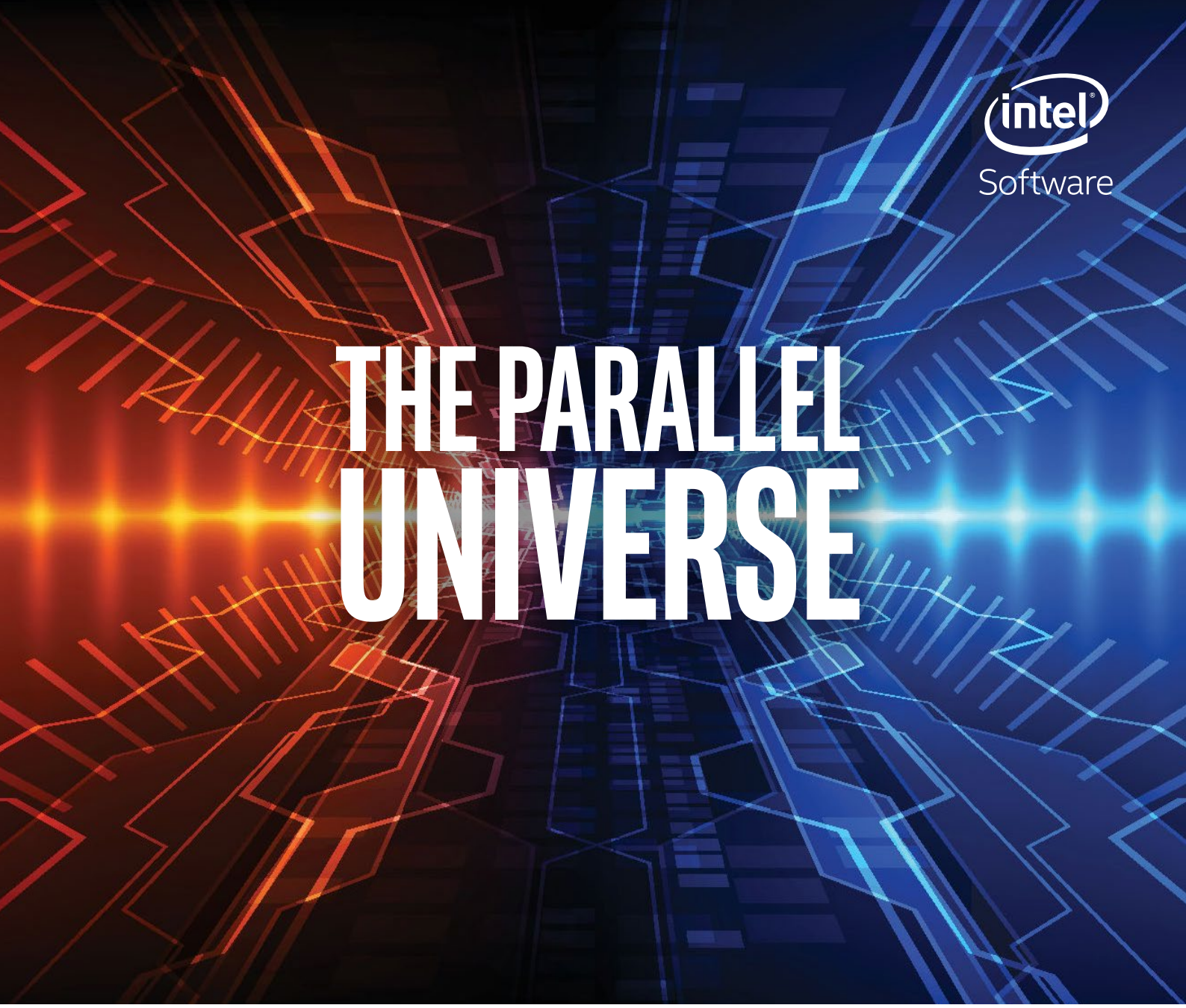


ソフトウェア定義の可視化を使用するインテル® Rendering Framework

単一クラスター上での AI、分析、HPC の統合

FPGA 向け OpenCL* の進展

Issue
35
2019

目次

編集者からのメッセージ
2019 年が皆様にハイパフォーマンスをもたらしますように 3
Henry A. Gabb インテル コーポレーション シニア主席エンジニア

注目記事

ソフトウェア定義の可視化を使用するインテル® Rendering Framework 5
インテル® Xeon® プロセッサーが可視化に優れている理由

単一クラスター上での AI、分析、HPC の統合 13
未来のエンタープライズに向けた効率の最大化とコストの軽減

FPGA 向け OpenCL* の進展 17
インテル® FPGA SDK for OpenCL* によりパフォーマンスを向上

Python* での並列処理 33
ツールにより通念を一掃して並列処理を実現

インテル® Advisor を使用してメモリー・ボトルネックを排除 39
プログラムのメモリーアクセス方法を理解してハードウェアを最大限に活用

インテル® MPI ライブラリーの MPI-3 非ブロッキング I/O 集合操作 63
HPC アプリケーションの I/O を高速化

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

編集者からのメッセージ

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

2019 年が皆様にハイパフォーマンスをもたらしますように

2019 年最初の The Parallel Universe へようこそ。2018 年の初めに、私はこれといった大胆な予測をしませんでした。並列コンピューティングの将来はヘテロジニアスであると述べただけです。この傾向はすでにかなり進んでおり、今年も勢いを増していくでしょう。今年も予測ではなく、現在注目しているいくつかの傾向について述べたいと思います。

ソフトウェア定義の可視化に関するオープンソース･コミュニティーのイニシアチブ (**SDVis.org** (英語)) は、メモリー制限と高コストの問題がある GPU ベースのソリューションよりも、CPU のほうが大規模レンダリングに適していることを実証しています。これについては本号の注目記事である「**ソフトウェア定義の可視化を使用するインテル® Rendering Framework**」で説明します。SDVis の利点は、長年 CPU ベースのレンダリングを行ってきた映画業界にとっては目新しいものではありませんが、SDVis は増え続けるデータセットの可視化が必要とされるほかの計算ドメインにも広がっています。

これは私が特に注目しており、最近**ブルックヘブン国立研究所で開催された 2018 New York Scientific Data Summit** (英語) で紹介した別の傾向「**将来のワークロードにおける HPC、BDA、および AI の収束**」(英語) につながります。インテル コーポレーションの副社長兼エクストリーム・コンピューティング本部長の Trish Damkroger は、最近 Top500.org の記事「**AI、HPC、HPDA の交差 : 次世代ワークフローが将来のブレークスルーをどのように推進するか**」(英語) で同様の見解を示しました。従来のハイパフォーマンス･コンピューティング、人工知能 (AI)、ビッグデータ分析の間の境界線はあいまいになりつつあります。これに関して、インテルのデータセンター・グループに寄稿を依頼した記事が「**単一クラスター上での AI、分析、HPC の統合**」です。

以前に述べたとおり、並列コンピューティングはヘテロジニアスへと向かっており、ソフトウェアを高速化するオフロードデバイスとして FPGA が注目を集めています。The Parallel Universe の名誉編集者である James Reinders は、昨年 FPGA プログラミングに関するいくつかの記事を発表しました。本号では、ボストン大学の Martin Herbordt 教授が FPGA 上での OpenCL* プログラミングに関するいくつかのベスト・プラクティスについて、「**FPGA 向け OpenCL* の進展**」で一般的な数値アルゴリズムの最適化を通して説明しています。

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

この号を締めくくるのは、コードの最適化に関する次の 3 つの記事です:「**Python* での並列処理**」、「**インテル® Advisor を使用してメモリー・ボトルネックを排除**」、「**インテル® MPI ライブラリーの MPI-3 非ブロッキング I/O 集合操作**」。

今後の The Parallel Universe では、Python* コードを最適化する JIT コンパイル、インテル® ソフトウェア開発ツールの新機能、パフォーマンス・ケース・スタディーに関する記事をお届けします。お見逃しなく。

また、コードの現代化、ビジュアル・コンピューティング、データセンターとクラウド・コンピューティング、データサイエンス、システムと IoT 開発向けのインテルのソリューションの詳細は、**Tech.Decoded** (英語) を参照してください。

Henry A. Gabb
2019 年 1 月

ソフトウェア定義の可視化を使用するインテル® Rendering Framework

インテル® Xeon® プロセッサーが可視化に優れている理由

Rob Farber TechEnablement グローバル・テクノロジー・コンサルタント

ソフトウェア定義の可視化 (SDVis) は、ソフトウェア定義ネットワーク、ソフトウェア定義インフラストラクチャー、および多数のスレッドとデータ並列処理を利用してソフトウェアにより最近の**インテル® Xeon® プロセッサー**の利点 (およびパフォーマンス) を最大限に引き出すためのインテルのその他の取り組みに似ています。優れたパフォーマンスを備え、メモリー制限のある専用デバイスと比較して多岐にわたる利点を提供します。これには、進化し続ける高度なアルゴリズムを導入することで得られる、次のような利点も含まれます。

- **大容量メモリー**を搭載したプロセッサー
- ハードウェアの換装と比較して、**柔軟でアップグレードが容易な**ソフトウェア
- 調達**コスト全体の削減**とハードウェアの寿命期間中の総所有コストの改善

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

可視化を再定義したパフォーマンスとスケーラビリティー

インテル コーポレーションのビジュアライゼーション・ソリューション担当シニア・ディレクター兼主席エンジニアの Jim Jeffers は、「**インテル® Rendering Framework** を使用することで、すべての処理を CPU 上で行い、専用グラフィックス・ハードウェアと同等またはそれ以上の体験が得られます。」と述べています。インテル® Rendering Framework は、**インテル® Embree** (英語)、**インテル® OSPRay** (英語)、および **インテル® OpenSWR** (英語) ライブラリーを介してスケーラブルでインタラクティブなレイトレーシングと OpenGL* による可視化を提供します。さらに、インテル® Rendering Framework には、新たに**インテル® Open Image Denoise Library** (英語) が追加されました。

当然のことながら、マルチコアおよびメニーコア・プロセッサーに搭載される高スループットのプロセッサー・コアは、多数のタスクをインタラクティブに実行することができ、以前のプロセッサーとは比較にならないパフォーマンスを実現します。「Mesa を使用した OpenGL* の三角形のレンダリング・ベンチマークでは、2016 年と比較して 100 倍のレンダリング・パフォーマンスを達成しました。」と Jeffers は指摘しています。

このレベルのパフォーマンスは科学的な可視化を再定義し、映画およびプロのビジュアライゼーション分野に深く浸透しています (**図 1**)。「利用可能な CPU メモリー (一般に CPU では 192GB 以上、ハイエンドの GPU では 16GB) を活用して、インテル® Rendering Framework は GPU とは比較にならない精度と同等以上のパフォーマンスを実現します。」と Jeffers は述べています。さらに、可視化タスクの規模に関係なく、対話に特別なハードウェアを必要とせず、さまざまなプラットフォーム環境で実行して可視化することができるため、ハイパフォーマンス・コンピューティング (HPC) センターでは可視化クラスター用の GPU を調達する必要がなくなりました。

1 **SDVis テクノロジーを利用するインテル® Rendering Framework は、クラウドと HPC クラスターを含むあらゆる規模のプラットフォームでレンダリングをサポート**

主要な HPC ビジュアル分析アプローチ

5 年前には「画像のレンダリングに CPU をよく使用する」という HPC ユーザーはいなかったでしょう。しかし、多くの HPC センターで CPU ベースのインタラクティブで写実的なレンダリングが GPU を凌駕したことでこれは変わりました。Texas Advanced Computing Center (TACC) のビジュアライゼーション・ディレクターの Paul Navrátil 氏は、「CPU ベースの SDVis は、インテル® Rendering Framework スタックを利用した、Frontera システムの主要なビジュアル分析モードになります。」と TACC の取り組みを強調しています。Frontera は 2019 年の稼働開始時に米国で最も速い学術スーパーコンピューターになると予想されています。

ロスアラモス国立研究所の Trinity スーパーコンピューターで Kitware*[1] が実施した 1.1 兆個の三角形を使用する OpenGL* Hero ベンチマークが示すように、一言でいえば、" 素晴らしい " スケーラビリティーです。しかし、SDVis を実行するのにスーパーコンピューターは必要ありません。インテル® OSPRay ライブラリーなどのインテル® Rendering Framework コンポーネントを ParaView* に統合することで、ほとんどのハードウェア・プラットフォームでレイトレーシングの利点を簡単に利用できます。Kitware* の主席エンジニアの David DeMarle 氏は、「インテル® Rendering Framework を利用することで、VTK および ParaView* ユーザーは 1 行変更するだけでインテル® OSPRay のレイトレーシングと OpenGL* レンダリングを切り替えることができます。」と述べています。

従来のバッチと新しい in-situ および in-transit 可視化ワークフロー

インテル® Rendering Framework はソフトウェア定義であるため、サイエンティストは計算と同じノードを使用して可視化する in-situ レンダリングを実行できます。in-situ 可視化は、エクサスケールでサイエンスを可能にする主要テクノロジーとして認識されてきました。[2] Jeffers は、「エクサスケールへの移行に伴い、エクサバイトのデータを管理する必要があります。データは計算できますが、I/O システムはまだエクサバイト・データの移動に対応していません。よって、in-situ が必要になります。さもないと、可視化には数日、数週間、あるいは数か月を要するでしょう。」と述べ、「1 枚の絵はエクサバイトに匹敵する。」(訳者注 : 1 枚の絵は一千語に匹敵する、の例え) と要約しています。

エクサスケールの可視化への道のり

アルゴンヌ国立研究所は、米国エネルギー省 (DOE) の複数機関における取り組みの一環として、民間企業やほかの国立研究所と共同で **SENSEI** (英語) フレームワークを利用して、インテルと Cray* の新しい共同開発システム Aurora の登場に向けて人々を支援しています。Aurora は、EXAFLOPS を超える浮動小数点パフォーマンスを実現できます。SENSEI は、エクサスケール・スーパーコンピューターにより生成される膨大なデータの分析とスケーラブルでインタラクティブなレンダリングのため、in-situ、in-transit、および従来のバッチ可視化ワークフローを利用可能な移植性の高いフレームワークの一例です。

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

アプリケーションによっては、研究者はレンダリング用のノード数よりも、計算負荷の高いシミュレーションにより多くのスーパーコンピューター・ノードを割り当てたい場合があるでしょう。この非対称ロードバランスは in-transit 可視化と呼ばれます。データをノード上の所定の位置にレンダリングする in-situ とは異なり、in-transit 可視化では、データが通信ファブリックを介してノード間を移動しなければならないため、オーバーヘッドが生じます。しかし、追加の計算能力をシミュレーションに使用できます。in-transit と in-situ ワークフローはどちらもデータをメモリーに保持し、ストレージへの書き込みは行いません。アルゴンヌ・リーダーシップ・コンピューティング施設のビジュアライゼーションおよび解析チームリーダーの Joseph Insley 氏は、次のように述べています。「SENSEI を利用することで、ユーザーは in-situ および in-transit 手法を使用して、フル解像度の I/O 負荷の高い事後解析を不可能にしている FLOPS と I/O 性能の間の拡大し続けるギャップに対応できます。」

特別なハードウェアを必要としない誰でも利用可能な可視化

CPU ベースのレンダリングの大きな利点は、特別なハードウェアが不要なことです。つまり、ラップトップやワークステーションから、組織的なクラスターやトップクラスのスーパーコンピューター、そしてクラウドまで、あらゆる計算ハードウェアで、誰でも利用できます。

インタラクティブで写真のようにリアルなレイトレーシングは、8 つの**インテル® Xeon® Platinum 8180 プロセッサー**で行うことも、in-situ ノードを使用するビッグデータの高品質なレンダリングにスケーリングすることもできます。[3,4,5,6] Jeffers は、「インテル® Rendering Framework によってもたらされるインタラクティブなパフォーマンスと、無料のインテル® OSPRay ライブラリーとビューアーによる写実的なレンダリングが " ニーズに応え、要求を生み出します "。」と述べています。特別なディスプレイ・ハードウェアの必要性を排除することは、エクサバイトのシミュレーション・データであっても「どこでも可視化」できることを意味します。ユーザーは、ラップトップで結果を表示し、壁や完全没入型 Cave の表示に切り替えられるようになります。

可視化タスクの規模に関係なく、対話に特別なハードウェアを必要とせず、CPU を使用してどこでも実行して可視化できるため、ハイパフォーマンス・コンピューティング (HPC) ユーザーは可視化タスクに CPU を使用するようになりました。SENSEI などのフレームワークとともに、一般的な VisIt*[7] や ParaView* ビューアーとインテル® Rendering Framework の SDVis 機能を統合することで、誰でも解析を実行したり、OpenGL* レンダリングを使用したり、写真のようにリアルな画像を作成できます。

図 2 は、ハードウェア定義の可視化と比較したソフトウェア定義の可視化の利点です。

OpenVINO™ ツールキット
マルチプラットフォーム向けコンピューター・ビジョン・ソリューションの開発
無料
ダウンロード

2 **ハードウェア定義の可視化と比較したソフトウェア定義の可視化の利点**

HPC からレンダリング・アプリケーションまで

Jeffers は、SDVis の採用を推進している主な要因は、レイトレーシングの視覚的な精度であると見ています。基本的に、ソフトウェアはスケーラブルでインタラクティブなパフォーマンスとともに、CPU 上でシリアル処理と並列処理の両方を使用して光の物理特性をモデル化できるため、ユーザーはフォトリアリズムが得られます。

SDVis テクノロジーを利用するインテル® Rendering Framework が複数の分野において魅力的であることは明らかです。Jeffers は、次のように述べています。「CAD や自動車などの分野で高い需要があります。フォトリアリズムは、民間航空機から軍用車両にいたるまで、「仮想」車両の設計と製造を改善する上で非常に重要です。基本的に、実際に製造することなく、製造する車両に関する仕様を決定できます。その一方で、アニメーションや写真のようにリアルな視覚効果のオフラインおよびインタラクティブなレンダリングを含む分野でも需要が高まりつつあります。」

すべてはコストの分離

ソフトウェアの観点から、インテル® Rendering Framework はチューニングおよび最適化された低レベルの操作を提供します。これが、レンダリング API を呼び出すだけでアプリケーション開発者に優れたパフォーマンスを提供できる理由であると Jeffers は主張しています。また、分散環境で実行するスケーラビリティーも備えており、これは、プロでなくても大規模で複雑なデータセットに対して完全な視覚効果に対応した「インタラクティブ」[8] なレンダリングとレイトレーシングが可能であるという大きな前進につながりました。これこそが、映画スタジオが数千ものインテル® CPU で構成されたレンダリング・ファームを所有している理由です。

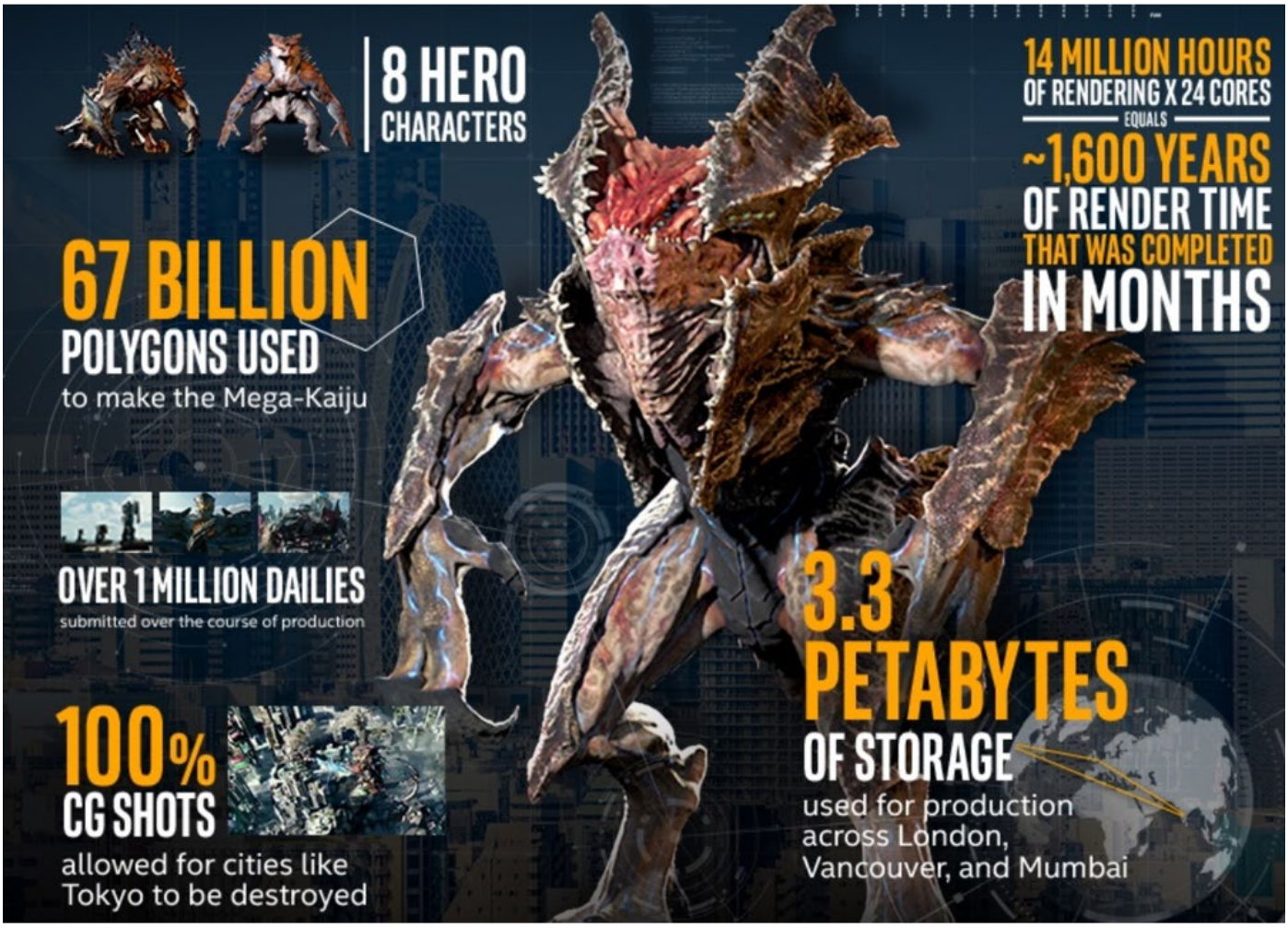
Jeffers は、アカデミー賞 3 部門にノミネートされた 1989 年公開の映画「The Little Mermaid」(邦題 :「リトル・マーメイド」) のアニメーションと 2016 年公開の「Moana」(邦題 :「モアナと伝説の海」) のイメージの違いから、インテル® Rendering Framework を使用したレイトレーシングによって可能な改善点を指摘しています。以前は、数秒のビデオを生成するために一晩かけてレンダリングを行っていました。研究および業界のベスト・プラクティスのため Walt Disney Animation Studios によって最近公開された、図 3 に示す 1,600 億個のオブジェクトで構成される**モアナの島のシーン** (英語) は、インテル® OSPRay、インテル® Embree レイトレーシング・ライブラリー、および**インテル® Open Image Denoise Library** (英語) を使用してライブ・レンダリングされました。このレンダリング処理は 100GB を超えるシステムメモリーを消費するため、システムメモリー容量が重要でした。

