド行のホットな領域で **[Imbalance (インバランス)]** メトリックの値がハイライトされています。マウスでこの値をポイントすると、動的スケジューリングによりインバランスを排除することを推奨するヒントが表示されます。領域内に複数のバリアがある場合は、領域ノードをバリアごとのセグメントに展開して、パフォーマンス・クリティカルなバリアを特定する必要があります。



このサンプルには、インテル® VTune™ Amplifier によって分類されない重大なインバランスを持つループバリアと並列領域のジョインバリアがあります。

注

アプリケーションの実行中にインバランスを視覚化するには、[Timeline (タイムライン)] ビューを調査します。有効な作業を実行している時間は緑色で示され、浪費時間は黒色で示されます。

動的スケジューリングを適用する

このインバランスは、静的な作業分散により、特定のスレッドに大きな数字を割り当てる一方で、一部のスレッドは小さな数字のチャンクを短時間で処理しバリアで時間を無駄にしていることが原因です。このインバランスを解消するには、デフォルトのパラメーターで動的スケジューリングを適用します。

```
    int sum = 0;

#pragma omp parallel for schedule(dynamic) reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }
```

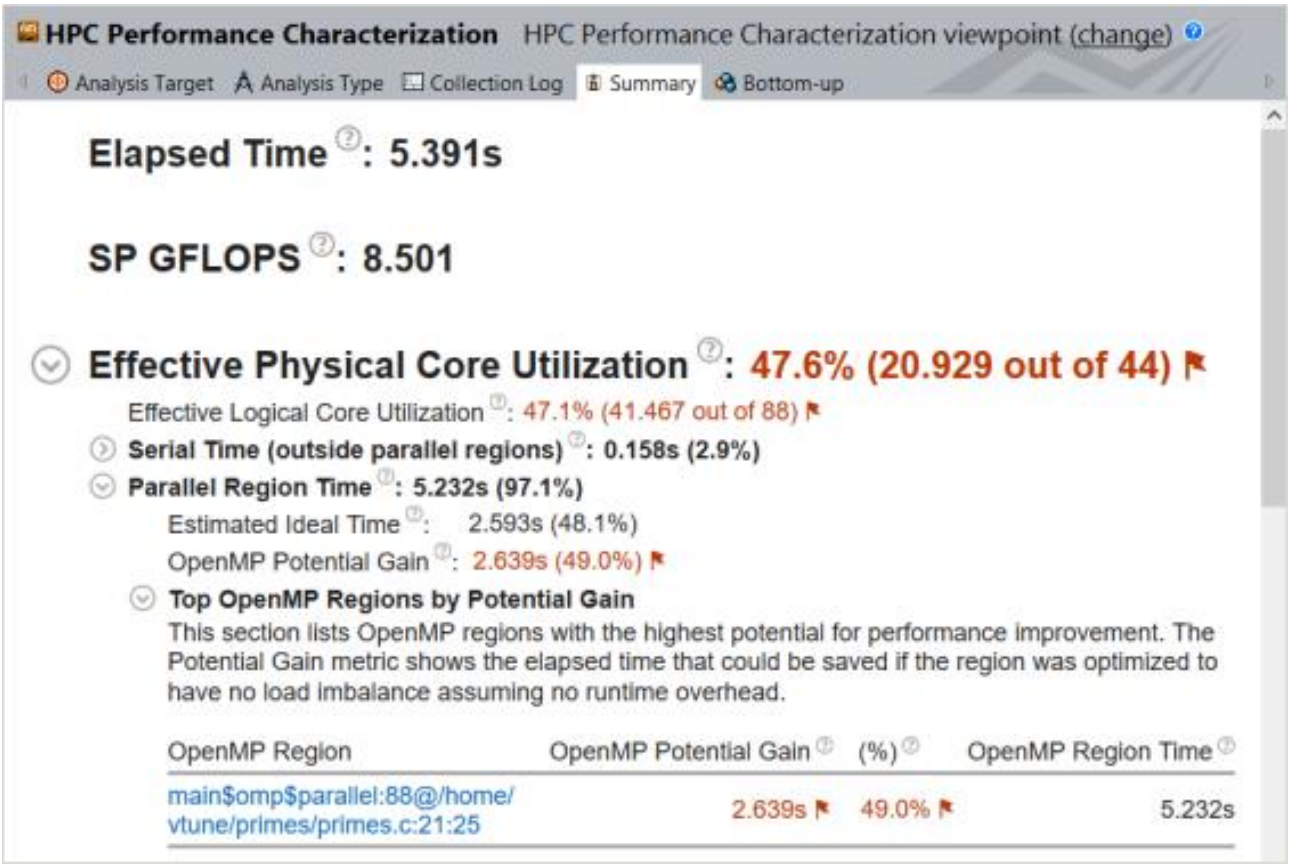
アプリケーションを再コンパイルして、その実行時間とオリジナルのパフォーマンス・ベースラインを比較して、最適化を検証します。

OpenMP* スケジュール・オーバーヘッドを特定する

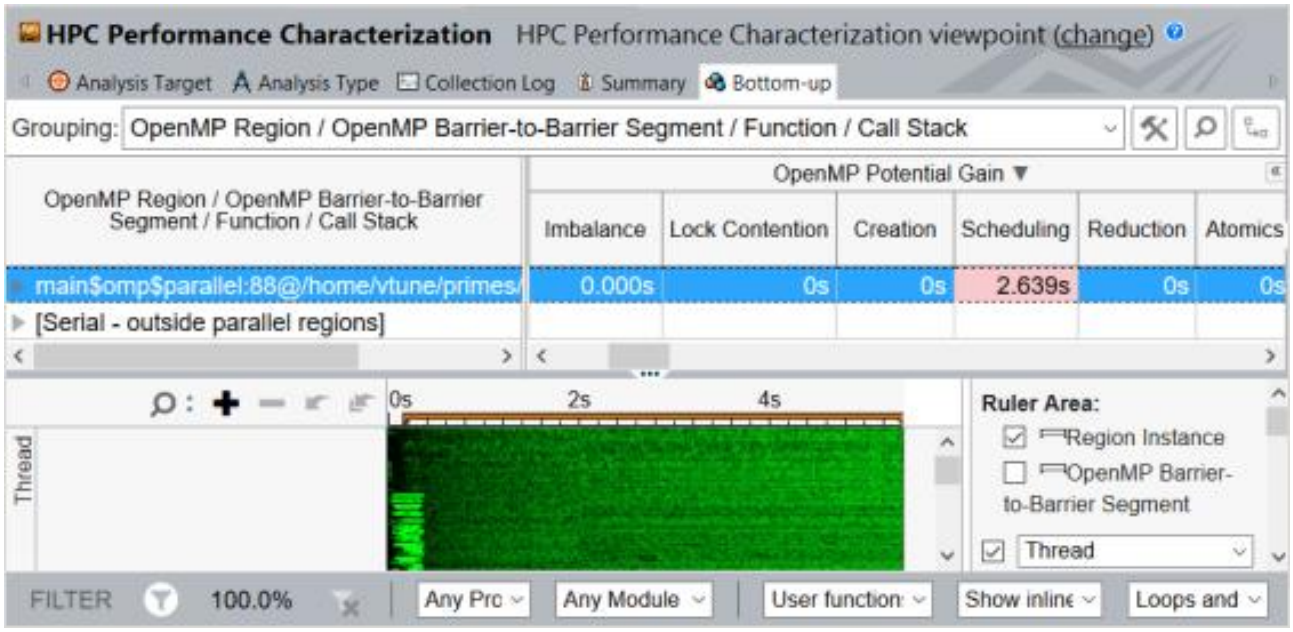
変更したサンプル・アプリケーションを実行すると、スピードアップせずに実行時間が 5.3 秒に増えます。これは、細粒度の作業チャンクで動的スケジューリングを適用した場合に起こる副作用です。潜在的なボトルネックの詳細を把握するには、HPC パフォーマンス特定解析を再度実行して、パフォーマンス低下の原因を確認することを検討してください。

1. メニューから **[New (新規)] > [HPC Performance Characterization Analysis (HPC パフォーマンス特性解析)]** を選択して、既存の `primes` プロジェクトの解析設定を開きます。
2. **[Start (開始)]** をクリックします。

[Summary (サマリー)] ビューでは、パフォーマンス・クリティカルとして **[Effective Physical Core Utilization (効率的な物理コア利用率)]** メトリックにフラグが付けられており、**[OpenMP Potential Gain (OpenMP* 潜在的なゲイン)]** の値がオリジナルの値の 1.084 秒よりも増えています。



