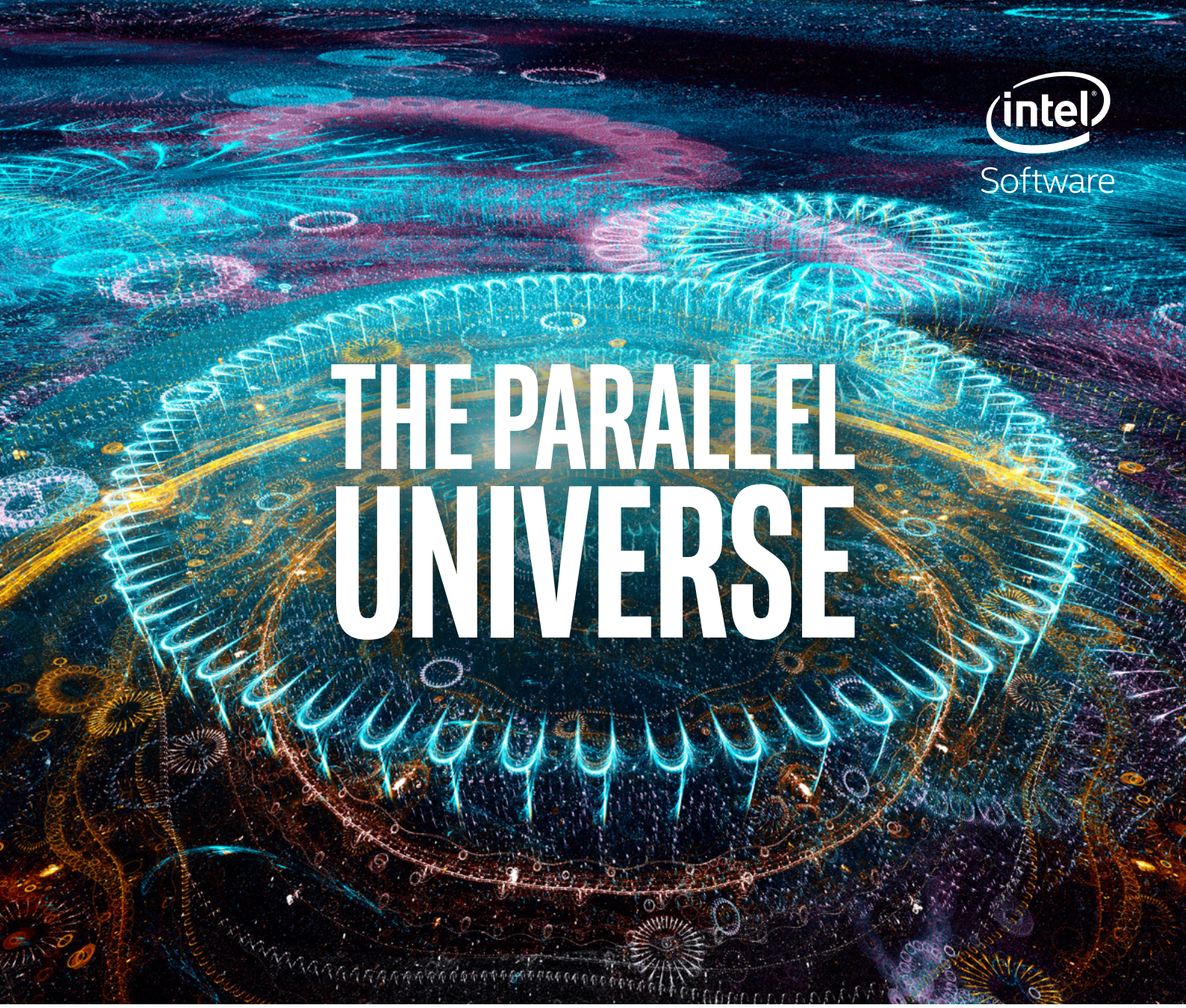


コンピューター・ビジョンの普及

ゲームをスピードアップ：ゲーム・プログラマー向けの
実践的な最適化

命令パイプラインに関する考察：適応性の鍵

Issue
32
2018

目次

注目記事

編集者からのメッセージ 3

コンピューター・ビジョン：間もなくブラウザーで利用可能に

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

コンピューター・ビジョンの普及 5

Open Web Platform* 内でコンピューター・ビジョンを利用する

ゲームをスピードアップ 23

インテル® Graphics Performance Analyzers を利用してゲーム開発のあらゆる作業を最適化する方法

ハイパフォーマンス・ビッグデータ・コンピューティング向け Harp-DAAL 31

生産性とパフォーマンスを同時に向上する鍵

命令パイプラインに関する考察 41

現代のアプリケーション・プログラミングにおける適応性の鍵

インテル® Xeon® スケーラブル・プロセッサー上の並列 CFD と HiFUN ソルバー 51

高速な数値シミュレーション向けに HPC プラットフォームの性能を最大化

VASP 材料シミュレーション・パフォーマンスの向上 61

最新のインテル® ソフトウェア開発ツールを利用してハードウェアを効率良く使用する

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。

コンピューター・ビジョン : 間もなくブラウザーで利用可能に

私はコンピューター・ビジョンについて、**Dr. Dobb's Journal に掲載されている Gary Bradski の OpenCV* の記事** (英語) で初めて知りました。Gary は当時インテル ラボに在籍していました。私はその数か月前にインテルに入社したばかりで、インテルがコンピューター・ビジョン分野で取り組みを行っていることに驚きました。OpenCV* は 17 年前に最初にリリースされて以来、長い道のりを歩んできました。その名のとおり、オープンソースとして、非営利団体の **OpenCV.org** (英語) によって支援されてきました。この号の注目記事「**コンピューター・ビジョンの普及**」では、カリフォルニア大学アーバイン校とインテル ラボの共同研究者グループによる、**Open Web Platform*** (英語) 内でハイパフォーマンスなコンピューター・ビジョンを実現する取り組みとして、OpenCV* の主要機能の JavaScript* への変換と、エッジ検出、顔認識、背景除去や、ディープ･ニューラル･ネットワークを用いた画像認識などの一般的なコンピューター・ビジョン・アプリケーションをブラウザー内で実行する魅力的な例を紹介します。

グラフィックスおよびビデオ処理に関する記事、「**ゲームをスピードアップ**」では、**インテル® Graphics Performance Analyzers** (英語) を利用してゲーム・パフォーマンスを最適化する方法を示します。これらの開発ツールが、アーティスト、デザイナー、ゲームプレイ･プログラマー、ゲーム･エンジン･プログラマーなど、ゲーム開発チームの異なるメンバーにもたらす利点をユースケースを用いて紹介します。

Google* File System (英語) と **MapReduce** (英語) は、2003 年に大規模な分散メモリー型データ解析時代を切り拓きました。それ以来、大規模なデータ解析とマシンラーニングの多くの選択肢が利用可能になりました。**Apache Spark*** (英語) が私の現在のお気に入りですが、これまでに **Dask** (英語)、**HPAT** (インテル ラボの High-Performance Analytics Toolkit プロジェクト)､その他いくつかの選択肢を試したことがあります。「**ハイパフォーマンス・ビッグデータ・コンピューティング向け Harp-DAAL**」でインディアナ大学の Judy Qiu 教授は、高度に最適化された**インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL)** を利用する新しいフレームワークを紹介し、Harp-DAAL を使用して K 平均法を実装する Java* コードの例を示しています。

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。

ハイレベルの抽象化とは正反対の「**命令パイプラインに関する考察**」は、命令パイプラインの概念的な概要と、現代のプロセッサーのパフォーマンスにおけるその重要性を説明します。この号の最後を締めくくるのは、計算負荷の高い科学アプリケーションのパフォーマンスの向上に関する 2 つの記事「**インテル® Xeon® スケーラブル・プロセッサー上の並列 CFD と HiFUN ソルバー**」と「**VASP 材料シミュレーション・パフォーマンスの向上**」です。前者は、実際の流体力学アプリケーションを使用して並列パフォーマンスのチューニングを行うケーススタディーです。後者は、**インテル® VTune™ Amplifier** のアプリケーション・パフォーマンス・スナップショットを使用してアプリケーション・パフォーマンスの要約を取得する方法と、**インテル® MPI ライブラリー**の新しいプレビュー機能を紹介します。

今後の The Parallel Universe では、Apache Spark* 解析フレームワーク内でのディープラーニングの実行、Java* パフォーマンスの拡張、I/O パフォーマンスのプロファイルとチューニング、インテル® ソフトウェア開発ツールの新機能などを含む、さまざまなトピックについて取り上げる予定です。お見逃しなく。

Henry A. Gabb
2018 年 4 月

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。

コンピューター・ビジョンの普及

Open Web Platform* 内でコンピューター・ビジョンを利用する

Sajjad Taheri カリフォルニア大学アーバイン校コンピューター・サイエンス学部 博士課程在籍
Alexeandru Nicolau カリフォルニア大学アーバイン校コンピューター・サイエンス学部 教授
Alexeander Vedienbaum カリフォルニア大学アーバイン校コンピューター・サイエンス学部 教授
Ningxin Hu インテル コーポレーション シニア・スタッフ・エンジニア
Mohammad Reza Haghighat インテル コーポレーション シニア主席エンジニア

Web は、世界で最も普及している計算プラットフォームであり、デジタル経済の基盤です。1990 年代初めに誕生して以来、Web の機能は量と質の両方で向上しています。しかし、コンピューター・ビジョンは、次のような理由からまだ Web 上で主流ではありません。

- Web の標準言語である **JavaScript* のパフォーマンスが十分ではない**
- 標準 Web API で**カメラがサポートされていない**
- **包括的なコンピューター・ビジョン・ライブラリーがない**

これらの問題は間もなく解決され、オンライン・ショッピング、教育、エンターテインメントなどに変革をもたらす、没入感のある知覚的な Web となっていくでしょう。

この 10 年間で JavaScript* のパフォーマンスが大幅に改善されたことに加えて、最近の WebAssembly の登場もあり、Web とネイティブ・コンピューティングのパフォーマンス・ギャップが埋まりつつあります。また、HTML5 WebRTC API により、Open Web Platform* でカメラがサポートされました。しかし、Web 向けの包括的なコンピューター・ビジョン・ライブラリーはまだありません。この記事では、OpenCV* を Open Web Platform* に追加することでこの最後の課題に対処します。

OpenCV* は、包括的なビジョン関数のセットと大規模なデベロッパー・コミュニティーを持つ、最もよく使用されているコンピューター・ビジョン・ライブラリーです。C++ で実装されており、これまでは、評判の良くないネイティブプラグインを使用しないと Web ブラウザーでは利用できませんでした。

この記事では、OpenCV* の効率、完全性、成熟した API、コミュニティーの共有ナレッジを活用して、数百の OpenCV* 関数を Open Web Platform* で利用できるようにします。それは、JavaScript* エンジンが最適化しやすい形式で提供され、Web プログラマーが簡単に習得してアプリケーション開発に使用できる API を備えています。さらに、SIMD (Single Instruction, Multiple Data) ユニットと複数のプロセッサー・コアをターゲットとする OpenCV* の並列実装を等価な Web プリミティブへ移植して、リアルタイムのインタラクティブなケースで要求されるハイパフォーマンスを達成する方法を示します。

Open Web Platform*

Open Web Platform* は、数十億のデバイスが接続された最も広く利用されている計算プラットフォームです。Web 上のマルチメディア・コンテンツの量と同様に、オンライン取引、エンターテインメント、科学、教育の分野において Open Web Platform* の採用が急速に拡大しています。しかし、Web ブラウザー上でのコンピューター・ビジョン処理はまだ一般的ではありません。いくつかの制限によりクライアントサイドでのビジョン処理が妨げられています。

- マルチメディア・コンテンツにアクセスし転送するための**標準 Web API がない**
- **JavaScript* のパフォーマンスが低い**
- アプリケーションの開発に必要な**包括的なコンピューター・ビジョン・ライブラリーがない**

ここで紹介するアプローチと Web フロントに関する最近の発展は、これらの制限に対処し、Web で適切なコンピューター・ビジョン機能を利用できるようにします。

カメラサポートの追加とプラグインフリーのマルチメディア配信

HTML5 により、サードパーティー・プラグインを使用せずに、ブラウザーでマルチメディア・コンテンツをキャプチャー、転送、表示する Web API が追加されました。その 1 つである Web リアルタイム通信 (WebRTC) は、マルチメディア・コンテンツとビデオ要素を取得して、ピアツーピア転送でビデオの表示を可能にします。

最近、仮想現実 (VR) や拡張現実 (AR) コンテンツを利用する没入型 Web が、新しい魅力的なユーザー体験を提供し始めています。

改善された JavaScript* パフォーマンス

JavaScript* は Web の主要言語です。動的型付けのスクリプト言語であるため、C++ などのネイティブ言語と比べてパフォーマンスが劣ります。マルチメディア処理では通常、複雑なアルゴリズムと膨大な量の計算を扱います。JIT (Just-in-Time) などのクライアントサイドのテクノロジーに加えて、Web 向けの移植性の高いバイナリー形式である WebAssembly (WASM) の登場により、Web クライアントは JavaScript* でネイティブに近いパフォーマンスを実現し、より負荷の高いタスクを処理できるようになりました。

包括的なコンピューター・ビジョン・ライブラリー

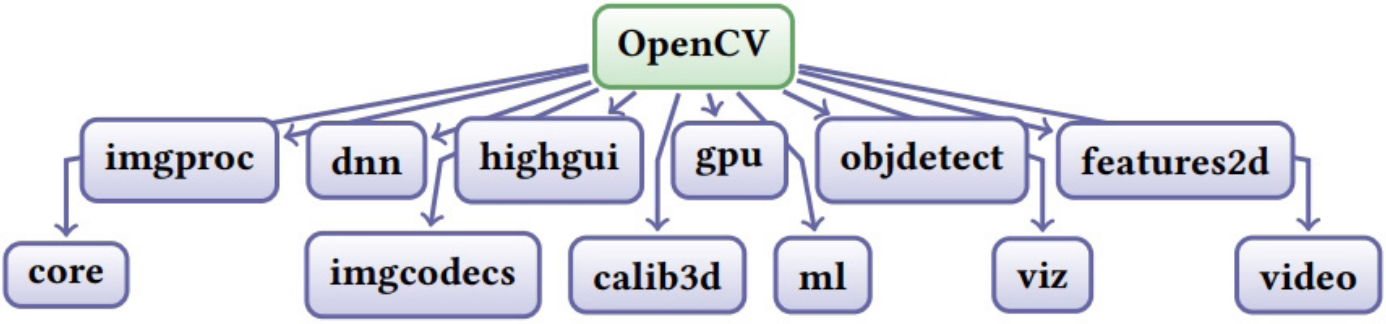
C++ などのネイティブ言語で開発されたコンピューター・ビジョン・ライブラリーはいくつかありますが、セキュリティーと移植性の問題を引き起こす可能性がある安定性の低いブラウザー拡張機能なしではブラウザーで使用することはできません。JavaScript* でコンピューター・ビジョン・ライブラリーを開発する取り組みもありますが、それらはビジョン関数の特定のカテゴリーに限定されています。それらを新しいアルゴリズムで拡張し実装を最適化することは、容易な作業ではありません。これまでに行われた作業では、機能、パフォーマンス、移植性のいずれかが欠けていました。

別のアプローチとして、ここでは C++ で開発された既存の包括的なコンピューター・ビジョン・ライブラリー (OpenCV*) を Web に対応させます。このアプローチは、いくつかの理由から Web に最適です。

- 広範な関数の最適化された実装を**提供**します。
- 通常の JavaScript* 実装よりも効率良く**動作**し、並列化によりパフォーマンスをさらに向上できます。
- チュートリアルやサンプルなど、充実した既存のリソースを**開発者が利用**できます。

OpenCV.js

OpenCV*[1] は、コンピューター・ビジョン開発向けの標準ライブラリーです。2000 年にインテル ラボで開発が開始されたオープンソース・ライブラリーであり、モジュールのセットとして実装されています (**図 1**)。画像処理、物体検出、追跡から、マシンラーニング、ディープ・ニューラル・ネットワークまで、広範な多数のプリミティブ・カーネルとビジョン・アプリケーションを提供します。OpenCV* は、ベクトルユニットを備えたマルチコア・プロセッサーなどの並列ハードウェア向けに効率良い実装を提供します。我々は、多くの OpenCV* 機能を JavaScript* に変換して、OpenCV.js と名付けました。



1 **モジュールのコレクションとしての OpenCV* 実装**

表 1 は、現在 OpenCV.js に含まれているカテゴリーと関数のリストです。2 つの理由から、いくつかの OpenCV* モジュールが省かれています。

1. **すべての OpenCV* モジュールが Web に適しているわけではありません。**例えば、カメラなどのメディアデバイスやグラフィカル・ユーザー・インターフェイスにアクセスする関数を提供する高度な GUI と I/O モジュール (**`highgui`**) は、プラットフォーム依存であり、Web 向けに変換できません。そのような関数では、代わりに JavaScript* モジュール (**`utils.js`**) で提供される HTML5 プリミティブを使用します。例えば、Web 上でホストされているファイルや getUserMedia を利用してメディアデバイスにアクセスし、HTML canvas でグラフィックスを表示する場合、HTML5 プリミティブを使用します。
2. **OpenCV* 関数の一部は、特定のアプリケーション・ドメインでのみ使用されており、通常 の Web プリケーションでは一般的ではありません。**例えば、ロボット工学では、カメラ・キャリブレーション・モジュール (**`calib3d`**) がよく使用されます。一般的な用途向けに生成されるライブラリーのサイズを減らすため、OpenCV* コミュニティーのフィードバックを基に、最も使用頻度の低い OpenCV* 関数をライブラリーの JavaScript* バージョンから除外しました。

表 1. OpenCV.js で提供される機能

モジュール	提供される関数
コア・コンポーネント	画像操作と基本的な演算
画像処理	画像の処理と解析を行う多数の関数
画像処理	追跡、バックグラウンド・セグメンテーション、オプティカル・フローなどのビデオ処理アルゴリズム
物体検出	Haar および HOG カスケード分類器
DNN	訓練済み Caffe*、Torch、TensorFlow* モデルの推論
GUI 機能	HTML canvas、ビデオ要素、カメラからのフレームにアクセスするヘルパー関数

特殊用途に役立つ関数が多数あるため、ユーザーが関数を選択してライブラリーをビルドできるようにしました。

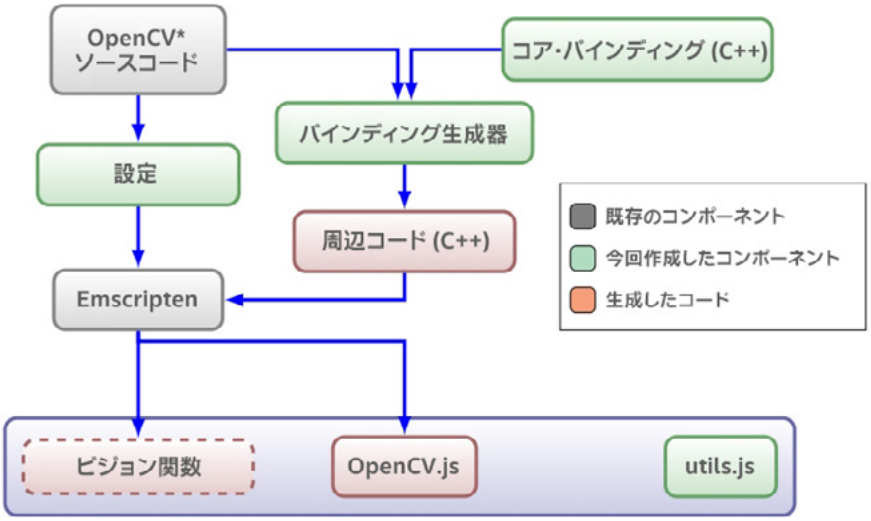
OpenCV* を JavaScript* と WebAssembly に変換する