3 **1,600 億個のオブジェクトを含むこのイメージは、インテル® Rendering Framework とインテル® Open Image Denoise Library を使用してライブでレイトレーシングされました (画像提供 : Walt Disney Animation Studios)**

今後の動向

Jeffers はまた、人工知能 (AI) のコンバージェンスとインテル® OSPRay とインテル® Embree のレイトレーシング機能にも注目しています。例えば、映画「Pacific Rim」(邦題 :「パシフィック・リム」) では、これらのライブラリーを使用してレンダリングされるロボットで、信頼できる動きの定義に AI が使用されました (**図 4**)。インテル® Xeon® スケーラブル・プロセッサーは、Ziva* AI ソフトウェアに身体の動きを漸進的に学習可能なリアルタイム・キャラクターの生成に必要なパフォーマンスをもたらすだけでなく、あるキャラクターから別のキャラクターへ機能や行動を簡単に適用できるようにします。[9]

写真のようにリアルなアニメーションが俳優に取って代わる可能性について Jeffers は、映画が要求する心理的な影響を与えるためには人間が必要であると考えています。しかし、最先端のレンダリング・テクノロジーのビジュアル精度が向上し続けると、吹き替えや俳優の役割がより重要になるほどテクノロジーが進化するかもしれません。



4 **AI とレイトレーシングを使用することで映画「Pacific Rim」では生きているリアルなキャラクターを実現 (画像提供 : インテル (英語))**

CPU ベースの可視化

前述のとおり、インテルではインテル® Rendering Framework のほかにも、現代のメニーコアおよびマルチコアのインテル® Xeon® プロセッサーのシリアルおよび並列パフォーマンスを利用して専用ハードウェア・デバイスを置き換えるための取り組みを行っています。インテル® Rendering Framework で作成された壮観なイメージは、CPU ベースの可視化の魅力を明確に示しています。これらのソフトウェア・ライブラリーはオープンソースであり、ダウンロードできます。

開発を行わず SDVis のみ体験したい場合は、ParaView* や VisIt* アプリケーション、または最近発表されたインテル® OSPRay Studio ビューアーを利用できます。一方、HPC 開発者は SENSEI などのフレームワークを使用して、大規模な in-situ と in-transit 可視化を実行できます。

SDVis の利点に興味がある組織ユーザーは、最新のインテル® Xeon® スケーラブル・プロセッサーと、**インテル® Omni-Path アーキテクチャー** (英語)、**インテル® SSD**、**OpenHPC*** (英語) クラスター・ソフトウェア・スタックなどのその他のテクノロジーを組み合わせた検証済みハードウェアとソフトウェア・ソリューションについて、「**プロフェッショナル・ビジュアライゼーション向けインテル® Select ソリューション**」を参照してください。

以下のサイトでは、アプリケーション開発者向けの情報が得られます。

- **インテル® Embree レイトレーシング・ライブラリー** (英語)
- **インテル® OSPRay 分散レイトレーシング・インフラストラクチャー** (英語)
- **インテル® OpenSWR OpenGL* ソフトウェア・ラスタライザー** (英語)
- インテル® Open Image Denoise Library は、近日中に **https://openimagedenoise.github.io/** (英語) で公開予定です。

記事の内容から外れますが、その他のインテルの取り組みに興味がある方は、**インテル® Software Defined Networking** と**インテル® Software Defined Infrastructure** (英語) でインテル® Xeon® スケーラブル・プロセッサーによりその他の専用ハードウェア・デバイスを換装する方法をご覧いただけます。

Rob Farber は、HPC 分野と国立研究所や営利団体で応用されているマシンラーニング・テクノロジーの開発において幅広い知識を持つ著者であり、グローバル・テクノロジー・コンサルタントです。Rob の連絡先は **info@techenablement.com** です。

参考文献 (英語)

[1] ベンチマークは 1.1 兆個の三角形のレンダリングにマシンの 1/19 しか使用しませんでした。Kitware* は、マシンをフル活用すれば、1 秒あたり 10-20 兆個の三角形をレンダリングできると考えています。(**http://www.techenablement.com/third-party-use-cases-illustrate-the-success-of-cpu-based-visualization/**)

[2] **https://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf**

[3] **http://sdvis.org/**

[4] **http://www.cgw.com/Press-Center/In-Focus/2018/Scalable-CPU-Based-SDVis-Enables-Interactive-Pho.aspx**

[5] **https://www.ixpug.org/documents/1496440983IXPUG_insitu_S1_Jeffers.pdf**

[6] **http://www.techenablement.com/third-party-use-cases-illustrate-the-success-of-cpu-based-visualization/**

[7] **https://tacc.github.io/visitOSPRay/**

[8] インタラクティブは、流動的なリアルタイムのフレームレートを意味しません。

[9] **https://ai.intel.com/ziva-pacific-rim/**

単一クラスター上での AI、分析、HPC の統合

未来のエンタープライズに向けた効率の最大化とコストの軽減

Allene Bhasker インテル コーポレーション データセンター・グループ ソリューション・アーキテクト
Keith Mannthey インテル コーポレーション データセンター・グループ ソリューション・アーキテクト

今後数年間は、マシンラーニングとディープラーニングを含む人工知能 (AI) が企業全体で主流になった時代として記憶されるでしょう。ある調査によると、現在、60% を超える企業が AI ソリューションを導入しており、最もよく使用されているアプリケーションは予測分析であると報告されています。

AI、分析、HPC ワークロードのホスティング戦略

AI ワークロードと AI 駆動型分析のホスティング方法を決定する際、多くの IT 組織は専用のアクセラレーターと GPU を搭載した専用サーバーの導入を検討します。将来を見据えている組織では、日常業務における AI の役割の成長を見越して、これらのサーバーから成るクラスターの構築を検討するかもしれません。より幅広い観点では、AI、分析、HPC ワークロードはすべて、パワフルなコアと高速なインターコネクトにより同様のクラスター・ハードウェアで実行できると考えられます。

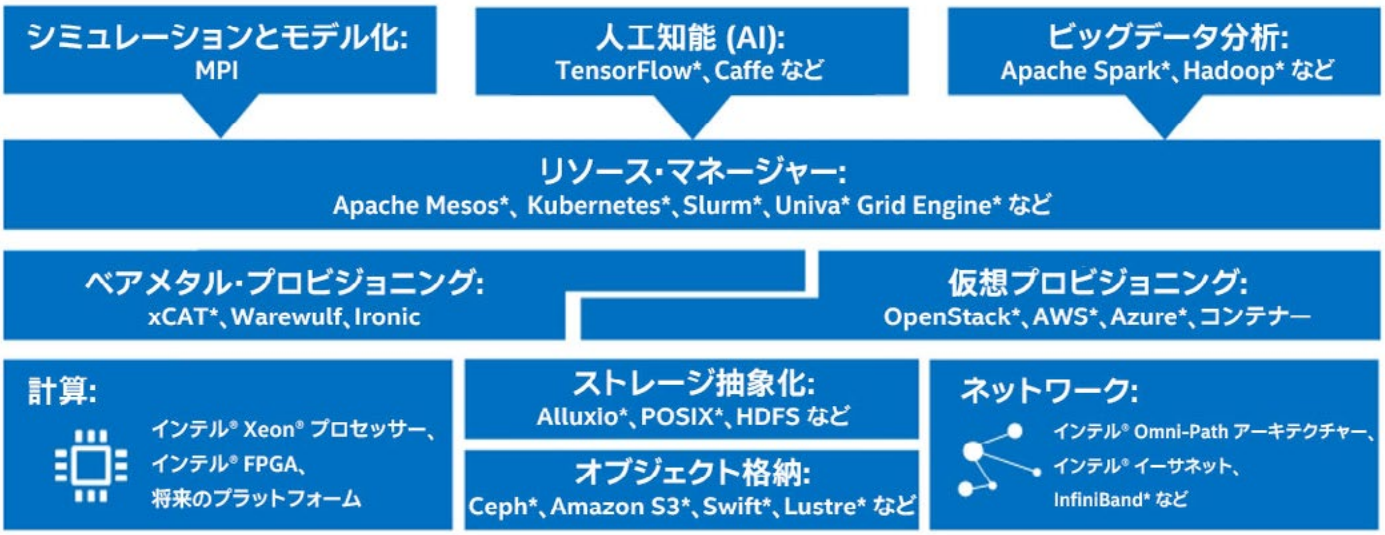
コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

ここで、"同じクラスター上で AI、分析、および HPC ワークロードを実行するには何が必要か？" という一般的な疑問が生じます。このアプローチは、これら 3 つの処理を利用するビジネスプロセスにとって特に魅力的です。例えば、シミュレーションとモデリング、データ・クリーニング、AI ベースの推論ステップをすべて同じクラスター上で実行するほうが、個別のクラスターを保有するよりもはるかに効率的です。

1 つのクラスターにまとめることは、費用面でもメリットがあります。単一クラスターのほうがサーバー使用率が高いため、導入するサーバー数を抑えることができます。単純な環境は、構成と保守にかかる費用が少なくて済み、複数のクラスター間でデータを移動したり、ステージングする費用を回避できます。これらのワークロードを 1 つの環境に統合することは、レイテンシーの軽減にもつながります。この重要性は、リアルタイムの要件が高まるにつれて、年々増しています。

既存の HPC プラットフォームで AI と分析機能を構築

統合アプローチは、HPC クラスター上に AI と分析機能を追加します。インテル® HPC Platform Specification は、インテルベースのソリューションの一般的な業界標準とプラクティスを含む、基本クラスター・ソリューションの要件を定義しています (**図 1**)。これは、HPC アプリケーションに共通の一貫したインターフェイスを提供し、多くの商用 HPC ソフトウェア・ベンダーがこのプラットフォーム要件に準拠するソリューションのサポートを検証してきました。



1 **統合クラスター向けの汎用インテル・ソリューション・スタック**

基本定義に加えて、特定の機能に対する追加要件が個別のセクションで説明されています。準拠ソリューションは、要件を満たす基本ソリューションと機能レイヤーで構成されます。これにより、プラットフォームをターゲットとするアプリケーションとの相互運用性を維持しつつ、機能の導入や拡張を合理的に行うことができます。統合プラットフォームへの道のりには、AI と分析機能の要件を定義した新しいセクションをインテル® HPC Platform Specification に追加することが含まれます。

さまざまな顧客環境向けに構築されたソリューションを組み合わせる

インテルは、AI、分析、および HPC ワークロードを 1 つの統合クラスターにまとめる要件定義を支援するため、一連のソリューション・アーキテクチャーを開発しています。複数のリソース・マネージャーを円滑に連携させることは容易ではありません。これらのソリューションは、あらゆる種類のワークロードに対して、ジョブキューの維持やジョブのスケジューリングなどの機能を集中的に統合するさまざまなアプローチを実装しています。

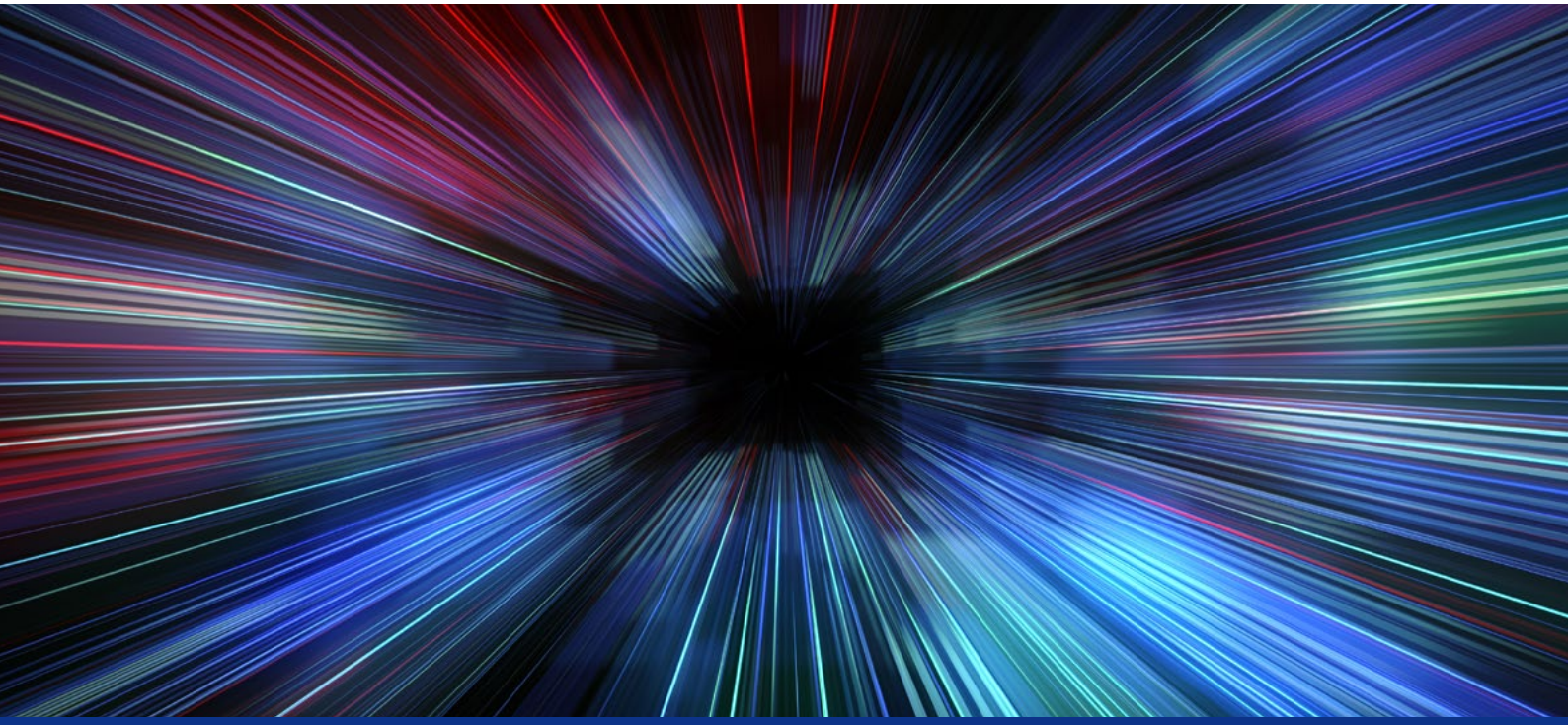
- **ソリューション 1: HPC バッチ・スケジューラーの拡張。**このアプローチは、AI と分析ワークロードの代わりにジョブを送信するラッパースクリプトを使用してバッチ・スケジューラーを拡張します。この単純なアプローチは、システム・オーバーヘッドをほとんど発生させません。
- **ソリューション 2: Univa* Grid Engine* と Resource Broker。**すでに Univa* Grid Engine* を使用している場合、このソリューションは Univa* Resource Broker* を使用して Apache Mesos* 互換の AI および分析ソフトウェアを統合します。
- **ソリューション 3: Apache Mesos* とバッチ・スケジューラー。**この次期ソリューション・アーキテクチャーは、Apache Mesos* とバッチ・スケジューラーを統合して HPC、AI、および分析ワークロードにわたってシームレスに連携します。

このソリューションは、ベアメタル上でも、ハイブリッド・クラウド上の仮想マシンやコンテナーであっても、さまざまなプロビジョニング方法をサポートするという点で柔軟です。また、ストレージを抽象化してオブジェクト・ストア間でデータを統合し、大規模なデータ移動を行うことなく、その場で使用される単一のデータソースを提供します。インテルは、オープンソースへの貢献やテクノロジー・プロバイダーとの共同開発を含む、ソフトウェア・エコシステム全体のイネーブルメント・アクティビティーに取り組んでいます。この最適化作業は、3 つのワークロードすべてがインテル® プラットフォームのパフォーマンスおよびセキュリティー機能を最大限に利用できるようにするための鍵です。

統合クラスター・ソリューションの配備を容易にするため、インテルは協賛 OEM を介して最適化済みの統合インフラストラクチャーである、**インテル® Select ソリューション**を提供しています。これらのアーキテクチャーは事前検証されているため、企業は将来の AI、分析、および HPC ワークロード向けの統合クラスターの効率と費用の利点を明確に把握することができます。

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

intel
Software
より優れた
より高速な
コードのビルド
インテルの強力なパフォーマンス・ライブラリーを入手して
コードを最適化し、開発時間を短縮しましょう。インテル®
アーキテクチャー上での革新と素晴らしいパフォーマンスを
サポートするというインテルのミッションに基づき、これらの
ライブラリーは無料でご利用いただけます。
無料のダウンロード (英語) >
コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation またはその子会社の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation.