このデータは、コードの並列効率が低下したことを裏付けています。原因を理解するため、[Bottom-up (ボトムアップ)] ビューに切り替えます。



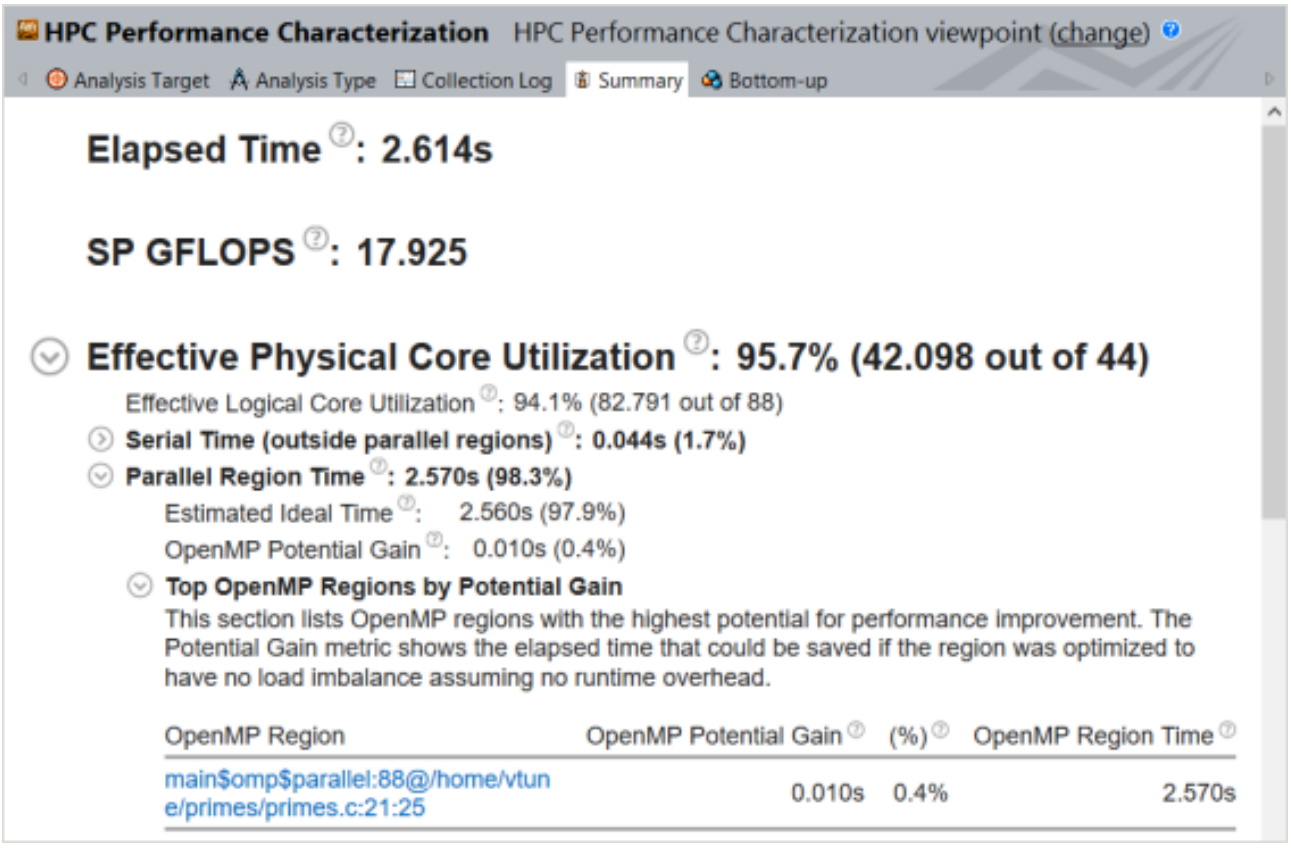
以前の解析結果と比較すると、**[Scheduling (スケジューリング)]** オーバーヘッドはインバランスよりもさらに深刻な状況にあります。これは、ワーカースレッドごとに 1 ループ反復を割り当てるスケジューラーのデフォルトの動作が原因です。スレッドがすぐにスケジューラーに戻るため、コンカレンシーが高すぎてボトルネックとなります。タイムラインの **[Effective Time (有効な時間)]** メトリックの分布 (緑色) は、スレッドによる通常の作業を示していますが、密度が低くなっています。グリッドでハイライトされている **[Scheduling (スケジューリング)]** メトリックの値にマウスホバーするとパフォーマンスに関するアドバイスが表示され、OpenMP* `parallel for` プラグマでチャンクを使用して並列処理を粗粒度にし、スケジュール・オーバーヘッドを排除するように推奨されます。

チャンク・パラメーターを使用して動的スケジューリングを適用する

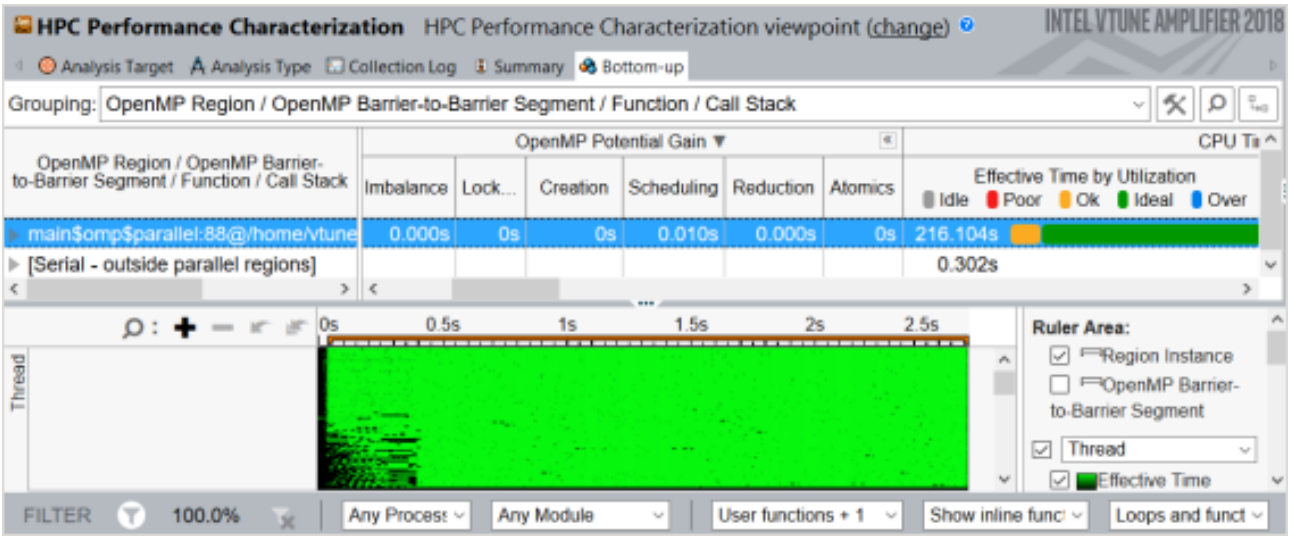
次のように、`schedule` 節のチャンク・パラメーターを 20 にします。

```
#pragma omp parallel for schedule(dynamic,20) reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }
```

再コンパイルすると、アプリケーションの実行時間は 2.6 秒になりました (ベースラインと比較して 30% の向上)。**[Summary (サマリー)]** ビューは、理想に近い **[Parallel Region Time (並列領域時間)]** (98.3%) を示しています。



[Bottom-up (ボトムアップ)] ビューのタイムラインは、良好な密度の有効な作業 (緑色) を示しています。



動的スケジューリングは、頻繁に新しい作業チャンクをスレッドへ割り当てるため、キャッシュの再利用が妨げられ、コード実行のキャッシュ効率が低下する可能性があります。つまり、CPU を効率良く使用しバランス良く最適化されたアプリケーションのほうが、静的スケジューリングを使用するバランスの悪いアプリケーションよりも低速になることがあります。この場合、**[HPC Performance Characterization (HPC パフォーマンス特性)]** ビューの **[Memory Bound (メモリー依存)]** セクションを調査します。

注

このレシピの情報は、デベロッパー・フォーラム (英語) を参照してください。

関連情報

- OpenMP* コード解析 (英語)
- HPC パフォーマンス特性解析 (英語)
- 潜在的なゲイン (英語)
- チュートリアル: MPI/OpenMP* ハイブリッド・アプリケーションの解析 (英語)
- 低いプロセッサー・コア利用率: OpenMP* シリアル時間

低いプロセッサー・コア利用率: OpenMP* シリアル時間

このレシピは、OpenMP* で並列化されたアプリケーションのシリアル実行部分を特定し、追加の並列化の機会を見つけ、アプリケーションのスケーラビリティーを向上します。

コンテンツ・エキスパート: Dmitry Prohorov (英語)

並列アプリケーションのシリアル時間は、アプリケーションが利用可能なハードウェア・リソース (アプリケーション・コードを実行するコアなど) を利用する能力である、アプリケーションのスケーラビリティーを妨げる要因の 1 つです。並列アプリケーションの最大スピードアップは、アムダールの法則で 1/((1-P)+(P/N)) と表されます。ここで、P はアプリケーション実行の並列部分であり、N はプロセッサー・エレメントです。そのため、P (アプリケーション実行のシリアル部分) が大きくなるほど、実行のシリアル部分によって制限される N (プロセッサー・エレメント) の数が多くなります。

ユーザー・アプリケーションが OpenMP* で並列化されている場合、シーケンシャル・コードの実行は、OpenMP* 領域外のコード実行、または `#pragma omp master` や `#pragma omp single` 構造内のコード実行の可能性があります。このレシピでは、1 つ目のケースの検出に注目します。このレシピを参考にして、インテル® VTune™ Amplifier で OpenMP* 領域外のコードの実行時間を検出して、シリアル・ホットスポット関数/ループの分布を解析し、コードの並列化の可能性を理解します。

- 使用するもの
- 手順:
    1. ベースラインを作成する
    2. HPC パフォーマンス特性解析を実行する
    3. OpenMP* シリアル時間を特定する
    4. コードを並列化する
    5. スレッドエラーを調査する

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション**: `miniFE` 有限要素ミニアプリケーション (OpenMP* バージョン)。https://github.com/Mantevo/miniFE (英語) からダウンロードできます。
- **コンパイラー**: インテル® コンパイラー 13 Update 5 以降。このレシピは、インテル® VTune™ Amplifier の解析で使用されるインテルの OpenMP* ランタイム・ライブラリー内のインストルメンテーションを実行するため、このコンパイラー・バージョンを必要とします。
- **パフォーマンス解析ツール:**
    - インテル® VTune™ Amplifier 2019: HPC パフォーマンス特性解析
    - インテル® Inspector 2019: スレッドエラー解析

注

- インテル® VTune™ Amplifier 評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。

    - インテル® Inspector 評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-inspector-xe/ を参照してください。
- **オペレーティング・システム:** Ubuntu* 16.04 LTS
- **CPU**: インテル® Xeon® プロセッサー E5-2699 v4 @ 2.20GHz

ベースラインを作成する

`openmp/src/Makefile.intel.openmp` メイクファイルを使用してアプリケーションをビルドします。インテル® コンパイラーの `-g` オプションと `-parallel-source-info=2` オプションを追加してコンパイルすることで、デバッグ情報が有効になり、OpenMP* 領域名でソースファイル情報が提供され、ユーザーが識別しやすくなります。