Mozilla* により開発された LLVM ベースのソースツーソースのコンパイラーである Emscripten[2] の登場により、C++ で開発された多くのプログラムとライブラリーを Web に移植できるようになりました。元々、Emscripten は、asm.js と呼ばれる JavaScript* の型指定されたサブセットをターゲットにしています。asm.js は単純であるため、JavaScript* エンジンは多くの最適化も適用できます。さらに asm.js 関数は、実行前にコンパイルすることも可能です。

素晴らしいパフォーマンスが得られますが、大きな JavaScript* ファイルの解析とコンパイルは、低性能のプロセッサーを搭載したモバイルデバイスでは、ボトルネックとなることがあります。これは、WASM[3] 開発の動機の 1 つでした。WASM は、Web コンパイル向けに設計された、移植性が高くサイズとロード時間の効率が良いバイナリー形式です。我々は、Emscripten を使用して OpenCV* ソースコードを asm.js と WASM の両方にコンパイルしました。どちらも同じ機能を提供し、同じように使用できます。

Emscripten のコンパイル中に、クラスや関数識別子などの C++ 高水準言語に関する情報はマングル名に置き換えられます。マングル名を使用してプログラムを開発するのはほぼ不可能なため、関数やクラスなど異なる OpenCV* エンティティーのバインディング情報を提供し、JavaScript* で利用できるようにします。これにより、ライブラリーは、プログラマーが慣れ親しんだ通常の OpenCV* に類似したインターフェイスを提供できます。OpenCV* は大規模で、今後も継続的に拡張されるため、移植を手動で更新し続けることは現実的ではありません。そこで、高品質な効率良いコードを生成するため、エキスパートの見識を取り入れつつ、単純な変換プロセス作業を半自動化するアプローチを採用しました。

図 2 は、OpenCV* の C++ コードを JavaScript* に変換するステップを示しています。最初に、OpenCV* ソースコードでプラットフォーム固有または Web 向けに最適化されないコンポーネントと実装を無効にします。次に、JavaScript* にエクスポートすべきクラスと関数の情報を OpenCV* ソースコードから取得します。ここでは、最終的な JavaScript* ビルドに含める OpenCV* クラスと関数のホワイトリストを使用しています。ビルド前にリストを更新して、OpenCV* モジュールや関数を含める / 除外することが可能です。効率上の理由から、OpenCV* のメインデータ構造 (**`cv::Mat`**) を含む OpenCV* コアモジュールのバインディング情報は、手動で提供します。バインディング情報と関数のホワイトリストを使用して、JavaScript* シンボルを C++ シンボルにマップし、それを残りの OpenCV* ライブラリーとともに Emscripten で JavaScript* にコンパイルする周辺コードを生成します。このプロセスは、OpenCV* 関数を実装する WASM または asm.js ファイルとともにライブラリー・インターフェイスとして機能する JavaScript* ファイル (OpenCV.js) を出力します。GUI、I/O、ユーティリティー関数を含む utils.js も OpenCV.js とリンクされます。



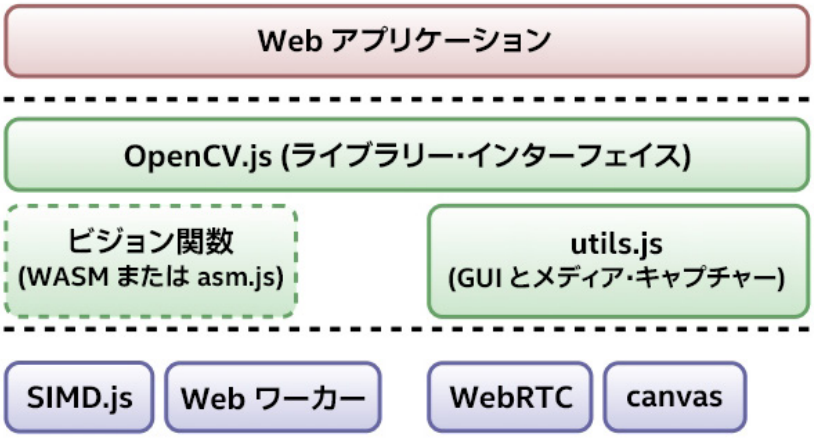
2 **OpenCV.js の生成ステップ**

Web アプリケーションで OpenCV.js を使用する

OpenCV.js を使用して Web アプリケーションを開発する方法を見てみましょう。**図 3** は、OpenCV.js の概要と Web アプリケーションとの関係を示しています。Web アプリケーションは、OpenCV.js API を使用して表 1 の関数にアクセスします。OpenCV* のビジョン関数は WASM または asm.js にコンパイルされますが、GUI 機能とメディア・キャプチャーは実装した JavaScript* モジュールによって提供されます。OpenCV.js は、Web ワーカーや SIMD.js などの標準 Web API を利用して高いパフォーマンスを達成し、canvas や WebRTC を利用してメディアおよび GUI 機能を提供します。

OpenCV.js API は、OpenCV* C++ API からインスピレーションを得ており、多くの類似点があります。例えば、C++ 関数は、同じ名前とシグネチャーで JavaScript* にエクスポートされます。関数の多重定義とデフォルト・パラメーターは、JavaScript* バージョンでもサポートされます。そのため、C++ で OpenCV* 開発を行っているユーザーは、簡単に JavaScript* へ移植できます。

OpenCV* C++ クラスは同じメンバー関数とプロパティーで JavaScript* オブジェクトに移植されますが、基本データ型は異なります。**表 2** は、C++ 基本データ型に対応する JavaScript* データ型です。JavaScript* エンジンは、ガベージコレクター (GC) を使用してプログラムメモリーを管理します。しかし、動的な GC の動作はパフォーマンスを低下させるため、OpenCV.js は静的メモリー管理を使用します。そのため、プログラマーは、OpenCV.js オブジェクトが不要になったら解放する責任があります。手動でのメモリー管理は面倒であるため、**`cv::Point`** などのプリミティブな OpenCV* データ型に JavaScript* データ型を使用しています。**`std::vectors`** は、**`cv::Mat`** のベクトルを除いて、すべて JavaScript* 配列に変換されます。これは、ベクトルを削除するとすべての **`cv::Mat`** 要素が削除されるため便利です。



3 **OpenCV.js コンポーネントとアプリケーションおよび Web API との関係**

表 2. C++ 基本データ型に対してエクスポートされる JavaScript* データ型

C++ データ型	JavaScript* データ型
数値型 (例 : int や float)	Number
bool	Boolean
enum	Constant
プリミティブ構造 (例 : cv::Point)	Value オブジェクト
(cv::Mats の) std::vector	cv.Vector
(プリミティブ型の) std::vector	Array
std::string	String

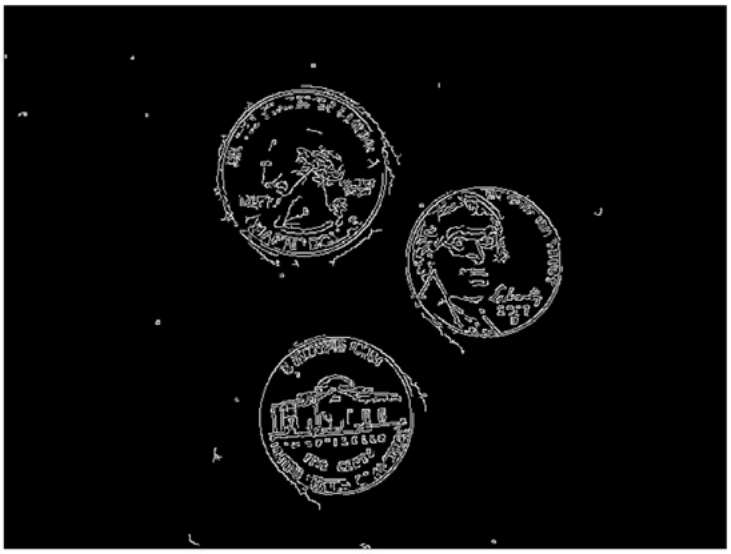
OpenCV.js を使用してコンピューター・ビジョン・タスクを実行するいくつかの例を紹介します。これらの例は、単純な HTML Web ページ上で動作します。ここでは、OpenCV.js API を使用するプログラムのロジック部分のみ紹介します。

図 4 は、Canny アルゴリズムを使用して画像のエッジを検出する方法を示しています。入力画像は、HTML canvas からロードされます。そのため、canvas 名を受け取ってカラー画像を返すヘルパー関数を作成しました。Canny アルゴリズムはグレースケール画像を扱うため、行 3 の **`cv.cvtColor`** を呼び出して入力画像をカラーからグレースケールに変換するステップが必要になります。最後に、Canny アルゴリズムの結果を取得して、出力用の canvas に画像をレンダリングします (行 6)。**図 5** は、ブラウザー内で実行されているこのプログラムのスナップショットです。

```
1   let src = cv.imread ('canvasInput');
2   let dst = new cv.Mat ();
3   cv.cvtColor (src, src, cv.COLOR_RGB2GRAY, 0);
4   // 異なるパラメーターを試すことができる
5   cv.Canny (src, dst, 50, 100, 3, false);
6   cv.imshow ('canvasOutput', dst );
7   src.delete (); dst.delete ();
```

4 **Canny アルゴリズムにより画像中のエッジを検出する例**

canvasInput Browse... coins.jpg



canvasOutput

5 **行 6 で画像をレンダリングする**

次の例 (**図 6** と **7**) は、Haar カスケードを使用して画像から顔を検出します。このアルゴリズムはグレースケール画像を扱うため、行 3 で入力画像を変換しています。行 7 でカスケード分類器を初期化して、顔検出モデルをロードします。猫や犬など、ほかの物体を検出するように訓練されたモデルを使用することもできます。行 9 で **`detectMultiscale`** を呼び出します。これは、異なるサイズでスケーリングされた入力画像の複数のコピーを検索して、処理が完了すると、画像中の顔の可能性がある四角のリストを返します。行 10 でこの四角を反復して、**`cv.rectangle`** 関数を使用して画像のその部分をハイライトします。

6 **カスケード分類器を使用した顔検出**

これで、OpenCV.js を使用して Web アプリケーションで単一イメージを処理する方法が分かりました。ビデオ処理は、連続した個々のフレーム処理です。次の例 (**図 8** と **9**) では、以下の操作方法を示します。

- ビデオ要素からフレームを**キャプチャーする方法**
- MOG2 アルゴリズムを使用して入力ビデオから背景を**除去する方法**
- 処理済みフレームを HTML canvas に**表示する方法**

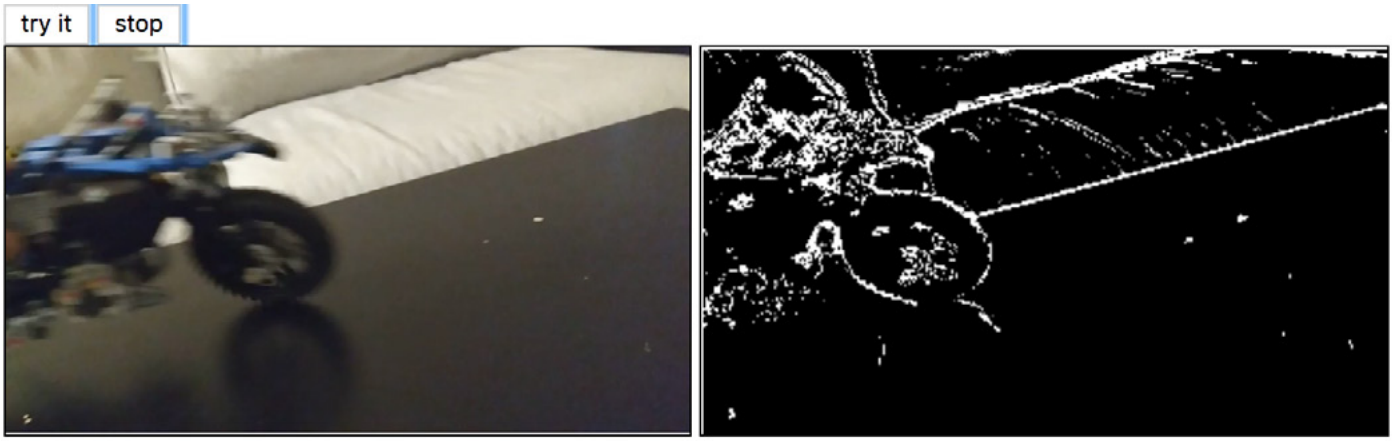
utils.js で提供される **`cv.VideoCapture`** オブジェクトは、WebRTC がカメラリソースにアクセスして管理できるようにします。この例では、入力ビデオは 30fps と仮定します。つまり、1/30 秒ごとに **`processVideo`** 関数が呼び出されます。この関数は、次の処理を行います。

- 次のビデオフレームの**読み取り** (行 18)
- 背景抽出関数の**適用** (行 19)
- 出力の前景マスクの**表示** (行 20)

最後に、行 23 で関数の次の呼び出しをスケジュールします。

```
let src = cv.imread ('canvasInput');
let gray = new cv.Mat();
cv.cvtColor (src, gray, cv.COLOR_RGBA2GRAY, 0);
let faces = new cv.RectVector ();
let faceCascade = new cv.CascadeClassifier ();
// 事前訓練済みの分類器をロードする
faceCascade.load ('haarcascade_frontalface_default.xml');
// 顔を検出する
faceCascade.detectMultiScale (gray, faces);
for (let i = 0; i < faces.size(); ++i) {
  let roiGray = gray.roi (faces.get(i));
  let roiSrc = src.roi (faces.get(i));
  let point1 = new cv.Point (faces.get(i).x, faces.get(i).y);
  let point2 = new cv.Point (faces.get(i).x + faces.get(i).width,
      faces.get(i).y + faces.get(i).height);
   cv.rectangle (src, point1, point2, [255, 0, 0, 255], 3);
   roiGray.delete (); roiSrc.delete ();
}
cv.imshow ('canvasOutput', src);
src.delete (); gray.delete (); faceCascade.delete ();
faces.delete ();
```

7 Haar カスケードを使用して画像中から顔検出を行う例



8 MOG2 メソッドを使用して入力ビデオの異なるフレームから背景を除去する

```
1   let video = document.getElementById ('videoInput');
2   let cap = new cv.VideoCapture (video);
3
4   let frame = new cv.Mat (video.height, video.width, cv.CV_8UC4);
5   let fgmask = new cv.Mat (video.height, video.width, cv.CV_8UC1);
6   let fgbg = new cv.BackgroundSubtractorMOG2 (500, 16, true);
7
8   const FPS = 30;
9   function processVideo () {
10     try {
11        if (! streaming) {
12           // 消去して停止する
13           frame.delete (); fgmask.delete (); fgbg.delete ();
14           return;
15        }
16        let begin = Date.now ();
17        // 処理を開始する
18        cap.read (frame);
19        fgbg.apply (frame, fgmask);
20       cv.imshow ('canvasOutput', fgmask);
21       // 次のフレームをスケジュールする
22       let delay = 1000/FPS - (Date.now () - begin);
23       setTimeout (processVideo, delay);
24     } catch (err) {
25        utils.printError (err);
26     }
27  };
28  // 最初のフレームをスケジュールする
29  setTimeout (processVideo, 0);
```

9 ビデオフレームをキャプチャーして背景を除去する

最後の例 (**図 10** と **11**) は、Web アプリケーションで事前訓練済み DNN を使用します。この例では、DNN を物体認識に使用していますが、バックグラウンド・セグメンテーションなどのほかの認識タスクを実行することもできます。行 2 でプログラムは、GoogLeNet の Caffe* フレームワーク・モデルを読み取ります。Torch や TensorFlow* など、ほかの形式もサポートされています。行 6 の次のステップでは、入力画像をネットワークに適合するブロブに変換します。そして、行 9 と 10 でブロブをネットワークに転送して、最も高い確率のクラスを見つけます。**図 10** は、サンプル画像を分類するプログラムのスナップショットです。(このほかの OpenCV.js の使用例は、**https://docs.opencv.org** (英語) を参照してください。)

Best class: #812 space shuttle **Probability**: 0.9910982251167297 **Elapsed time**: 1174 ms

canvasInput Browse... shuttle-s.jpg

10 **GoogLeNet モデルを使用した物体検出の例**

```
let src = cv.imread ('canvasInput');
let net = cv.readNetFromCaffe ('bvlc_googlenet.prototxt',
                               'bvlc_googlenet.caffemodel');
if (net.empty ());
   throw "Failed to read net";
let inputBlob = cv.blobFromImage (src, 1, new cv.Size (224, 224),
                                  new cv.Scalar (104, 117, 123));

net.setInput (inputBlob, data);
let prob = net.forward ("prob");
let minMax = cv.minMaxLoc (prob);

console.log ("Best class: #" + minMax.maxLoc.x + " " +
      keywords [minMax.maxLoc.x] + "Probability: " + minMax.maxVal);

prob.delete (); inputBlob.delete ();
net.delete (); src.delete ();
```

11 Web アプリケーションで事前定義済み DNN を使用する

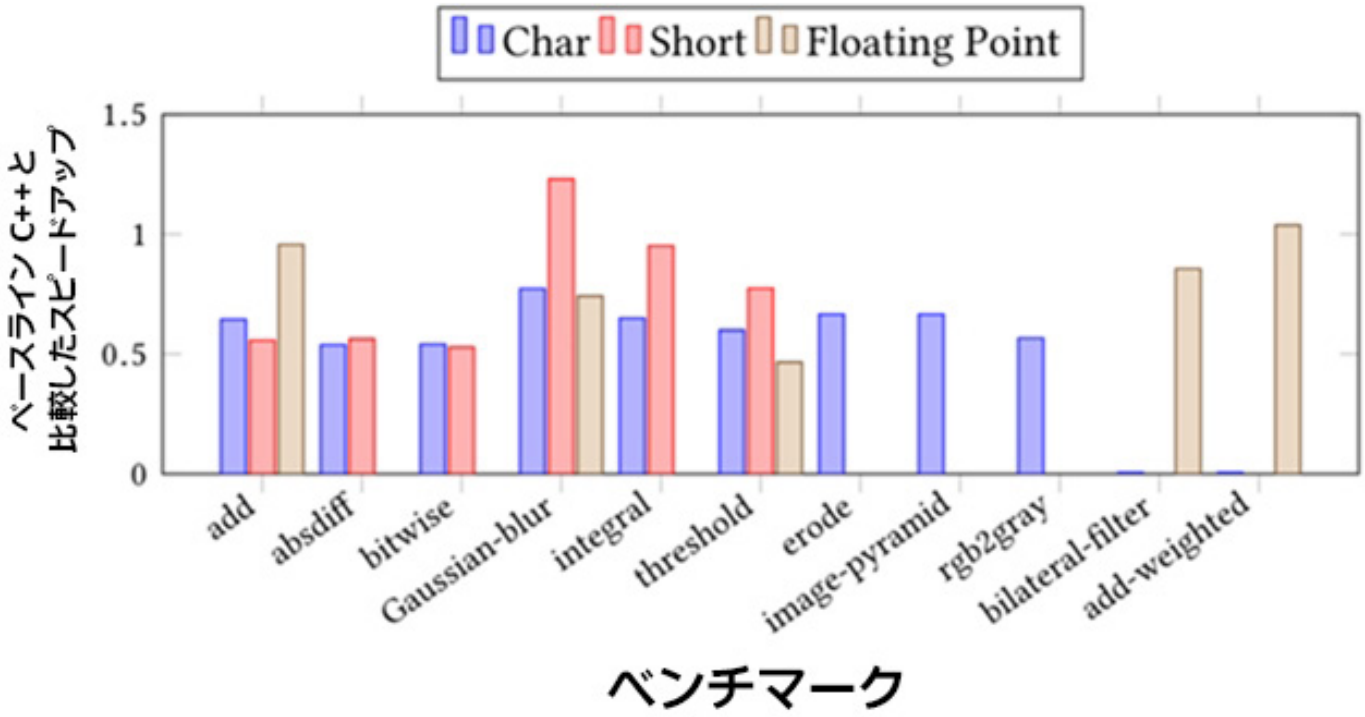
パフォーマンスの評価

OpenCV.js は、多くのコンピューター・ビジョン機能を Web にもたらします。OpenCV.js のパフォーマンスを評価するため、ここでは、プリミティブ・カーネルと洗練されたビジョン・アプリケーションを含むいくつかのビジョン・ベンチマークを選択しました。