FPGA 向け OpenCL* の進展

インテル® FPGA SDK for OpenCL* によりパフォーマンスを向上

Martin C. Herbordt ボストン大学 電気・コンピューター工学部教授

FPGA (Field programmable gate array) は、非常にハイパフォーマンスで、特に電力パフォーマンスに優れています。これは驚くべきことではありません。FPGA ハードウェア自体は柔軟で、アプリケーションに合わせて構成することができます。またこれも驚くべきことではありませんが、アプリケーション開発者がソフトウェアだけでなくハードウェアも変更できるため、アプリケーション開発ワークフロー全体がさらに複雑になります。

事実、最近まで、FPGA アプリケーションのほとんどの開発者が、ソフトウェア・プログラマーよりもハードウェア設計者が使用しているものと共通点が多いツールや手法に頼ってきました。使用言語は、Verilog* や VHDL* などのハードウェア記述言語 (HDL) で、これらの言語は命令フローよりもロジックの性質を表現します。コンパイル時間 (合成と呼ばれる) が長く、特にデバッグとテストには非常に多くのシステムの知識が必要です。

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

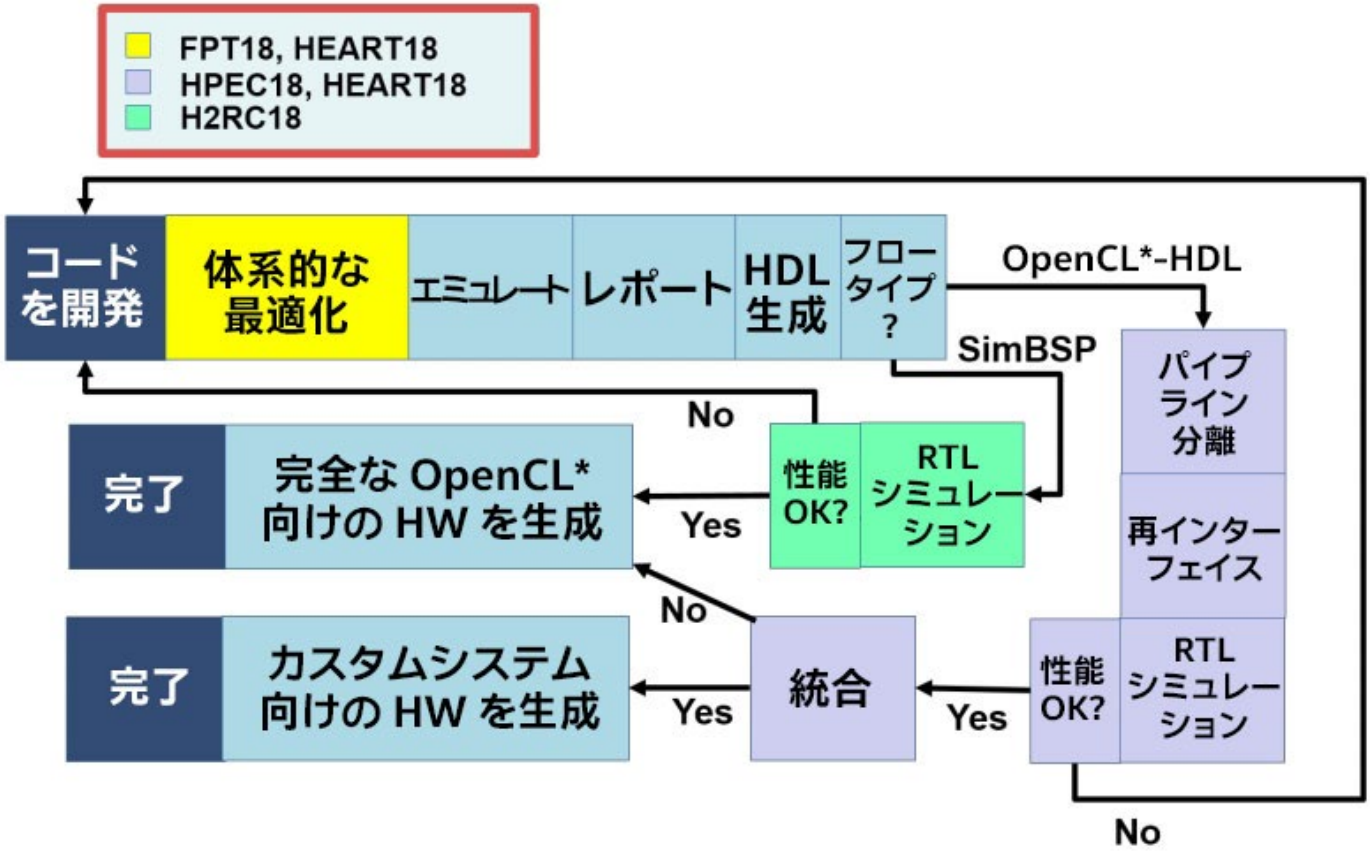
これらすべてが変わりつつあります。インテルは、HDL プログラミングの代替手段となる**インテル® FPGA SDK for OpenCL***[1] (英語) を開発しました。このテクノロジーと関連ツールは、高レベルの抽象化により設計を表現できるハイレベル合成 (HLS) と呼ばれる手法に属します。インテル® FPGA SDK for OpenCL* は、現在広範に使用されています。驚くべきことに、長年 FPGA アプリケーション開発者は、HDL コードに近いか、それを上回るパフォーマンスを達成してきました。しかし、このパフォーマンスを達成できるのは、通常 C 言語からハードウェアへの変換作業に精通し、内製の最適化ツールキットを持っている開発者に限られていました。

ボストン大学では、そのような最適化ツールキットの 1 つとして列挙、特徴付け、および体系化に取り組んできました。FPGA OpenCL* ドキュメントには、すでに多くのベスト・プラクティスがあります。ここでは、主にハイパフォーマンス・コンピューティング (HPC) コミュニティーでよく知られている追加の手法を適用することでこれまでの取り組みを拡張します。[2] これが理にかなった方法であると我々は信じています。HPC パフォーマンス・プログラミングが現在の高度なレベルに達するには数十年かかりました。インテル® FPGA SDK for OpenCL* のユーザーが同様の道のりと学習カーブをたどらざるを得ないことは、驚くべきことではありません。

ここで紹介する取り組みの詳細は、この記事の最後にある参考資料の 3 と 4 を参照してください。最初に、詳細なケーススタディーとして FFT を使用します。次に、経験に基づいた最適化フレームワークを説明します。関連する取り組みとして、参考資料の 5 と 6 は、システム全体のハードウェアを生成せずに設計機能とパフォーマンスのテスト / 検証に使用できるツールフローの拡張を示します。これにより、設計上のボトルネックと最適化の影響を正確に特定して、迅速に改善できます。**図 1** は、これらを既存のツールフローに統合したものです。

経験に基づくコードの最適化

ボストン大学では、現在のベスト・プラクティスを拡張し、パフォーマンスを大幅に向上する体系的で経験に基づいたコードの最適化を提案してきました。我々は、これらすべての最適化の影響を分類し、測定しました。これは、プログラマーがスクリプトに従って自身のカーネルを最適化できるようにするだけでなく、自動的に最適化する自動チューナーの開発の道も拓きます。



1 **青色はインテル® FPGA SDK for OpenCL* の標準ツールフロー。黄色、緑色、紫色は拡張部分。この記事では体系的な最適化を説明します。**

このドメインのコード最適化は、次の 3 つに分類されます。

1. インテルのベスト・プラクティス (IBP)
2. 汎用的なコード最適化 (UCO)
3. FPGA 固有の最適化 (FSO)

IBP は、OpenCL* セマンティクスを使用してハードウェアを表現する方法を示す、インテルのベスト・プラクティス・ガイド[7] に記載されている設計手法です。IBP は FPGA OpenCL* コミュニティーではよく知られており、動作を特徴付けるいくつかの研究が行われているため、UCO や FSO とは区別しています。

UCO は計算プラットフォームから独立して、プログラムを最適化する一般的なアプローチから成ります。次に例を示します。

- 1D 配列の使用
- 配列レコード
- プレディケーション
- ループ結合
- スカラー置換
- 定数の事前計算

これらはさまざまな資料 (例えば、参考資料 2) に記載されていますが、IBP ドキュメントではほとんど取り上げられていません。

FSO は通常 IBP を補完する FPGA 固有の最適化から成ります。以下をベースにしています。

- IBP にはない特定の FPGA 固有のマッピングの取得
- IBP に記載されているものを利用して最適化に変換したもの
- よく使用されているが回避すべきプラクティス

7 つのコードバージョンがあり (詳細は参考資料 4 と 6 を参照)、それらは段階的に開発されています。各バージョンには、1 つ以上の最適化が適用されています。**表 1** は、最適化とそのタイプ (IBP、FSO、UCO) の要約です。

表 1. コードバージョンと最適化の要約

バージョン	最適化	タイプ
0	(FPGA OpenCL* に移行するオリジナルの **GPU コード**)	—
1	**キャッシュの最適化を含むシングルスレッド・コード**	IBP、FSO
2	**タスク並列計算を別々のカーネル**で実装してチャネルで接続	IBP
	`#pragma unroll` ですべてのループを完全にアンロール	IBP、UCO
	計算ループの外側での変数宣言を最小化 (可能であれば一時変数を使用)	IBP、UCO
	問題サイズとデータ値に**定数を使用** (オフチップ・メモリー・アクセスに依存しない)	IBP、FSO、UCO
	メモリー操作の**融合**	IBP、UCO
3	**単一のカーネル**内に計算全体を実装してチャネルの使用を回避	FSO
4	パイプライン・レジスターを BRAM ではなくレジスターとして推論するため配列サイズを縮小	FSO
5	一時変数を使用して**詳細な計算**を実行し、中間結果を格納	FSO、UCO
6	データパスで fork を定義する際に条件付き分岐の代わりに**プレディケーションを使用**	FSO、UCO

バージョン 0: 最適化されていないベースライン・コード

一般的な開始点 (例えば参考資料 8) は、GPU などに適した MWI (Multiple Work Items) カーネルです。ここから始める利点には、SIMD によるデータ並列処理や MWI 構造専用の CUR (Compute Unit Replication) を容易に利用できることが含まれます。

アルゴリズム 1 は、V0 タイプのカーネル (参考資料 9 に基づく) です。主な操作は、最初の行と最初の列の既知の値を使用して行列を生成することです。不明な各エントリーは、左、上、左上の値を基に計算されます。すべての行列エントリーを順番に反復するループを使用します。**`max`** 関数は "**`if-else`**" 文を使用して実装されています。**アルゴリズム 1** では、**`SIZE`** は処理される行列ブロックの次元を表します。

アルゴリズム 1. Needleman-Wunsch カーネル - V0

```
int tx = get_local_id(0)
_local int* temp
_local int* ref
Initialize temp from global memory
barrier(CLK_LOCAL_MEM_FENCE);
Initialize ref from global memory
barrier(CLK_LOCAL_MEM_FENCE);
for i = 1 : SIZE do
   if tx≤i then
    compute t_idx_x and t_idx_y based on tx and i
    temp[t_idx_y][t_idx_x] =
       max (temp[t_idx_y-1][t_idx_x-1] +
       ref[t_idx_y-1][t_idx_x-1],
       temp[t_idx_y][t_idx_x-1] - penalty,
       temp[t_idx_y-1][t_idx_x] - penalty);
    barrier(CLK_LOCAL_MEM_FENCE);
barrier(CLK_LOCAL_MEM_FENCE);
for i = SIZE - 2 : 0 do
    Perform computations similar to above
barrier(CLK_LOCAL_MEM_FENCE);
Store temp to global memory
```

バージョン 1: 推奨ベースライン・コード (参照用)

直感的ではありませんが、ベースラインとしてシングルスレッド の CPU コードを使用することを推奨します。特に初期設計は、IBP が推奨するように SWI (Single Work Item) カーネルとして実装すべきです。SWI カーネルは、MWI カーネルよりも効率的な方法であらゆる形式の並列処理を効果的に推論して利用できます。CPU のようなベースライン・コードもキャッシュ・パフォーマンスを最適化すべきです。これにより、次の利点が得られます。

- コンパイラーが並列パイプライン間の接続性を推論するのに**役立ちます** (つまり、データがメモリーに格納される代わりに、パイプライン間で直接転送される可能性があるかどうか)。
- オンチップ・データ・アクセスの帯域幅が**向上します**。
- オフチップ・メモリー・トランザクションを担当するロード・ストア・ユニットの内部キャッシュを**効率良く使用します**。

アルゴリズム 2 は、推奨ベースライン・カーネルです。行列の最初の行と列は、それぞれ **`Vector A`** と **`Vector B`** です。

アルゴリズム 2. Needleman-Wunsch カーネル - V1

```
for i = 1 : Vector_B_Size do
   for j = 1 : Vector_A_Size do
      Out[i,j] = max( Out[i-1,j ] - penalty,
         Out[i-1,j -1] + ref[i,j ] , Out[i,j -1] - penalty)
```

バージョン 2: IBP

推奨ベースライン・コードに、次の IBP を適用します。

- マルチタスク並列カーネル
- すべてのループを完全にアンロール
- 状態レジスターの使用を最小化
- 定数配列
- 融合

アルゴリズム 3 は、IBP 適用後の Needleman-Wunsch カーネルの構造です。並列処理はシストリック・アレイを使用して行われ、各プロセシング要素 (PE) は別々のカーネルに実装されます。チャネルは、指定された順序で PE を接続するために使用されます。内部ループの反復ごとに、PE は同じ行内の連続する列を計算します。これにより、メモリー・トランザクションの空間的局所性が保証されます。欠点は、カーネル間のデータ依存関係です。コンパイラーは各カーネルを個別のエンティティーとして最適化するため、データ依存関係を確実に解消できません。そのため、データパスの同期に伴うオーバーヘッドによりパフォーマンスが低下します。

アルゴリズム 3. Needleman-Wunsch カーネル - V2

```
N ← Size of systolic array
P E_k → Kernel Begin
int up, left, up_left, cached_up, cached_up_left
   Initialize cached_up & cached_up_left
   for j = 1 : 1 : Vector_B_Size do
      left ← read_channel (PE_{k-1})
      up = cached_up
      up left = cached_up_left
      cached_up = max(up - penalty,
         left - penalty , up_left + ref [j,i])
      cached_up_left = left
      Out[j,i] = cached up
      write_channel (PE_{k-1}) ← cached_up
```

バージョン 3: シングルカーネル設計

バージョン 3 では、IBP で最適化されたタスク並列カーネルをマージして、同じカーネル内ですべての計算ループを定義します。これは、コンパイラーが依然として自動的にタスク並列パイプラインを推論できるためです。シングルカーネルには、マルチカーネル・アプローチと比較して、次のような利点があります。

- **内在的な**グローバル同期
- パイプラインのマージ / 並べ替えによるリソース使用量と遅延の**軽減**
- **簡素化された**制御ロジック

アルゴリズム 4 は、シストリック・アレイを単一のカーネルとして実装するカーネル構造を示します。これで、コンパイラーは個々の PE ではなく、計算全体を最適化できます。ほぼすべての計算が 1 つのループ変数 (j) に関連付けられているため、同期オーバーヘッドも軽減されます。このケースでは、リソース使用の軽減量が開始間隔のコストを上回るため、入れ子のループが使用されます。これは、ループ融合時にコンパイラーがデータ・アクセス・パターンを推論できなかったためです。

アルゴリズム 4. Needleman-Wunsch カーネル - V3

```
N ← Size of systolic array
int value[N+1], left[Vector_B_Size]
left ← Vector_B
for i = 1 : 1 : Vector_A_Size=N do
   base = f(i)
   value ← Vector_A[base:base+N+1]
   for j = 1 : 1 : Vector_B_Size do
      int up_left[N+1]
      for k = 2 : 1 : N + 1 do
         up_left[k] = value[k-1]
      value[1] = left[j]
      #pragma unroll
      for k = 2 : 1 : N + 1 do
         value[k] = max(value[k-1] - penalty,
           up_left[k] + ref[j, base+k] , value[k] - penalty)
      left[j] = value[N+1]
      Out value[2:N+1]
```

バージョン 4: 配列サイズの縮小

大きな変数配列を使用すると、パイプライン・レジスターがレジスターではなく BRAM として推論され、設計上の問題になります。BRAM はレジスターと同じスループットでデータをソースおよびシンクできないため、バレルシフターとメモリーの複製処理が必要になります。これにより、リソースの使用量が大幅に増加します。さらに、コンパイラーはメモリー依存関係のため、ストールフリーの計算ループの反復を生成できません。この問題を解決するには、中間変数に対応する大きな配列をより小さな配列に分割します。

アルゴリズム 5 は、パイプライン・レジスターをレジスターとして推論するためのカーネル構造です。スループット要件が低い "**left**" の **Vector B** のローカルストレージを除いて、すべての配列は別々の変数として表現され、スクリプトで生成されます。

アルゴリズム 5. Needleman-Wunsch カーネル - V4

```
N ← Size of systolic array
int value_1, value_2 ... Value_N_plus_1
int left [Vector_B_Size]
left Vector_B
for i = 1 : 1 : Vector_A_Size=N do
    base = f(i)
    value_1 Vector_A[base]
            ↓
    value_N plus 1 Vector_A[base+N+1]
    for j = 1 : 1 : Vector_B_Size do
        int up_left_2 ... Up_left_N_plus_1
        up_left_2 = value_1
            ↓
    up_left_N_plus_1 = value_N
    value_1 = left [j]
    value_2 = max(value_1 - penalty,
        up_left_2 + ref[j,base+2], value_2 - penalty)
            ↓
    value_N_plus_1 = max(value_N - penalty,
        up_left_N_plus_1 + ref[j,base+N+1],
        value_N_plus_1 - penalty)
    left[j] = value_N_plus_1
    Out ← value 2 ... value N plus 1
```

インテル® MKL
インテル® アーキテクチャー・ベースのシステムで演算処理を高速化
無料
ダウンロード

バージョン 5: 詳細な計算

OpenCL* コンパイラーは、1 つの変数に割り当てられている大規模な計算を中間ステージに確実に分割できません。そのため、実行可能なパイプライン・ステージの数が減り、クリティカル・パスが大きくなり、データ依存ストールが発生する可能性があります。この問題を回避するため、コンパイラーがパイプラインを推論しやすいように、中間変数を使用して可能な限り詳細な計算を行います。ロジックがすでに最適化されている場合、これらの変数は合成されリソースが有効活用されます。

アルゴリズム 6 は、いくつかの中間変数を追加して詳細な計算を実行した後のカーネル構造です。"**`max`**" 関数も明示的に実装しています。

アルゴリズム 6. Needleman-Wunsch カーネル - V5

```
1: N ← Size of systolic array
2: int value_1, value_2 ... Value_N_plus_1
3: int left [Vector B Size]
4: left ← Vector_B
5: for i = 1 : 1 : Vector_A_Size=N do
6:    base = f(i)
7:    value_1 Vector_A[base]
8:       ↓
9:    value_N_plus_1 Vector_A[base+N+1]
10:   for j = 1 : 1 : Vector_B_Size do
11:     int a_2 = value_1 + ref[j,base+2];
12:     value_1 = left[j]
13:
14:   int b_2 = value_1 - penalty
15:   int a_3 = value_2 + ref[j,base+3];
16:   int c_2 = value_2 - penalty
17:
18:   if ((a_2 ≥ b_2) && (a_2 ≥ c_2))
19:     value_2 = a_2
20:   else if ((b_2 > a_2) && (b_2 ≥ c_2))
21:     value_2 = b_2
22:   else
23:     value_2 = c_2
24:
25:   int b_3 = value 2 - penalty
26:   int a_4 = value 3 + ref[j,base+4];
27:   int c_3 = value 3 - penalty
28:     .
        .
        .
29: left[j] = value_N_plus_1
30: Out ← value 2 ... Value_N_plus_1
```

バージョン 6: プレディケーション

アーキテクチャーの状態を明示的に指定して、条件付き操作を最適化し、計算の有効性を保証します。ハードウェアは永続的であり、いったん合成されると常に存在するため、条件付き分岐文を使用しないようにします。代わりに、無効な操作の出力がコミットされて結果全体に影響しないように、変数値を条件付きで代入します。**アルゴリズム 7** は、"**`if-else`**" 操作を条件付き代入に置換します。

アルゴリズム 7. Needleman-Wunsch カーネル - V6

```
1: .
   .
   .
2: int a_2 = value_1 + ref[j,base+2];
3: value_1 = left[j]
4:
5: int b_2 = value_1 - penalty
6: int a_3 = value_2 + ref[j,base+3];
7: int c_2 = value_2 - penalty
8:
9: int d_2 = (a_2 > b_2) ? a_2 : b_2
10: value_2 = (c_2 > d_2) ? C_2 : d_2
11: .
    .
    .
```

ハードウェア構成

インテル® Arria® 10AX115H3F34I2SG FPGA と**インテル® FPGA SDK for OpenCL* 16.0** を使用して設計を実装します。この FPGA は 427,200 ALM、1,506K ロジックエレメント、1,518 DSP ブロック、53MB オンチップストレージで構成されます。GPU 実装には、NVIDIA* Tesla* P100 PCIe* 12GB GPU と CUDA* 8.0 を使用します。この GPU には 3,584 の CUDA* コアがあり、ピーク帯域幅は 549GB/秒です。CPU コードは 14 コア、2.40GHz の**インテル® Xeon® プロセッサー E5-2680 v4** と**インテル® C++ コンパイラー 16.0.1** を使用して実装しました。

最適化の特徴付け

以下のベンチマークを使用して、完全な OpenCL* コンパイルフローで最適化をテストします。

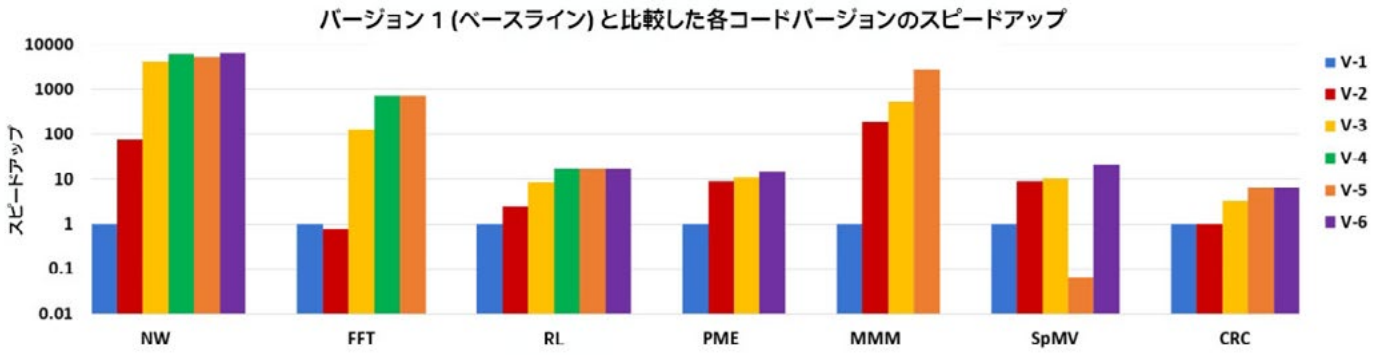
- Needleman-Wunsch (NW)
- 高速フーリエ変換 (FFT)
- 範囲限定分子動力学 (RL)
- Particle Mesh Ewald (PME)
- 密行列 - 行列乗算 (MMM)
- 疎行列 - 密ベクトル積 (SpMV) と巡回冗長検査 (CRC)