引数 nx=200、ny=200、nz=200 でコンパイルしたアプリケーションを、物理コア数に対応する OpenMP* スレッド数とコアごとに 1 スレッド (OMP_NUM_THREADS=44、OMP_PLACES=cores) で実行すると、約 12 秒かかります。これが、以降の最適化で使用するパフォーマンスの**ベースライン**となります。

HPC パフォーマンス特性解析を実行する

サンプル・アプリケーションの潜在的なパフォーマンス・ボトルネックを理解するため、まず、インテル® VTune™ Amplifier の HPC パフォーマンス特性解析を実行します。

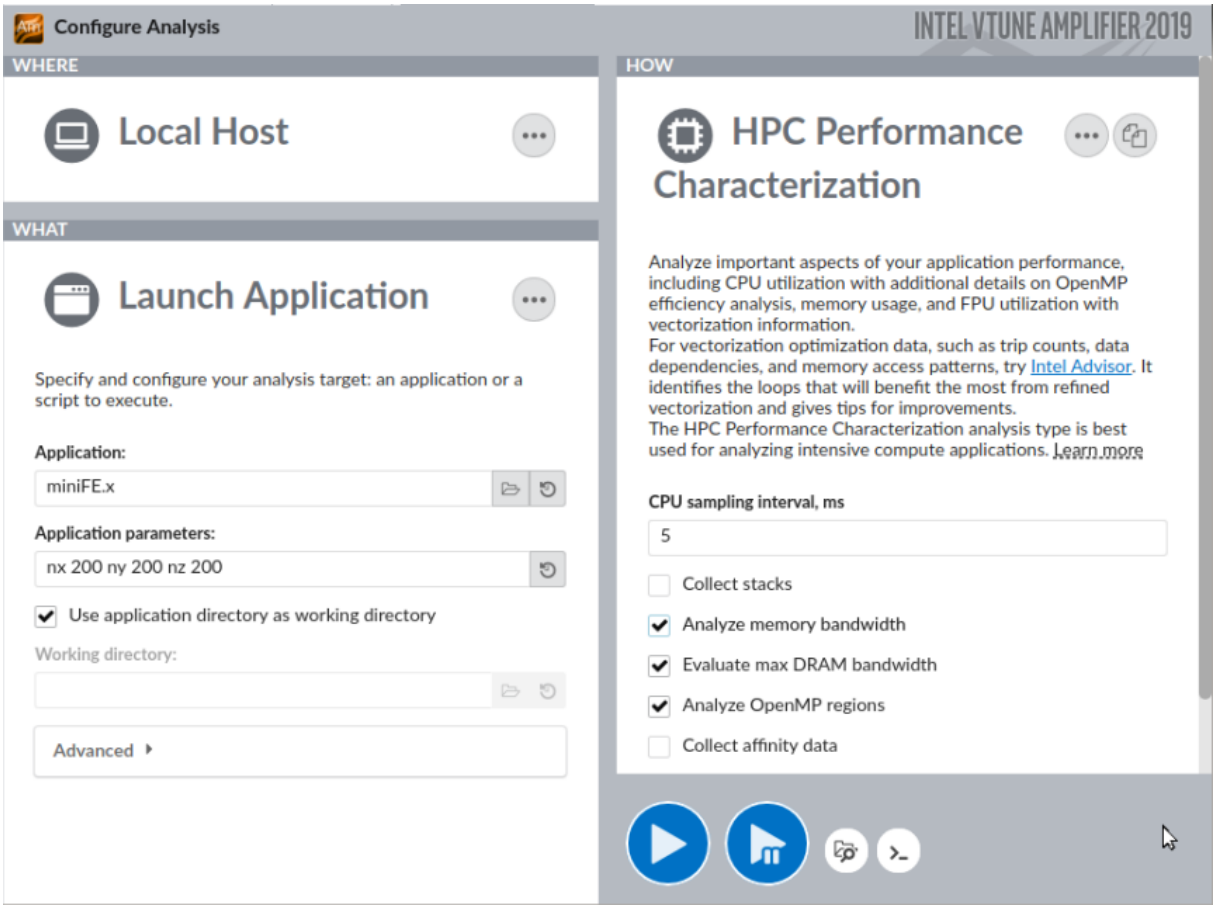
1. ツールバーの **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `miniFE`) を指定します。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

2. **[WHERE (どこを)]** ペインで、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。

   [WHAT (何を)] ペインで、**[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、解析するアプリケーションと引数 `nx 200 ny 200 nz 200` を指定します。

3. **[HOW (どのように)]** ペインで、[...] ボタンをクリックして **[Parallelism (並列処理)]** グループから **[HPC Performance Characterization (HPC パフォーマンス特性)]** 解析を選択します。
4.  [Start (開始)] ボタンをクリックして、解析を実行します。



コマンドラインから解析を実行するには、次の構文を使用します。

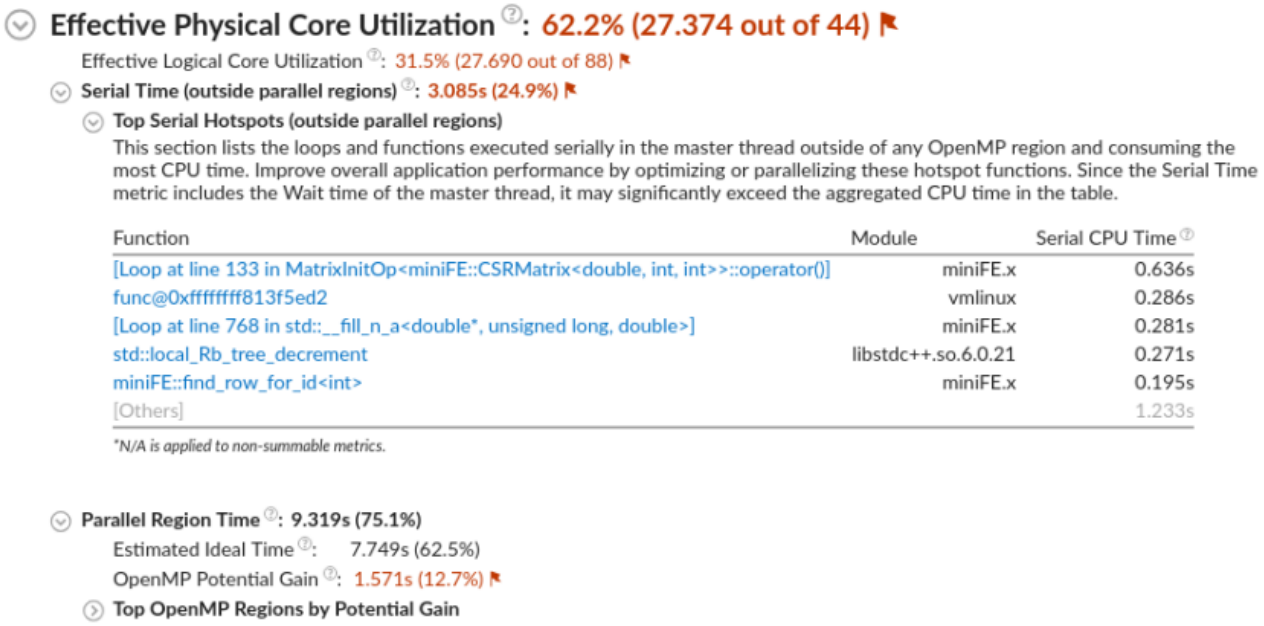
```
amplxe-cl -collect hpc-performance -data-limit=0 ./miniFE.x nx 200 ny 200 nz 200
```

インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

OpenMP* シリアル時間を特定する

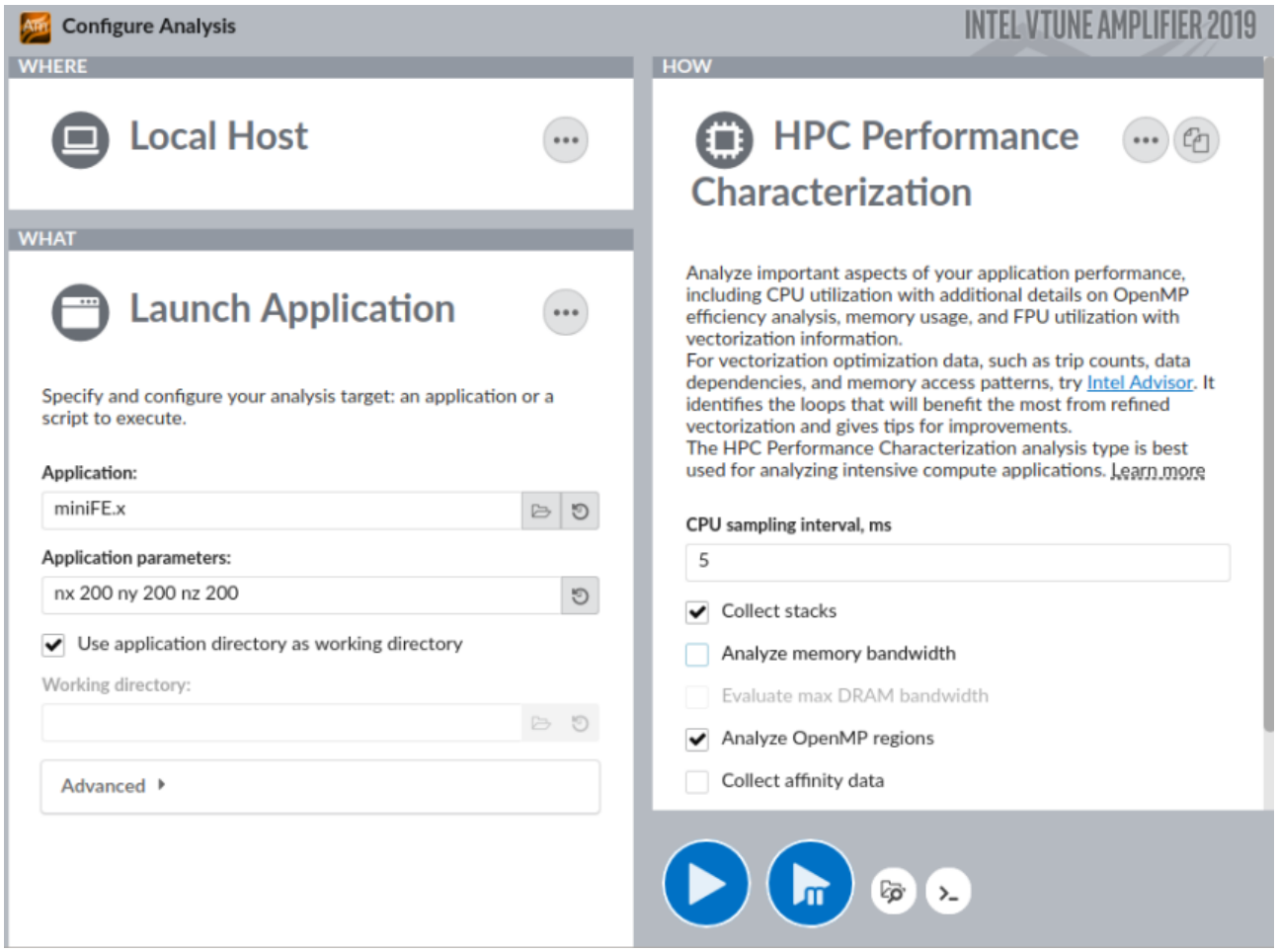
HPC パフォーマンス特性解析は、CPU 使用率 (並列性)、メモリーアクセス効率、ベクトル化などのパフォーマンス・ボトルネックの理解に役立つ重要な HPC メトリックを収集して表示します。このレシピのようにインテルの OpenMP* ランタイムを使用するアプリケーションでは、スレッド並列処理の問題の特定を支援する特別な OpenMP* 効率メトリックが役立ちます。