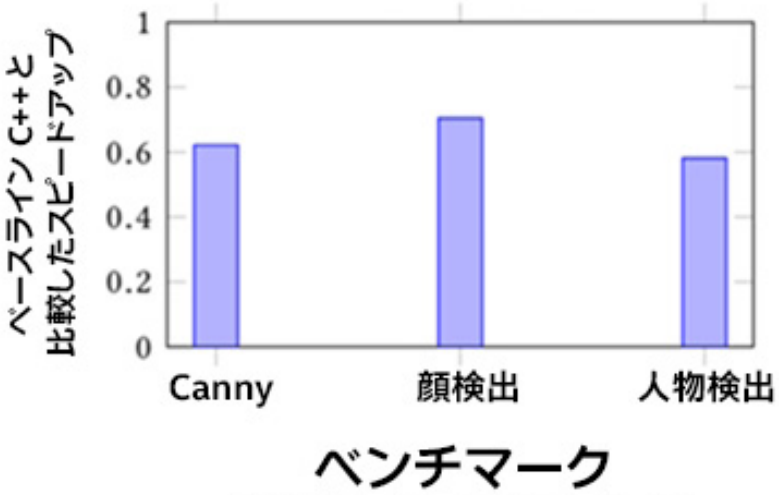
- エッジ検出を行う **Canny** アルゴリズム
- Haar カスケードを使用する**顔検出**
- 勾配ヒストグラムを使用する**人物検出**

一連のビデオの設定とテストは、**インテル® Core™ i7-3770 プロセッサー**、8GB RAM 搭載の Ubuntu* 16.04 ベースのシステムで Firefox* ブラウザーを使用して行いました。

図 12 は、OpenCV* スカラービルドを使用するネイティブバージョン (並列処理を使用しない) と比較した、単純な JavaScript* カーネルとビジョン・アプリケーションの平均スピードアップです。JavaScript* のパフォーマンスが良いことが分かります。WASM と asm.js のパフォーマンスはほぼ同じですが、ライブラリーの WASM バージョンは、初期化が非常に高速で、かつコンパクトです (WASM のサイズは 5.3MB、asm.js は 10.4MB)。



(a) 整数および浮動小数点演算のプリミティブ・カーネル



(b) ビジョン・アプリケーション

12 **Firefox* 上でのプリミティブ・カーネルとビジョン・アプリケーションのパフォーマンス**

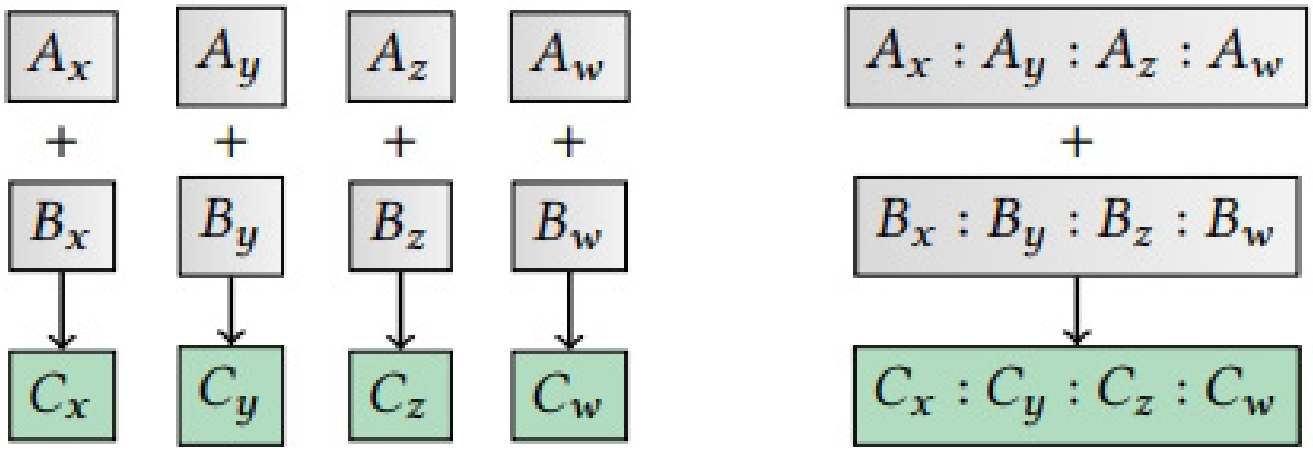
さらなる高速化

コンピューター・ビジョンは、計算要件が厳しく、膨大な数のピクセルに対して多くの計算を行う必要があります。例えば、ベースラインの Canny、顔検出、人物検出ベンチマーク・タスクの各反復は、単一の 800 x 600 解像度の画像の処理に、それぞれ平均 7 ミリ秒、345 ミリ秒、323 ミリ秒かかります。Canny、顔検出、人物検出の計算は高速ですが、リアルタイムのインタラクティブなユースケースにはまだ十分ではありません。

幸いにも、コンピューター・ビジョン・アルゴリズムは、本質的に並列性を備えており、優れたアルゴリズムと最適化された実装により、並列ハードウェアにおいて大幅なスピードアップが得られます。OpenCV* では、異なるアーキテクチャー向けにアルゴリズムの並列実装が用意されています。ここでは、マルチコア・プロセッサーと SIMD ユニットをターゲットとする 2 つのメソッドを利用して、JavaScript* バージョンを高速化します。GPU 実装は複雑なためここでは省略します。GPU 上では、今後リリースされる WebGPU API を利用して OpenCV* を高速化できる可能性があります。

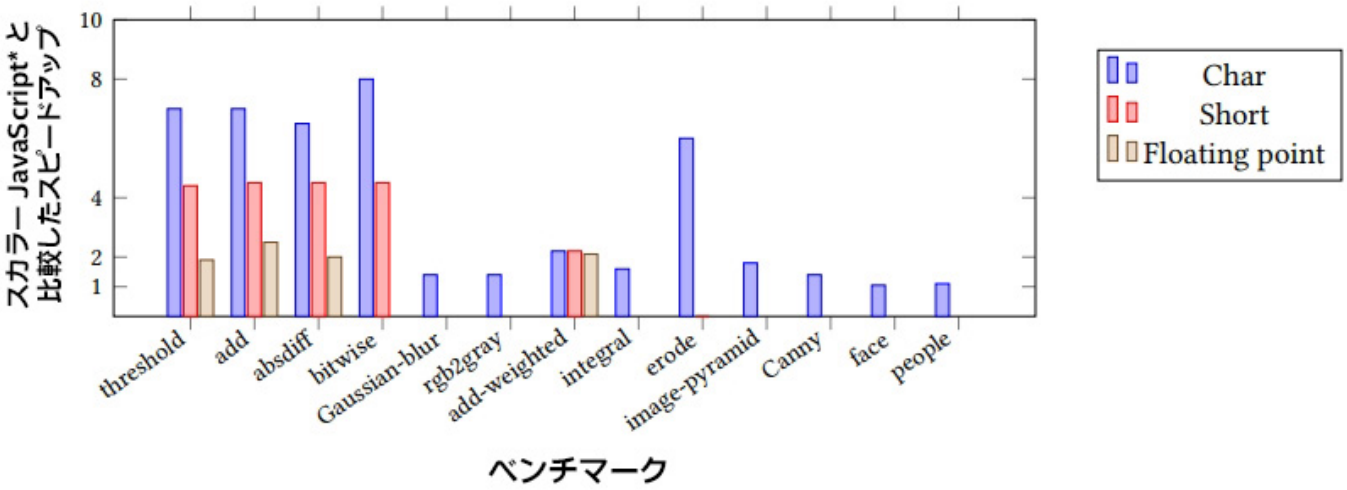
SIMD.js

SIMD.js[4, 5] は、プロセッサーのベクトル機能を Web で利用できるようにする新しい Web API です。インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2) と ARM* NEON* 命令セットのどちらのアーキテクチャーでも効率良く動作する共通のサブセットをベースにしています。128 ビットのベクトルレジスター (4 つの整数、4 つの単精度浮動小数点数、または 16 文字を保持できる) を操作するベクトル命令を定義します。**図 13** は、ベクトルレジスターを使用して単一の CPU 命令で 4 つの整数を加算する方法を示しています。



13 スカラーと SIMD の 4 つの整数の加算処理

SIMD は、特にマルチメディア、グラフィックス、科学アプリケーションのパフォーマンスの向上に効果があります[5, 6]。実際、コアルーチンなどの多くの OpenCV* 関数がベクトル組込み関数を使用して実装されています。ここでは、Peter Jensen、Ivan Jibaja、Ningxin Hu、Dan Gohman、John Mc-Cutchan[7] の取り組みを応用して、インテル® SSE2 を使用する OpenCV* のベクトル化された実装を SIMD.js 命令を使用する JavaScript* に変換しました。SIMD.js 実装の命令は、ライブラリー・インターフェイスに影響しません。**図 14** は、特定のカーネルとアプリケーションを Firefox* で実行した場合に、SIMD.js によってもたらされるスピードアップを示しています。プリミティブ・カーネルでは最大 8 倍のスピードアップを達成しています。予想どおり、小さなデータ型のほうが大幅なスピードアップが得られています。Canny、顔検出、人物検出などの複雑な関数では、ベクトル化の可能性が低くなります。現在、SIMD.js は asm.js コンテキストでのみ利用することができ、Firefox* と Microsoft Edge* ブラウザーでサポートされています。WebAssembly の SIMD は、現時点では SIMD.js と同じ仕様になる予定です。そのため、同様のパフォーマンス結果が期待されます。



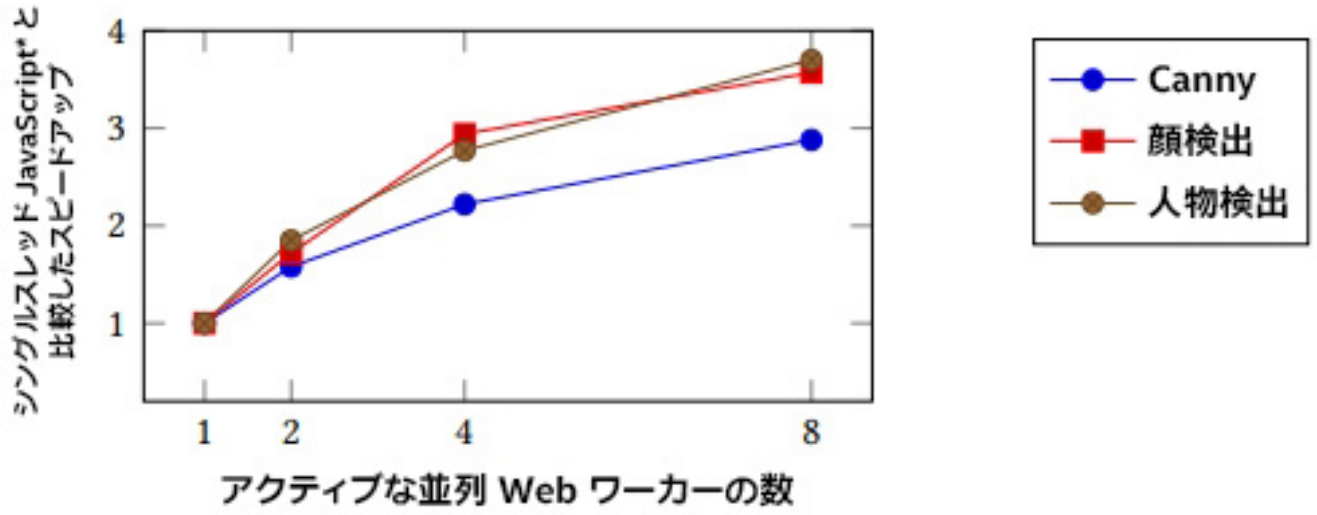
14 **SIMD.js によるパフォーマンスの向上**

Web ワーカーを使用したマルチスレッド化

JavaScript* プログラムは、計算負荷の高いタスクの並列処理に Web ワーカーを使用します。Web ワーカーは、メッセージパッシングを介して通信するため、特に画像などの大きなメッセージを送信する場合、大きなコストが発生する可能性があります。Web ワーカー間で共有可能なストレージシステムとして、SharedArrayBuffer[8] が最近提案されており、共有メモリー並列プログラミングを実装できます。OpenCV* は、**`parallel_for_`** フレームワークを使用して、ビジョン関数のマルチスレッドバージョンを実装しています。また、**`parallel_for_`** フレームワークは、POSIX* スレッド (Pthreads) を含む複数のマルチスレッド化モデルをターゲットにすることができます。Emscripten* の登場により、Pthreads API を Web ワーカーと共有配列バッファーを使用する等価な

JavaScript* へ変換することが可能です。マルチスレッド化をサポートする OpenCV.js は、Web ワーカーのプールを使用して、新しいスレッドがスポーンされるとワーカーを割り当てます。さらに、OpenCV* API が同時スレッド数などの JavaScript* への同時性を動的に調整できるようにします。

複数の Web ワーカーによるパフォーマンスへの影響を検証するため、SIMD ベクトル化では有意なパフォーマンス向上が見られなかった 3 つのアプリケーション・ベンチマークについて、異なるワーカー数 (最大 8) で測定しました。OpenCV* のロードバランス・アルゴリズムは、ワークロードをスレッド間で均等に分割します。**図 15** に示すように、8 つの論理コアを持つプロセッサーでは、3 ~ 4 倍のスピードアップを得ることができました。同様の傾向は、同じ関数のネイティブの Pthreads 実装でも見られました。



15 **複数の Web ワーカーの使用によるパフォーマンス**

コンピューター・ビジョンの普及

ここで紹介した作業により、コンピューター・ビジョン処理における長年の OpenCV* 開発を効率良く Web で利用できるようになります。OpenCV.js は、画像処理、物体検出、ビデオ解析、特徴抽出、DNN などを含む関数のコレクションを提供します。我々のテスト結果は、これらのフレームワークの高い能力を示しています。JavaScript* の高い移植性により、Web ブラウザー、そして組込みデバイスやデスクトップ・アプリケーションで多くのビジョン関数の利用が可能になります。例えば、Node.js ベースの IoT デバイスや Electron などの JavaScript* デスクトップ開発フレームワークでコンピューター・ビジョンを利用できます。

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。

Web プラットフォームの最近の発展と組み合わせることで、仮想現実 (VR) や拡張現実 (AR) などの新しい Web アプリケーションや体験を効率良く開発できます。また、OpenCV.js を使用した多くのコンピューター・ビジョン・チュートリアルも用意しています。これらのチュートリアルが教育や研究に役立つことを願っています。

最後に、Google* Summer of Code プログラムを介して貢献してくれた Congxiang Pan、Gang Song、Wenyao Gan、そして有益なフィードバックを提供してくれた OpenCV* 設立者の Gary Bradski 博士、同チーフ・アーキテクトの Vadim Pisarevsky および Alex Alekin に感謝します。

関連資料

開発者と研究者向けに OpenCV.js とコンピューター・ビジョン全般に関する情報収集を支援するため、幅広いオンラインリソースを用意しています。

- **OpenCV.js ドキュメントとチュートリアル** (英語)
- **OpenCV.js デモ** (英語)

OpenCV.js は、Node.js ベースの環境でも使用できます。Node Package Manager (NPM) (**https://www.npmjs.com/package/opencv.js** (英語)) から入手できます。

参考資料

1. Gary Bradski and Adrian Kaehler, "Learning OpenCV: Computer vision with the OpenCV library," O'Reilly Media, Inc., 2008.
2. Alon Zakai, "Emscripten: An LLVM-to-JavaScript compiler," in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pages 301–312. ACM, 2011.
3. Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien, "Bringing the Web up to speed with WebAssembly," in Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 185–200. ACM, 2017.
4. SIMD.js specification: **http://tc39.github.io/ecmascript_simd/** (英語). 2017.
5. Ivan Jibaja, Peter Jensen, Ningxin Hu, Mohammad R Haghighat, John McCutchan, Dan Gohman, Stephen M Blackburn, and Kathryn S McKinley, "Vector parallelism in JavaScript: Language and compiler support for SIMD," in Parallel Architecture and Compilation (PACT), 2015 International Conference on, pages 407–418. IEEE, 2015.
6. Sajjad Taheri, "Bringing the power of simd.js to gl-matrix": **https://hacks.mozilla.org/2015/12/bringing-the-power-of-simd-js-to-gl-matrix/** (英語), 2015.
7. Peter Jensen, Ivan Jibaja, Ningxin Hu, Dan Gohman, and John Mc-Cutchan, "SIMD in JavaScript via C++ and Emscripten," in Workshop on Programming Models for SIMD/Vector Processing, 2015.
8. ECMAScript 2018 language specification: **https://tc39.github.io/ecma262/** (英語), 2017.