表 2 は、これらのベンチマーク、関連する DWARF[8]、テストした問題サイズ、コードバージョンの要約です。

表 2. ベンチマークの要約

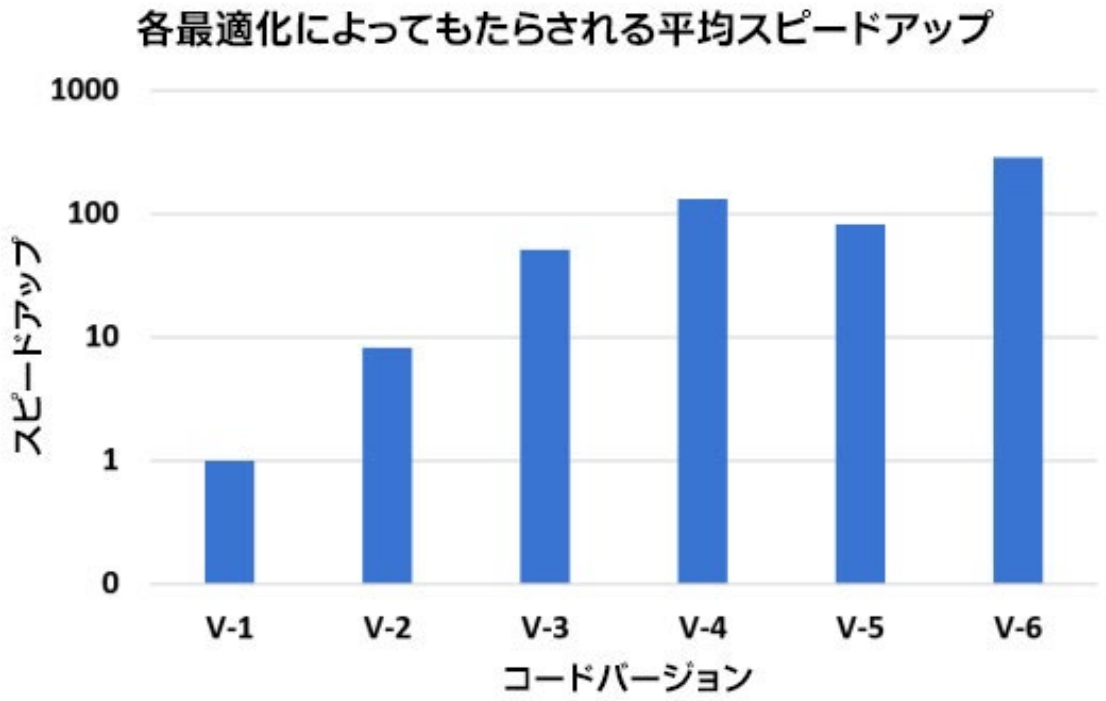
ベンチマーク	DWARF	問題サイズ	V1	V2	V3	V4	V5	V6
NW	動的プログラミング	16K x 16K 整数テーブル	●	●	●	●	●	●
FFT	スペクトル法	64 ポイントの基数 2 の 1D FFT、8,192 ベクトル	●	●	●	●	●	●
RL	N 体	セルあたり 180 粒子、パスレート 15%	●	●	●	●	●	●
PME	構造格子	1,000,000 粒子、323 格子、3D 三次補間	●	●	●			●
MMM	密線形代数	1K x 1K 行列、単精度	●	●			●	●
SpMV	スパース線形代数	1K x 1K 行列、単精度、疎性 5%、NZ=51,122	●	●	●		●	●
CRC	組み合わせ論理	100MB CRC32	●	●	●		●	●

図 2 は、個々の最適化の結果です。ほぼすべてのケースで同じ傾向が見られます。従来の最適化 (V2) では、わずかなスピードアップしか得られません。V2 に追加の最適化を適用することで、パフォーマンスが大幅に向上します。



2 **キャッシュが最適化された CPU ベースライン・コードに提案した最適化を体系的に適用した結果。ほぼすべてのケースで後続のコードバージョンのパフォーマンスが向上しており、完全に最適化されたカーネルは IBP (V2) のみを適用した場合と比較してパフォーマンスが大幅に向上しています。**

個々の最適化の平均スピードアップを**図 3** に示します。一般に、追加の最適化を適用するにつれてパフォーマンスが向上しています。例外は V5 です。V5 は NW と SpMV で実行時間が長くなるためです。どちらのケースでも、計算を可能な限り詳細に実行しようとすると条件文を使用しなければならず、最適化の利点が得られません。V6 でこれらの条件文が排除されるとスピードアップが得られます。



3 **実行したすべてのベンチマークのスピードアップを平均したコードバージョンごとのパフォーマンス**

このアプローチの全体的な有効性を示すため、最適化したカーネルのパフォーマンスと既存の CPU、GPU、Verilog*、FPGA-OpenCL* 実装を比較します。**表 3** は、既存の実装の参照元です。文献から取得されたか、利用可能なソースコード / ライブラリーを使用して実装 (* 付きのもの) されました。参考資料 3 から取得された Verilog* FFT の測定結果は、オフチップアクセスのオーバーヘッドを含めるように拡張されています。

表 3. 既存の実装の参照元

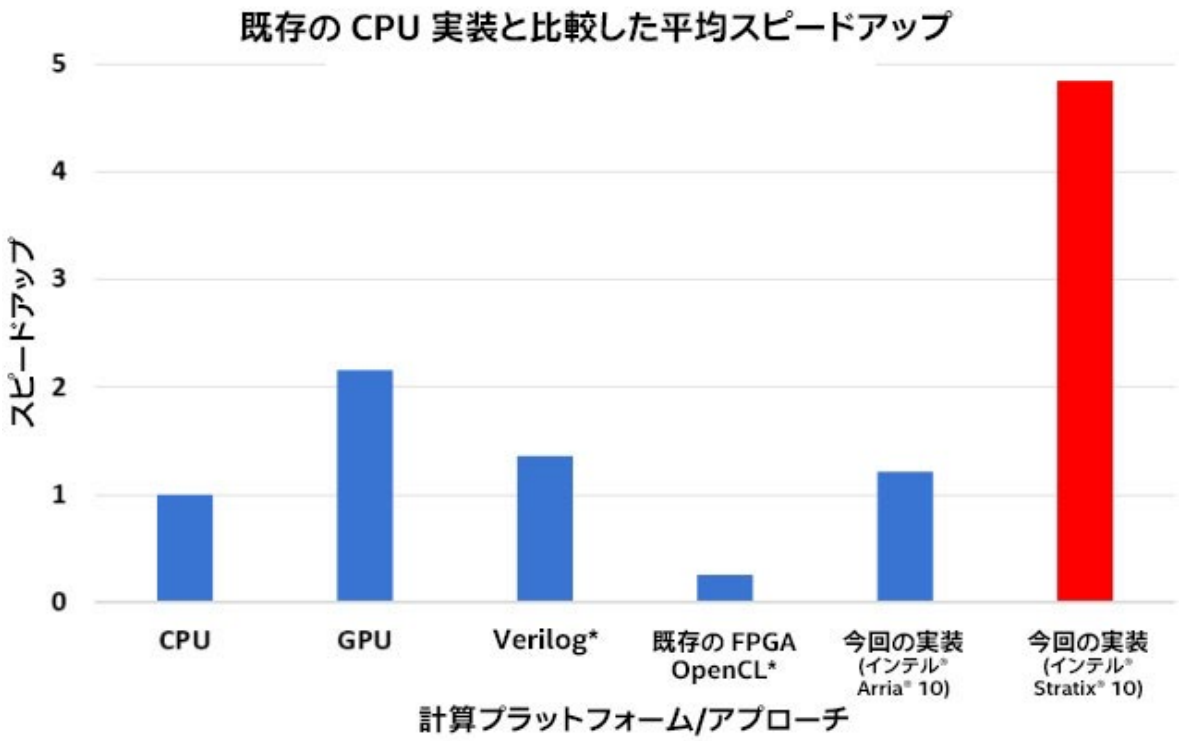
ベンチマーク	CPU	GPU	Verilog*	OpenCL*
NW	Rodinia*[9]	Rodinia*[9]	Benkrid*[10]	Zohouri*[11]
FFT	インテル® MKL*[12]	cuFFT**[13]	Sanaullah*[3]	インテル[14]
RL	—	—	Yang*[15]	Yang[15]
PME	Ferit[16]	Ferit*[16]	Sanaullah[17]	—
MMM	インテル® MKL*[12]	cuBLAS*[18]	Shen*[19]	Spector*[20]
SpMV	インテル® MKL*[12]	cuSPARSE*[18]	Zhou*[22]	OpenDwarfs*[8]
CRC	Brumme*[23]	—	Anand*[24]	OpenDwarfs[8]

図 4 は CPU コードと比較した平均スピードアップで、**図 5** はすべての実行の正規化された実行時間です。**インテル® マス・カーネル・ライブラリー (インテル® MKL)** を使用して記述されたコードのパフォーマンスにより、我々の実装はマルチコア CPU 実装の約 1.2 倍のスピードアップを達成しています。また、平均実行時間は既存の FPGA OpenCL* 実装の約 1/5 であることが分かります。GPU は我々の実装と比較して 2.4 倍のスピードアップを達成しています。これは、我々がミッドレンジの FPGA (**インテル® Arria® 10 FPGA**) を使用しているのに対して、ハイエンドの GPU (NVIDIA* Tesla* P100) が使用されているためです。参考までに、リソースの増加のみを考慮した控えめな 4 倍の係数を使用して、ハイエンドの FPGA (インテル® Stratix® 10) の予測パフォーマンスを示しています。最適化されたカーネルは、インテル® Stratix® 10 上で GPU 実装よりも平均 65% 高速になることが予想されます。

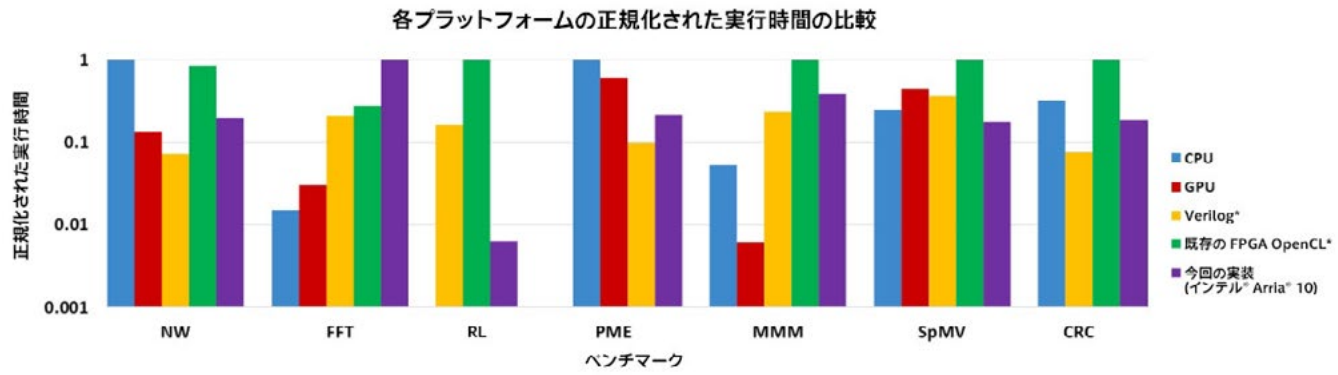
まとめ

既存の Verilog* 実装との比較では、OpenCL* カーネルは平均して手動でチューニングした HDL の 12 % 以内を達成しています。これは、最適化が FPGA のパフォーマンスとプログラマビリティーのギャップを埋め、OpenCL* で HDL のようなパフォーマンスを実現できることを示しています。

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



4 CPU 実装と比較したすべてのベンチマークの平均スピードアップ



5 既存の CPU、GPU、Verilog*、および FPGA OpenCL* 実装と我々の実装のパフォーマンスの比較。我々の実装は、ほとんどのベンチマークで CPU と OpenCL* 実装を上回っています。さらに、ベンチマークによっては、GPU (SpMV、PME) と Verilog* (SpMV、RL) よりも高速です。

参考資料

[1] **インテル® FPGA SDK for OpenCL***

[2]S. Chellappa, F. Franchetti, and M. Pueschel, "How To Write Fast Numerical Code: A Small Introduction," in Generative and Transformational Techniques in Software Engineering II, Lecture Notes in Computer Science v5235, 2008, pp. 196 - 259.

[3]A. Sanaullah and M. Herbordt, "FPGA HPC using OpenCL: Case Study in 3D FFT," in Proceedings of the International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, 2018.

[4]A. Sanaullah, R. Patel, and M. Herbordt, "An Empirically Guided Optimization Framework for FPGA OpenCL," in Proceedings of the IEEE Conference on Field Programmable Technology, 2018.

[5]A. Sanaullah, C. Yang, D. Crawley, and M. Herbordt, "SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels," in Proceedings on Heterogeneous High-Performance Reconfigurable Computing, 2018.

[6]A. Sanaullah and M. Herbordt, "Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolow," in IEEE High Performance Extreme Computing Conference, 2018.

[7]"**Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide** (英語)"

[8]K. Krommydas, A. E. Helal, A. Verma, and W.C. Feng, "Bridging the Performance Programmability Gap for FPGAs via OpenCL: A Case Study with Opendwarfs," Department of Computer Science, Virginia Polytechnic Institute and State University, Tech. Rep., 2016.

[9]S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in IEEE International Symposium on Workload Characterization, 2009, pp. 44 - 54.

[10]K. Benkrid, Y. Liu, and A. Benkrid, "A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 17, No. 4, pp. 561 - 570, 2009.

[11]H. R. Zohouri, N. Maruyamay, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs" in International Conference for High Performance Computing, Networking, Storage and Analysis, SC16, 2016, pp. 409 - 420.

[12] **インテル® マス・カーネル・ライブラリー**

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

参考資料 (続き)

[13]C. Nvidia, "CuFFT Library," 2010.

[14]**FFT (1D) 設計例** (英語)

[15]C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. Herbordt, "OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics," in IEEE High Performance Extreme Computing Conference, 2017.

[16]F. Buyukkececi, O. Awile, and I. F. Sbalzarini, "A Portable OpenCL Implementation of Generic Particle—Mesh and Mesh—Particle Interpolation in 2D and 3D," Parallel Computing, Vol. 39, No. 2, pp. 94 - 111, 2013V

[17]A. Sanaullah, A. Khoshparvar, and M. C. Herbordt, "FPGA-Accelerated Particle-Grid Mapping," in Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on. IEEE, 2016, pp. 192 - 195.

[18]Nvidia, "CUBLAS Library," NVIDIA Corporation, Santa Clara, CA, 2008.

[19]J. Shen, Y. Qiao, Y. Huang, M. Wen, and C. Zhang, "Towards a Multi-Array Architecture for Accelerating Large-Scale Matrix Multiplication on FPGAs," in Circuits and Systems (ISCAS), 2018 IEEE International Symposium on. IEEE, 2018, pp. 1 - 5.

[20]Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An OpenCL FPGA Benchmark Suite," in International Conference on Field-Programmable Technology, 2016.

[21]Nvidia, "CuSparse Library," NVIDIA Corporation, Santa Clara, CA, 2014.

[22]L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," in Proceedings of the International Symposium on Field-Programmable Gate Arrays, 2005.

[23]S. Brumme, "Fast CRC32," http://create.stephan-brumme.com/crc32/, 2018.

[24]P. A. Anand et al., "Design of High Speed CRC Algorithm for Ethernet on FPGA using Reduced Lookup Table Algorithm," in India Conference (INDICON), 2016 IEEE Annual. IEEE, 2016, pp. 1- 6.

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

Python* での並列処理

ツールにより通念を一掃して並列処理を実現

David Liu インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア
Anton Malakhov インテル コーポレーション ソフトウェア開発エンジニア

プログラミング言語としての Python* は、業界および学術分野でほぼ 10 年にわたり使用されてきました。この高生産性言語は科学計算およびマシンラーニングで最も人気のある言語の 1 つですが、ベースの Python* 言語はまだシングルスレッドのままです。これらの分野の生産性は、シングルスレッドの言語でどのように維持されているのでしょうか。

グイド・ヴァンロッサム氏による Python* 言語の設計では、型の柔軟性や予測可能なスレッドセーフな動作と、静的な型やスレッド・プリミティブを管理することのトレードオフが行われました。次に、この設計思想を守るため、実行を一度に 1 つのスレッドに制限するグローバル・インタープリター・ロック (GIL) が実装されました。この 10 年間で、多くの並行処理が Python* に実装されましたが、並列処理はまだわずかです。言語のパフォーマンスはこれ以上改善できないのでしょうか？もう少し詳しく調べてみましょう。

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

ベース言語は for ループや非同期 / 同時呼び出しをシングルスレッド GIL に従って構築するため、**`[x*x for x in range(0,10)]`** などのリスト内包表記も常にシングルスレッドになります。ベース言語のスレッド・ライブラリーは、スレッドを実装してはいますが GIL の下で動作するため、誤解を招くことがあります。Python* の並行処理の機能の多くも、GIL の下で動作します。なぜこの高生産性言語では、これらの規則で言語を制限しているのでしょうか。

その理由は、言語設計で採用された抽象化のレベルです。この言語は、ctypes から cffi まで、C コードをラップする多くのツールとともに提供されています。ネイティブ Python* ライブラリーのマルチプロセス・ライブラリーから明らかなように、ベース言語ではマルチスレッドよりもマルチプロセスが好まれています。これらの 2 つの設計思想は、Python* API で C コードを**インテル® マス・カーネル・ライブラリー (インテル® MKL)** や OpenBLAS などの数学ランタイム・ライブラリーにディスパッチする、NumPy* や SciPy* などのパッケージにも明確に現れています。コミュニティーは、高速な C ベースのライブラリーにディスパッチする方法を採用し、これが Python* の並列処理の実装に推奨される方法となりました。

これらの方法と言語制限の組み合わせでは、独自の並列処理フレームワークを使用して Python* に並列処理を適用するオプションがあります。

- **Numba** は、Python* コードの JIT ベースのコンパイルが可能で、LLVM ベースの Python* 互換コードを実行できます。
- **Cython** は、C モジュールをコンパイルしてハードウェア・ベクトル化をターゲットとすることができる Python* 形式の構文とコンパイルされたモジュールを提供します。
- **numexpr** は、記号評価によりコンパイラーおよび高度なベクトル化を利用できます。

これらの 3 つの手法は、言語の当初の意図を損なうことなく、異なる方法で Python* の GIL を回避しており、すべて異なる並列処理モデルを実装しています。

最も一般的な言語構造である for ループに並列処理を適用する例を考えてみましょう。次のループは、リストの 50 未満のすべての数を返します。

```
def test_func(list_of_items):
        final_list = []
        for item in list_of_items:
                if item < 50:
                final_list.append(item)
        return final_list
```

このコードを実行すると、結果は次のようになります。

```
import random
random_list = [random.randint(0,1000000) for x in range(0,1000000)]
%timeit test_func(random_list)
27.4 ms ± 331 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

このコードはピュアな Python* で記述されているため、Python* は GIL の下、シングルスレッドで項目のリストを制御します。そのため、処理はすべてシーケンシャルに行われ、コードに並列処理は適用されません。その記述方法から、このコードは Numba フレームワークの適切な候補となります。Numba は、デコレーター (@ 記号) を使用して JIT (ジャストインタイム) コンパイルを行う関数にフラグを付けます。

```
@jit(nopython=True)
def test_func(list_of_items):
    final_list = []
    for item in list_of_items:
        if item < 50:
            final_list.append(item)
    return final_list
```

このコードを実行すると、結果は次のようになります。

```
%timeit test_func(random_list)
15.7 ms ± 173 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

この単純なデコレーターを追加することで、パフォーマンスはほぼ倍になりました。これは、オリジナルの Python* コードが、簡単にコンパイルして CPU でベクトル化できるプリミティブおよびデータ型で記述されているためです。Python* のリストは最初に注目すべき場所です。通常、このデータ構造は、loose 型およびビルトイン・アロケーターで占められます。しかし、**`random_list`** を含むデータ型はすべて整数です。これにより、Numba の JIT コンパイラーはループをベクトル化することができます。

リストが混在する項目を含む場合 (例えば、**char** と **int** のリスト)、コンパイルされたコードはヘテロジニアス・リストを処理できないため、TypeError をスローします。また、関数が混在するデータ型操作を含む場合、Numba はハイパフォーマンス JIT コンパイルコードを生成しないで、Python* オブジェクト・コードに戻します。

ここでの教訓は、Python* で並列処理を行えるかどうかはオリジナルのコードがどのように記述されているかに依存するということです。同一のデータ型およびベクトル化可能なデータ構造を利用することにより、Numba は単純なデコレーターを挿入してコードを並列化することができます。歴史的に見て、Python* 辞書は適切にベクトル化されないため、その使用には注意を払う必要があります。ジェネレーターと内包表記には同じ問題があります。それらのコードをリスト、セット、配列にリファクタリングすると、ベクトル化を容易にできます。

並列処理は、数値および記号数学では容易に実装できます。NumPy* および SciPy* は、Python* の GIL の外部の計算を低レベルの C コードとインテル® MKL ランタイムにディスパッチします。例えば、単純な NumPy* シンボリック式 **((2*a + 3*b)/b)** は次のように表現されます。

```
import numpy as np
a = np.random.rand(int(1e6))
b = np.random.rand(int(1e6))

%timeit (2*a + 3*b)/b
8.61 ms ± 108 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

NumPy* の構造および設計により、この式は、シングルスレッド Python* インタープリターにより複数回実行されます。NumPy* のリターンはそれぞれ C にディスパッチされ、Python* レベルに戻されます。次に、Python* オブジェクトはそれぞれ後続の呼び出しに送られ、再度 C にディスパッチされます。この繰り返しジャンプは計算のボトルネックになるため、NumPy* または SciPy* で記述できないカスタムカーネルを計算する必要がある場合は、numexpr を使用します。

```
import numexpr as ne
%timeit ne.evaluate('(2*a + 3*b)/b')
2.22 ms ± 52.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

numexpr は約 4 倍のスピードアップをどのように達成しているのでしょうか。前のコードは、計算のシンボリック表現を numexpr のエンジンに渡してインテル® MKL のベクトル・マス・ライブラリーのベクトル化コマンドで動作するコードを生成します。つまり、計算が完了して Python* に結果を戻すまで、すべての計算は低レベルコードで行われます。この方法は、Python* インタープリターによる複数回の実行を回避し、簡潔な構文を提供する一方で、シングルスレッドのセクションを減らします。