アプリケーション・レベルの統計が表示される **[Summary (サマリー)]** ビューから解析を始めます。フラグが付いた **[Effective Physical Core Utilization (効率的な物理コア利用率)]** メトリック (一部のシステムでは **[CPU Utilization (CPU 使用率)]**) は、調査すべきパフォーマンスの問題を示しています。



メトリックの階層を深く掘り下げると、アプリケーションの **[Serial Time (outside parallel regions) (シリアル時間 (並列領域外))]** が経過時間の約 25% を占めていることが分かります。主なシリアル hotspot は、行列の初期化コードにあります。

コールスタック付きの HPC パフォーマンス特性解析を実行して、利用可能な最適化の可能性を調査することを検討してください。コールスタックは、適切な粒度で並列化の候補を見つけるのに役立ちます。コールスタック収集はメモリー帯域幅の解析と組み合わせることができないため、必ず **[Analyze memory bandwidth (メモリー帯域幅を解析)]** 設定オプションを無効にしてください。



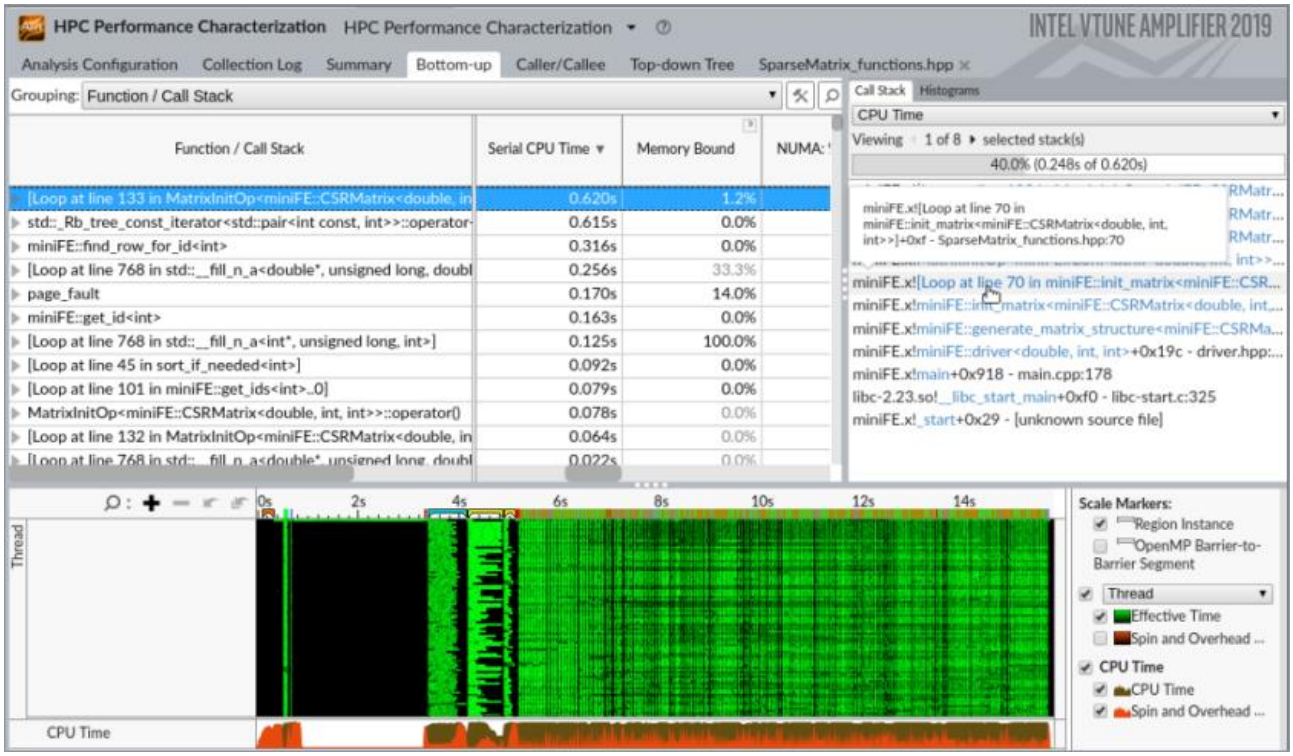
コマンドラインからこの設定を実行するには、次のコマンドを入力します。

```
amplxe-cl -collect hpc-performance -data-limit=0 -knob enable-stack-collection=true -knob collect-memory-bandwidth=false ./miniFE.x nx 200 ny 200 nz 200
```

注

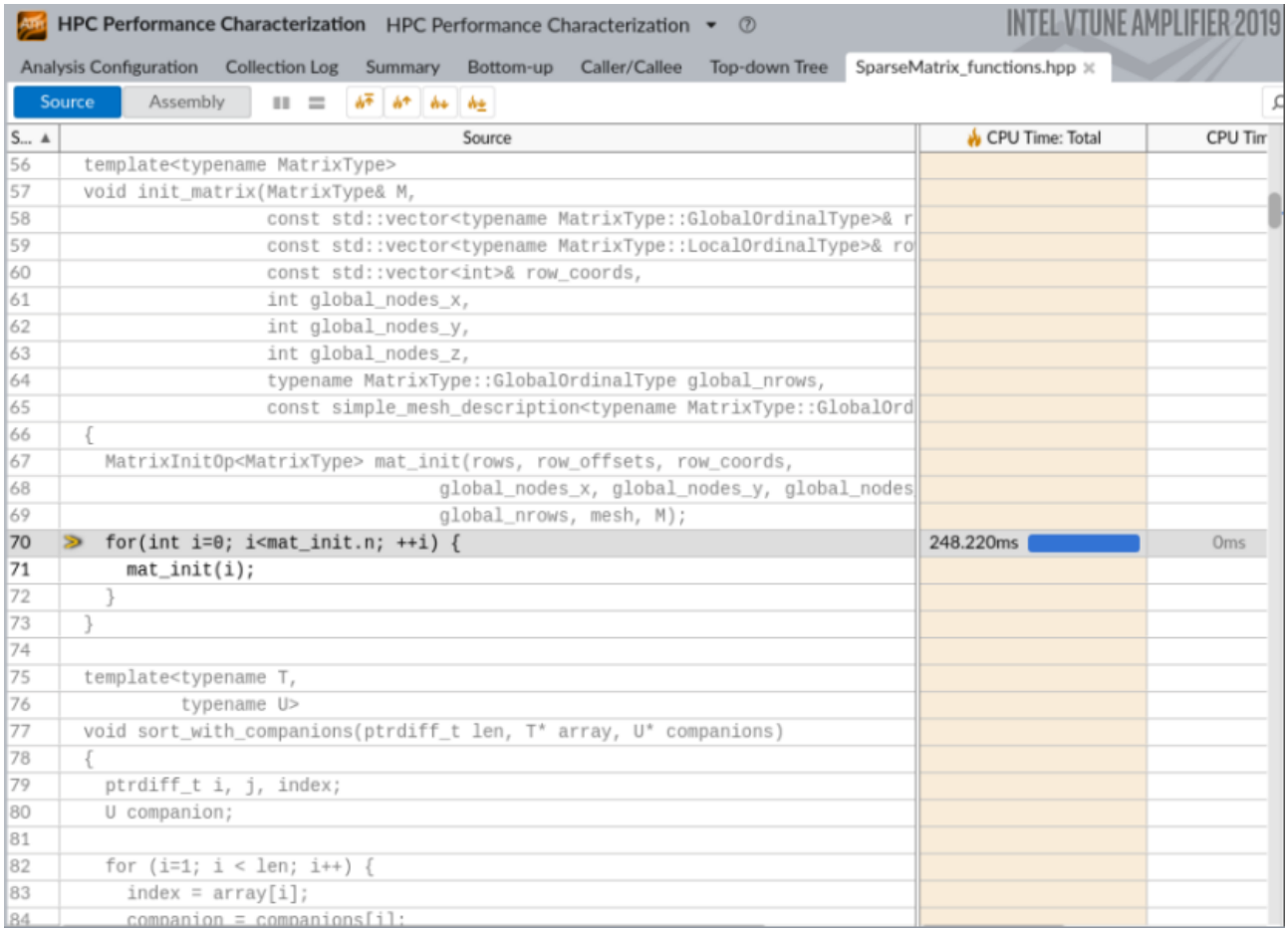
[Threading (スレッド化)] 解析を実行して、スタックを使用して OpenMP* シリアル時間を解析することもできます。ただし、どのパフォーマンス特性がボトルネックになっているのか分からない場合は、HPC パフォーマンス特性解析を出発点としたほうが良いでしょう。