ゲームをスピードアップ

インテル® Graphics Performance Analyzers を利用してゲーム開発のあらゆる作業を最適化する方法

Giselle Gomez 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

パフォーマンスの最適化は、良いゲームと素晴らしいゲームの違いを意味します。そして、スムーズなゲームプレイを提供することは困難な作業です。しかし、**インテル® Graphics Performance Analyzers (インテル® GPA)** (英語) を利用することで、プロセスを合理化して、開発チームの誰もが効率良く作業できるようになります。この記事では、最適化プロセスのどの位置にいるのかを正確に特定して、インテル® GPA ツールスイートにより、忠実度やフレームレートの向上などの最適化の目標を達成する方法を説明します。

最適化プロセスのどの位置にいるのか？

ゲーム開発者は、3 つのグループに分けることができます。

1. アーティスト / デザイナー
2. ゲームプレイ・プログラマー
3. ゲーム・エンジン・プログラマー

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。

それぞれのグループが重要な役割を果たします。そして、最適化プロセスにおけるそれぞれのグループの取り組みは異なります。その違いを理解するため、いくつかのユースケースを紹介します。

アーティスト / デザイナー

最初に、風景、オブジェクト、サーフェス・テクスチャー、乗り物など、ゲーム内の環境とキャラクターに関連するアセットを作成するアーティストから見ていきます。

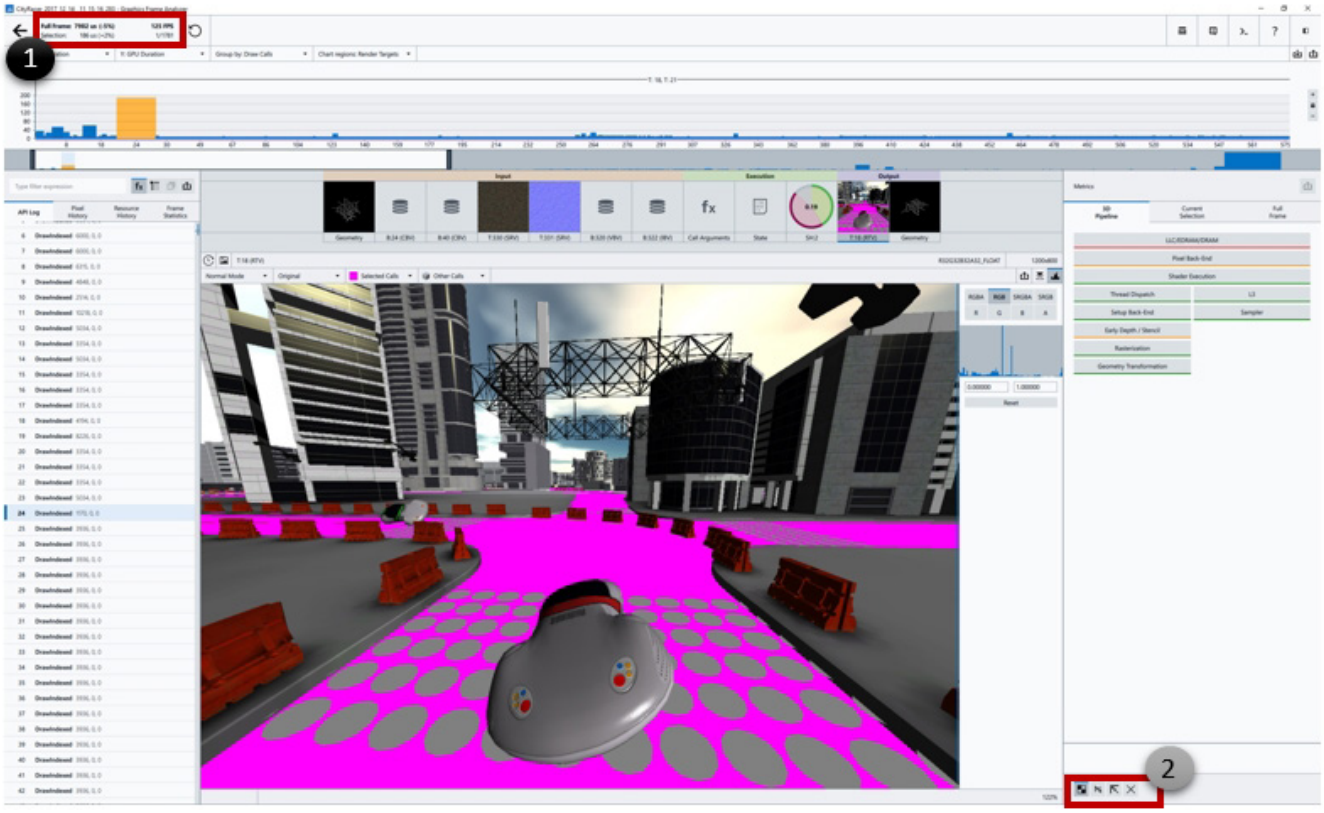
最高の視聴体験を作成するため、アーティストは通常、最高の忠実度 (4K テクスチャーやさらに複雑なジオメトリー) で要素を開発しますが、最適化により忠実度が下げられる可能性があります。ターゲット・ハードウェア上でスムーズなゲームプレイを実現するテクスチャーとアセットを作成することが、アーティストの役割です。また、ゲームプレイ・プログラマーと協力して、ライティング、動作、全体のデザインに関連したデザインを行います。

新しい最小要件向けの再調整や、不適切なフレームレートのため最適化を行う際に、次のような課題に直面する可能性があります。

- **テクスチャー帯域幅 :** 大きすぎる、または解像度が高すぎるテクスチャー
- **ジオメトリーの複雑さ :** 非常に小さなポリゴン

インテル® GPA などの最適化ツールは、テクスチャーやアセットの追加によってフレームレートが目標値以下に低下したり、ほかのゲームプレイの問題が発生しないかを、アーティストがセルフチェックできるようにします。

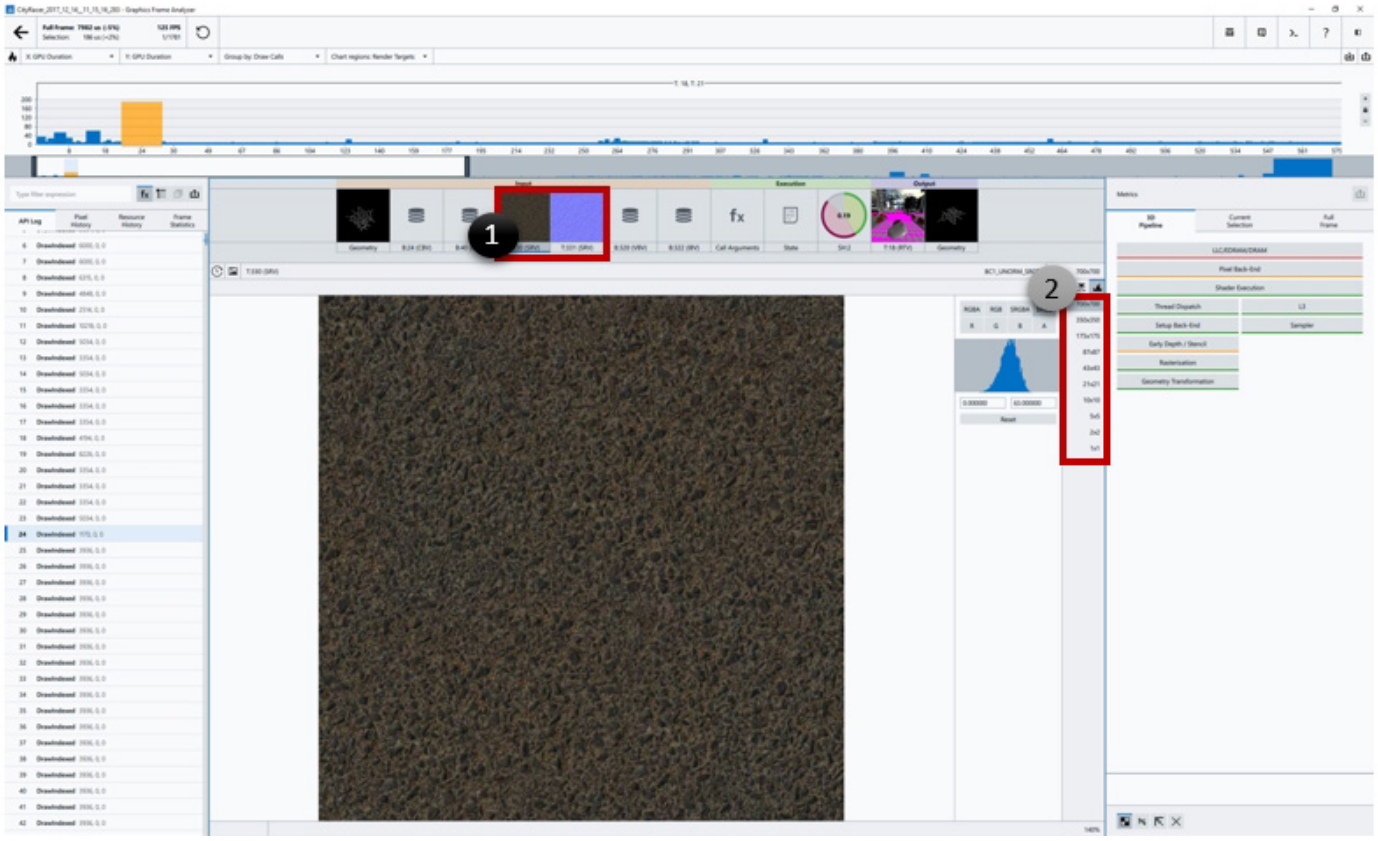
インテル® GPA の Graphics Monitor でフレームをキャプチャーして、Frame Analyzer でそのフレームをプロファイルすることで、アーティストは任意のシーンにおいて全体の影響を判断できます。Frame Analyzer では、アセットのジオメトリーに関連する適切なメトリック (レンダリングされたプリミティブの合計など) を取得したり、テクスチャーに関連する異なるミップマップ (MIP) レベルを選択してテクスチャーとアセットの影響を確認できます。"2*2 Texture" (**図 1**) のようなインタラクティブな実験を行うことで、テクスチャーやオブジェクトがゲーム・パフォーマンスを低下させるボトルネックとなるかどうか、そして解像度が低いテクスチャーを使用して MIP レベルを下げたり (**図 2**)、詳細レベル (LOD) を使用してカメラから遠ざかるオブジェクトの詳細レベルを下げることで改善されるかどうかを判断できます。アセットの作成とパフォーマンスの影響を検証するツールとしてインテル® GPA を使用することで、アセットの追加によるゲームプレイのクロスリファレンスを向上し、高いフレームレートと忠実度を維持することができます。



1 **右下の 2*2 テクスチャーの実験 (②) を使用することで、フレーム全体のパフォーマンスが 5% 向上します (①)。これは、選択された描画呼び出しに使用されるテクスチャーを表示して、最終的なシーンの品質に大きく影響しないことが確認できる場合、テクスチャーの MIP レベルを下げることで利点が得られる可能性があることを示しています。**

ゲームプレイ・プログラマー

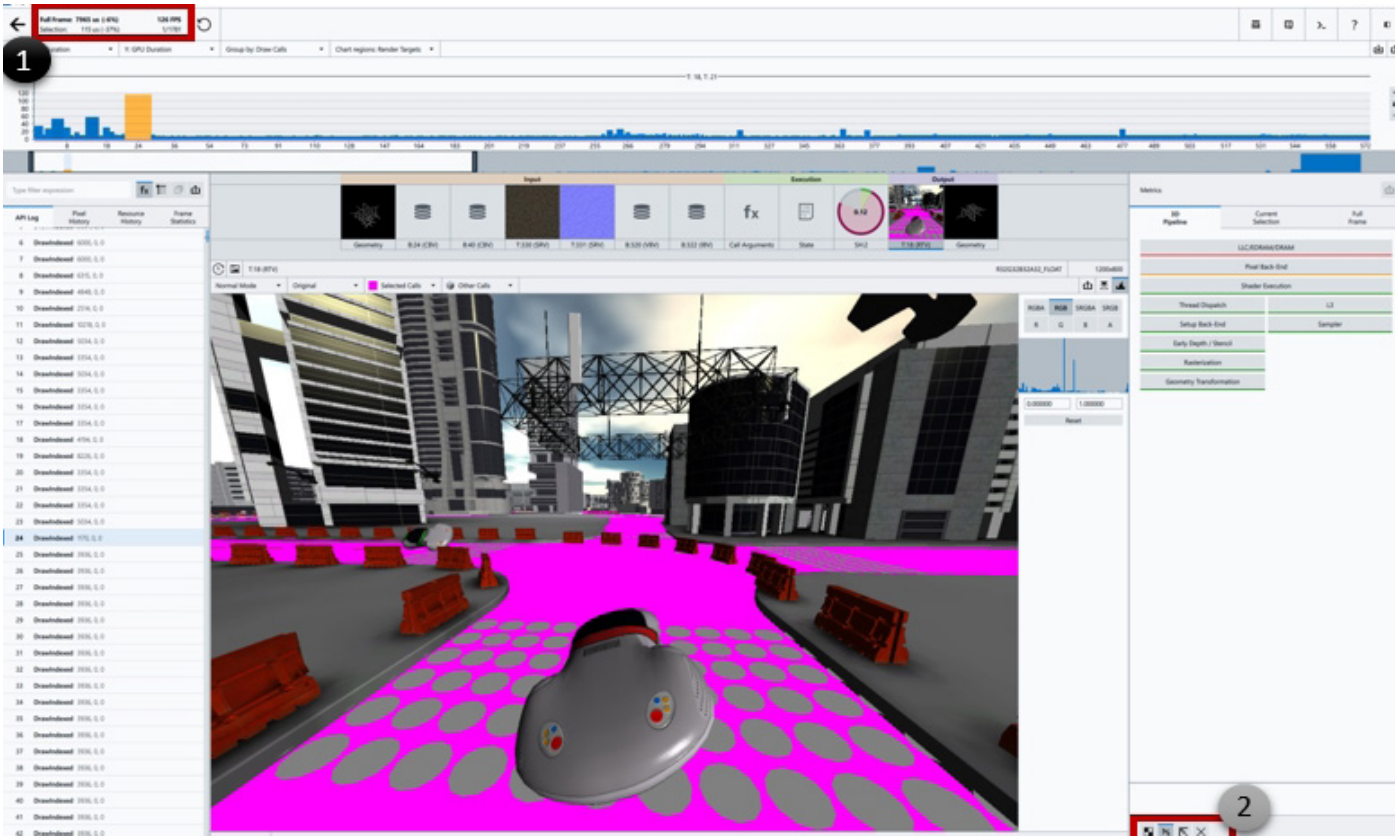
ゲームプレイ・プログラマーは、ゲーム・エンジン・プログラマーによって作成されたゲームに依存しないコンポーネントとアーティストによって作成されたアセットを組み合わせて、可能な限り最高のゲームプレイ体験を作成します。最適化では、アーティストのデザインに従う一方で、ユーザー・インターフェイス、キャラクターの動作、コントロール、オーディオなどのゲーム機能に注目します。



2 **リソースビューアー (①) から (入力と出力の) テクスチャーを選択して表示します。テクスチャーに異なる MIP が関連付けられている場合、リソースビューアーの左側に MIP レベルが表示されます。ここで、シーンや最終レンダーターゲットへのテクスチャーの影響を確認できます。**

最適化時にゲームプレイ・プログラマーは、アセットの基本レイヤーを適切に作成できるように、効率良いコードのパフォーマンスに注目します。効率的なバックエンド・コードのほうが、良い結果が得られます。ゲームプレイ・プログラマーにとってインテル® GPA を使用する主な利点は、ライブ編集と再生、CPU/GPU プロファイル、実験です (ただし、これらに限定されません)。

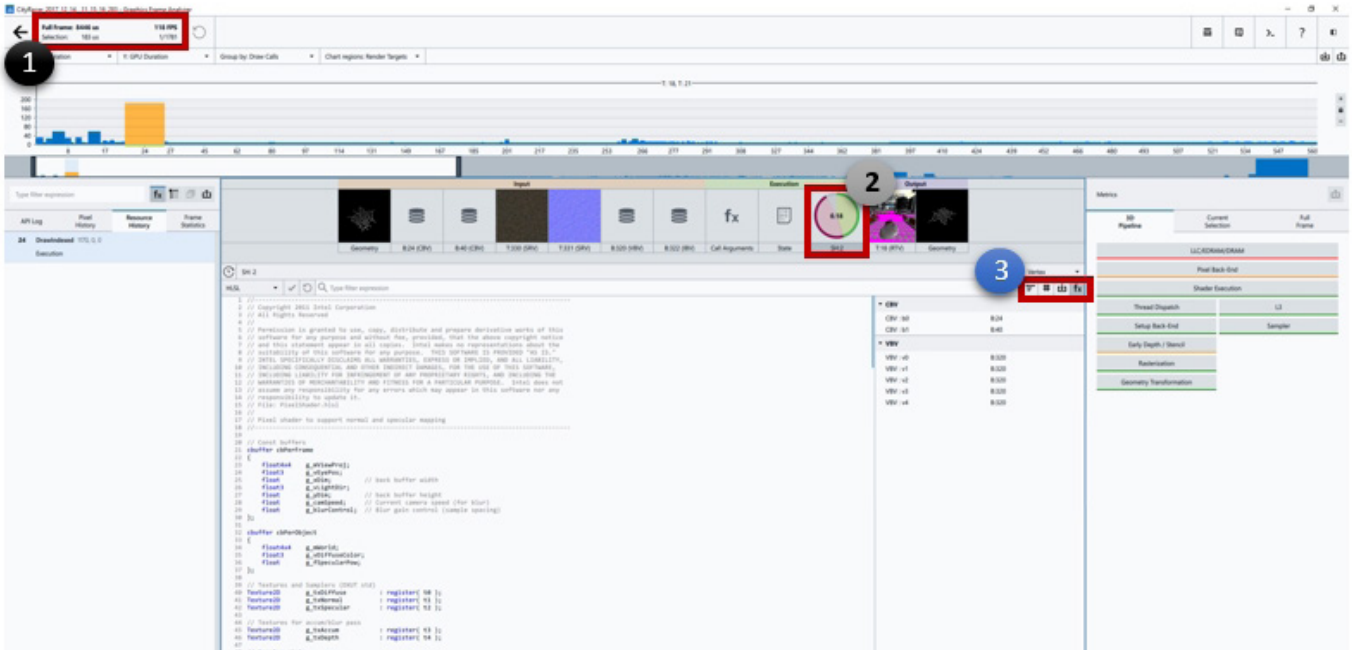
インテル® GPA は、コードを最適化する場所とタイミングを特定したり、レンダリング方法を最適化するオブジェクト (オブジェクトの数とバッチ) を特定する、さまざまな方法を提供します。ゲームプレイ・プログラマーが最適化できる一般的な領域の 1 つは、シェーダーです (スタジオの規模やチームにゲーム・エンジン・プログラマーがいるかどうかによって異なります)。Frame Analyzer で "Simple Pixel Shader" または "1*1 Scissor Rect" を有効にすると (**図 3**)、ボトルネックをピクセルシェーダーまたは頂点シェーダーのいずれかに絞り込むことができます。そして、ライブ再生と編集を使用してツール内でシェーダーを編集するか、最適化されたシェーダーをインポートして、変更の結果を確認できます (**図 4**)。選択したリソースを使用するすべてのイベントをリストする [Resource History] タブを利用して、シェーダーの依存関係も簡単に素早く特定できます。ゲームプレイ・プログラマーが注意すべきほかの領域には、次のものが含まれます。