Python* のエコシステムを見て異なる並列処理フレームワークを評価すると、優れたオプションが存在することは明らかです。Python* の並列処理を習得するには、ツールと制限を理解することが重要です。Python* は、フレームワーク開発を単純化して予測可能な言語動作を提供するための設計思想として GIL を採用しています。しかし、並列処理を行うには、結局のところ、別のツールを使用して GIL とそのシングルスレッドの制限を回避することが望ましいのです。

関連資料

- **インテル® Distribution for Python***
- **インテル® MKL**

BLOG HIGHLIGHTS

エッジ AI の大きな恩恵をもたらすソフトウェア・ツール

CHARLOTTE DRYDEN, INTEL CORPORATION

人工知能 (AI) は日常生活でも見かける機会が増えています。エッジ AI の進歩により、その存在は歓迎されるとともに、広く受け入れられるようになりました。多くの AI のユースケースは、データソースまたはその近くで処理が行われるエッジに最も適しており、コストを抑え、アプリケーションやサービスのレイテンシーを軽減し、信頼性やデータのプライバシーを向上できます。

我々が認識しているかどうかに関係なく、エッジ AI テクノロジーは、デジタルで接続された世界に莫大な利益を提供しています。このエッジ AI の急速な進歩は、わずかな「キラーアプリ」によるものではなく、さまざまな新しいソリューションや使用法が絶えず出現しているためです。

この記事の続きはこちら (英語) でご覧になれます。>

テクノロジーの未来を紐解く

intel Software

開発者向けのナレッジハブ、Tech.Decoded へようこそ

現代のハードウェアを最大限に活用できるように、
さまざまな情報ライブラリーを公開しています。
競争力を高め、市場投入期間を短縮しましょう。

エキスパートの考察を得る

テクノロジーの将来を予測する人々による、
今日の技術展望 (コードの現代化、システムと IoT、
データサイエンスなど) について視聴できます。

さらに深く掘り下げる

今日の最も重要な戦略、プラクティス、ツールに関する
オンデマンドのセミナーからコードのパフォーマンスを
最大限に引き出す方法が分かります。

概念のコード化

特定の開発ツールで主なプログラミング・タスクを
実行する方法を説明した短いビデオや記事を
利用できます。

Tech.Decoded を見る (英語) >

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation またはその子会社の商標です。
© Intel Corporation

インテル® Advisor を使用してメモリー・ボトルネックを排除

プログラムのメモリーアクセス方法を理解してハードウェアを最大限に活用

Kevin O' Leary インテル コーポレーション テクニカル・コンサルティング・エンジニア
Alex Shinsel インテル コーポレーション テクニカル・コンサルティング・エンジニア

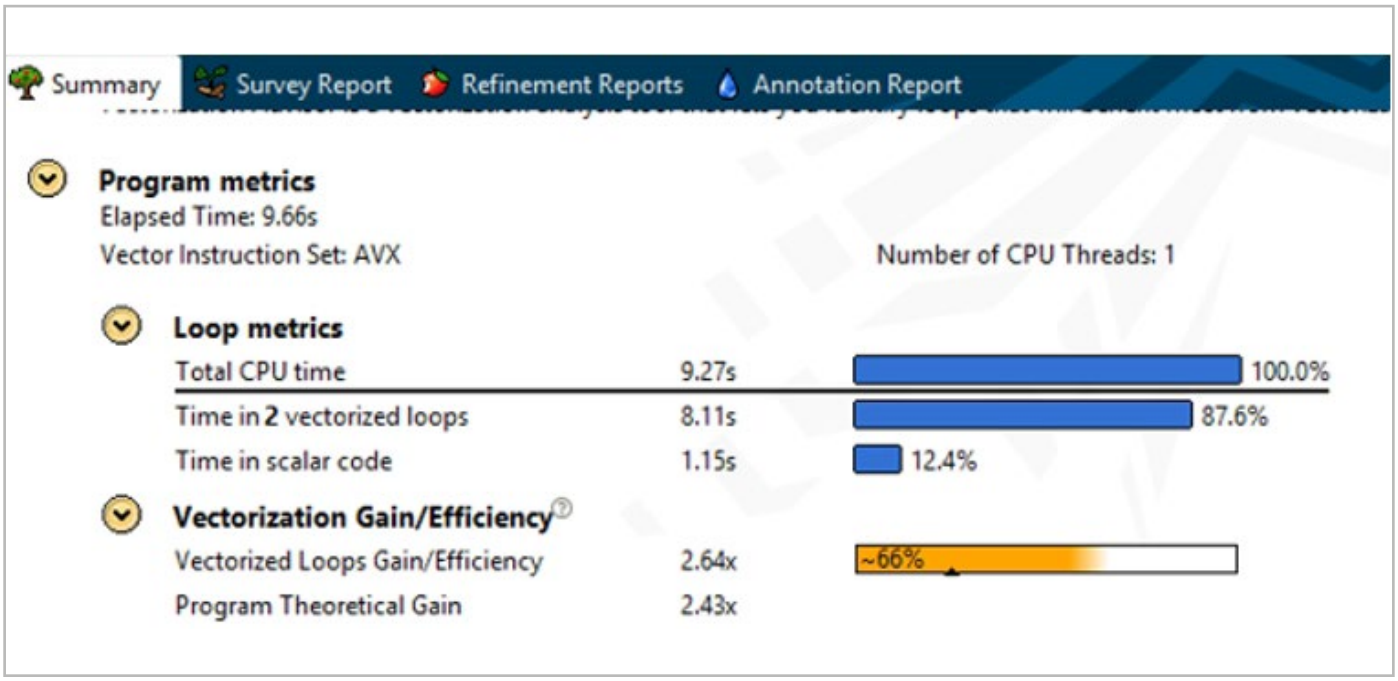
アプリケーションのメモリーアクセスは、パフォーマンスに大きく影響します。スレッド化とベクトル化によりアプリケーションを並列化するだけでは十分ではありません。メモリー帯域幅の効率的な利用も重要です。しかし、多くの場合、ソフトウェア開発者はそのことに気づいていません。メモリー・レイテンシーを最小限に抑え、帯域幅の向上を支援するツールを利用することで、パフォーマンス・ボトルネックをピンポイントで特定し、その原因を診断できます。**インテル® Advisor** はその代表的なツールの 1 つで、メモリーアクセスの最適化とメモリー・ボトルネックの排除を支援する機能を備えています。

- **ルーフライン解析** (新しい統合ルーフライン機能)
- **メモリー・アクセス・パターン (MAP) 解析**
- **メモリー・フットプリント解析**

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

優れたパフォーマンスを達成する

アプリケーションのパフォーマンスを最大限に引き出すには、すべてのシステムリソースを適切に使用しているか知る必要があります。インテル® Advisor の [Summary (サマリー)] ビュー (**図 1**) で、アプリケーションがどの程度適切にベクトル化されているかを示す、プログラム全体のメトリックを確認します。



1 **インテル® Advisor の [Summary (サマリー)] ビュー**

プログラム中の最も時間がかかっているループを系統的に調査する必要もあります。このための重要なメトリックはベクトル化効率 (**図 2**) です。この例で、インテル® Advisor はベクトル化による向上を 2.19 倍と示しています。しかし、ここには 55% のベクトル化効率スコアしか示されていません。45% の効率はどこに行ったのでしょうか。非効率なベクトル化を引き起こす要因はさまざまです。

2 **インテル® Advisor の [Vectorization Efficiency (ベクトル化効率)] ビュー**

パフォーマンス問題

非効率なアクセスパターン

間接メモリーアクセスはパフォーマンス低下の原因の 1 つです。次のコードは、最初に **B[i]** をデコードしないと **A** をデコードできません。

```
for (i = 0; i < N; i++)
      A[B[i]] = C[i] * D[i];
```

この結果、アクセスパターンは不規則になります。コンパイラーは、このコードをギャザー / スキャッターと呼ばれる手法を使用してベクトル化します。ループをベクトル化できることはメリットですが、これらのギャザー / スキャッター命令はシーケンシャル・アクセスほど速くありません。高速なコードを実現するには、データがユニットストライドでアクセスされるようなデータ構造を選択することが重要です。(後でインテル® Advisor がこの情報をどのように示すか説明します。)

メモリー・サブシステムのレイテンシー / スループット

システムのパフォーマンスを最大限に引き出すには、コードをさまざまなキャッシュに収めて、適切にデータを再利用することが非常に重要です。次の例では、インデックス **i** を使用して非常に大きなデータにアクセスしています。このデータは非常に大きくキャッシュに収まりません。**A** が多次元配列であることも問題です。

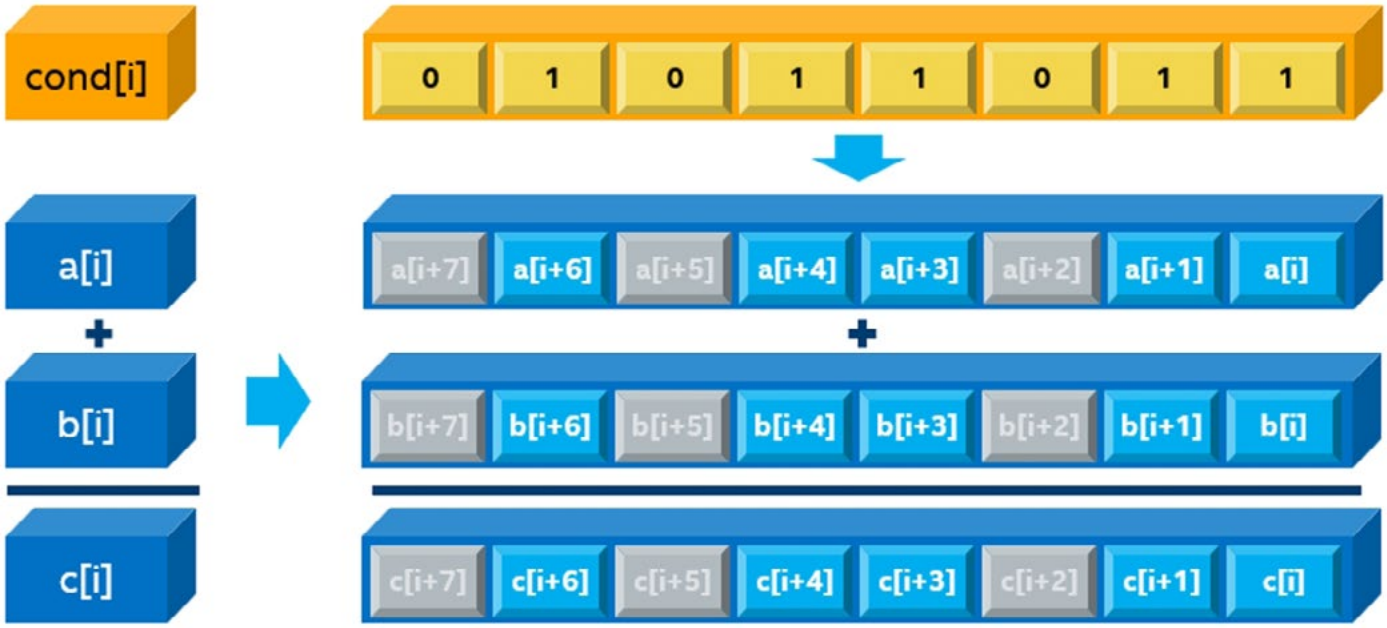
```
for (i = 0; i < VERY_BIG; i++)
      C[i] = z * A[i][j];
```

A[i][j] と **A[i+1][j]** はメモリーで隣り合っていないため、新しい配列要素を参照するたびに、キャッシュラインを入れ替える (潜在的にキャッシュラインを退避する) 必要があります。この「キャッシュ・スラッシング」は、パフォーマンスを低下させる原因となります。この種のアプリケーションの最適化には、キャッシュに収まる小さな範囲のインデックスを使用する内部ループを追加する、キャッシュ・ブロッキングなどの手法が役立ちます。

分岐の多いコード

分岐の多いアプリケーション (例えば、下記の **if(cond(i)** を含む **for** ループ) は、条件が真でない SIMD レーンをブロックするようにマスクレジスターを使用してベクトル化できます。これらの反復では、SIMD レーンは有益な処理を行っていません。インテル® Advisor のマスク使用率メトリック (**図 3**) を確認すると、3 つの要素が除外され、マスク使用率は 5/8 = 62.5% になっています。

```
for (i=0; i< MAX; i++)
     if (cond(i))
          C[i] = A[i] + B[i];
```



3 **マスク使用率メトリック**

潜在的にユニットストライド方式でデータにアクセスし、優れたベクトル効率を得ることができましたが、マスク使用率が低いため必要なパフォーマンスが得られていません (**図 4**)。**表 1** は、メモリーアクセスのタイプを示しています。

FLOPS			Vectorized Loops				Instruction Set Analysis
GFLOPS	AI	Mask Utilization	Vector ISA	Efficiency	Gain Esti...	VL	Traits
0,800	0,1000		AVX512	~56%	8.98x	16	FMA
1,684	0,0968	100%	AVX512	~41%	13.05x	16;	Blends; Divisions; Extracts;
4,951	0,0833	88%	AVX512	~79%	12.68x	16	Unpacks
1,251	0,0833	78%	AVX512	~41%	13.05x	16;	Unpacks
1,111	0,0833	78%	AVX512	~41%	13.05x	16;	Unpacks
10,001	0,0833	89%	AVX512	~37%	11.70x	16;	FMA

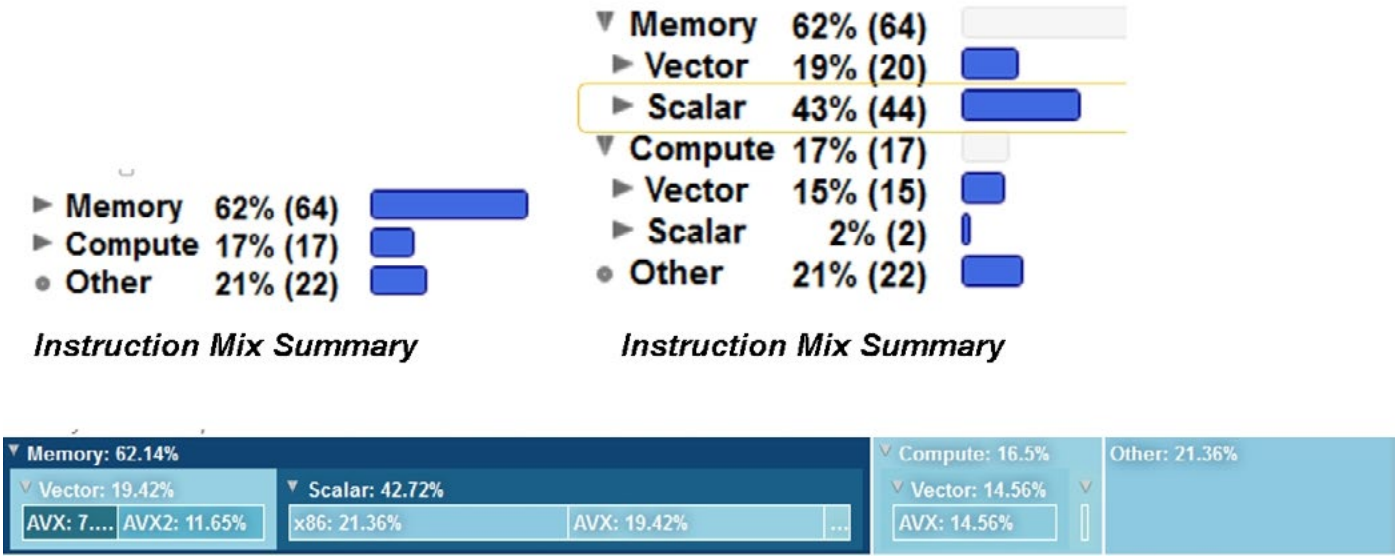
4 **マスク使用率と効率**

表 1. メモリーアクセスのタイプ

アクセスパターン	小さなメモリー・フットプリント	大きなメモリー・フットプリント
ユニットストライド	• **効率的な SIMD** • **レイテンシー / 帯域幅ボトルネックなし**	• **効率的な SIMD** • 帯域幅ボトルネック
定数ストライド	• 中程度の SIMD • レイテンシー / ボトルネックの可能性	• 中程度の SIMD • レイテンシー / ボトルネックの可能性
不規則アクセス、ギャザー / スキャッター	• 非効率な SIMD • レイテンシー / ボトルネックの可能性	• 非効率な SIMD • レイテンシー / ボトルネック

CPU/VPU 依存かメモリー依存か

アプリケーションがメモリー依存の場合、インテル® Advisor が最適化に役立ちます。しかし、まず最初に、メモリー依存なのか CPU/VPU 依存なのかを判断する必要があります。この判断を行うには、使用している命令を確認します。インテル® Advisor の [Code Analytics (コード解析)] ウィンドウ (**図 5**) で、コードが実行している命令を確認できます。



5 **[Code Analytics (コード解析)] ウィンドウ**

確かな経験則は、多くのメモリー命令を実行しているアプリケーションはメモリー依存になる傾向があり、多くの計算命令を実行しているアプリケーションは計算依存になる傾向があるというものです。**図 5** を詳しく見てみましょう。スカラー命令とベクトル命令の比率は特に重要です。できるだけベクトル命令の比率が高くなるようにします。

別の方法としては、インテル® Advisor の [Survey (サーベイ)] ビュー (**図 6**) の [Traits (特徴)] 列を使用することができます。

Function Call Sites and Loops	Self Time	Type	Instruction Set Analysis			
			Traits	Data Types	Vector Widths	Instruction Sets
[loop in Intel::CompilerDevSuit ...	5,370s	Scalar				
[loop in Intel::CompilerDevSu ...	**1,380s**	**Vectorized (Body)**	**Compress...**	**Float32; Int32; UIn...**	**512**	**AVX512F_512**

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

File: DataCompress.cpp:115 Intel::CompilerDevSuite::DataCompress::RunCCode

Line	Source	Total Time	%	Loop Time	%	Traits
114	`#pragma ivdep`					
115	`for (i=0; i<BUFF_SIZE; i++)`	0,130s		1,380s		
116	`{`					
117	`if ( source[i] > 0 )`	0,710s				Mask Manipulations
118	`{`					
119	`dest[j++] = source[i];`	0,550s				Compresses
120	`}`					
121	`}`					

6 **インテル® Advisor の [Survey (サーベイ)] ビューの [Traits (特徴)] 列**

特徴を、ループをベクトル化するためにコンパイラーが行う必要がある操作と考えます。**インテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512)** などの最新のベクトル命令セットでは、コードをベクトル化するためにコンパイラーが使用できる、多くの新しい命令および表現が用意されています。**図 6** に示されている、レジスターマスクおよび圧縮命令のような手法を利用すると、以前はできなかったベクトル化を行うことができますが、場合によってはコストがかかります。データ構造をベクトル化するためにコンパイラーが行う必要がある操作 (メモリー操作など) は、通常、[Traits (特徴)] 列に表示されます。これらの特徴は、メモリー・アクセス・パターン解析で調査できる問題を示すことがあります。

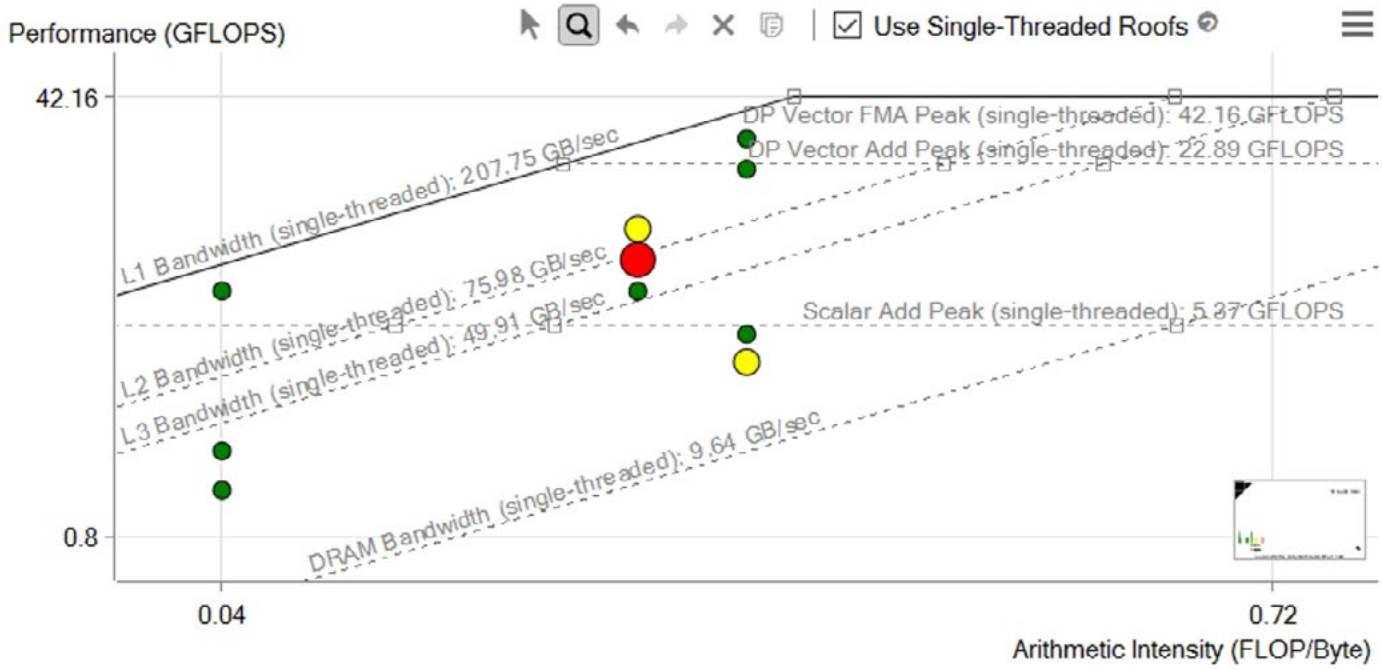
便利な最適化機能

ルーフライン解析

ルーフライン・グラフは、メモリー帯域幅や計算のピークを含む、ハードウェアの制限に関連するアプリケーション・パフォーマンスのビジュアルな表現です。このモデルは、2008 年のカリフォルニア大学バークレー校の研究者による論文「**Roofline: An Insightful Visual Performance Model for Multicore Architectures (ルーフライン : マルチコア・アーキテクチャー向けの洞察力に富んだビジュアルなパフォーマンス・モデル)**」(英語) で最初に提唱され、2014 年には、リスボン工科大学の研究者による論文「**Cache-Aware Roofline Model: Upgrading the Loft (キャッシュを考慮したルーフライン・モデル : ロフトの改良)**」(英語) で改良されました。従来は、ルーフライン・グラフは手動で計算および作成されていましたが、インテル® Advisor の最新バージョンではルーフライン・グラフを自動的に作成できるようになりました。