上位の hotspot を特定し、そのコールスタックを調査するには、**[Bottom-up (ボトムアップ)]** ビューに切り替えて、**[Serial CPU Time (シリアル CPU 時間)]** カラムでソートします。



行列要素でループを反復している `SparseMatrix_functions.hpp` の 70 行目が並列化に適していることが分かります。

[Call Stack (コールスタック)] ペインの行をダブルクリックして、ソースファイルを開きます。上位の hotspot ループの最もホットな行が自動的に表示されます。



コードを並列化する

`#pragma omp parallel for` を追加して、行列の初期化を OpenMP* で並列化します。

```
#pragma omp parallel for
for(int i=0; i<mat_init.n; ++i) {
    mat_init(i);
```

アプリケーションを再コンパイルして、その実行時間とオリジナルのパフォーマンス・ベースラインを比較して、最適化を検証します。

このレシピでは、最適化したアプリケーションの経過時間は約 10 秒になり、アプリケーションの実行は約 16% スピードアップしました。

最適化したアプリケーションで HPC パフォーマンス特定解析を再度実行します。

全体的な **[Effective Physical Core Utilization (効率的な物理コア利用率)]** は 10% 向上しました。OpenMP* シリアル時間は 9% に減り、インテル® VTune™ Amplifier によって問題フラグが付けられなくなりました (メトリックのしきい値は 15% です)。

並列効率をさらに向上するには、「OpenMP*インバランスとスケジュール・オーバーヘッド」クックブック・レシピで示すように、最もバランスの悪いバリアを解析できます。

スレッドエラーを調査する

並列処理の解析を完了するには、データ競合やデッドロックなどのスレッドエラーをチェックします。チェックには、一部のハードウェアでは発生せずに別の環境では問題となる、あるいは同じ環境の異なる設定で発生する潜在的なデータ競合とデッドロックも検出できる、インテル® Inspector を使用します。

コマンドライン・インターフェイスを使用し、ワークロード・サイズを減らすと、チェックを高速に行い代表的な結果が得られます。

```
inspxe-cl -collect ti3 ./miniFE.x nx 40 ny 40 nz 40
```

インテル® Inspector は並列コードの問題は検出しません。

注

このレシピの情報は、デベロッパー・フォーラム (英語) を参照してください。

関連情報

- OpenMP* コード解析
- HPC パフォーマンス特性解析 (英語)
- スレッド解析 (英語)
- チュートリアル: MPI/OpenMP* ハイブリッド・アプリケーションの解析 (英語)

インテル® TBB アプリケーションのスケジュール・オーバーヘッド

このレシピは、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) アプリケーションのスケジュール・オーバーヘッドを検出して修正する方法を説明します。

コンテンツ・エキスパート: Dmitry Prohorov (英語)

スケジュール・オーバーヘッドは、細粒度の作業チャンクをスレッド間で動的に分散する場合に発生する典型的な問題です。スケジュール・オーバーヘッドが発生している場合、ワーカースレッドに作業を割り当てるためスケジューラーが費やす時間と、ワーカースレッドが新しい作業の待機に費やす時間が長くなると、並列処理の効率が低下し、極端なケースでは、プログラムのスレッド化したバージョンのほうがシーケンシャル・バージョンよりも遅くなります。ほとんどのインテル® TBB 構文は、オーバーヘッドを回避するためチャンク数をデフォルトの粒度よりも大きくする、デフォルトの自動パーティショナーを使用します。意図的にまたは `parallel_deterministic_reduce` などの構文を使用して単純なパーティショナーを使用する場合、単純なパーティショナーは最大で 1 反復のデフォルトの粒度と等しいチャンクサイズに作業を分割するため、粒度を調整する必要があります。インテル® VTune™ Amplifier は、インテル® TBB アプリケーションのスケジュール・オーバーヘッドの検出を支援し、粒度を大きくしてオーバーヘッドによる速度低下を回避するためのアドバイスを提供します。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** インテル® TBB の `parallel_deterministic_reduce` テンプレート関数を使用してベクトル要素の合計を計算するサンプル・アプリケーション
- **コンパイラー:** インテル® コンパイラーまたは GNU* コンパイラー。次のコンパイラー/リンカーオプションを指定します。

  ```
  -I <tbb_install_dir>/include -g -O2 -std=c++11 -o vector-reduce vector-reduce.cpp -L <tbb_install_dir>/lib/intel64/gcc4.7 -ltbb
  ```

- **パフォーマンス解析ツール:** インテル® VTune™ Amplifier 2019: スレッド解析

注

- インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
- このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。

    - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Ubuntu* 16.04 LTS
- **CPU:** インテル® Xeon® プロセッサー E5-2699 v4 @ 2.20GHz

ベースラインを作成する

サンプルコードの初期バージョンは、デフォルトの粒度で `parallel_deterministic_reduce` を使用します (行 17-23)。

```
#include <stdlib.h>
#include "tbb/tbb.h"

static const size_t SIZE = 50*1000*1000;
double v[SIZE];

using namespace tbb;

void VectorInit( double *v, size_t n )
{
    parallel_for( size_t( 0 ), n, size_t( 1 ), [=](size_t i){ v[i] = i *
2; } );
}

double VectorReduction( double *v, size_t n )
{

    return parallel_deterministic_reduce(
        blocked_range<double*>( v, v + n ),
        0.f,
        [](const blocked_range<double*>& r, double value)->double {
            return std::accumulate(r.begin(), r.end(), value);
        },
        std::plus<double>()
    );
}

int main(int argc, char *argv[])
{
    task_scheduler_init( task_scheduler_init::automatic );

    VectorInit( v, SIZE );

    double sum;

    for (int i=0; i<100; i++)
                sum = VectorReduction( v, SIZE );

    return 0;
}
```

統計解析向けに大きく測定可能な計算処理にするため、行 35 のループでベクトル合計計算を繰り返しています。

コンパイルしたアプリケーションの実行には約 9 秒かかりました。これが、以降の最適化で使用するパフォーマンスの**ベースライン**となります。

スレッド解析を実行する

アプリケーションのスレッド並列性とスケジュール・オーバーヘッドに費やされた時間を予測するには、並行性解析を実行します。

1. ツールバーの  **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `vector-reduce`) を指定します。
2. **[Create Project (プロジェクトの作成)]** をクリックします。

   [Configure Analysis (解析の設定)] ウィンドウが表示されます。

3. **[WHERE (どこを)]** ペインで、**[Local Host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
4. **[WHAT (何を)]** ペインで、**[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、解析するアプリケーションを指定します。
5. **[HOW (どのように)]** ペインで、[...] ボタンをクリックして **[Parallelism (並列性)] > [Threading (スレッド化)]** を選択します。
6.  **[Start (開始)]** ボタンをクリックします。

インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

注

スレッド解析はインストルメンテーション・ベースでスタックのスティッチを使用するため、収集オーバーヘッドにより、インストルメントされたアプリケーションの経過時間はオリジナルのアプリケーションの実行よりも長くなります。

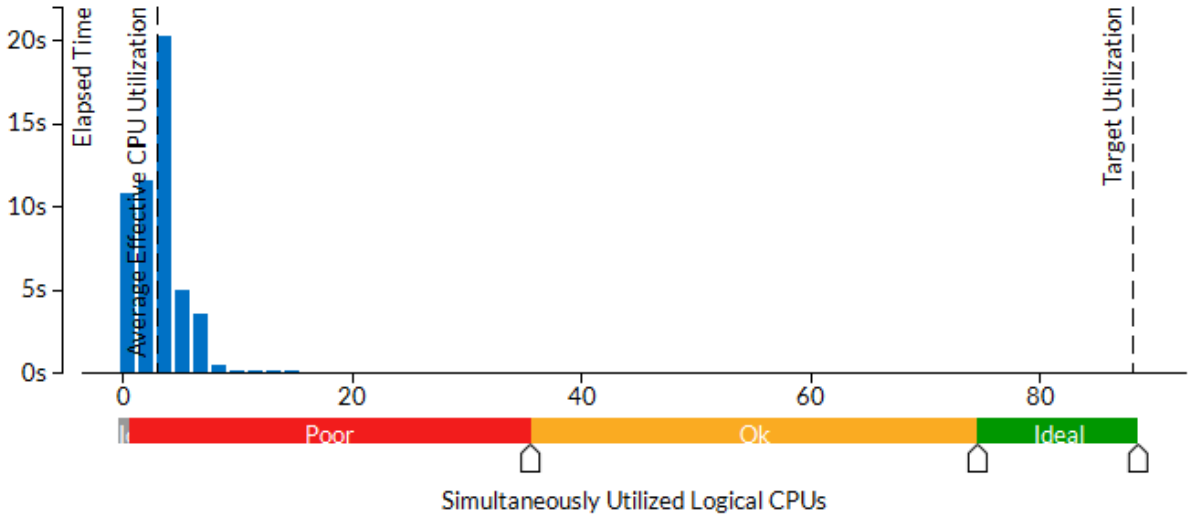
スケジュール・オーバーヘッドを特定する

アプリケーション・レベルの統計が表示される **[Summary (サマリー)]** ビューから解析を始めます。

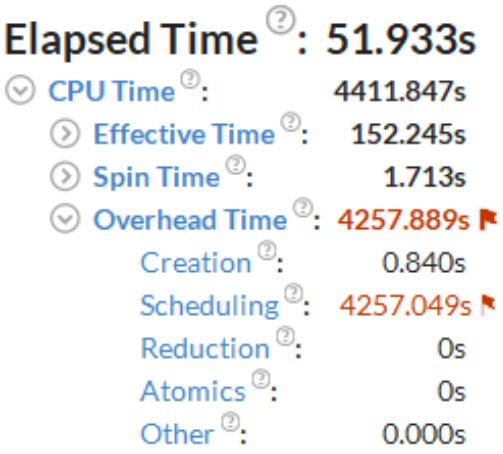
[Effective CPU Utilization Histogram (効率良い CPU 利用率の分布図)] は、アプリケーションが利用可能な 88 コアのうち平均で約 3 コアしか使用していないことを示しています。