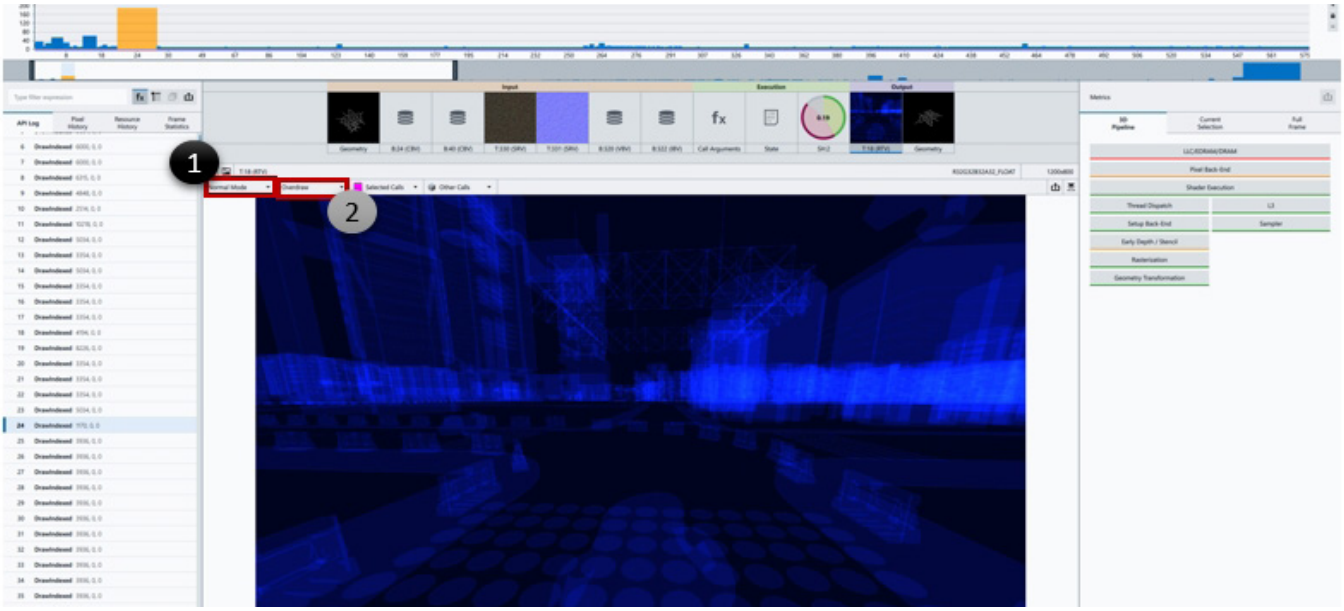
3 **Frame Analyzer の左下にある Simple Pixel Shader を有効にすると、フレーム全体のパフォーマンスが 6% (①) 向上します。シェーダーに最適化の余地があることを特定できたら、シェーダー・リソース・ビューでシェーダーをインポートまたは変更します (図 2)。**

- パーティクル・エフェクトの**レンダリング方法**
- **ライティング** (反射、光の数など)
- シーン内の**オブジェクトの配置方法**
- **描画呼び出し**のバッチ方法 (動的バッチまたは静的バッチ)

別の簡単な最適化として、API ログで重複する描画呼び出しを確認したり、棒グラフで余分なデプスパスなどの重複するパスを確認します。最適化時にゲームプレイ・プログラマーが最初に使用するツールは、Trace Analyzer です。ライブ解析中にトレースをキャプチャーすることで、CPU および GPU 使用率をプロファイルできます。トレースから 、CPU および GPU キュー、フレームのタイミング、VSYNC、処理時間が分かります。Trace Analyzer はカスタマイズ可能で、ユーザーはキャプチャー中に収集されるメトリックを変更することができます。待機が発生している場所、項目がキュー内にとどまっている時間、ワークロードの CPU と GPU への分散方法を確認することで、ゲームプレイ・プログラマーは最適化が必要な場所 (CPU プロセスまたは GPU プロセス) とチューニングすべき場所を特定できます。



4 リソースメニューの実行部分からシェーダービューを選択すると (②)、ローカルに保存されているシェーダーのライブ編集またはインポート (③) が可能になります。シェーダーを編集した後、Frame Analyzer は新しいシェーダーを使用してフレームを再生し、ウィンドウの右上にフレーム・パフォーマンスへの影響を示します (①)。



5 レンダーターゲットのオーバードロー・ビュー (ドロップダウン ②)。明るい色の領域は、オーバードローが多く存在します。オーバードローを作成するオブジェクトがシーンの以降の描画呼び出しにある場合、Z テストは実装されません。シーンの以降の呼び出しにあるか特定するには、Scrub Mode (ドロップダウン ①) を有効にして、オーバードローが消えるかどうか確認します。Scrub Mode は、現在選択されている描画呼び出し以前に描画された要素のみ表示します。Scrub Mode を有効にすると消える要素は、シーンの以降の描画呼び出しにあります。

ゲーム・エンジン・プログラマー

最後に、ゲーム・エンジン・プログラマーは、ゲームプレイ・プログラマーの作業に必要なゲームを作成する際の基盤となる、ゲームに依存しないコンポーネントを作成します。これには、次のものが含まれます。

- アニメーションの実現方法の**実装**
- ゲームプレイの背後にある**物理特性**
- **衝突**判定
- スクリーンに**ピクセルをレンダリングする方法**

ゲームプレイ・プログラマーのセクションで前述したように、ゲーム・エンジン・プログラマーがいない場合、ゲームプレイ・プログラマーがシェーダーを最適化します。しかし、ゲーム・エンジン・プログラマーがいる場合、シェーダーの最適化 (頂点、ピクセル、計算など) はゲーム・エンジン・プログラマーが行います。Frame Analyzer のオーバードロー・ビューを使用して、Z テストや深度テストに関連する最適化を見つけることができます。レンダーターゲットのオーバードロー・ビュー (**図 5**) では、シーン内の近接するオブジェクトによって隠されるピクセルを、エンジンがどのように処理しているかを特定できます。シーン内で多くのオーバードローが行われている場合、さらに詳しく調査して、エンジン内で深度テストが使用されているかどうか特定すると良いでしょう。Z 除去を追加することで、表示されないピクセルが完全にレンダリングされる回数を制限し、シーンにおける合計描画呼び出し数を軽減できます。ゲーム・エンジン・プログラマーは、最適化時にレンダリング・パイプラインのほかの側面もカバーするとともに、Trace Analyzer を使用して CPU/GPU 使用率を調査し、AI アルゴリズムや物理特性計算の最適化など、CPU で必要な最適化を特定します。ゲーム・エンジン・プログラマーには、CPU パイプラインのボトルネックを特定する GPU パイプライン・ビューからバッファービューまで、インテル® GPA のすべての機能が役立ちます。

ゲーム・エンジン・プログラマー向けのインテル® GPA を使用した実践的な最適化の詳細は、「**インテル® Graphics Performance Analyzers を使用した実践的なゲーム・パフォーマンス解析**」(英語) (注 : この記事はインテル® GPA の古い UI を使用して記載されています) または**こちらのビデオ** (英語) を参照してください。

良いゲームから素晴らしいゲームへ

インテル® GPA は、特定のゲーム開発者のためのものではなく、ゲーム開発チームの全員に機能を提供します。この記事で紹介した最適化プロセスに関連した役割と一般的な課題が、目標のフレームレートを達成し、さらなる向上に役立つことを願っています。この記事では、最適化プロセスの各段階におけるインテル® GPA の利点を簡単に説明し、最適化時に確認すべきいくつかの項目を示しました。インテル® GPA を利用してアプリケーションを最適化するさまざまな方法については、**software.intel.com/gpa** (英語) を参照してください。サポートされるオペレーティング・システムごとの入門ガイドおよび包括的なヘルプ・ドキュメントを入手できます。

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、開発期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 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

ハイパフォーマンス・ビッグデータ・コンピューティング向け Harp-DAAL

生産性とパフォーマンスを同時に向上する鍵

Judy Qiu インディアナ大学 情報、コンピューティング、エンジニアリング学部 知能システム工学科

大規模なデータ解析が多くのビジネスや科学分野に革命をもたらしています。ビッグデータを処理して意味のある情報を得るには、簡単でスケーラブルな並列手法が重要です。この記事では、Harp-DAAL という名前の新しいハイパフォーマンス･コンピューティング (HPC) クラウド畳み込みフレームワークを紹介し、ビッグデータ (Hadoop*) と HPC 手法を組み合わせて生産性とパフォーマンスを同時に向上する方法を紹介します。

Harp は、ノードの同期を調整する分散 Hadoop* ベースのフレームワークです [1]。**インテル® Xeon® プロセッサー**および**インテル® Xeon Phi™ プロセッサー**・アーキテクチャー向けに高度に最適化された Harp のカーネルは、**インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL)**[2] を使用しています。そのため、ビッグデータ･ツールの高水準 API と、HPC プラットフォーム向けに最適化されているノード内の細粒度の並列処理を組み合わせることができます。

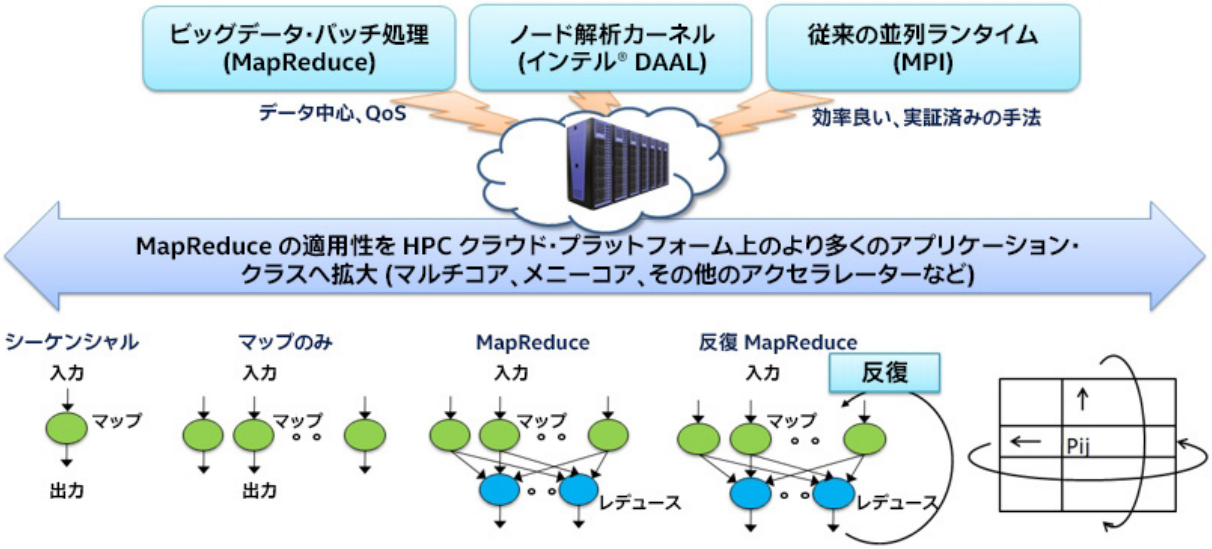
ここでは、画像クラスタリングで使用される計算負荷の高いアルゴリズム、K 平均法を用いて詳しく説明します。また、ほかの 3 つのビッグデータ・アルゴリズムのパフォーマンスを考察することで、Harp-DAAL の幅広い適用性を示します。

1. カラーコードによるサブグラフカウント
2. 行列因数分解
3. 潜在的ディリクレ配分法 (LDA)

これらのアルゴリズムには、パフォーマンスを妨げるロードバランス、不規則な構造、通信の問題などの特性があります。

図 1 のカテゴリーは、5 つの異なるシステム・アーキテクチャーにマップされる 5 つのモデルへのデータ集約型計算の分類を示しています。シーケンシャルで始まり、マップのみ、MapReduce、そして 反復 MapReduce の 3 つの MapReduce 形式に対応する集中型バッチ・アーキテクチャーが続きます。5 つ目のカテゴリーは、古典的なメッセージ・パッシング・インターフェイス (MPI) モデルです。

Harp は、手動による並列処理からマシンラーニングやシミュレーションまで、データ集約型計算の 5 つのクラスすべてのサポートを Hadoop* ユーザーに提供します。これにより、Hadoop* を (Harp プラグインを使用して) さらに多くのビッグデータ・アプリケーションのクラス、特にマシンラーニングやグラフ解析などの複雑なデータ解析に適用できます。マシンラーニングおよびデータ解析アプリケーション向けのスケールアップ・サーバーを効率良く利用できるように、モジュール型のソフトウェア・スタックとネイティブカーネル (およびインテル® DAAL) で構成されています。Harp-DAAL は、豊富な集合操作とライブラリーを利用して、一般的なプログラミング環境でシミュレーションとビッグデータを活用する方法を提供します。



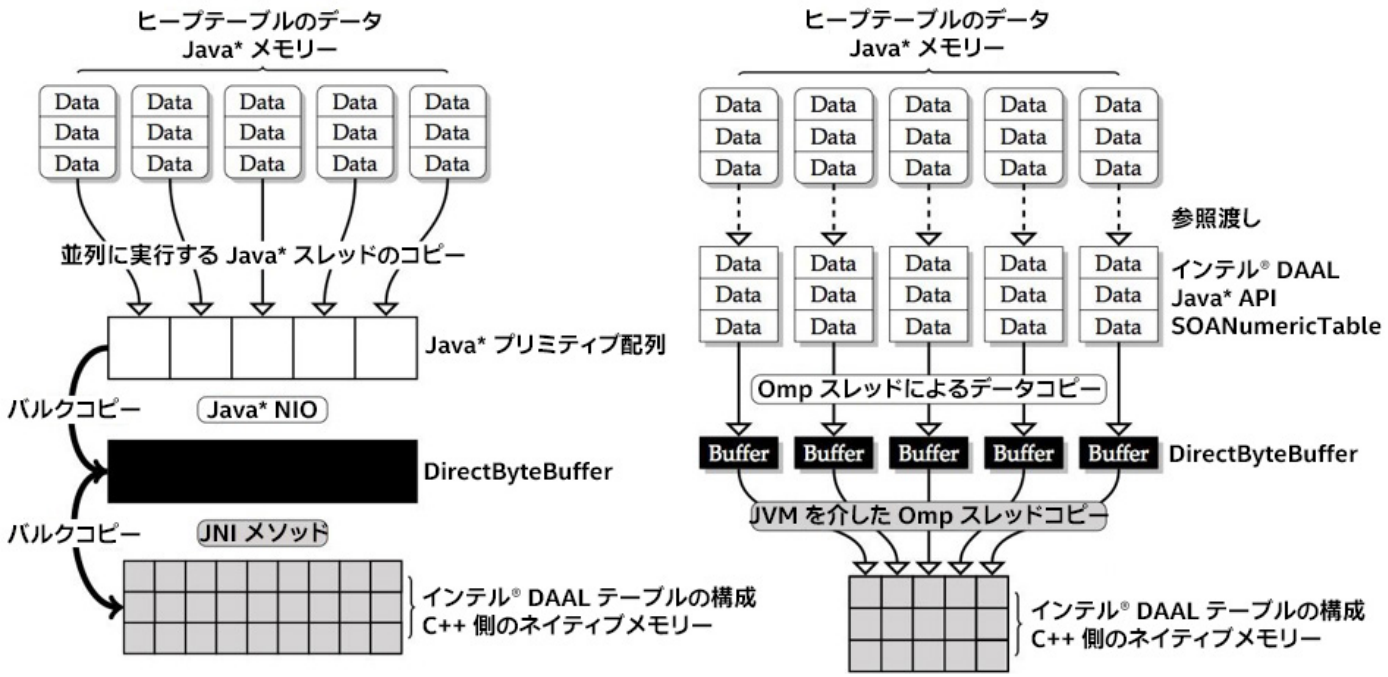
1 **大規模なハイパフォーマンス・ビッグデータ解析向けクラウド HPC 相互運用ソフトウェア**

Harp とインテル® DAAL のインターフェイス

インテル® DAAL は、ネイティブ C/C++ API だけでなく、Java* や Python* などの高水準プログラミング言語へのインターフェイスも提供しています。Harp は Java* で記述された Hadoop* エコシステムの拡張であるため、Harp とインテル® DAAL のインターフェイスとして Java* は当然の選択肢でした。

Harp-DAAL では、Harp-Table と呼ばれる、タグ付けされたパーティションで構成される階層的なデータ構造にデータが格納されます。各パーティションには、パーティション ID (メタデータ) とユーザー定義のシリアル化可能な Java* オブジェクト (プリミティブ配列など) が含まれます。通信を行う場合、Harp 集合通信操作を利用して分散クラスターノード間でデータが転送されます。ローカルの計算では、Harp-Table (JVM ヒープメモリー) からインテル® DAAL のネイティブカーネル (JVM ヒープメモリー外) へデータが移動します。Java* オブジェクトと C/C++ 割り当てメモリー空間との間でコピーが必ず発生します。**図 2** は、このデータコピーに対する 2 つのアプローチです。

1. **直接バルクコピー :** インテル® DAAL カーネルが密な問題に対して連続するメモリー空間を割り当てる場合、Harp-DAAL は Harp-Table とネイティブ・メモリー・アドレス間でバルクコピー操作を行います。
2. **マルチスレッド化された不規則なコピー :** インテル® DAAL カーネルが不規則な疎データ構造を使用する場合 (つまり、データが連続しないメモリーセグメントに格納される場合)、Harp-DAAL は Java*/OpenMP* スレッドを使用して、2 つ目のコピー操作を行い、スレッドはデータセグメントを並列に転送します。



2 **直接バルクコピー (左) とマルチスレッド化された不規則なコピー (右)**

Harp-DAAL のデータ解析への適用

K 平均法は、広く使用されている比較的単純なクラスタリング・アルゴリズムで、Harp-DAAL の分かりやすい使用例を示します。K 平均法は、クラスターの中心を使用して、データをモデル化し、反復改良によって迅速に収束します。Flickr* からの大きなデータセットに対して、K 平均法を実行します。このデータセットには、それぞれ ImageNet* で訓練した畳み込みディープ・ニューラル・ネットワーク・モデルを使用して抽出された 4,096 次元の特徴を持つ 1 億の画像が含まれます。データの前処理には、形式の変換や主成分分析 (PCA) を使用した 4,096 から 128 への次元削減が含まれます。(**Harp-DAAL チュートリアル** (英語) にデータの準備に関する説明があります。)[3]

Harp-DAAL は、K 平均法アルゴリズムをカスタマイズしたい開発者向けにモジュラー Java* 関数と、エンドユーザー向けにチューニング・パラメーターを提供しています。プログラミング・モデルは、集合操作によりリンクされたマップ関数で構成されています。K 平均法の例は 7 つのステップから成ります。

ステップ 1: 訓練データ (特徴ベクトル) とモデルデータ (クラスターの中心) をロードする

この関数を使用して、Hadoop* 分散ファイルシステム (HDFS) から訓練データをロードします。

```
// pointArray を作成
List<double[]> pointArrays = LoadTrainingData();
```

同様に、Harp テーブル・オブジェクト **cenTable** を作成して、HDFS から中心をロードします。すべてのマッパーが中心を要求するため、マスターマッパーがロードして、ほかのすべてのマッパーにブロードキャストします。異なる中心の初期化メソッドは、次のようにサポートできます。

```
// クラスターの中心を格納するテーブルを作成
Table<DoubleArray> cenTable = new Table<>(0, new DoubleArrPlus());

if (this.isMaster()) {
  createCenTable(cenTable);
  loadCentroids(cenTable);
}
// ほかのマッパーへ中心をブロードキャスト
bcastCentroids(cenTable, this.getMasterID());
```

ステップ 2: Harp からインテル® DAAL へ訓練データを変換する

HDFS からロードされた訓練データは、Java* ヒープメモリーに保持されます。インテル® DAAL カーネルを呼び出すため、このステップではインテル® DAAL の **NumericTable** へデータを変換します。これには、**NumericTable** のネイティブメモリーの割り当てと、**pointArrays** から **trainingdata_daal** へのデータコピーが含まれます。

```
// Harp からインテル® DAAL へ訓練データを変換
NumericTable trainingdata_daal = convertTrainData(pointArrays);
```

ステップ 3: インテル® DAAL の K 平均法カーネルの作成とセットアップ

インテル® DAAL は、各ノードで K 平均法のネイティブカーネルを呼び出す Java* API を提供します。この Java* API は、次のパラメーターを指定して呼び出します。