ルーフラインは、次のような項目に対する詳しい情報を提供します。

- パフォーマンス・ボトルネックの場所
- パフォーマンス向上の余地
- 対応可能なボトルネックと対応する価値のあるボトルネック
- ボトルネックが発生している理由
- 次のステップ

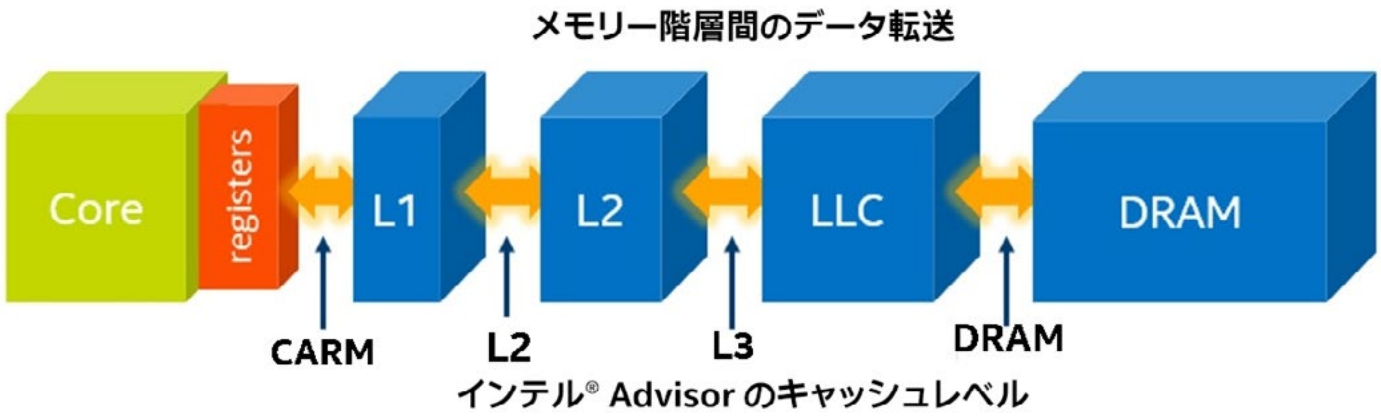


7 **ルーフライン解析**

図 7 の水平線は、ハードウェアが一定の時間内に実行できる、指定された型の浮動小数点演算または整数演算の数を表します。対角線は、メモリー・サブシステムが 1 秒ごとに処理できるデータのバイト数を表します。点はそれぞれ、プログラムのループまたは関数で、その位置はパフォーマンス (最適化により影響を受けます) および演算密度 (AI) を示します。

インテル® Advisor の統合ルーフライン

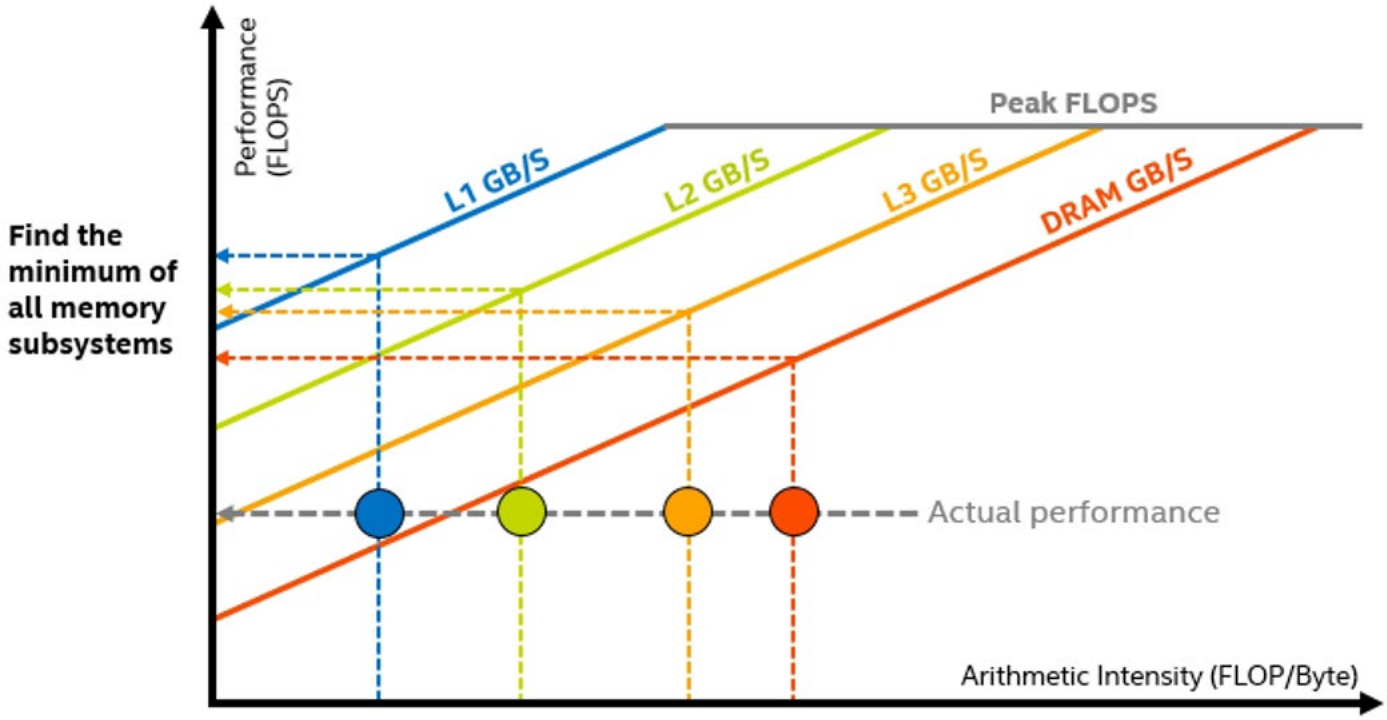
統合ルーフライン・モデルは、詳細な解析により、ボトルネックの情報を直接表示します。インテル® Advisor は、キャッシュ・シミュレーション (**図 8**) を使用して、すべてのメモリータイプのデータを収集します。



8 **インテル® Advisor のキャッシュ・シミュレーション**

収集したデータから、インテル® Advisor は指定されたキャッシュレベルのデータ転送の数をカウントし、各ループおよび各メモリーモデル固有の演算密度を計算します。あるレベルから別のレベルへのトラフィックの変化を観察し、これらのレベルの最大帯域幅を表す各ルーフと比較することにより、カーネルのメモリー階層ボトルネックをピンポイントで特定して、最適化ステップを判断することができます (**図 9**)。

インテル® Advisor
現代のハードウェア向けにコードを最適化
詳細

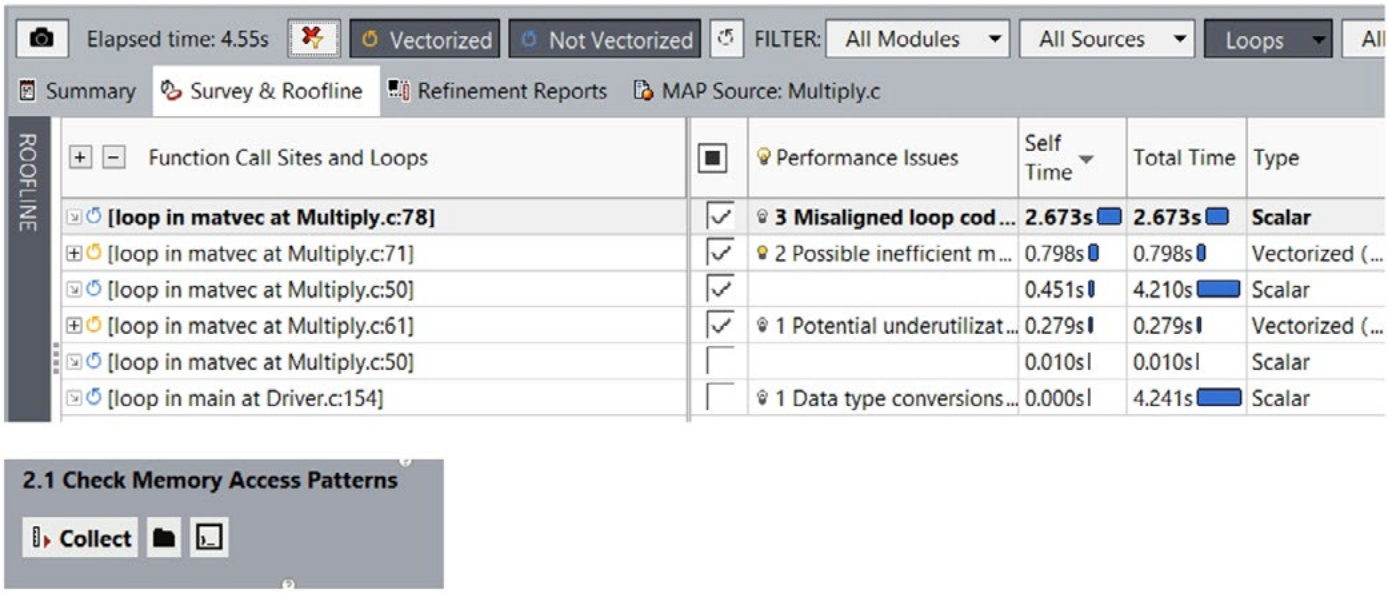


9 **メモリー階層ボトルネックをピンポイントで特定**

メモリー・アクセス・パターン (MAP) 解析

インテル® Advisor の MAP 解析は、メモリーのアクセス方法について詳細な情報を提供します。メモリー・アクセス・パターンは、ベクトル化の効率および達成できるメモリー帯域幅の両方に影響します。MAP 収集は、実行中のデータアクセスを観察して、メモリーアクセスを含む命令を見つけます。収集および解析されたデータは、[Refinement Reports (リファインメント・レポート)] ウィンドウの [Memory Access Patterns Report (メモリー・アクセス・パターン・レポート)] タブに表示されます。

GUI (**図 10**) から MAP 解析を実行するには、サーベイレポートのチェックボックスを使用してループを選択し、MAP 収集を実行する必要があります。



10 **メモリー・アクセス・パターン・レポート**

コマンドラインから MAP 収集を実行することもできます。**`-mark-up-list`** オプションを使用して解析するループを選択します。

```
advixe-cl -collect map -mark-up-
list=Multiply.c:78,Multiply.c:71,Multiply.c:50,Multiply.c:6
1 -project-dir C:/my_advisor_project -- my_application.exe
```

メモリー・アクセス・パターン・レポートは、ループ実行中のメモリーアクセス操作で観察されたストライドの型に関する情報を提供します。ツールは、ユニット / 均一ストライドと定数ストライドの両方をレポートします (**図 11**)。

ユニット / 均一ストライド型

- **ユニットストライド (ストライド 1)** 命令は、反復ごとに 1 つの要素で一貫して変化するメモリーにアクセスします。
- **均一ストライド (ストライド 0)** 命令は、反復ごとに同じメモリーにアクセスします。
- **定数ストライド (ストライド N)** 命令は、反復ごとに N の要素 (N>1) で一貫して変化するメモリーにアクセスします。

さまざまなストライド型

- **不規則ストライド**命令は、ごとに予測できない数の要素で変化するメモリーアドレスにアクセスします。
- **ギャザー (不規則) ストライド**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・アーキテクチャーの **`v(p)gather*`** 命令で検出されます。

Memory Access Patterns Report | Dependencies Report | Recommendations

ID		Stride	Type	Source	Nested Function	Variable references	Max. Per-Instruction Addr. Range	Modules	Site Name	Access Type
P1			Parallel site information	Multiply.c:78				vec_samples.exe	loop_site_3	
P3		0	Uniform stride	Multiply.c:79		sumx	4B	vec_samples.exe	loop_site_3	Write
P4		1	Unit stride	Multiply.c:79		x	188B	vec_samples.exe	loop_site_3	Read

```
76  #pragma omp simd reduction(+:sumx)
77  #endif
78      for (k = 0;k < size2; k++) {
79          sumx += x[k]*b[k];
80      }
```

```
77  #endif
78      for (k = 0;k < size2; k++) {
79          sumx += x[k]*b[k];
80      }
81      }
```

```
77  #endif
78      for (k = 0;k < size2; k++) {
79          sumx += x[k]*b[k];
80      }
81      }
```

11 **ストライド型**

[Memory Access Patterns Report (メモリー・アクセス・パターン・レポート)] タブの任意の行をダブルクリックすると、選択した操作のソースコードが表示されます (**図 12**)。

[Source (ソース)] ビューと [Details (詳細)] ビュー (**図 13**) はどちらも、別の重要なインテル® Advisor のメモリー機能である、メモリー・フットプリントの詳細を表示します。

メモリー・フットプリント解析

メモリー・フットプリントは、指定されたループがアクセスするメモリーの範囲です。このフットプリントは、メモリー帯域幅の重要な指標となります。範囲が非常に広い場合、キャッシュに収めることはできません。キャッシュ・ブロッキングなどの最適化手法は、これらのケースで大きな違いをもたらします。インテル® Advisor には 3 つの異なるメモリー・フットプリント・メトリックがあります (**図 14**)。

Source | Assembly | Details

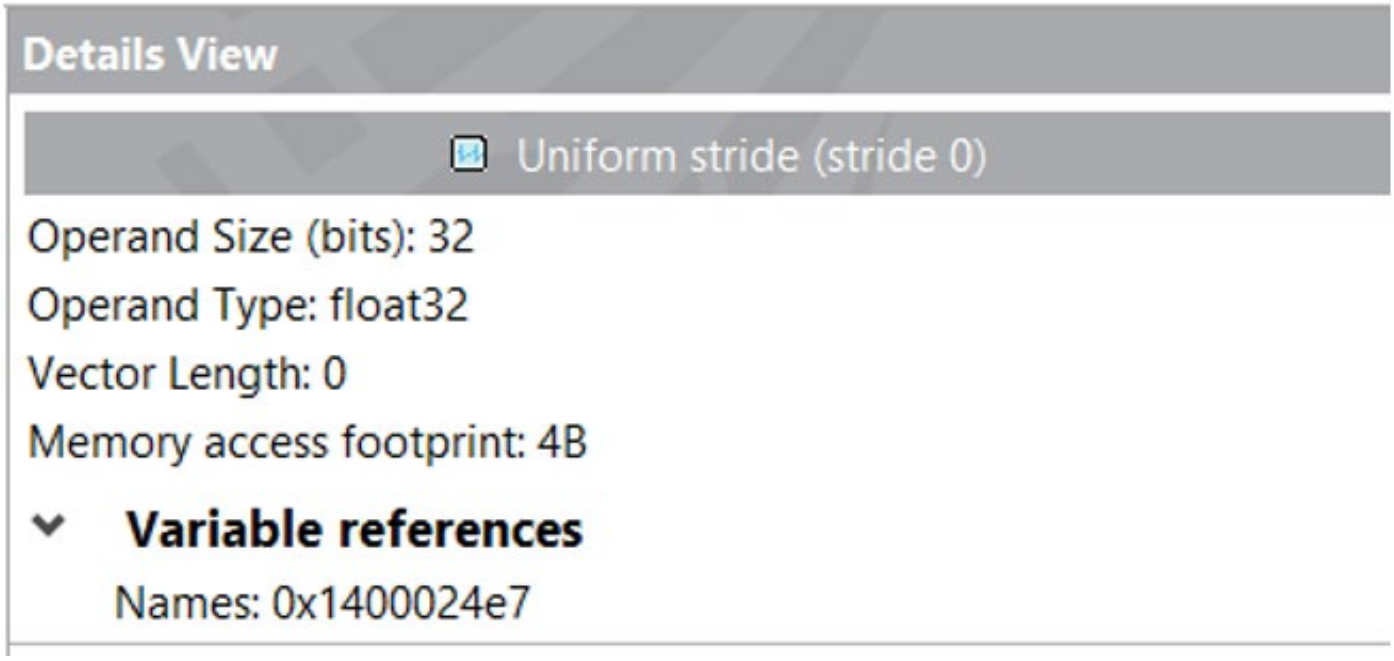
File: Multiply.c:78

Line	Source	Stride	Operand Type	Vector Length	Operand Size (bits)	Loop instance footprint
68	#endif					
69	#pragma nounroll					
70	#pragma vector always					
71	for (l = 0;l < size2; l++) {					
72	b[i] += a[l][i] * x[l];					
73	}					
74	#pragma nounroll					
75	#ifdef REDUCTION					
76	#pragma omp simd reduction(+:sumx)					
77	#endif					
78	for (k = 0;k < size2; k++) {					
79	sumx += x[k]*b[k];	[0] [1]	float32		32	188B
80	}					
81	}					

Module: vec_samples.exe!0x1400022e0

Address	Line	Assembly	Physical Stride	Operand Info	Vector Length	Operand Size (bits)	Address range	Memory access footprint
0x1400024c5	51	mov dword ptr [r9+r10*4], r15d						
0x1400024c9		**Block 21:**						
0x1400024c9	78	mov ebx, 0x0						
0x1400024ce	78	jle 0x14000253c <Block 29>						
0x1400024d0		**Block 22:**						
0x1400024d0	78	mov rsi, qword ptr [rsp+0x50]						
0x1400024d5		**Block 23:**						
0x1400024d5	79	vmovss xmm0, dword ptr [r9+rbx*4]	1	float32		32	0xde44 - 0xdefc	188B
0x1400024db	79	vmulss xmm2, xmm0, dword ptr [rsi+rbx*4]	1	float32		32	0xdcc4 - 0xdd...	188B
0x1400024e0	78	inc rbx						
0x1400024e3	79	vaddss xmm1, xmm2, xmm1						
0x1400024e7	79	vmovss dword ptr [rip+0x9481], xmm1	0	float32		32	0xb970 - 0xb9...	4B
0x1400024ef	78	cmp rbx, rdx						
0x1400024f2	78	jb 0x1400024d5 <Block 23>						

12 選択した操作のソースコードを確認



13 [Details (詳細)] ビュー

Summary | Survey & Roofline | Refinement Reports

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Footprint Estimate		
				Max. Per-Instruction Addr. Range	First Instance Site Footprint	Simulated Memory Footprint
[loop in main at tiling_inter.cpp:56]	No Information Available	98% / 0% / 2%	Mixed Strides	1019MB	5GB	0B
[loop in output_] at output.c:1073]	No Information Available	93% / 1% / 6%	Mixed Strides	4KB	9KB	0B
[loop in stencil_2d at tiling_inter.cpp ...	No Information Available	No Strides Found	No Strides Found	0B	0B	0B
[loop in stencil_2d at tiling_inter.cpp ...	No Information Available	No Strides Found	No Strides Found	0B	0B	0B