Effective CPU Utilization Histogram

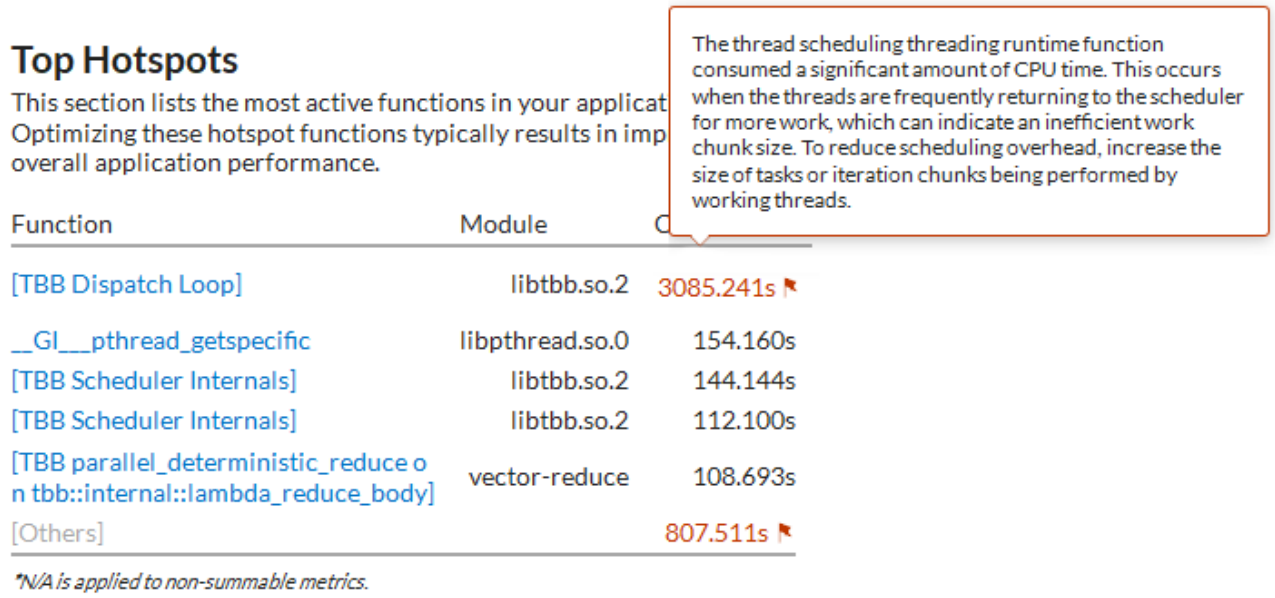
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



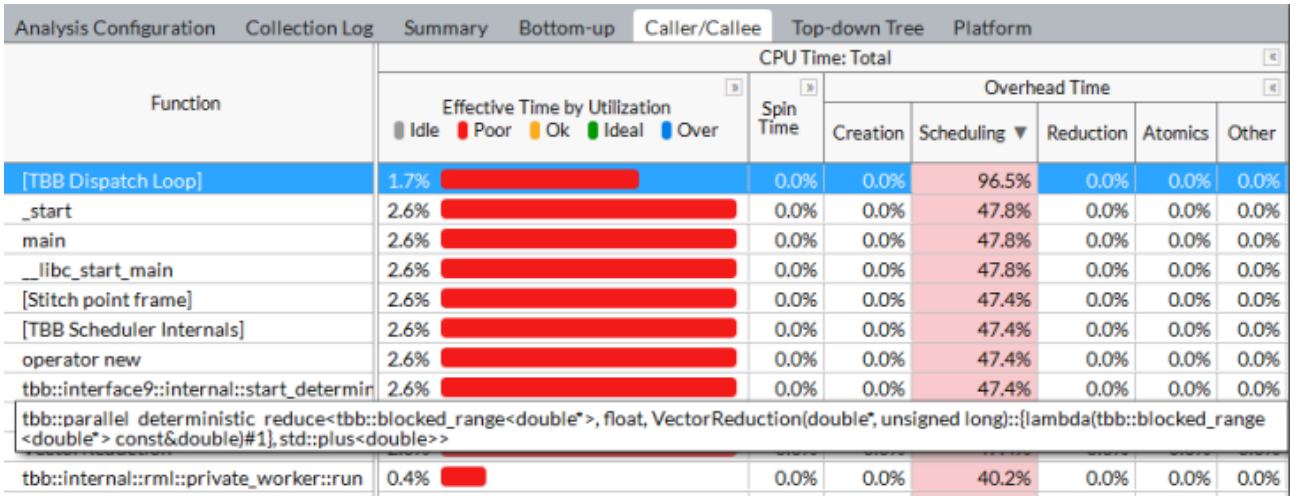
フラグが付いた **[Overhead Time (オーバーヘッド時間)]** メトリックと **[Scheduling (スケジューリング)]**) サブメトリックは、調査すべき非効率的なスレッド化の問題を示しています。



[Top Hotspots (上位 hotspot)] セクションから `[TBB Dispatch Loop]` が最も時間を費やしている関数であることが分ります。関数に関連するフラグのヒントは、並列処理の粒度を大きくして対応すべきスケジュール・オーバーヘッドに関する情報を示しています。



サンプル・アプリケーションには、ベクトル要素の合計を計算する 2 つのインテル® TBB 構文 `parallel_for` の初期化と `parallel_deterministic_reduce` が含まれています。**[Caller/Callee (呼び出し元/呼び出し先)]** データビューを使用して、どちらのインテル® TBB 構文でオーバーヘッドが発生しているか確認します。**[CPU Time: Total (CPU 時間: 合計)]** > **[Overhead Time (オーバーヘッド時間)]** カラムを展開して、**[Scheduling (スケジューリング)]** カラムでグリッドをソートします。**[Function (関数)]** リストでインテル® TBB の並列構文を含む最初の行を見つけます。



この行は、スケジュール・オーバーヘッドが最も大きい `VectorReduction` 関数の `parallel_deterministic_reduce` 構文をポイントしています。この構文の並列処理でオーバーヘッドを排除できるように、作業チャンクをより粗粒度にします。

注

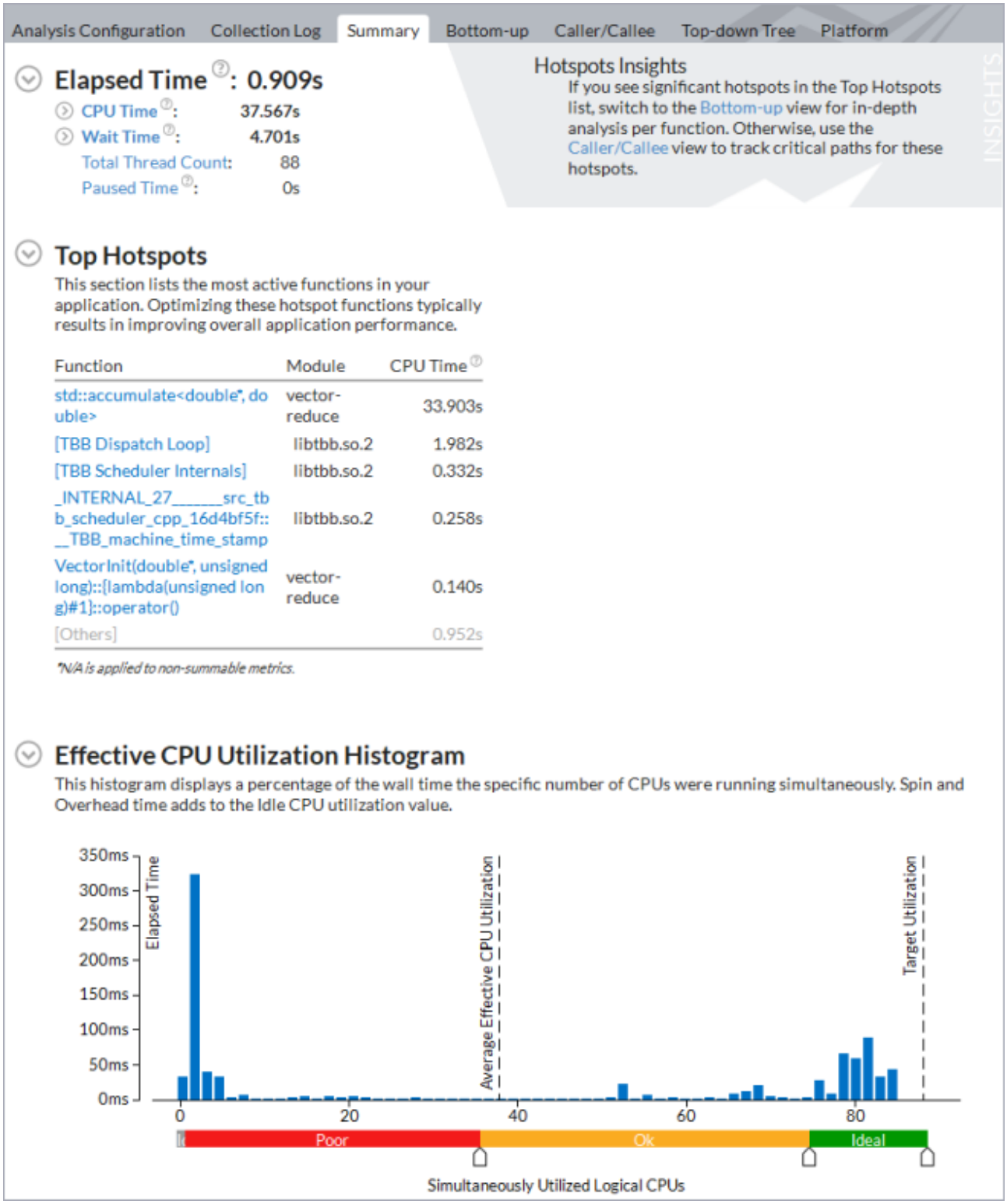
アプリケーションの実行中にインバランスを視覚化するには、**[Timeline (タイムライン)]** ビューを使用します。有効な作業を実行している時間は緑色で示され、浪費時間は黒色で示されます。