- 入力訓練データ・オブジェクト
- 中心の数
- スレッド・スケジューラー (**インテル® スレッディング・ビルディング・ブロック (インテル® TBB)** と**インテル® マス・カーネル・ライブラリー (インテル® MKL)**) によって使用されるスレッドの数

```
// インテル® DAAL の K 平均法カーネル・オブジェクトを作成
DistributedStep1Local kmeansLocal = new DistributedStep1Local(daal_Context,
Double.class, Method.defaultDense, this.numCentroids);
// 入力訓練データのセットアップ
kmeansLocal.input.set(InputId.data, trainingdata_daal);
// インテル® DAAL カーネルによって使用されるスレッド数を指定
Environment.setNumberOfThreads(numThreads);
// インテル® DAAL 側の cenTable を作成
NumericTable cenTable_daal = createCenTableDAAL();
```

ステップ 4: Harp からインテル® DAAL へ中心の形式を変換する

中心は、プロセス (マッパー) 間の通信用に Harp テーブル **cenTable** に格納されており、各反復でインテル® DAAL 形式に変換されます。

```
// Harp からインテル® DAAL 形式へ中心を変換
convertCenTableHarpToDAAL(cenTable, cenTable_daal);
```

ステップ 5: インテル® DAAL カーネルによるローカル計算

各反復でローカル計算を実行する インテル® DAAL の K 平均法カーネルを呼び出します。

```
// クラスターの中心をインテル® DAAL カーネルに指定
kmeansLocal.input.set(InputId.inputCentroids, cenTable_daal);
// 部分解を求めるインテル® DAAL カーネルによるローカル計算の最初のステップ
PartialResult pres = kmeansLocal.compute();
```

ステップ 6: マッパー間の通信

Harp-DAAL K 平均法は、各マッパーがモデルデータ全体 (クラスターの中心) のローカルコピーを保持する **AllReduce** 通信モデルを使用します。しかし、Harp はマッパー間のモデルデータの同期に異なる通信操作を提供します。

- **regroup & allgather** (デフォルト)
- **allreduce**
- **broadcast & reduce**
- **push & pull**

regroup & allgather 操作では、最初に異なるマッパーからの同じ中心を結合して、それを指定された順序でマッパーに再配布します。中心を平均化した後、**allgather** 操作は、各マッパーに平均化した中心の完全なコピーを取得させます。

```
comm_regroup_allgather(cenTable, pres);
```

allreduce 操作では、中心がレデュースされ、各マッパーにコピーされます。そして、各マッパーで中心に平均操作が適用されます。

```
comm_allreduce(cenTable, pres);
```

各反復の最後に、**printTable** を呼び出してクラスタリングの結果を確認します。

```
// 反復 i の最初の 10 次元の最初の 10 個の中心を出力
printTable(cenTable, 10, 10, i);
```

ステップ 7: メモリーを解放してクラスターの中心を格納する

すべての反復が終了したら、インテル® DAAL と Harp テーブル・オブジェクトに割り当てられたメモリーを解放します。中心の値は HDFS に出力として格納されます。

```
// メモリーを解放して時間を記録
cenTable_daal.freeDataMemory();
trainingdata_daal.freeDataMemory();
// クラスターの中心を書き出す
if (this.isMaster()) {
      KMUtil.storeCentroids(this.conf, this.cenDir,
      cenTable, this.cenVecSize, "output");
}
cenTable.release();
```

パフォーマンス結果

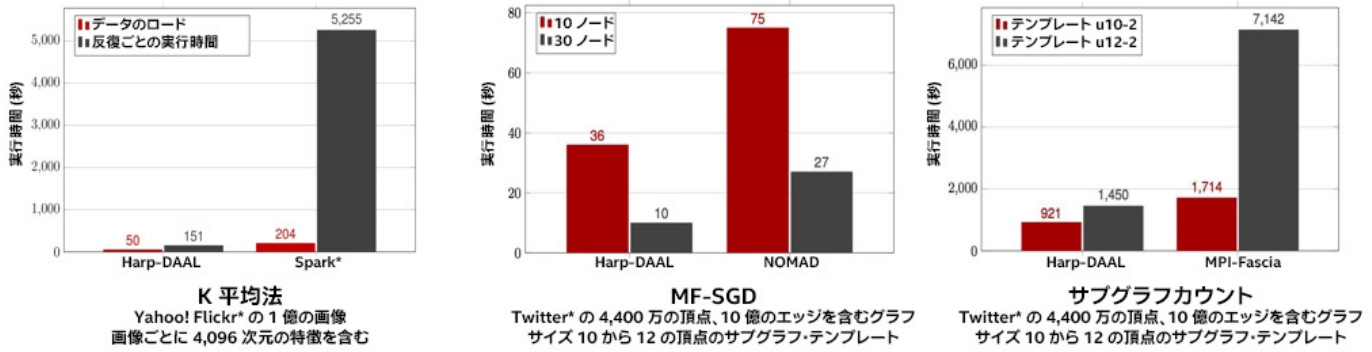
Harp-DAAL のパフォーマンスは、異なるアルゴリズムを用いた 4 つのアプリケーションの結果 [4,5] で示します。

1. **K 平均法 :** 規則的なメモリーアクセスを持つ高密度のクラスタリング・アルゴリズム
2. **MF-SGD (確率的勾配降下法の行列因数分解):** 不規則なメモリーアクセスと大きなモデルデータを持つ高密度の推奨アルゴリズム
3. **サブグラフカウント :** 不規則なメモリーアクセスを持つ疎グラフ・アルゴリズム
4. **潜在的ディリクレ配分法 (LDA):** 大きなモデルデータと不規則なメモリーアクセスを持つ疎トピック・モデリング・アルゴリズム

テストには 2 つのクラスターを使用しました。

1. **インテル® Xeon® プロセッサー E5-2670** と InfiniBand* インターコネクト
2. **インテル® Xeon Phi™ プロセッサー 7250** とインテル® Omni-Path ファブリック・インターコネクト

図 3 では、Harp-DAAL は 30 ノードの K 平均法において、インテル® DAAL に含まれるインテル® MKL の高度にベクトル化されたカーネルを使用して、Spark* と比較して約 30 倍のスピードアップを達成しています。MF-SGD は最大 30 ノードで実行され、最先端の MPI C/C++ ソリューションの NOMAD* と比較して、3 倍のスピードアップを達成しています。これは、推奨システムで大きなモデルデータの通信を高速化する、Harp の回転集合操作によってもたらされます。

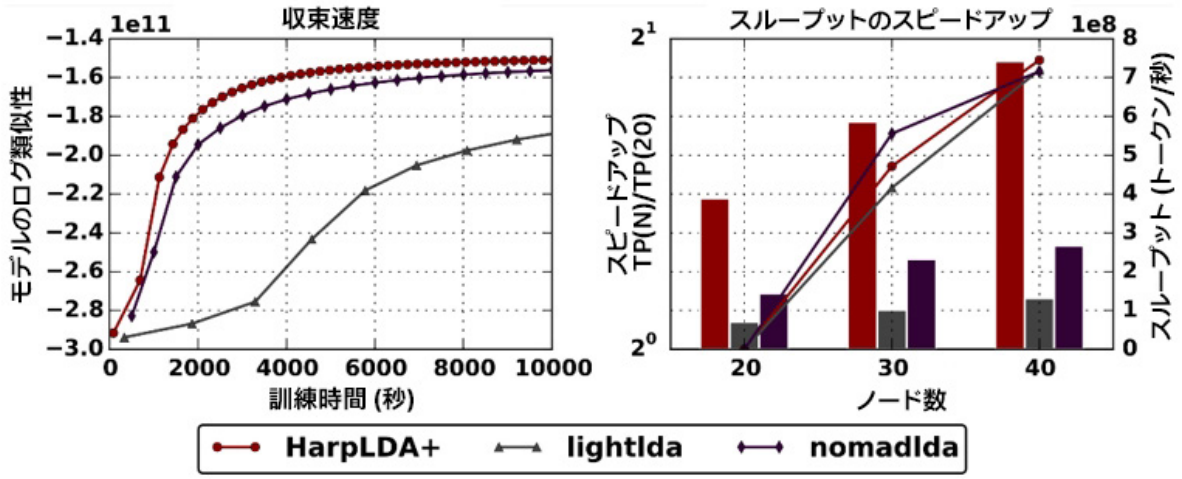


3 **3 つの主要マシンラーニング・アルゴリズムのパフォーマンスの比較：K 平均法、MF-SGD、およびサブグラフカウント**

16 ノードのインテル® Xeon® プロセッサー E5 ファミリーベースの構成において、Harp-DAAL のサブグラフカウントは、膨大な数のエッジを含む Twitter* グラフデータの大規模なサブテンプレートで、MPI-Fascia* と比較して 1.5 ~ 4 倍のスピードアップを達成しています。このパフォーマンスの向上は、計算と通信のノードレベルのパイプラインのオーバーラップによってもたらされます。グラフ頂点の隣接リスト分割により、シングルノードのスレッドの並行性が改善されます。

図 4 は、Harp LDA が、LightLDA* や NomadLDA* などのほかの最先端の MPI 実装と比較して、収束とスピードアップに優れていることを示しています[5]。これは、並列処理の効率を向上する 2 つの最適化によってもたらされます。

1. Harp は、最新モデルのノード間通信に回転計算モデルを採用し、同時にタイマー制御により同期オーバーヘッドを軽減しています。
2. ノード内レベルでは、動的なスケジューリング・メカニズムにより、ロードインバランスを軽減しています。



4 **ClueWeb データセットに対するさまざまな LDA 実装のパフォーマンス (300 億トークン、5,000 トピック)**

現在の Harp-DAAL システムは、インテル® DAAL 2018 の K 平均法などのローカル計算カーネルを利用して、13 の分散データ解析とマシンラーニング・アルゴリズムを提供します。さらに、Harp-DAAL は独自のデータ集約型カーネルも開発しています。これには、前述のように、膨大な数のエッジと 10 個の頂点のサブテンプレートを含むソーシャル・ネットワークの Twitter* グラフを 15 分で処理できる、大規模なサブグラフ・カウント・アルゴリズムが含まれます。Harp-DAAL フレームワークとマシンラーニング・アルゴリズムは、一般公開されているため[5]、ソフトウェアをダウンロードして、チュートリアルを調査し、Harp-DAAL をほかのデータ集約型アプリケーションに適用することができます。

参考資料

1. Harp コード・リポジトリー : **https://github.com/DSC-SPIDAL/harp** (英語)
2. インテル® DAAL: **https://github.com/intel/daal** (英語)
3. Harp-DAAL チュートリアル : **https://github.com/DSC-SPIDAL/harp/tree/tutorial** (英語)
4. Langshi Chen et. al, "Benchmarking Harp-DAAL: High Performance Hadoop on KNL Clusters," in the Proceedings of the 10th IEEE International Conference on Cloud Computing (IEEE Cloud 2017), June 25-30, 2017.
5. Bo Peng et. al, "HarpLDA+: Optimizing Latent Dirichlet Allocation for Parallel Efficiency," the IEEE Big Data 2017 conference, December 11-14, 2017.

インテル® DAAL
マシンラーニングとビッグデータ解析を高速化
無料
ダウンロード

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 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

命令パイプラインに関する考察

現代のアプリケーション・プログラミングにおける適応性の鍵

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

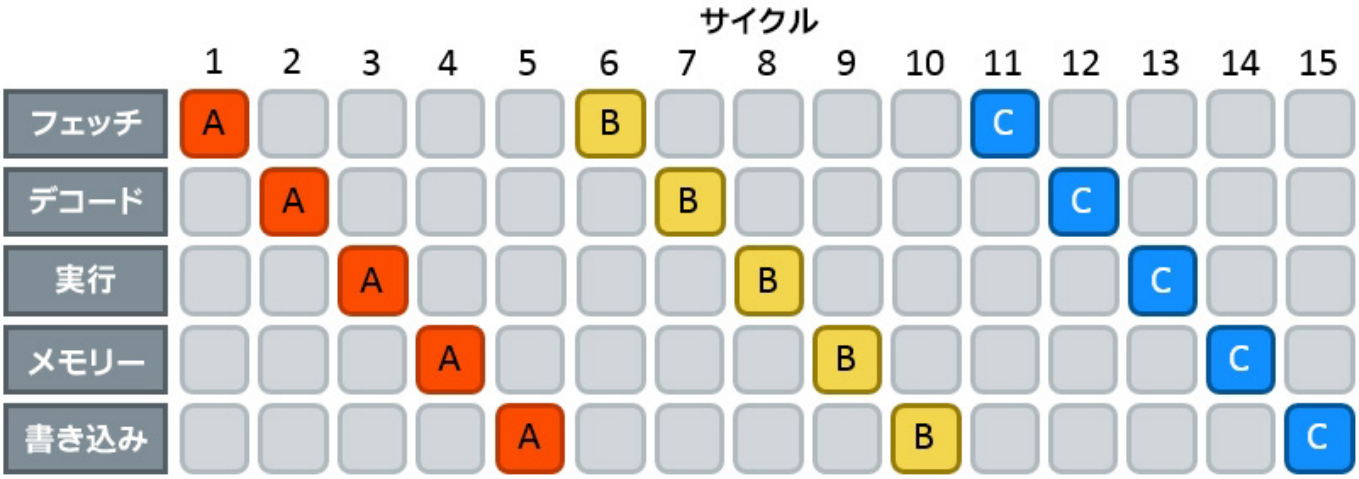
現代のアプリケーション・プログラミングにおいて高い効率を達成するには、絵を描く上で色彩理論を理解するのと同じくらい、命令パイプラインを少なくとも基本レベルで理解することが重要です。それは、基本的かつ普遍的な概念です。命令パイプラインの誕生時期と定義についてはさまざまな説がありますが、1970 年代から 1980 年代のある時点から広まり始めて、現代ではほとんどのマシンで採用されています。

命令の処理は瞬時には行われません。いくつかのステップが含まれます。実装の正確な詳細はマシンにより異なりますが、概念的には 5 つのステップがあります。

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

1. 命令の**フェッチ**
2. 命令の**デコード**
3. 命令の**実行**
4. メモリー**アクセス**
5. 結果の**書き込み**

パイプライン化しない場合、1 つの命令を開始して終了するまで、次の命令を処理することはできません。5 つのステップにはそれぞれ 1 サイクルかかると仮定すると、3 つの命令の処理には 15 サイクルかかります (**図 1**)。

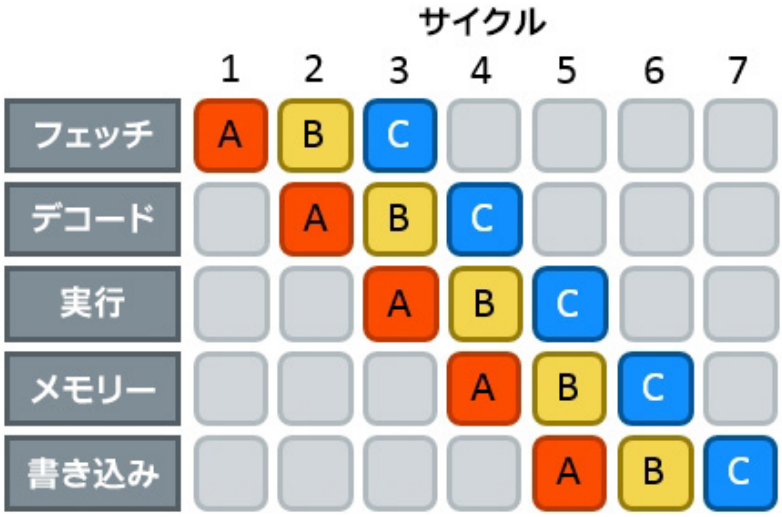


1 **クロックサイクルごとに 1 ステップのシーケンシャルな命令処理**

各ステップは異なるハードウェア・セクションによって処理されるため、現代のプロセッサーは命令をパイプライン化して、さまざまなハードウェア・セクションがそれぞれ異なる命令を同時に進行できるようにすることで効率を向上しています。例えば、**図 2** のサイクル 3 では、プロセッサーは命令 C のフェッチ、命令 B のデコード、命令 A の実行を行っています。サイクル 7 の最後には 3 つの命令がすべて完了します。つまり、シーケンシャルに処理した場合よりも 8 サイクル早く完了できます。

これは、最初に洗った洗濯物を乾燥機で乾かしながら、次の洗濯物を洗濯機で洗うのに似ています。もちろん、命令の処理には洗濯よりも多くのステップが含まれており、それは大きく 2 つに分類できます。

1. **フロントエンド :** 命令のフェッチとデコードを行う CPU 部分
2. **バックエンド :** 命令の実行とリタイアを行う CPU 部分



2 **パイプライン化された命令処理**

図 2 は、命令パイプラインを簡略化したものです。実際には、パイプラインに含まれるステップの数は実装に依存し、例で使用されている各ステップは、多くの場合、複数のサブステップで構成されます。しかし、これは概念の理解には影響しないため、ここでは簡略化した 5 ステップのモデルを使用します。また、この簡略化されたモデルでは、スーパースカラー設計は考慮していません。スーパースカラー設計は、演算ユニット (ALU) などの機能ユニットを複製して、追加のユニットがビジーになるように一度に複数の命令をフェッチするため、プロセッサー・コアごとに複数のパイプラインを持ちます。

利用可能なパイプラインの数は幅と呼ばれます。**図 3** は、幅 2 の設計を示しており、サイクル 1 で命令 A と B をフェッチして、サイクル 2 で命令 C と D をフェッチします。幅は、(理論的に) サイクルごとに発行可能な命令の数として定義されます。ただし、広く普及している x86 などの CISC 設計では、パイプライン処理はさらに複雑になります。

パイプラインは、少数の単純な命令を高速に実行する RISC 設計に最適です。x86 で見られるような精密な命令は、複雑さと実行時間が異なるため、いくつかの理由からパイプライン化することが困難です。

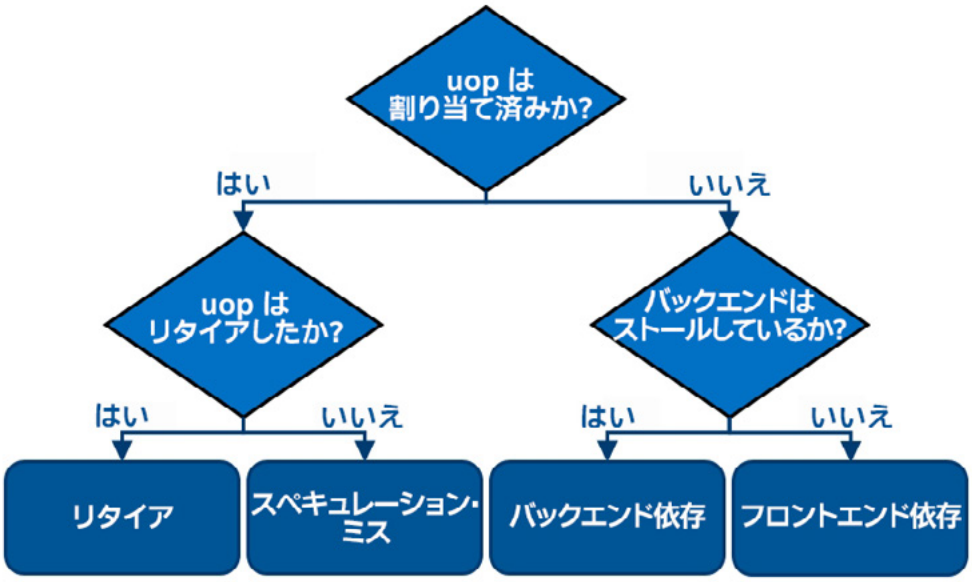
3 **幅 2 の命令パイプライン**

- **処理に時間のかかる命令**は、パイプラインを停滞させる可能性があります。
- **複雑な命令**は、データ依存性によりストールする可能性が高まります。

この問題の解決策は、複雑な命令を小さなマイクロオペレーション (µop) に分割することです。便宜上、µ はよく u (つまり uop) と表記されます。x86 命令は、フェッチ、デコード、uop への変換、バッファーからディスパッチして実行を経て、最終的にリタイアされます。この x86 命令のフェッチと uop のディスパッチ間の溝が、この手法を使用するプロセッサーの幅の定義を困難にしており、uop のペアは融合できる場合があるという事実が状況をさらに難しくしています。