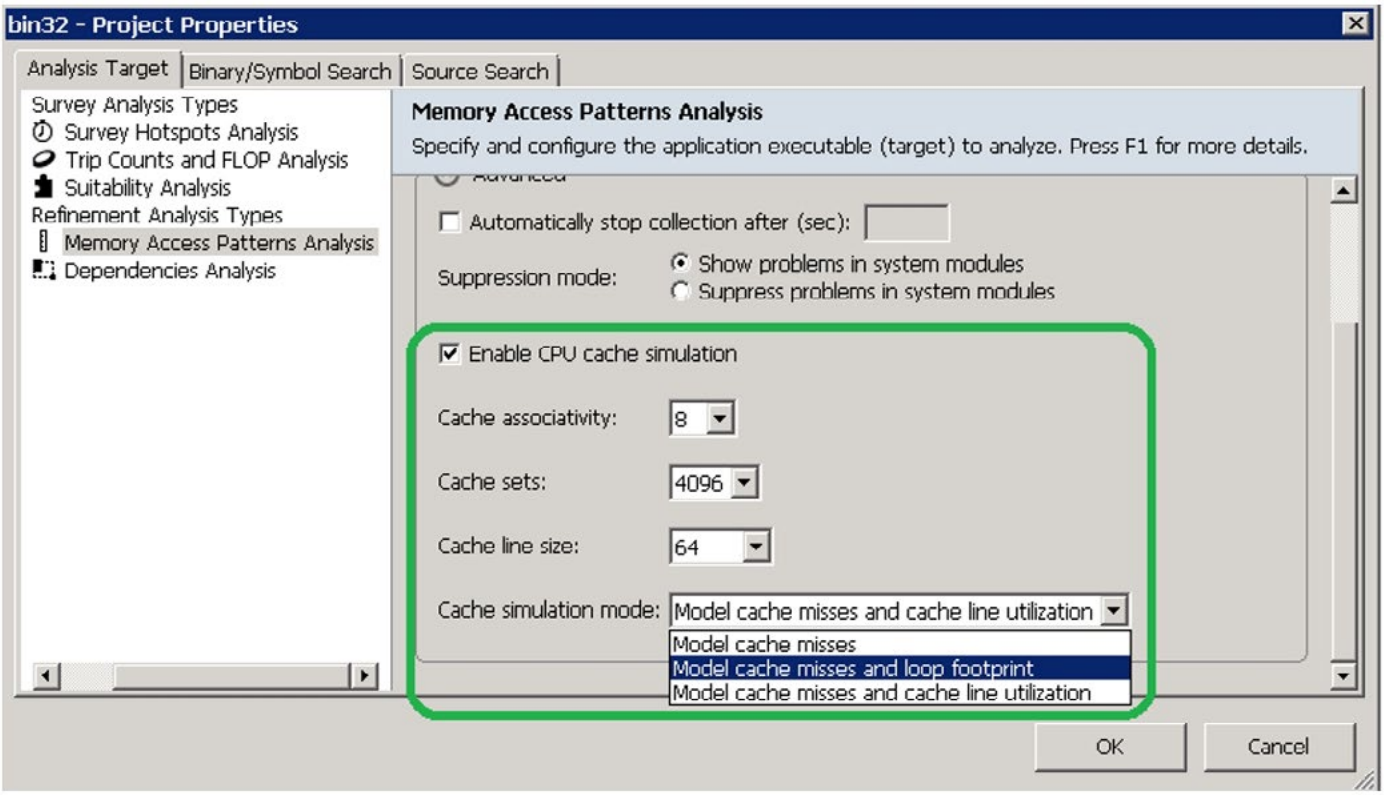
```
54
55   //    #pragma block_loop factor(512)
56       for (int i = 0; i < 80; ++i)
57           stencil_2d(resultArr1d, inputArr2d, i);
58
```

14 **メモリー・フットプリント・メトリック**

2 つの基本的なフットプリント・メトリックは、メモリー・フットプリントの一面のみ表します。これらのメトリックは、メモリー・アクセス・パターン (MAP) 解析でデフォルトで収集されます。

- **最大命令あたりアドレス範囲**は、このループの命令によりアクセスされた最小メモリーアドレス値と最大メモリーアドレス値の間の最大距離を表します。各メモリーアクセス命令について、そのアクセスの最小アドレスおよび最大アドレスが記録され、すべてのループ命令で最大範囲のアドレスがレポートされます。いくつかのフィルターにより、2 つ以上のループのインスタンスをカバーします。インテル® Advisor により信頼性が低いと判断されたメトリックはグレーでレポートされます。
- **最初のインスタンス・サイト・フットプリント**は、ループ反復のアドレス範囲のオーバーラップおよびループによりアクセスされたアドレス範囲間のギャップを考慮するため、メモリー・フットプリントの精度は高くなりますが、ループの最初のインスタンス (呼び出し) でのみ計算されます。

Simulated Memory Footprint (シミュレートされたメモリー容量) と呼ばれる、キャッシュ・シミュレーションに基づいて計算される高度なフットプリントもあります。このメトリックは、すべてのループのインスタンスを要約して示し、オーバーラップを考慮しますが、1 つのスレッドに制限されます。このメトリックは、キャッシュ・シミュレーション中にアクセスされたキャッシュラインの数とキャッシュラインのサイズを掛けた値として計算されます。GUI で有効にするには、[Project Properties (プロジェクトのプロパティー)] の [Memory Access Patterns Analysis (メモリー・アクセス・パターン解析)] タブの [Enable CPU cache simulation (CPU キャッシュのシミュレーションを有効にする)] チェックボックスをオンにして、[Cache simulation mode (キャッシュ・シミュレーション・モード)] ドロップダウン・リストで [Model cache misses and loop footprint (キャッシュミス・モデルおよびループ・フットプリント)] を選択します (**図 15**)。次に、[Survey (サーベイ)] ビューで調査するループのチェックボックスをオンにして MAP 解析を実行します。

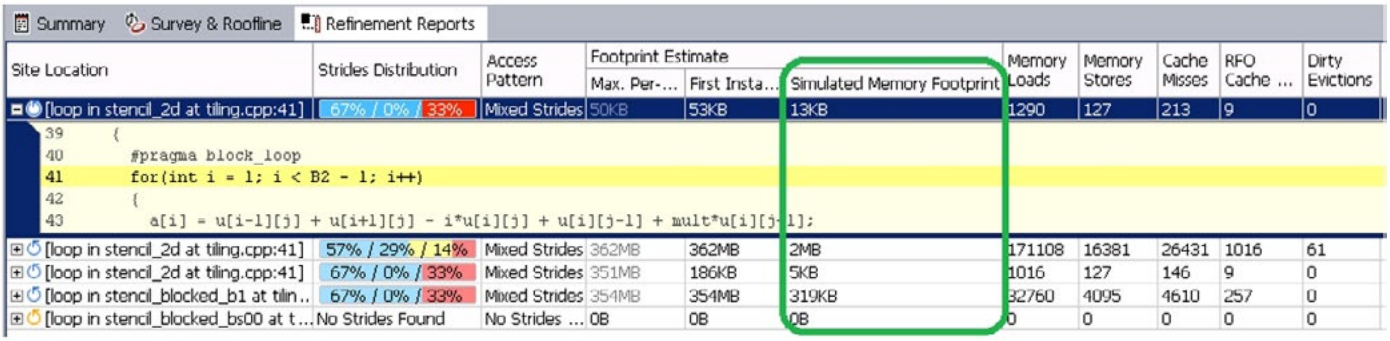


15 **メモリー・フットプリントのシミュレーション**

コマンドラインで有効にするには、MAP コマンドを使用して、**-enable-cache-simulation** および **-cachesim-mode=footprint** オプションを指定します。

```
advixe-cl -collect map -mark-up-
list=tiling_inter.cpp:56,output.c:1073 -enable-cache-
simulation -cachesim-mode=footprint -project-dir
C:\my_advisor_project -- my_application.exe
```

インテル® Advisor GUI の [Refinement Reports (リファインメント・レポート)] ビュー (**図 16**) で解析の結果を確認できます。詳細なキャッシュ関連のメトリック (メモリーロード、ストア、キャッシュミスの数、およびキャッシュ・シミュレート・メモリー・フットプリント) は、詳細なメモリーに関連するループの動作を示します。**表 2** は、インテル® Advisor のフットプリント・メトリックの適用性、制限、異なる種類のコードの解析との関連性を示したものです。



16 **インテル® Advisor GUI の [Refinement Reports (リファインメント・レポート)] ビュー**

表 2. インテル® Advisor のフットプリント・メトリック

	最大命令あたりのアドレス範囲	最初のインスタンス・サイト・フットプリント	シミュレートされたメモリー容量
ループ / サイトを解析するスレッド	1	1	1
解析するループのインスタンス	すべてのインスタンス、ショートカットを含む	1、最初のインスタンスのみ	ループ呼び出し回数制限オプションに依存
アドレス範囲のオーバーラップを考慮	×	○	○
ランダム・メモリー・アクセスを行うコードに適している	×	×	○

実際の例

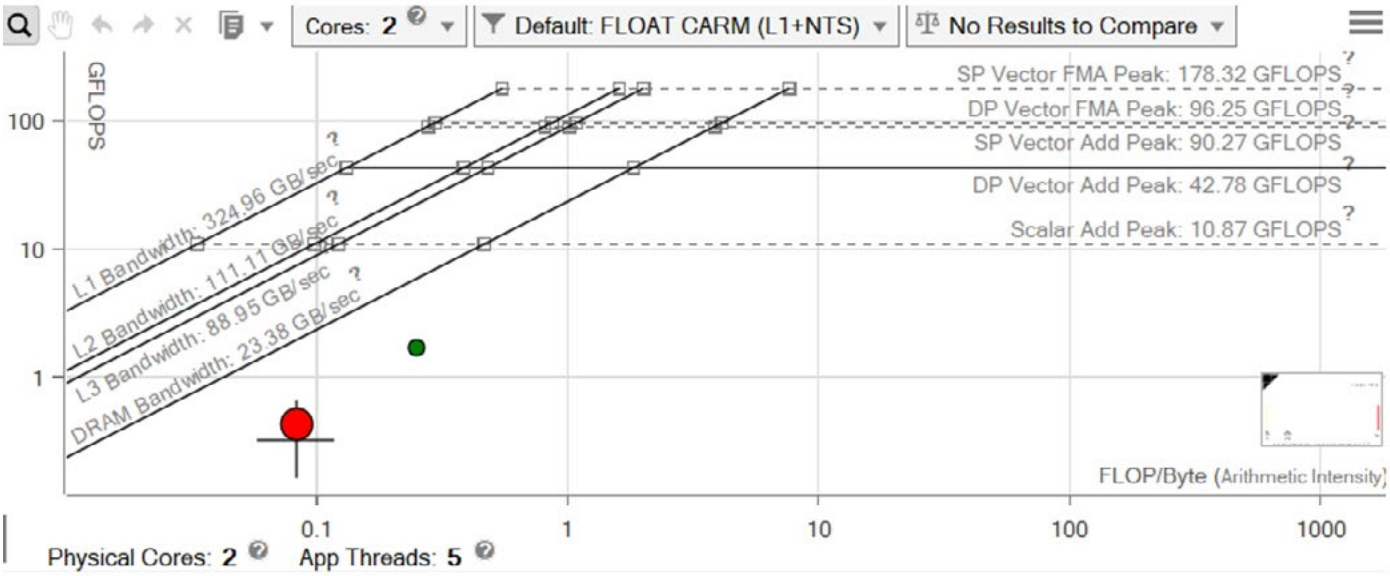
計算科学で最も一般的な問題は行列乗算を行います。行列を使用する分野はほぼ無限であり、人工知能、シミュレーション、モデリングはその一例に過ぎません。下記のサンプル・アルゴリズムは、三重の入れ子のループで各反復の乗算および加算を行っています。非常に計算負荷が高いだけでなく、多くのメモリーにアクセスします。インテル® Advisor を使用して確認してみましょう。

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

ベースラインの作成

最初の実行の経過時間は 53.94 秒でした。**図 17** は、キャッシュを考慮したルーフライン・グラフです。赤のドットはメインの計算ループです。達成しようとしている最大帯域幅の L1 帯域幅よりも下であるばかりか、DRAM 帯域幅よりもはるか下にあります。ループの [Code Analytics (コード解析)] タブを使用して、メモリー階層の各レベルで達成している正確な帯域幅を確認します (**図 18**)。

なぜパフォーマンスがこれほど低いのでしょうか。どうすればパフォーマンスを向上できるのでしょうか。インテル® Advisor でこれらの質問の答えを得ることにしましょう。最初に、[Survey (サーベイ)] ビュー (**図 19**) で、現在の状況とインテル® Advisor の推奨事項を確認します。インテル® Advisor は、メモリー・アクセス・パターンが非効率であること、および依存関係が仮定されているためにループがベクトル化されていないことを示しています。メモリー・アクセス・パターンを調べるために、メモリー・アクセス・パターン (MAP) 解析を実行します (**図 20**)。



17 キャッシュを考慮したルーフライン・グラフ

Data Transfers and Bandwidth

	Per Loop	Per Instance	Per Iteration	Float AI
L1, Gb	206.15843	0.00005	4.80000e-08	0.0833333
L2, Gb	617.65834	0.00015	1.43810e-07	0.0278145
L3, Gb	309.46159	0.00007	7.20521e-08	0.0555154
DRAM, Gb	64.16093	0.00002	1.49386e-08	0.267762

Self bandwidth by memory levels

L1 Gb/s	5.14634
L2 Gb/s	15.4186
L3 Gb/s	7.72511
DRAM Gb/s	1.60165

18 データ転送と帯域幅

Function Call Sites and Loops	Performance Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in multiply1 at multiply.c:50]	3 Inefficient memory access patterns present	158.567s	158.567s	Scalar	vector dependence prevents vectorizati
[loop in multiply1 at multiply.c:49]		0.206s	158.773s	Scalar	outer loop was not auto-vectorized: co
[loop in init_arr at matrix.c:50]	2 Potential underutilization of FMA instructions	0.010s	0.010s	Vectorized (Bod...	
[loop in multiply1 at multiply.c:48]		0.000s	158.773s	Scalar	outer loop was not auto-vectorized: co
[loop in init_arr at matrix.c:49]	**1 Data type conversions present**	**0.000s**	**0.010s**	**Scalar**	**inner loop was already vectorized**

19 [Survey (サーベイ)] ビュー

ID		Stride	Type	Source	Nested Function	Variable references
P1		2; 4096	Constant stride	multiply.c:51		block 0x18a25040 allocated at matrix.c:116, block 0x1aadb

```
49          for(j=0; j<msize; j++) {
50              for(k=0; k<msize; k++) {
51                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
52              }
53          }
```

P2			Parallel site information	multiply.c:50		
P4		0	Uniform stride	multiply.c:51		block 0x1cafa040 allocated at matrix.c:126

```
49          for(j=0; j<msize; j++) {
50              for(k=0; k<msize; k++) {
51                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
52              }
53          }
```

20 メモリー・アクセス・パターン (MAP) 解析

インテル® Advisor は、読み取りアクセスで定数ストライド、書き込みアクセスで均一ストライド (ストライド 0) を検出しました。アクセスしているメモリーの範囲は 32MB で、キャッシュサイズよりもはるかに大きな値です (**図 21**)。MAP レポート (**図 22**) を使用して、キャッシュの状況を確認します。2,300 を超えるキャッシュラインがミスしており、パフォーマンスが低いのも当然と言えます。この状況を改善する方法はいくつかあります。

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Footprint Estimate Max. Per-Instruction Addr. Ra
[loop in multiply1 at multiply.c:...	No Information Available	50% / 50% / 0%	Mixed Strides	32MB

```
48      for(i=tidx; i<msize; i=i+numt) {
49          for(j=0; j<msize; j++) {
50              for(k=0; k<msize; k++) {
51                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
52              }
```

21 **ストライド分布**

Memory Loads	Memory Stores	Cache Misses
4092	2046	2302

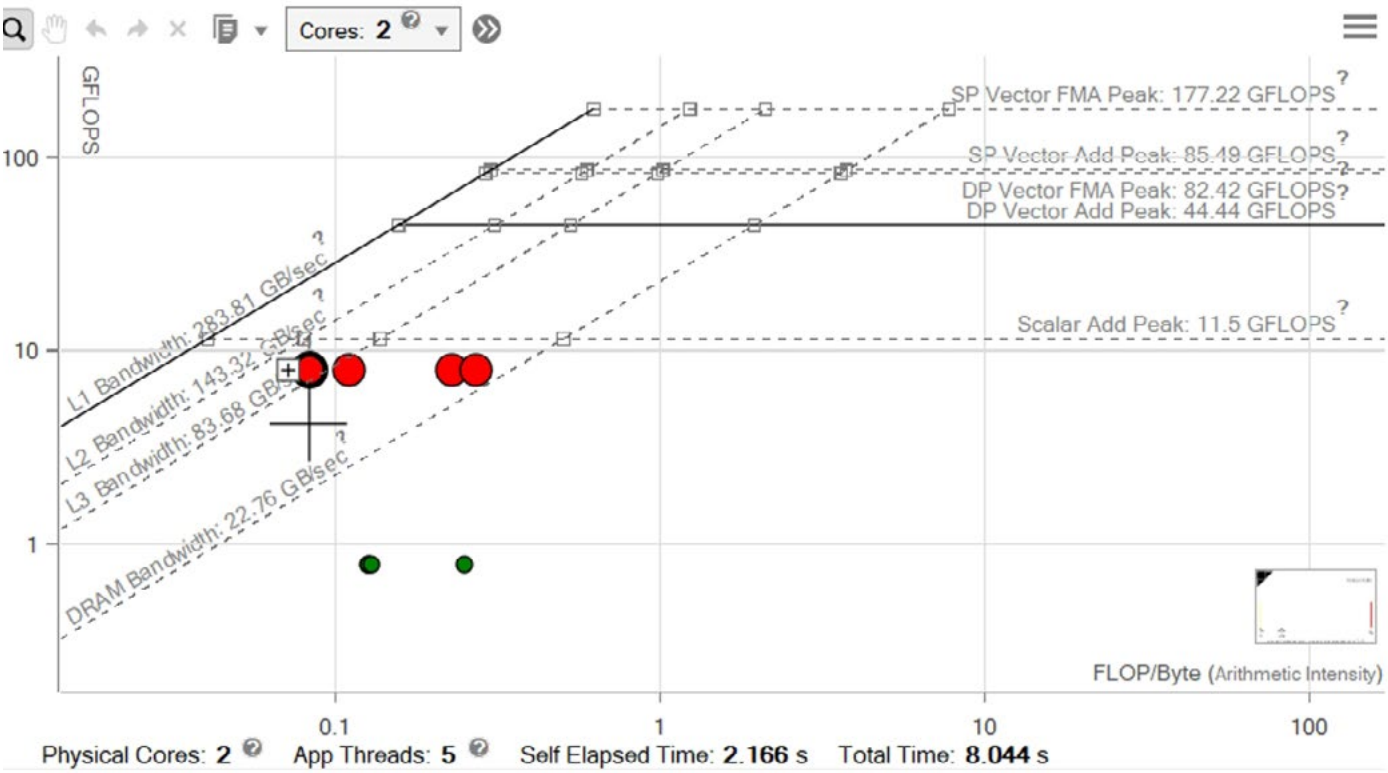
22 **MAP レポート**

ステップ 1

定数ストライドと広い範囲のメモリーアクセスが必要ないように、ループ交換を行います。また、**`#pragma ivdep`** を追加して、ベクトル化を妨げる依存関係が存在しないことをコンパイラーに知らせ、ループをベクトル化します。

```
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][k] * b[k][j];
        }
    }
}
```

修正後の実行経過時間は 4.12 秒と、パフォーマンスは 12.87 倍向上しました。なぜパフォーマンスは大幅に向上したのでしょうか。最初に、統合ルーフライン・グラフ (**図 23**) を見てみましょう。赤の円はそれぞれ、対応するメモリー階層 (L1、L2、L3、DRAM) の帯域幅を表します。左端の赤い円で表された、計算ループの L1 メモリー帯域幅が 95GB/秒になったことが分かります。[Survey (サーベイ)] ビュー (**図 24**) の、インテル® AVX2 命令のベクトル化効率は 100% になりました。



23 **統合ルーフライン・グラフ**

Function Call Sites and Loops	Performance Issues	Self Time	Total Time	Type	W. N.	Vectorized Loops			
						Vector...	Efficiency	Gain E...	VL (Ve...
[loop in multiply2 at multiply.c:65]	**1 Misaligned lo...**	**8.044s**	**8.044s**	**Vectorized (Body)**		**AVX2**	**100%**	**4.04x**	**4**
[loop in multiply2 at multiply.c:63]		0.058s	8.102s 99.7%	Scalar					
[loop in init_arr at matrix.c:50]	1 Data type conv...	0.021s	0.021s	Vectorized (Body)		AVX2	100%	6.21x	4
[loop in multiply2 at multiply.c:62]		0.000s	8.102s 99.7%	Scalar					
[loop in init_arr at matrix.c:49]	1 Data type conv...	0.000s	0.021s	Scalar					

24 **[Survey (サーベイ)] ビュー**

MAP レポート (**図 25**) は、アクセスがすべてユニットストライドで、最大アドレス範囲が 16KB と、キャッシュサイズ内に収まっていることを示しています。キャッシュのパフォーマンスも向上しました (**図 26**)。キャッシュミスの数は、2,302 から 512 に減りました。パフォーマンスは大幅に向上しましたが、まだピークには達していません。

Memory Access Patterns Report | Dependencies Report | Recommendations

ID	Stride	Type	Source	Nested Function	Variable references
P1		Parallel site information	multiply.c:65		
P3	1	Unit stride	multiply.c:66		block 0x19eeb040 allocated at matrix.c:121, block 0x1bef9(

```
64    #pragma ivdep
65                for(j=0; j<msize; j++) {
66                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
67                }
68            }
```

P4	1	Unit stride	multiply.c:66		block 0x1bef9040 allocated at matrix.c:126

```
64    #pragma ivdep
65                for(j=0; j<msize; j++) {
66                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
67                }
68            }
```