並列処理の粒度を大きくする

前述のとおり、ワーカースレッドに細粒度の作業チャンクを割り当てると、スケジューラーが作業割り当てに費やす時間を相殺できません。単純なパーティショナーを使用する `parallel_deterministic_reduce` のデフォルトのチャンクサイズは 1 です。つまり、ワーカースレッドは 1 ループ反復を実行しただけで、スケジューラーに新しい作業を要求します。次のコードのように、最小チャンクサイズを 10,000 に増やします (行 5)。

```
double VectorReduction( double *v, size_t n )
{

    return parallel_deterministic_reduce(
        blocked_range<double*>( v, v + n, 10000 ),
        0.f,
        [](const blocked_range<double*>& r, double value)->double {
            return std::accumulate(r.begin(), r.end(), value);
        },
        std::plus<double>()
    );
}
```

そして、スレッド解析を再度実行します。



アプリケーションの経過時間が大幅に減り、効果的な CPU 使用率の平均は約 38 論理コアになります (このメトリックにはウォームアップ・フェーズが含まれるため、計算フェーズの CPU 使用率は 80 コア近くに上ります)。インテル® TBB のスケジューリングやその他の並列処理関連の作業に費やされた CPU 時間はわずかです。このわずかなコード変更によって、アプリケーション全体では (収集時間を除いて) オリジナルバージョンと比較して 10 倍のスピードアップを達成しました。

注

このレシピの情報は、デベロッパー・フォーラムを参照してください。

関連情報

- スレッド解析 (英語)

PMDK アプリケーション・オーバーヘッド

このレシピは、PMDK ベースのアプリケーションのメモリーアクセスのオーバーヘッドを検出して修正する方法を説明します。

コンテンツ・エキスパート: Kirill Uhanov (英語)

不揮発性メモリー開発キット (PMDK) は、データの一貫性と耐久性を維持するためのトランザクションおよびアトミック操作のサポートを提供します。Linux* と Windows* の両方で利用可能なオープンソースのライブラリーとツールのコレクションです。詳細は、不揮発性メモリー・プログラミングのウェブサイト pmem.io (英語) を参照してください。PMDK は、高水準言語のサポートにより、不揮発性メモリー・プログラミングの採用を容易にします。現在、完全に検証済みの C/C++ サポートを Linux* で利用できます。Windows* では早期アクセスとして利用できます。

インテルの新世代の不揮発性メモリーは、メモリー層とストレージ層に加え、DRAM よりも大容量でストレージよりも高速な第 3 のメモリー層を提供します。アプリケーションは、従来のメモリーと同様に不揮発性メモリー内のデータ構造にアクセスでき、メモリーとストレージ間のデータブロック転送の必要性を排除します。

しかし、PMDK ライブラリーを利用すると、アプリケーションのパフォーマンスに影響することがあります。このレシピでは、インテル® VTune™ Amplifier でそのような問題を検出する方法を説明します。

使用するもの

以下は、パフォーマンス解析シナリオで使用するハードウェアとソフトウェアのリストです。

- **アプリケーション:** PMDK メモリー・アロケーターを使用して 2 つのベクトルの要素単位の合計を計算するサンプル・アプリケーション
- **コンパイラー:** GNU* コンパイラー。次のコンパイラー/リンカーオプションを指定します。

```
gcc -c -o array.o -O2 -g -fopenmp -I <pmdk-install-dir>/src/include -I
<pmdk-install-dir>/src/examples array.c

gcc -o arrayBefore array.o -fopenmp -L <pmdk-install-dir>/src/nondebug -
lpmemobj -lpmem -pthread
```

- **パフォーマンス解析ツール:** インテル® VTune™ Amplifier 2018: メモリーアクセス解析/高度な hotspot 解析

注

  - インテル® VTune™ プロファイラー評価版のダウンロードと製品サポートについては、https://www.isus.jp/intel-vtune-amplifier-xe/ を参照してください。
  - このクックブックのレシピはすべてスケーラブルであり、インテル® VTune™ Amplifier 2018 以降に適用できます。バージョンにより設定がわずかに異なることがあります。
  - ベータ版インテル® oneAPI ベース・ツールキット向けのバージョンから、インテル® VTune™ Amplifier の名称がインテル® VTune™ プロファイラーに変わりました。引き続き、インテル® Parallel Studio XE またはインテル® System Studio のコンポーネントとして、あるいはスタンドアロン版のインテル® VTune™ プロファイラーをご利用いただけます。
- **オペレーティング・システム:** Ubuntu* 16.04 LTS
- **CPU:** インテル® Core™ i7-6700K プロセッサー @ 4.00GHz

PMDK アプリケーションのメモリーアクセス解析を実行する

このレシピは、不揮発性メモリーを利用するサンプル・アプリケーションから始めます。このアプリケーションは、よく知られている STREAM ベンチマークの Triad カーネルを使用して、DRAM 帯域幅を最大限に利用します。

統計解析向けに大きく測定可能な計算処理にするため、ループでベクトル合計計算を繰り返しています。

```
#include <ex_common.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <libpmemobj.h>
#include <omp.h>

#define REPEATS 32


POBJ_LAYOUT_BEGIN(array);
POBJ_LAYOUT_TOID(array, int);
POBJ_LAYOUT_END(array);

int
main()
{
    size_t size = 82955000;
    size_t pool_size = 16200000000;
    int i,j;
    int multiplier = 3;

    PMEMobjpool *pop;
    char* path = "test_file1";
    if (file_exists(path) != 0)
    {
        if ((pop = pmemobj_create(path, POBJ_LAYOUT_NAME(array),
            pool_size, CREATE_MODE_RW)) == NULL)
        {
            printf("failed to create pool\n");
            return 1;
        }
```

```
    }
    else
    {
        if ((pop = pmemobj_open(path, POBJ_LAYOUT_NAME(array))) == NULL)
        {
            printf("failed to open pool\n");
            return 1;
        }
    }

    TOID(int) a;
    TOID(int) b;
    TOID(int) c;

    POBJ_ALLOC(pop, &a, int, sizeof(int) * size, NULL, NULL);
    POBJ_ALLOC(pop, &b, int, sizeof(int) * size, NULL, NULL);
    POBJ_ALLOC(pop, &c, int, sizeof(int) * size, NULL, NULL);

    for (i = 0; i < size; i++)
    {
        D_RW(a)[i] = (int)i;
        D_RW(b)[i] = (int)i+100;
        D_RW(c)[i] = (int)i+3;
    }

    pmemobj_persist(pop, D_RW(a), size * sizeof(*D_RW(a)));
    pmemobj_persist(pop, D_RW(b), size * sizeof(*D_RW(b)));
    pmemobj_persist(pop, D_RW(c), size * sizeof(*D_RW(c)));

    for (j = 0; j < REPEATS; j++)
    {
        #pragma omp parallel for
        for (i = 0; i < size; i++)
        {
            D_RW(c)[i] = multiplier * D_RO(a)[i] + D_RO(b)[i];
        }
    }

    POBJ_FREE(&a);
    POBJ_FREE(&b);
    POBJ_FREE(&c);

    pmemobj_close(pop);
    return 0;
}
```

サンプルコード内のパフォーマンスの問題を特定し、メモリーアクセスに費やされた時間を予測するには、インテル® VTune™ Amplifier を起動してメモリーアクセス解析を実行します。

1. ツールバーの **[New Project (新規プロジェクト)]** ボタンをクリックして、新規プロジェクトの名前 (例: `arraysum`) を指定します。
2. **[Analysis Target (解析ターゲット)]** ウィンドウで、ホストベースの解析として **[local host (ローカルホスト)]** ターゲット・システム・タイプを選択します。
3. **[Launch Application (アプリケーションを起動)]** ターゲットタイプを選択して、右ペインで解析するアプリケーションを指定します。

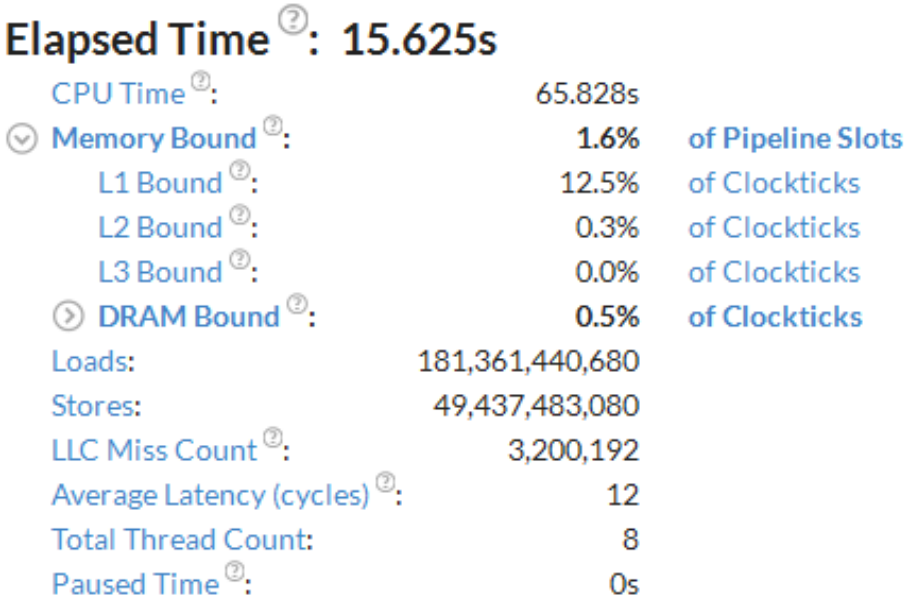
4. 右の **[Choose Analysis (解析の選択)]** ボタンをクリックし、**[Microarchitecture Analysis (マイクロアーキテクチャー解析)] > [Memory Access (メモリーアクセス)]** を選択して、**[Start (開始)]** をクリックします。