このような状況では、プロセッサーの幅を正確に定義することは難しく、抽象化のほうが理解しやすいでしょう。セマンティクスやハードウェアに関係なく、サイクルごとにフロントエンドから発行できる uop 数と、サイクルごとにバックエンドからリタイアできる uop 数は固定です。これは利用可能なパイプライン・スロットの数で、一般的な経験則として、最近のインテル® プロセッサーでは通常 4 です。

各スロットは uop の処理に応じて、任意のサイクルで 4 つのカテゴリーのいずれかに分類できるため、パイプライン・スロットの概念はアプリケーションの最適化に役立ちます (**図 4**)。各パイプライン・スロット・カテゴリーは、適切にチューニングされた特定のタイプ (例えば、クライアント、デスクトップ、サーバー、データベース、科学) のアプリケーションでは、特定のパーセンテージ範囲内になることが予想されます。**インテル® VTune™ Amplifier** などのツールは、各カテゴリーに分類されるアプリケーションのパイプライン・スロットのパーセンテージを測定して、予想範囲と比較するのに役立ちます。リタイア以外のカテゴリーが適切なアプリケーション・タイプの予想範囲を超える場合、パフォーマンス・ボトルネックの存在と性質を示します。

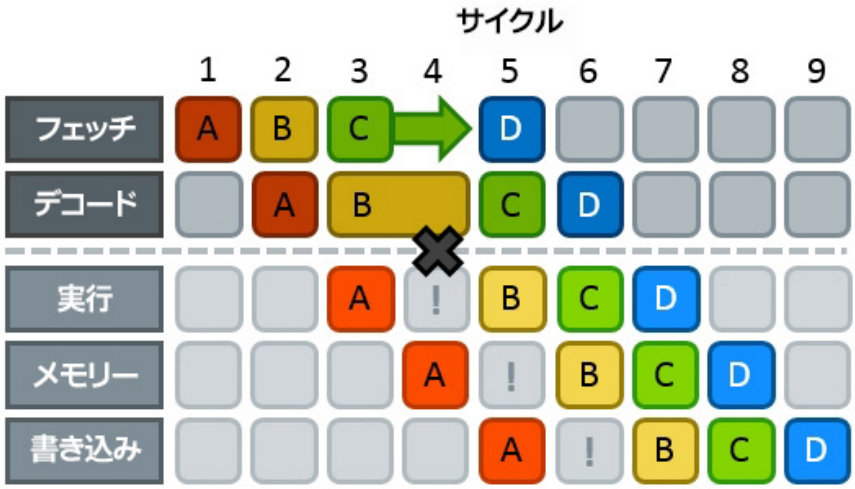


4 パイプライン・スロットの分類フローチャート

測定結果を利用してパフォーマンスを最適化する手法については、インテル® VTune™ Amplifier チューニング・ガイドなどで説明されています。それらの手法については、この記事の範囲外であるため、ここでは取り上げません。(チューニング・アドバイスに関しては、この記事の最後にある関連資料を参照してください。) 代わりに、ここでは、これらの状況下でパイプライン内で何が起こっているのかを考察します。分かりやすくするため、1 つのパイプラインを使用して説明します。

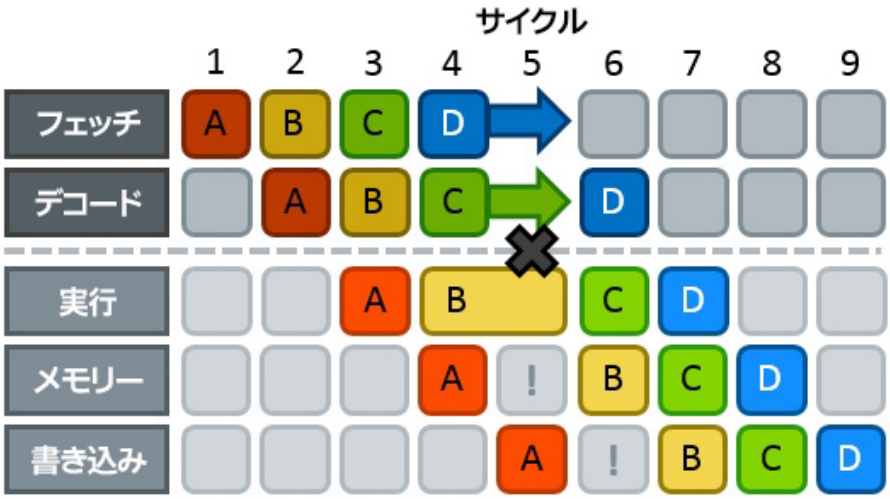
リタイアカテゴリーについては前述のとおりです。ストールや中断のない、パイプラインの通常の機能を示します。一方、バックエンド依存およびフロントエンド依存カテゴリーは、ストールによって命令がフロントエンドからバックエンドに移動できない状況を示しています。バックエンド依存およびフロントエンド依存スロットを引き起こすストールには、キャッシュミスから特定のタイプの実行ユニットに対する過度に高い需要まで、さまざまな原因が考えられます。その結果、uop が現在のステージをスケジュールどおりに通過しません。

フロントエンド依存スロットは、バックエンドが受け入れ可能な状態にもかかわらず、命令がフロントエンドからバックエンドに移動できない場合に発生します。**図 5** では、命令 B はデコードに 2 サイクルかかっており、サイクル 4 でバックエンドに移動するのではなく、同じステージにとどまっています。これにより、パイプライン・バブルと呼ばれる空のスペース (図では ! で示します) がパイプライン中に発生します。



5 フロントエンド依存スロットの例

バックエンド依存スロットは、(フロントエンドが実際に供給可能かどうかに関係なく) バックエンドが uop を受け入れることができない場合に発生します (**図 6**)。この例では、命令 B は実行に 2 サイクルかかっており、サイクル 5 においても実行ステージにとどまっているため、命令 C はバックエンドに移動できません。この場合もパイプライン・バブルが生じます。



6 バックエンド依存スロットの例

遅延は、常にデコードまたは実行ステージで発生するとは限りません。**図 5** で、もし B がフェッチに 2 サイクルかかった場合、サイクル 3 でデコードステージに命令が渡されず、バブルが生じるため、サイクル 4 でバックエンドに渡す命令がありません。同様に、**図 6** で、もし 命令 A がメモリーステージに 2 サイクルかかった場合、サイクル 5 で B は (準備できているかどうかに関係なく) 実行ステージから移動できません。そのため、実行ステージにとどまり、C がバックエンドへ移動するのを妨げます。

最後のカテゴリーは、スペキュレーション・ミスです。これは、一部処理済みの uop が完了前にキャンセルされた場合に発生します。uop がキャンセルされる最も一般的な原因は分岐予測ミスですが、ほかにも原因があります (自己修正コードなど)。分岐命令は、分岐するかどうかを判別する前に、あるポイントまで処理を進める必要があります。ここでも、実装の詳細は異なりますが、概念は似ています。説明のため、ここでは分岐命令が実行ステージの最後に到達したら、X と Y のどちらのパスに分岐するか判定されると仮定します (**図 7**)。分岐は頻繁に発生するため、毎回分岐命令が完了するのを待ってから次の命令をロードするのでは、非常に大きなパフォーマンス･ペナルティーを被ります。代わりに、分岐予測アルゴリズムがどちらのパスに分岐するのかを予測します。

7 分岐が解決される前にパス X の命令をパイプラインにロードする

8 **正しい分岐予測**

結果は次のいずれかになります。

1. **分岐予測が正しく**(**図 8**)、通常どおりに処理が続行されます。
2. **分岐予測ミス**が発生し(**図 9**)、正しくない命令は破棄されてバブルになり、正しい命令がパイプラインにロードされます。

パフォーマンス・ペナルティーは、分岐が実行され解決されるのを待ってから命令をロードする場合と同じです。ただし、分岐のたびに発生するのではなく、予測アルゴリズムが間違っていた場合にのみ発生します。そのため、予測アルゴリズムを改善するための取り組みが絶えず行われています。

9 **分岐予測ミス**

インテル® VTune™ Amplifier を利用すると、アプリケーションのスペキュレーション・ミス、フロントエンド依存、およびバックエンド依存スロットの数を確認できます。しかし、それらの数値がどのように算出され、何を意味しているのかを理解しないことは、ガイドの手順に闇雲に従い、著者の推奨事項に完全に依存することと同じです。理解こそが適応する鍵であり、流動的なソフトウェアの世界では、いつか未知のシナリオに遭遇する可能性があるため、個々のアプリケーション固有のニーズに対応できるようになることが重要です。

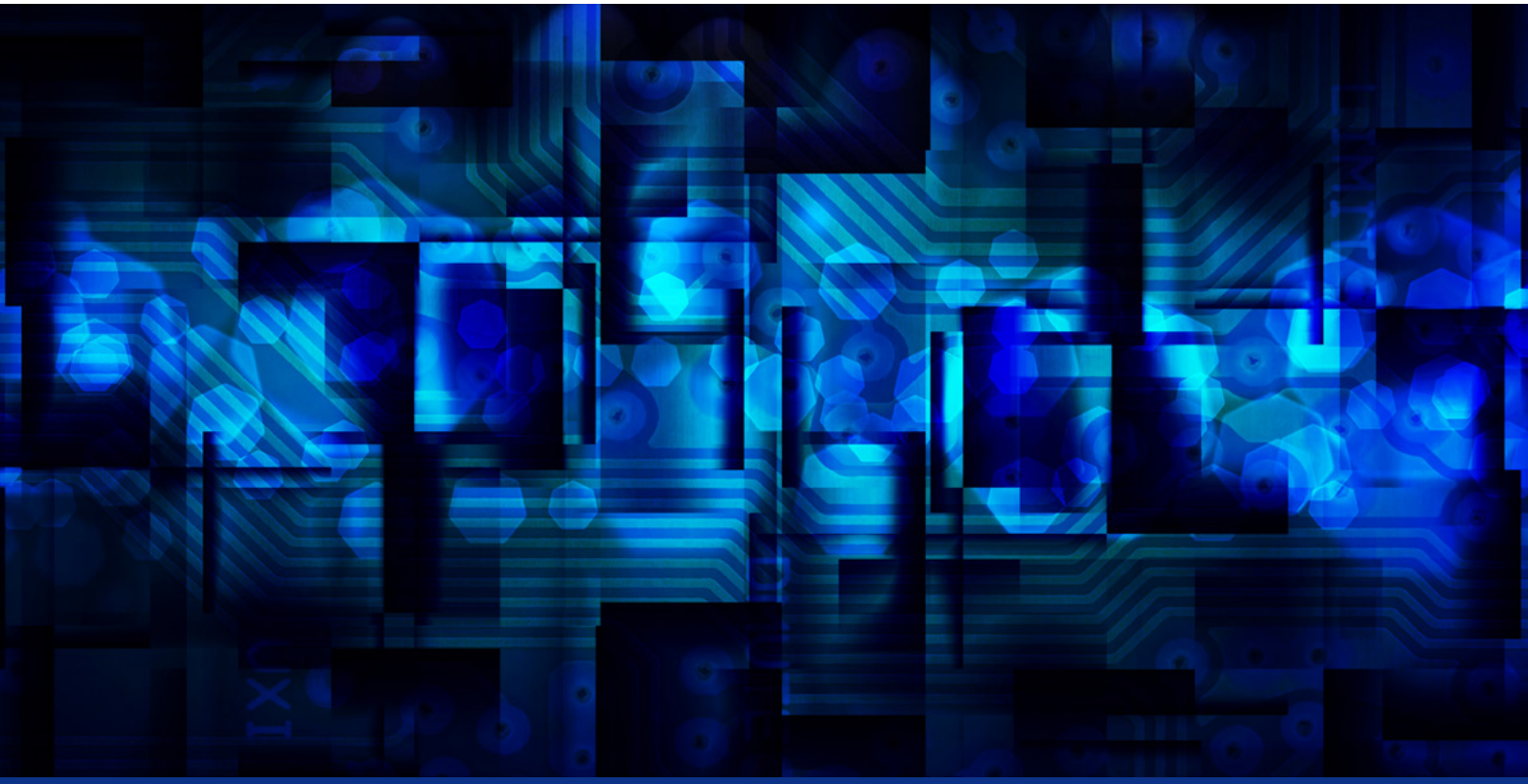
関連資料

- Patterson, Jason R. C, "**Modern Microprocessors: A 90-Minute Guide!** (英語)," Lighterra, May 2015. Sections 2-7, 12.
- Walton, Jarred, "**Processor Architecture 101 – the Heart of your PC** (英語)," PCGamer. Future plc, 28 December 2016.
- "**Tuning Guides and Performance Analysis Papers** (英語)," Intel Developer Zone, Intel. 日本語版 :「**チューニング・ガイドとパフォーマンス解析**」
- "**Tuning Applications Using a Top-Down Microarchitecture Analysis Method** (英語)," Intel Developer Zone, Intel. 日本語版 :「**マイクロアーキテクチャーのトップダウン解析法を使用してアプリケーションをチューニングする**」

インテル® VTune™ Amplifier
現代のプロセッサーのパフォーマンス解析

無料評価版のダウンロード

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 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® Xeon® スケーラブル・プロセッサー上の並列 CFD と HiFUN ソルバー

高速な数値シミュレーション向けに HPC プラットフォームの性能を最大化

S&I Engineering Solutions Pvt. Ltd. 研究分析チーム
Rama Kishan Malladi テクニカル・マーケティング・エンジニア
S.V. Vinutha テクニカル・マーケティング・エンジニア
Austin Cherian アカウント・エグゼクティブ

計算流体力学 (CFD) は、高速なコンピューターを利用して、流体制御方程式の数値解を求める科学の一分野です。CFD の解は大量のデータを基に計算されるため、今日のハイパフォーマンス・コンピューティング (HPC) ハードウェア・プラットフォームを最大限に活用することが不可欠です。

CFD は、流体の質量、運動量、エネルギー保存を表す非線形偏微分方程式の**ナビエ - ストークス方程式**を使用します。CFD において離散化は、ナビエ - ストークス方程式を一連の代数方程式に変換する手法です。

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。

産業応用に関連する形状の複雑さと複雑な流体物理特性により、代数系のサイズはさまざまで、通常、数百万から数十億を超える式になります。つまり、設計サイクルに影響を及ぼすのに十分な短い期間内に設計データを取得するには、現実的な数値シミュレーションを大規模な HPC プラットフォームで実行する必要があります。

この記事では、超並列スーパーコンピューティング・プラットフォームのアーキテクチャーを最大限に活用できる CFD アプリケーションの例として **S&I Engineering Solutions (SandI) Pvt. Ltd.** (英語) の独自ソフトウェアである **HiFUN** (英語) ソルバーを検証します。[1,2]

スケーラブルなパフォーマンスの達成

HPC アプリケーションのパフォーマンスに影響する要因は 3 つあります。

1. 単一プロセスのパフォーマンス
2. ロードバランス
3. アルゴリズムのスケーラビリティー [3]

ロードバランスを達成するには、各プロセッサー・コアに割り当てられる計算作業がほぼ同じになるように、離散化された計算ドメイン (ワークロード、グリッド、またはメッシュとも呼ばれる) を分割すべきです。しかし、ロードバランスのために、分散プロセッサー間で過度なデータ通信を行うべきではありません。この 2 つの要件はしばしば競合するため、両者のバランスをとる必要があります。METIS を使用するドメイン分割は[2]、CFD シミュレーションでロードバランスを達成する一般的な方法です。アルゴリズムのスケーリングは、並列アプリケーションにおける別の重要なパフォーマンス要因です。[2] 理想的には、並列ソルバーのパフォーマンスは、プロセッサー数の増加に伴って低下すべきではありません。

HiFUN ソルバーのような CFD アプリケーションの並列パフォーマンスを制限する 3 つ目の要因は、高速なプロセッサーと低速なメモリーアクセスのギャップです。これは、単一プロセスのパフォーマンスの低下につながります。この問題に対応するため、多くのコンピューターはメモリー階層を採用しており、すぐに計算で必要となる少量のデータは、キャッシュと呼ばれる高速にアクセスできるメモリーに保持されます。つまり、キャッシュにあるデータを最大限活用することが重要です。そのためには、グリッドデータの空間的局所性を保証する適切なデータレイアウトが必要です。単一プロセスのパフォーマンスの問題は、**カットヒル・マキー法**のような順序付けアルゴリズムを使用してアプリケーション・レベルである程度対応できますが、CFD ソルバーに含まれる計算操作のさまざまな性質を考慮すると、プロセスのパフォーマンスを最大限に引き出すことは面倒な作業です。これに関連して、インテル® プロセッサー・テクノロジーの進化は、CFD アプリケーションの単一プロセスのパフォーマンスの大幅な向上をもたらしています。最新世代の**インテル® Xeon® プロセッサー**は、大容量キャッシュ、プロセッサーとメモリー間のデータチャネル数増加による高メモリー帯域幅、メモリー速度の向上を提供します。CFD アプリケーションでは、単一プロセスのパフォーマンスの向上により、当然ながら、シングルノードのパフォーマンスも向上します。

プロセッサー・テクノロジーの進化がどのようにしてソフトウェア・パフォーマンスの向上をもたらすのかを検証するため、**インテル® Xeon® スケーラブル・プロセッサー**上で HiFUN のシングルノードとマルチノードのパフォーマンスを評価し、前世代のプロセッサーと比較しました。

HiFUN CFD ソルバー

HiFUN は、ロバストかつ高速で正確な最先端の汎用 CFD ソルバーで、魅力的なターンアラウンド・タイムで空力設計データを提供します。典型的な産業環境において、複雑な形状と複雑な流体物理特性を処理する能力により、有用性を発揮します。任意の多面体の体積を処理可能な非構造化データを使用することで、HiFUN は複雑な形状を比較的容易にシミュレーションできます。さらに、行列を使用しない暗黙のプロシージャーの使用は、定常状態への迅速な収束をもたらし、ソルバーを効率良くロバストにします。

HiFUN の正確さは、**AIAA Drag Prediction Workshop** (英語) や **AIAA High Lift Prediction Workshop** (英語) など、さまざまな国際的 CFD コード評価で十分に実証されています。HiFUN は、低亜音速から極超音速まで、広範な流れ問題のシミュレーションでの使用実績があります。