25 **MAP レポート**

Memory Loads	Memory Stores	Cache Misses
1022	511	512

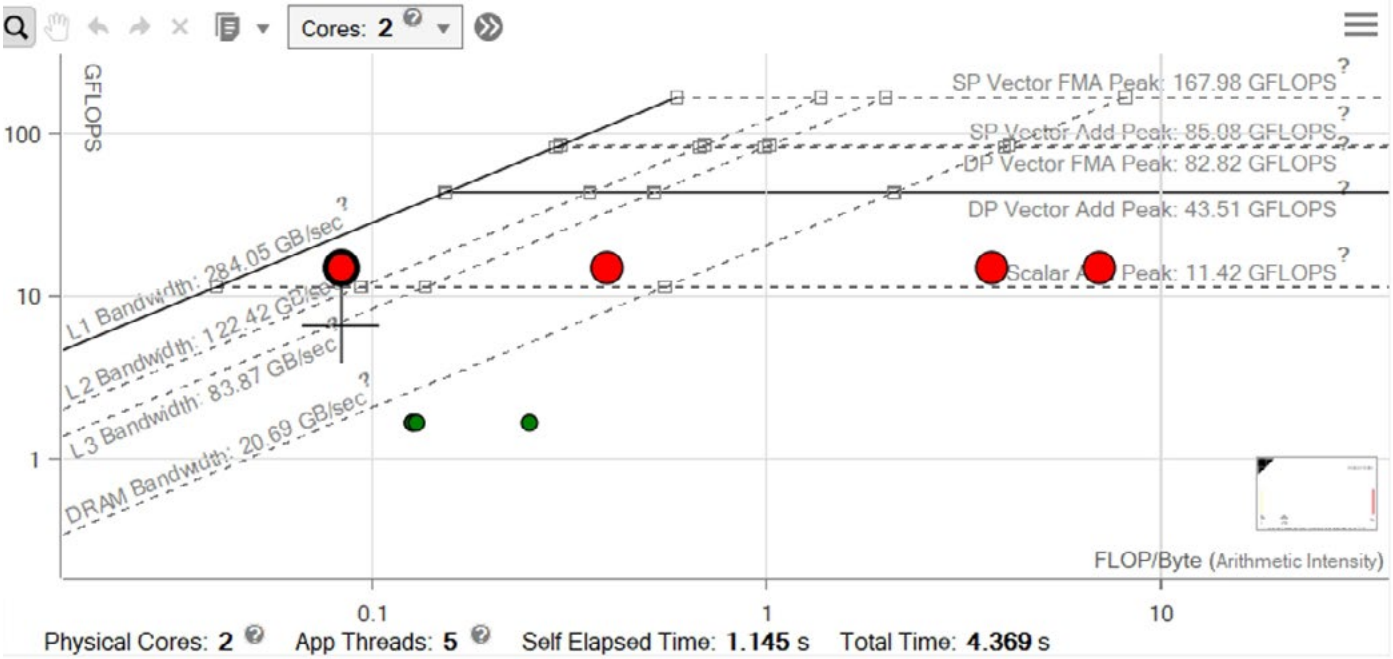
26 **キャッシュ・パフォーマンス**

ステップ 2

メモリーの小さな範囲で計算が行われるように、キャッシュ・ブロッキングを実装します。

```
for (i0 = ibeg; i0 < ibound; i0 +=mblock) {
    for (k0 = 0; k0 < msize; k0 += mblock) {
        for (j0 =0; j0 < msize; j0 += mblock) {
            for (i = i0; i < i0 + mblock; i++) {
                for (k = k0; k < k0 + mblock; k++) {
#pragma ivdep
#ifdef ALIGNED
     #pragma vector aligned
#endif //ALIGNED
               #pragma nounroll
                    for (j = j0; j < j0 + mblock; j++) {
                        c[i][j]  = c[i][j] + a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

上記のコードでは、ブロックで計算が行われるように、3 つの入れ子のループを追加しています。1 つのブロックが完了すると、次のブロックに移動します。キャッシュ・ブロッキング実装後の経過時間は 2.60 秒と、前回の実行からパフォーマンスは 1.58 倍向上しました (**図 27**)。ループの L1 メモリー帯域幅は 182GB/秒と、L1 ルーフにかなり近くなりました。ベクトル化およびストライドは変更していませんが、内部ループのキャッシュミスは 15 に、アドレス範囲は 480 バイトに減りました (**表 3**)。



27 **パフォーマンスの向上**

表 3. 結果のサマリー

解析の実行	経過時間 (秒)	合計 GFlops	メモリー アドレス範囲	キャッシュミス	向上率
ベースライン	53.94	0.32	32MB	2,302	N/A (ベースライン)
ループ交換	4.19	4.17	16KB	511	12.87 倍
ブロック	2.6	6.61	480 バイト	15	20.74 倍

メモリーアクセスの最適化

プログラムのメモリーアクセスを最適化することは極めて重要です。インテル® Advisor のようなツールを使って、プログラムがどのようにメモリーにアクセスしているか理解することは、ハードウェアのパフォーマンスを最大限に引き出すのに役立ちます。インテル® Advisor のルーフラインおよび新しい統合ルーフライン機能を使用することにより、メモリーのボトルネックを視覚化して確認できます。ルーフライン機能とメモリー・アクセス・パターン解析を組み合わせると、詳細なメモリーの情報が得られます。

関連情報

- **インテル® Advisor のルーフライン (英語)**
- **インテル® Advisor の整数ルーフライン (英語)**
- **インテル® Advisor の統合ルーフライン (英語)**
- **インテル® Advisor 日本語版 (iSUS 翻訳版)**

コンパイラーの最適化に関する詳細は、最適化に関する注意事項（software.intel.com/articles/optimization-notice#opt-jp）を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation またはその子会社の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation

インテル® MPI ライブラリーの MPI-3 非ブロッキング I/O 集合操作

HPC アプリケーションの I/O を高速化

Nitya Hariharan、Amarpal Singh Kapoor、Rama Kishan Malladi インテル コーポレーション
コア / ビジュアル・コンピューティング・グループ テクニカル・マーケティング・エンジニア
Md Vasimuddin インテル ラボ 並列コンピューティング・ラボ リサーチ・サイエンティスト

アプリケーションを真にスケーラブルにするには、すべてのセクションが線形にスケーリングする (少なくとも、線形スケーラビリティーの傾向がある) 必要があります。アムダールの法則は、小さなシリアルセクションまたは不適切にスケーリングされた並列コードセクションがアプリケーション全体のスケーラビリティーに大きな影響を与えると述べています。この記事では、シリアルになる傾向がある HPC アプリケーションの 1 つのセクション、ファイル I/O に注目します。実際の HPC アプリケーションのコンテキストで非ブロッキング MPI I/O 呼び出しを使用して、パフォーマンス上の利点を説明します。

開発者が通信と計算、および通信とほかの通信のオーバーラップから利点が得られるように、I/O 負荷の高いコードを支援するため、MPI は非ブロッキング呼び出しを提供しています。[**The Parallel Universe Issue 33**[1] の「MPI-3 の非ブロッキング集合操作による通信レイテンシーの隠蔽」では、非ブロッキング集合 (NBC) 通信の使用方法と利点を説明しました。この記事では、非ブロッキング I/O (**`MPI_File_iwrite_at`** など) に注目します。]

NBC I/O

NBC I/O 操作は非ブロッキングおよび集合操作の両方を利用できます。[2] 非ブロッキング呼び出しは、別の独立した計算を並列に行うことにより、I/O コストを相殺できる可能性があります。ハードウェアを最大限に利用できるだけでなく、これらの非ブロッキング I/O 呼び出しは、遅延プロセスの同期コストの隠蔽にも役立ちます。計算と比較して I/O のパフォーマンスが低い場合にもこのような最適化を検討します。

LAMMPS および BWA-MEM の 2 つのアプリケーション・コードを見て、非ブロッキング I/O 呼び出しの使用によるパフォーマンス・ゲインを考察します。LAMMPS は、材料のモデリングに重点を置いた分子動力学アプリケーションです。[3] BWA-MEM は、短鎖 DNA 断片 (リードと呼ばれる) をマッピングしてヒトゲノムなどの配列を決定する最も有名なツールの 1 つです。[4] どちらも MPI I/O を使用しており、NBC I/O テストの典型的な候補です。

テストケース 1: LAMMPS

LAMMPS は各行にコマンドを含む入力ファイルを使用して実行します。コマンドごとに LAMMPS は、内部変数を設定する、または指定された時間ステップ数のシミュレーションを実行するなどの処理を行います。例えば、**`dump`** コマンドは次のようになります。

```
dump ID group-ID style N file args
```

この記事の I/O テストでは、3D Lennard-Jones (LJ) ワークロードと、次の **`dump`** コマンドを入力ファイルに追加して使用しました。

```
dump myDump all atom/mpiio 20 dump.atom.mpiio
```

(**`dump`** コマンドの各入力の詳細は、この記事の最後にある参考資料 4 を参照してください。)

コードは、ダンプのリストを管理し、リストをチェックして N 時間ステップごとに行うダンプを決定します。書き込まれる原子データは文字列バッファーにコピーされます。出力ファイルのオフセットは各 MPI ランクで計算されます。次に、集合、ブロッキング MPI I/O 呼び出し **MPI_File_write_at_all** を使用してデータをファイルに書き込みます。

ベースラインのコードはブロッキング **MPI_File_write_at_all** を使用しています。

```
//Iterate over n time steps, check for dump at N time steps
//Copy atom data to string and calculate offset into file
call MPI_File_write_at_all(mpifh, offset, sbuf, nsme, MPI_CHAR, status);
```

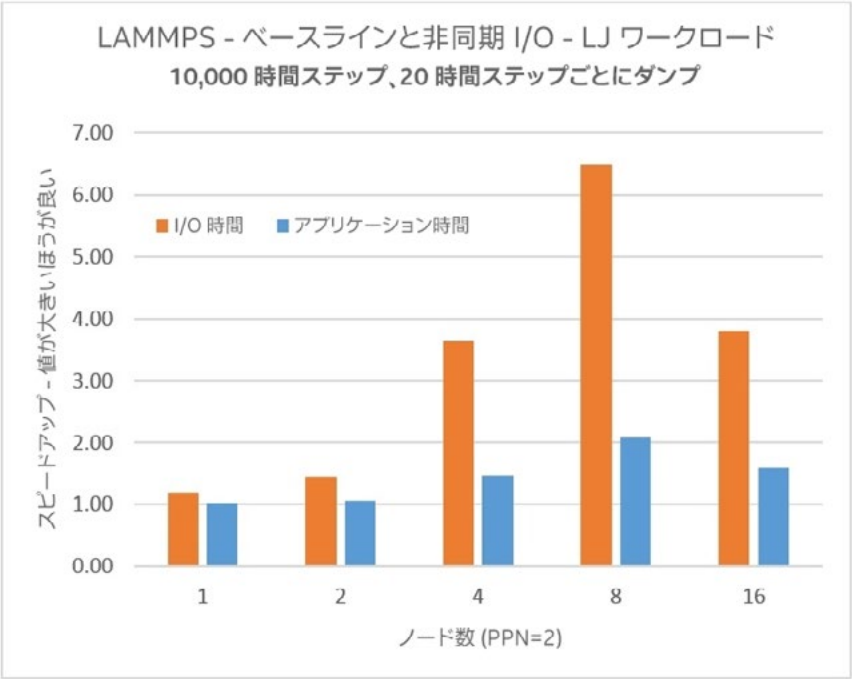
最適化後のコードは非ブロッキング **MPI_File_iwrite_at** を使用しています。

```
        //Iterate over n time steps, check for dump at N time steps
if(countWrites%2==0){
        //Copy atom data to string buffer sbuf, and calculate offset into file
        MPI_File_iwrite_at(mpifh, offset, sbuf, count, MPI_CHAR, request); }
else {
        //Copy atom data to string buffer sbuf1, and calculate offset into file
        MPI_File_iwrite_at(mpifh, offset, sbuf1, count, MPI_CHAR, request1);}
//Compute for next iteration
if(countWrites%2==0) MPI_Wait(&request1, &status1);
else MPI_Wait(&request, &status);
```

各反復で書き込まれるデータが独立していれば、ここで NBC I/O 呼び出しを使用して計算とファイル I/O をオーバーラップすることができます。ファイルのオフセットは MPI I/O の準備で各ランクごとにすでに計算されているため、NBC I/O 呼び出しを使用し、対応する待機を追加するようにコードを変更します。

また、各 MPI ランクはファイルに独立して書き込むため、非ブロッキングの集合書き込み操作を行う **MPI_File_iwrite_at_all** の代わりに非ブロッキング **MPI_File_iwrite_at** 呼び出しを使用します。計算と I/O が適切にオーバーラップするように、異なるバッファーを使用して異なる反復に非同期書き込みを行う、“prologue” および “epilogue” フェーズ (つまり、ダブルバッファリング・アルゴリズム) を使用します。書き込み数を把握して、使用するバッファーを決定し、現在の反復で前の反復用に **MPI_Wait** 呼び出しを行うことにより、正しくバッファーを再利用します。

図 1 は、ノード数を増やした (ノードあたり 2 つの MPI プロセス、MPI ランクあたり 20 の OpenMP* スレッド) LJ ワークロード (10,000 時間ステップ、20 時間ステップごとにデータをダンプ) の MPI I/O と合計実行時間のスピードアップを示しています。テストは、**インテル® Xeon® Gold 6148 プロセッサー**・ベースのクラスター (ノードあたり 2 ソケット、**インテル® Omni-Path アーキテクチャー**で接続) で行われました。I/O と合計実行時間は LAMMPS のデフォルトのタイマーを使用して測定されました。I/O 時間のスピードアップは、対応する合計実行時間のスピードアップには反映されていませんが、並列の非ブロッキング I/O により、全体的なパフォーマンスは向上し、8 ノードのアプリケーションのパフォーマンスは 2 倍になりました。



1 **LAMMPS の MPI I/O 時間と合計実行時間のスピードアップ**

テストケース 2: BWA-MEM

次世代シーケンシング (NGS) アプリケーションにおけるシーケンスマッピングは計算負荷の高い処理です。ゲノム・アセンブリーおよび遺伝的変異発見で広く利用されている GATK (Genome Analysis ToolKit) ワークフローの初期段階[5] では、シーケンス・マッピングは実行時間全体の 30% を占めます。BWA-MEM[6] は、短鎖 DNA 断片 (リード) をマッピングしてゲノム配列を決定する最も有名なツールの 1 つです。入力シーケンスのセットが指定されると、BWA-MEM は参照シーケンス内で最も可能性の高い場所を探します。

BWA-MEM は分散メモリーシステムをサポートし、適切にスケーリングします。また、出力の書き込みに MPI I/O を使用します。下記のコードサンプルは、ブロッキング MPI I/O 呼び出しを使用した実装と非ブロッキング MPI I/O を使用した最適化を示しています。ベースラインのコードはブロッキング **`MPI_File_write_at`** を使用しています。

```
    MPI_Status status;
for (int l=0; l<index; l++) {
        std::pair<int64_t, ktp_data_t*> pr = samCache[l];
        assert(pr.first < aux->ntasks);
        int64_t offset = totSamSize[pr.first];
        ret = pr.second;
        std::string str;

        for (i = 0; i < ret->n_seqs; ++i) {
                if (ret->seqs[i].sam)
                        str += ret->seqs[i].sam;
                free(ret->seqs[i].sam);
         }
         free(ret->seqs);


         /* Write to output SAM file */
         MPI_File_write_at(aux->mfp, offset, str.c_str(), \
                           samSize[pr.first], MPI_INT8_T, &status);
```

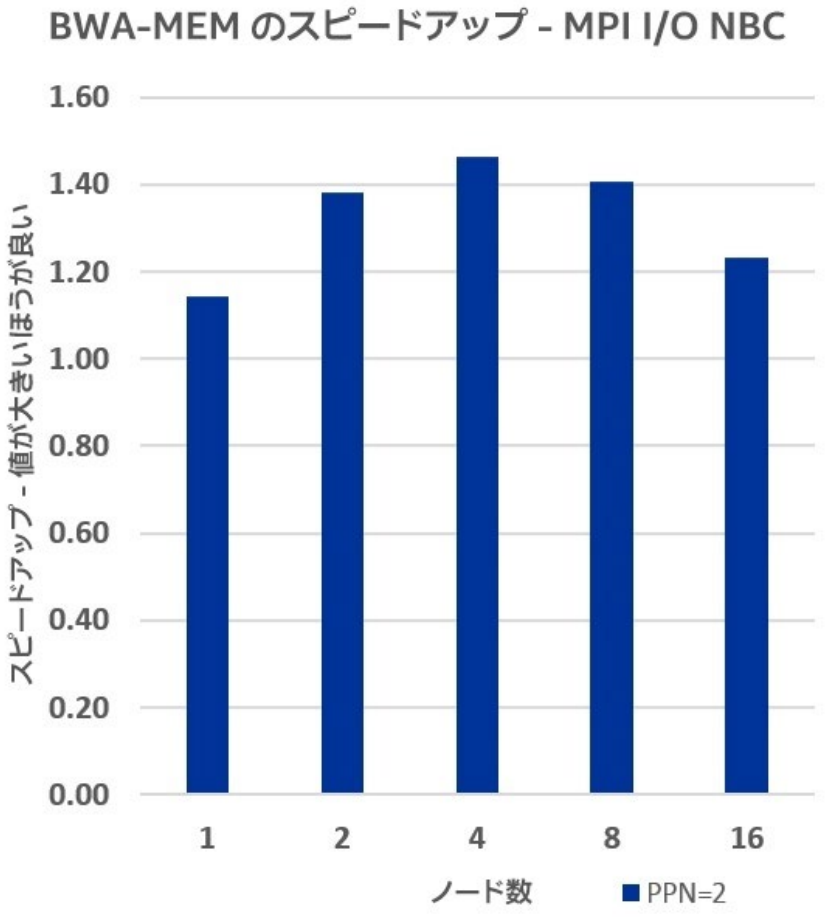
最適化後のコードは非ブロッキング **MPI_File_iwrite_at** を使用しています。

```
    MPI_Request request[index];
        int64_t offset[index];
        int64_t first[index];
        int test_flag;
        std::vector<std::string> str(index);
        for (int l=0; l<index; l++)
        {
                std::pair<int64_t, ktp_data_t*> pr = samCache[l];
                assert(pr.first < aux->ntasks);
                first[l] = pr.first;
                offset[l] = totSamSize[first[l]];
                ret = pr.second;

                for (i = 0; i < ret->n_seqs; ++i) {
                        if (ret->seqs[i].sam)
                                str[l] += ret->seqs[i].sam;
                        free(ret->seqs[i].sam);
                }
                free(ret->seqs);

                /* Write to output SAM file */
                MPI_File_iwrite_at(aux->mfp, offset[l], str[l].c_str(), \
                                   samSize[first[l]], MPI_INT8_T, &request[l]);
        }
        MPI_Waitall(index,request,MPI_STATUSES_IGNORE);
```

図 2 は、NBC MPI I/O (ノードあたり 2 MPI プロセス、MPI ランクあたり 20 スレッド) を使用した BWA-MEM パフォーマンスの向上を示しています。テストは、インテル® Xeon® Gold 6148 プロセッサー・ベースのクラスター (ノードあたり 2 ソケット、インテル® Omni-Path アーキテクチャーで接続) で行われました。並列、非ブロッキング I/O を利用することにより、各テストのパフォーマンスは向上しました。



2 **NBC MPI I/O を使用した BWA-MEM パフォーマンスの向上**

HPC アプリケーションの I/O を高速化

近年の HPC アプリケーションは非常に複雑で、反復的なソリューションが多用されます。その結果、複数のソリューション・ステップで、現在の実行が失敗した場合に備えて、ファイルへの書き込みが必要となるだけでなく、追加の再起動や一時ファイルへの書き込みのような安全性を考慮したメカニズムも必要とされます。MPI I/O を利用することは、HPC 環境で I/O を効率良く処理するための最初のステップです。この記事では、LAMMPS および BWA-MEM の 2 つのアプリケーションで非ブロッキング MPI I/O 呼び出しを使用してパフォーマンスを向上する方法を紹介しました。コードの変更は、これらのアプリケーションで NBC MPI I/O を使用することの容易さを示しています。

参考資料

[1]**The Parallel Universe, Issue 33**

[2]Seo S., Latham R., Zhang J., and Balaji P. “Implementation and Evaluation of MPI Nonblocking Collective I/O.”

[3]S. Plimpton, “Fast Parallel Algorithms for Short-Range Molecular Dynamics,” J Comp Phys, 117, 1-19 (1995).

[4]LAMMPS dump command: **https://lammps.sandia.gov/doc/dump.html** (英語)

[5]M. Vasimuddin, S. Misra, and S. Aluru, “Identification of Significant Computational Building Blocks through Comprehensive Investigation of NGS Secondary Analysis Methods,” [Preprint] bioRXiv, April 2018

[6]Li H. (2013) Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv:1303.3997v1 [q-bio.GN].

この Python* は役に立つ

インテル® Distribution for Python* で
アプリケーションをスーパーチャージ

無料のダウンロード (英語) >

intel Software

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation またはその子会社の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation

Software

THE PARALLEL UNIVERSE

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。§ さらに詳しい情報をお知りになりたい場合は、https://www.intel.com/benchmarks (英語) を参照してください。

パフォーマンス結果は 2018 年 10 月 1 日現在の測定値であり、すべての公開済みセキュリティー・アップデートが適用されていない可能性があります。詳細は、システム構成を参照してください。絶対的なセキュリティーを提供できる製品はありません。

インテル® ソフトウェア開発製品のパフォーマンスおよび最適化に関する詳細は、最適化に関する注意事項 (https://software.intel.com/articles/optimization-notice#opt-jp) を参照してください。

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。実際の性能はシステム構成によって異なります。絶対的なセキュリティーを提供できるコンピューター・システムはありません。詳細については、各システムメーカーまたは販売店にお問い合わせください。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。

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。

本資料には、開発中の製品、サービスおよびプロセスについての情報が含まれています。本資料に含まれる情報は予告なく変更されることがあります。最新の予測、スケジュール、仕様、ロードマップについては、インテルの担当者までお問い合わせください。

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。

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

© 2019 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Arria、Stratix、Xeon、OpenVINO は、アメリカ合衆国および / またはその他の国における Intel Corporation またはその子会社の商標です。

OpenCL および OpenCL ロゴは、Apple Inc. の商標であり、Khronos の使用許諾を受けて使用しています。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

JPN/1902/PDF/XL/IAGS/SS