   インテル® VTune™ Amplifier は、アプリケーションを起動してデータを収集し、収集したデータをファイナライズして、シンボル情報を解決します。この情報は、ソース解析で必要になります。

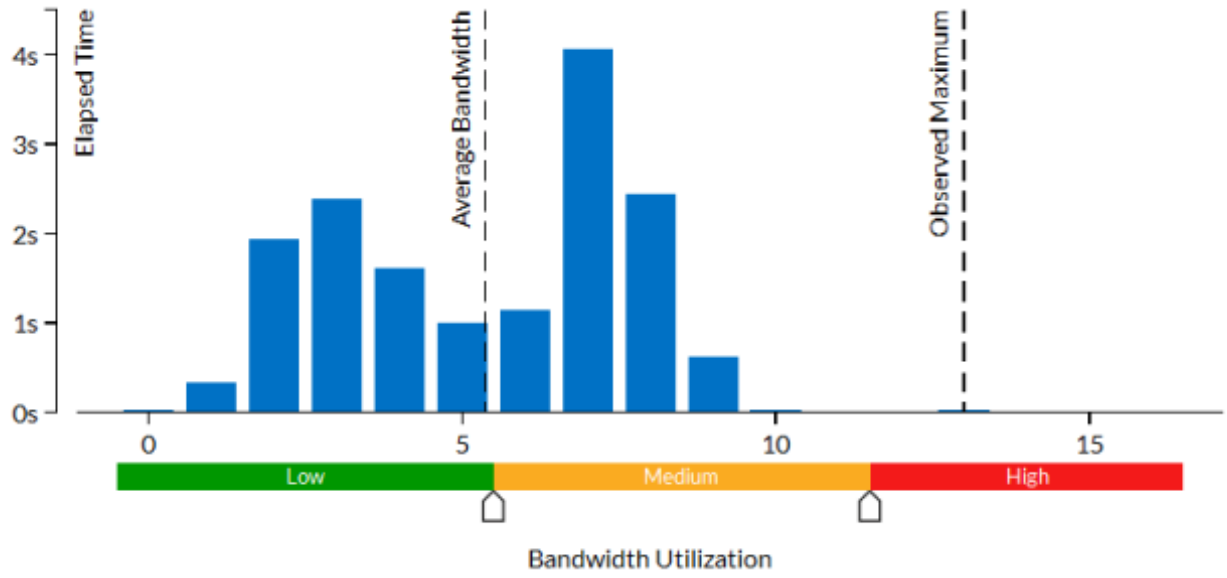
PMDK ベースのアプリケーションの hotspot を特定する

ハードウェア・メトリックごとのアプリケーション・レベルの統計が表示される **[Summary (サマリー)]** ビューから解析を始めます。通常、基本パフォーマンス・ベースラインはアプリケーションの経過時間です。このサンプルコードでは約 16 秒です。

この PMDK コードでは高い DRAM 使用率が予測されましたが、[Summary (サマリー)] ビューのメトリックはこのサンプル・アプリケーションを DRAM 帯域幅依存として定義していません。



[Bandwidth Utilization Histogram (帯域幅利用率分布図)] もアプリケーションが DRAM 帯域幅を最大限に使用していないことを示しており、**[Observed Maximum (観察された最大値)]** は予想を大きく下回る約 13GB/秒です。



PMDK によりコードのオーバーヘッドが増えているのは明らかです。詳細を確認するため、**[Bottom-up (ボトムアップ)]** ビューに切り替えて **[Function/Call Stack (関数/コールスタック)]** グループレベルを適用します。

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Memory Bound	Loads	Stores	LLC Miss Count
pmemobj_direct_inline	31.794s	1.1%	83,252,897,512	3,116,893,504	0
main._omp_fn.0	29.132s	0.0%	95,970,879,040	44,609,338,240	800,048
func@0x11c80	2.343s	52.4%	50,401,512	0	0
[vmlinux]	1.330s	32.5%	948,028,440	380,811,424	2,400,144
func@0x11b12	0.871s	49.6%	12,800,384	0	0
main	0.340s	0.0%	1,084,832,544	1,311,239,336	0
gen8_irq_handler	0.003s	0.0%	0	0	0
drm_get_last_vbltimest	0.003s	0.0%	0	0	0

最も大きな hotspot は `pmemobj_direct_inline` です。この関数は、`D_RO` マクロと `D_RW` マクロから呼び出されています。関数をダブルクリックして、*`<pmdk-install-dir>`*`/src/include/libpmemobj/types.h` にあるソースコードを表示します。

```
#define DIRECT_RW(o) \
    (reinterpret_cast < __typeof__((o)._type) > (pmemobj_direct((o).oid)))
#define DIRECT_RO(o) \
    (reinterpret_cast < const __typeof__((o)._type) > \
    (pmemobj_direct((o).oid)))

#endif /* (defined(_MSC_VER) || defined(__cplusplus)) */

#define D_RW    DIRECT_RW
#define D_RO    DIRECT_RO
```

注

アプリケーション実行中の DRAM 帯域幅の使用状況を視覚化するには、**[Platform (プラットフォーム)]** ビューを使用します。DRAM 帯域幅は、緑色と青色で表示されます。

冗長な PMDK 関数呼び出しを削除する

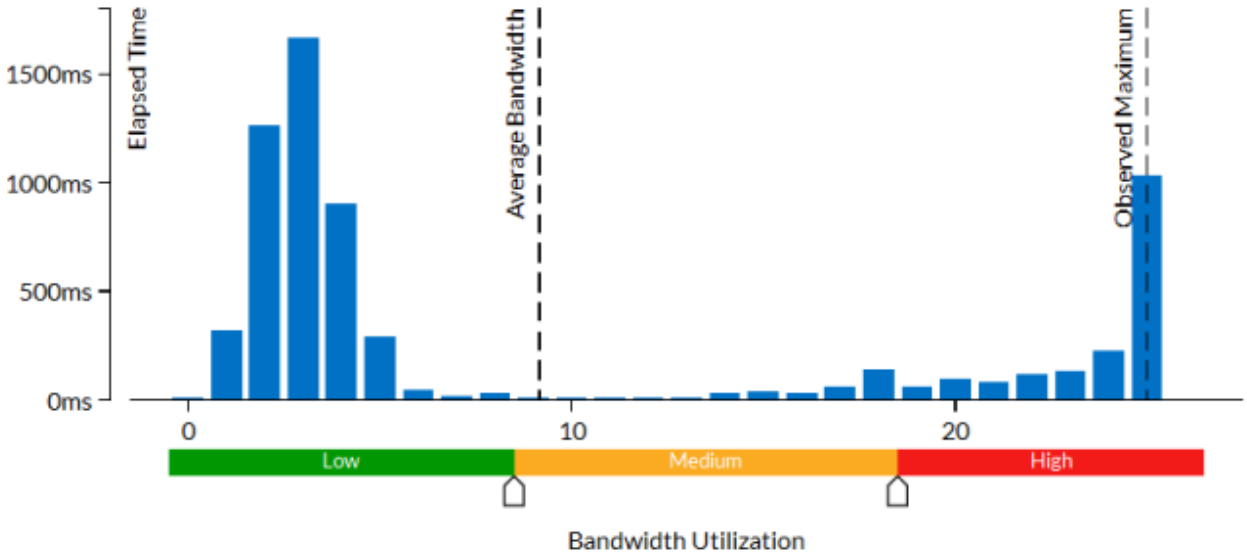
各配列のメモリーは 1 つのチャンクとして割り当てられるため、`D_RO` と `D_RW` は配列の開始アドレスを取得するため計算前に一度だけ呼び出します。

```
int* _c = D_RW(c);
const int* _a = D_RO(a);
const int* _b = D_RO(b);

for (j = 0; j < REPEATS; j++)
{
    #pragma omp parallel for
    for (i = 0; i < size; i++)
    {
      _c[i] = multiplier * _a[i] + _b[i];
    }
}
```

アプリケーションを再コンパイルしメモリーアクセス解析を再度実行して、この変更がパフォーマンスにどのように影響したか確認します。

アプリケーションの経過時間が大幅に減ったことが分ります。PMDK オーバーヘッドはパフォーマンスに影響しません。

[Bandwidth Utilization Histogram (帯域幅利用率分布図)] はアプリケーションが DRAM 帯域幅を最大限に使用していないことを示しており、**[Observed Maximum (観察された最大値)]** は約 25GB/秒です。



注

このレシピの情報は、デベロッパー・フォーラム (英語) を参照してください。

関連情報

- インテルの不揮発性メモリーを利用するプログラミングの概要 (英語)
- [Memory Usage (メモリー使用)] ビューポイント (英語)

法務上の注意書き

インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、またはサービスの有効化が必要となる場合があります。

絶対的なセキュリティーを提供できる製品またはコンポーネントはありません。

実際の費用と結果は異なる場合があります。

本資料は、(明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず) いかなる知的財産権のライセンスも許諾するものではありません。

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

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

実際の性能は利用状況、システム構成、およびその他の要因によって異なります。詳細については、www.Intel.com/PerformanceIndex (英語) を参照してください。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。