並列パフォーマンスの評価

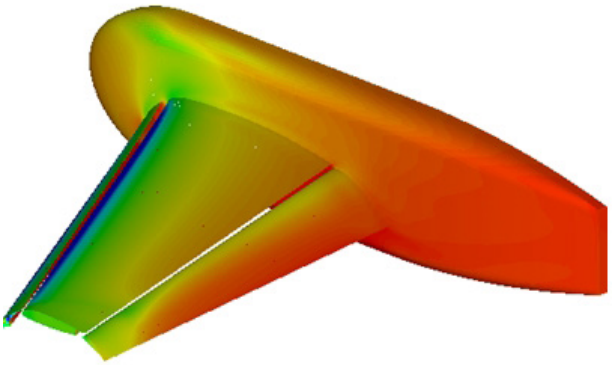
HiFUN ソルバーの並列パフォーマンスを評価するには、次のメトリックを考慮する必要があります。

1. **理想的なスピードアップ :** 特定の実行で使用されたコア数と参照コア数の比率 (評価では最小コア数を使用)
2. **実際のスピードアップ :** 指定した計算で参照コア数が使用された場合の反復ごとの時間と指定したコア数が使用された場合の反復ごとの時間の比率
3. **並列処理の効率 :** 実際のスピードアップと理想的なスピードアップの比率
4. **マシン・パフォーマンス・パラメーター (MPP):** 100 反復あたりの時間にスケーラビリティー実行で使用されたコア数を掛けた積とグリッドサイズ (100 万単位) の比率

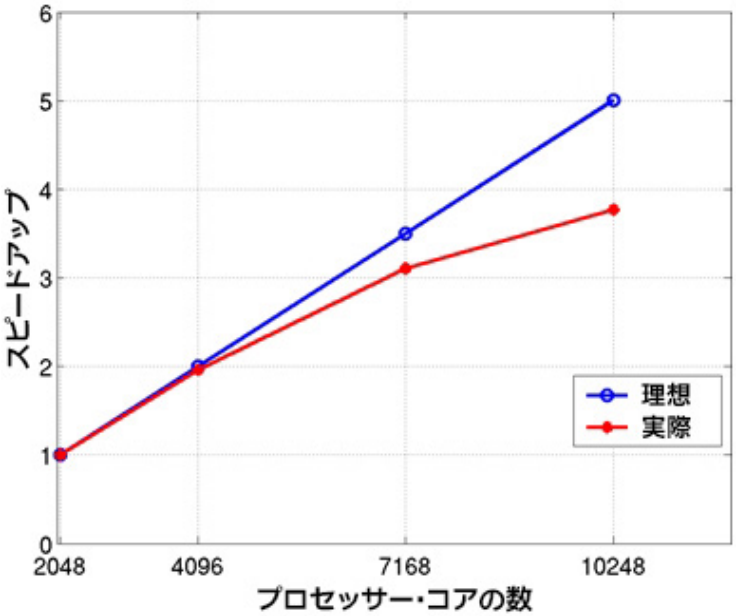
最初の 3 つのパラメーターは、並列コンピューティングの文献でよく見かけます。4 つ目のパラメーターの MPP は、特定の CFD アプリケーションを異なるコンピューティング・プラットフォームで評価する方法を提供します。[1] MPP 値が小さいコンピューティング・プラットフォームは、ほかのプラットフォームよりも優れたコンピューティング・パフォーマンスをもたらすと予想されます。

HiFUN ソルバーの重要な強みは、数千のプロセッサー・コアにスケールできることです。これは、HiFUN が **NASA の Pleiades スーパーコンピューター** (英語) 上の 10,000 を超えるインテル® Xeon® プロセッサー・コアにスケールできることを示した **SandI とインテルの共同研究** (英語) で実証されています。評価には、NASA の

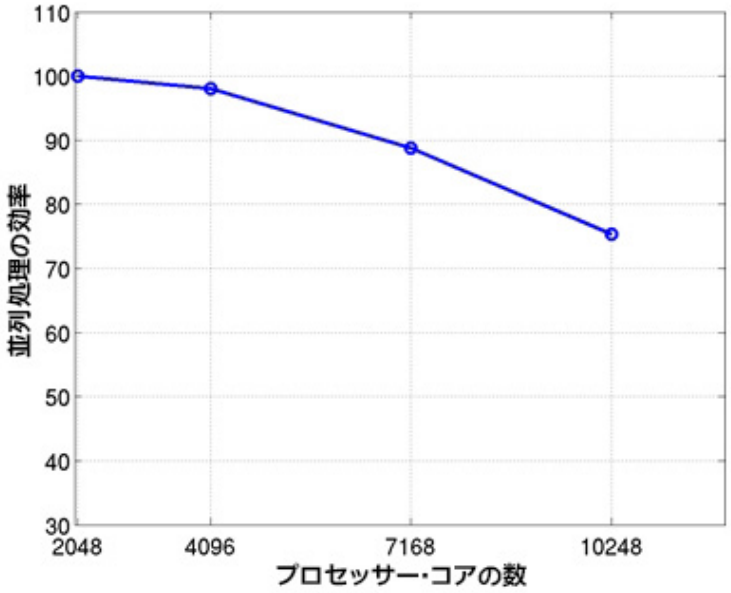
Trap Wing が使用されました (**図 1**)。**図 2** と **3** は、それぞれ 6350 万のグリッドを使用した場合のスピードアップと並列処理の効率を示すグラフです。**図 2** から、HiFUN は 4,096 プロセッサー・コアで理想に近いスピードアップを示していることが分かります。また、Pleiades 上の 7,168 プロセッサー・コアでは、HiFUN の並列処理の効率は約 88% に達します。さらに、約 6,350 万のグリッドのが控えめな大きさになる 10,248 プロセッサー・コアの場合でも HiFUN の並列処理の効率は約 75% と比較的良好です。この HiFUN のスケーラブルな並列パフォーマンスは、問題サイズに依存しないターンアラウンド・タイムが期待できることを示しているため、デザイナーにとって利点と言えます。



1 **NASA Trap Wing の表面圧力分布図**

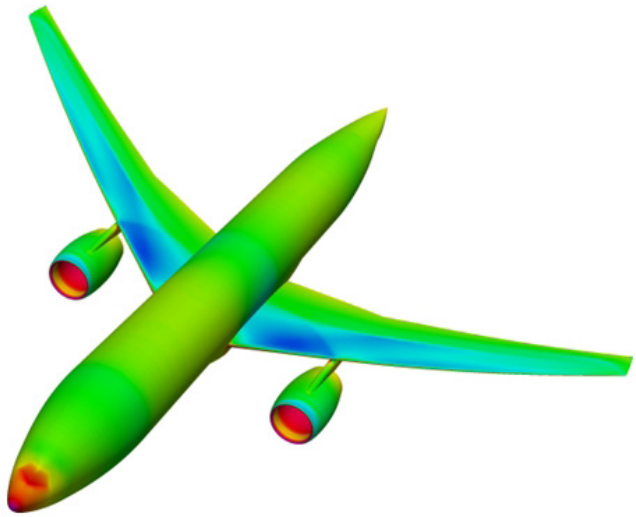


2 **スピードアップ・グラフ**



3 **並列処理の効率グラフ**

評価は、第 6 回 AIAA Drag Prediction Workshop で使用された NASA Common Research Model (CRM) で行われました。**図 4** に示すように、翼、機体、エンジン保護カバーおよびその支柱を備えた輸送機を表します。空力平均翼弦に基づく自由流マッハ数、迎え角、およびレイノルズ数は、それぞれ 0.85、約 2.6°、500 万です。解析のワークロードは、約 510 万 の六面体です。計算された NASA CRM の表面圧力分布を**図 4** に示します。



4 **NASA CRM の表面圧力分布図**

この評価では、HiFUN ソルバーのシングルノード・パフォーマンスの比較に、次の 2 つの世代のインテル® Xeon® プロセッサーを使用しました (**表 1**)。

1. インテル® Xeon® プロセッサー E5-2697 v4 (以降「インテル® Xeon® プロセッサー」と呼びます)
2. インテル® Xeon® Gold 6148 プロセッサー (以降「インテル® Xeon® スケーラブル・プロセッサー」と呼びます)

インテル® Xeon® スケーラブル・プロセッサーは 20 コアで、インテル® Xeon® プロセッサーは 18 コアです。両方のプロセッサーの各プロセッサー・コアのクロック周波数はほぼ同じであるため、コア数の違いのみを考慮して線形スケーラビリティーを仮定すると、11% のパフォーマンス向上が期待できます。しかし、**図 5** に示すように、HiFUN ソルバーはインテル® Xeon® スケーラブル・プロセッサーにおいて、インテル® Xeon® プロセッサーと比較して約 22% のパフォーマンス向上を達成しています。これは、次の要因によるものと考えられます。

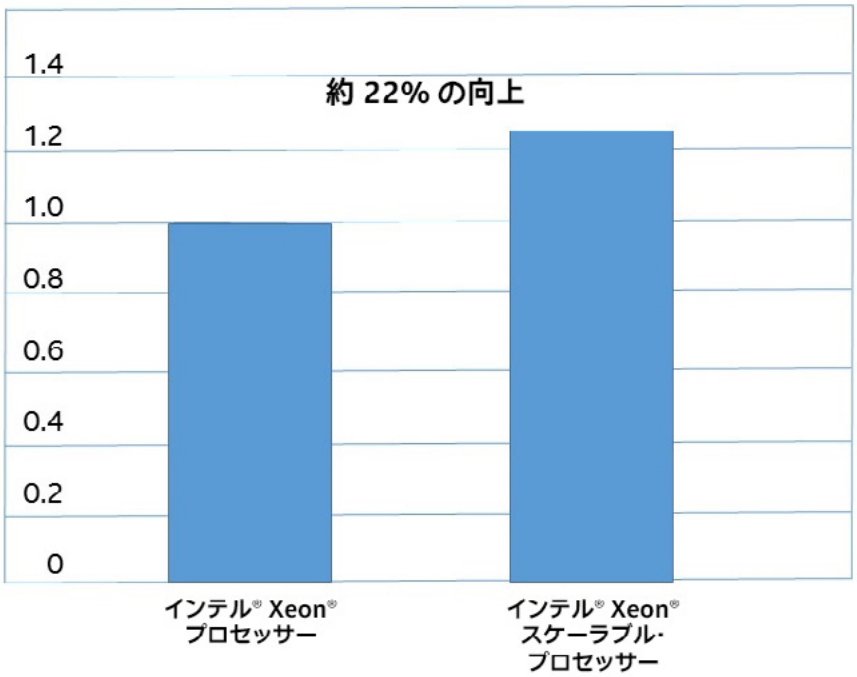
- 追加のプロセッサー・コア
- 大きな L2 キャッシュ
- 高速なメモリー
- インテル® Xeon® スケーラブル・プロセッサーで利用可能なメモリーチャネル数の増加による高メモリー帯域幅

表 1. 評価に使用したプロセッサー

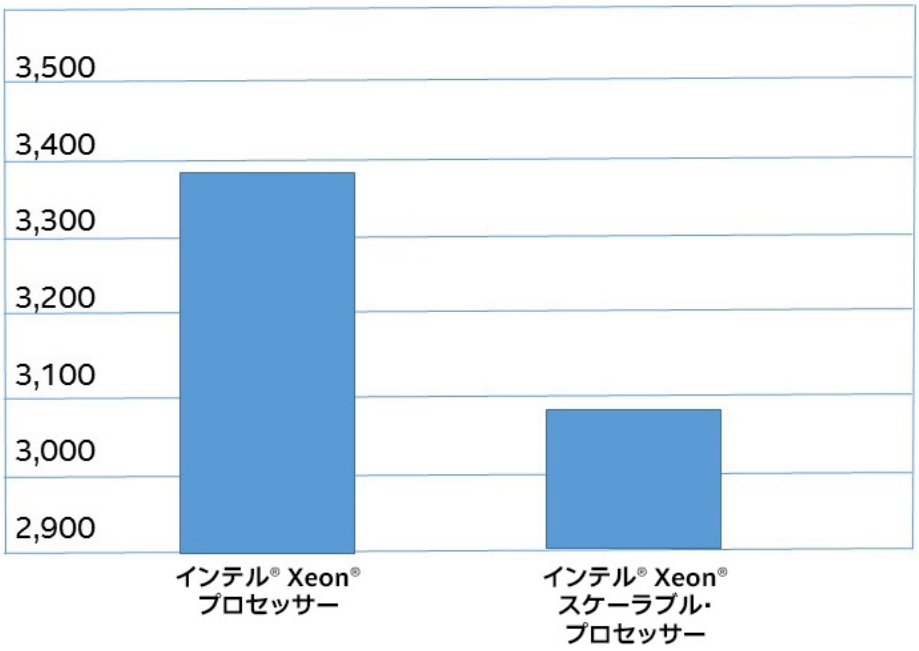
機能	インテル® Xeon® プロセッサー	インテル® Xeon® スケーラブル・プロセッサー
プロセッサー	インテル® Xeon® プロセッサー E5-2697 v4	インテル® Xeon® Gold 6148 プロセッサー
ソケットごとのコア数	18	20
周波数	2.30GHz	2.40GHz
キャッシュ (L2/L3)	256KB/45MB	1MB/27MB
メモリーサイズ	128GB	192GB
メモリー速度	2,400MHz	2,666MHz
メモリーチャネル	4	6

図 6 は、それぞれのプロセッサーの MPP 値です。予想どおり、インテル® Xeon® スケーラブル・プロセッサーのほうが、インテル® Xeon® プロセッサーよりも MPP 値が低く、計算パフォーマンスに優れていることは明らかです。ここで注目すべきことは、インテル® Xeon® スケーラブル・プロセッサーのほうがコア密度が高く、ノード内並列パフォーマンスを向上して、指定されたプロセッサー・コア数に対してコンパクトな並列クラスターを実現できるということです。

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。



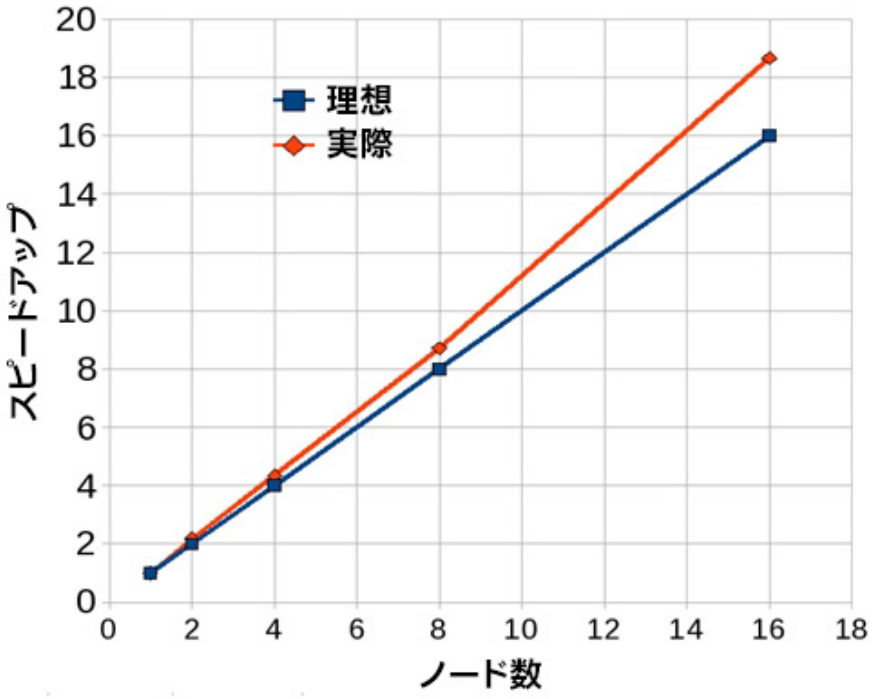
5 インテル® Xeon® プロセッサーとインテル® Xeon® スケーラブル・プロセッサーのパフォーマンスの比較



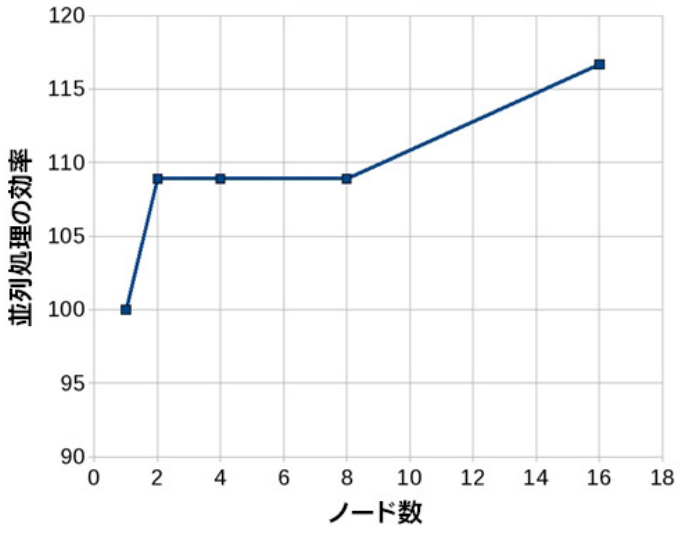
6 インテル® Xeon® スケーラブル・プロセッサーとインテル® Xeon® プロセッサーの MPP 値の比較

図 7 は、HiFUN ソルバーの理想的なスピードアップと実際のスピードアップを比較したグラフです。ノード数の増加に伴い、実際のスピードアップは理想的なスピードアップよりも高くなることが分かります。並列処理の効率が 2 つ以上のノードで超線形性を示す**図 8** でも同様に確認できます。これは、スケーラビリティーの評価中、問題が多数の部分に分割され、プロセッサー・コアごとのメモリー要件が軽減されることで、キャッシュ効率が向上するためです。しかし、一般にキャッシュ効率の向上は、問題が多数の部分に分割されることによる、プロセッサー・コア全体のデータ転送の増加に関連したオーバーヘッドによって相殺されます。

ここでは、優れたノード間インターコネクトと最適化された**インテル® MPI ライブラリー**により、HiFUN ソルバーのキャッシュ効率を向上し、超線形パフォーマンスを達成しています。



7 **マルチノードのインテル® プロセッサー・ベースのクラスターにおける HiFUN のスピードアップ**



8 マルチノードのインテル® プロセッサー・ベースのクラスターにおける HiFUN の並列処理の効率

HPC プラットフォームの性能を最大化

ここでは、最新世代のインテル® Xeon® スケーラブル・プロセッサーが、大きなキャッシュ、CPU あたりのコア密度の増加、高速なメモリー、メモリーチャネル数の増加による高メモリー帯域幅によって、HiFUN ソルバーのシングルノード・パフォーマンスを向上できることを示しました。コア密度が高くなると、ノード内の並列処理パフォーマンスが向上し、想定されたプロセッサー・コア数でよりコンパクトなクラスターを構築できます。インテル® Xeon® スケーラブル・プロセッサー・ベースの HPC プラットフォームにおける HiFUN ソルバーの超線形パフォーマンスは、優れたノード間インターコネクトと最適化されたインテル® MPI ライブラリーにより達成できます。

参考資料

1. N. Balakrishnan, "Parallel performance of the HiFUN solver on CRAY XC40," Technical Note No. CAd TN 2015:03, November 2015.
2. Manke J. W., "Parallel Computing in Aerospace," guest editorial, Parallel Computing, Volume 27, 2001, pp. 329-336.
3. George Karypis and Vipin Kumar, "A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Ordering of Sparse Matrices," Version 4.0, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN-55455, 1998.

インテル® MPI ライブラリー
柔軟で、効率良い、スケーラブルなクラスター・メッセージング
無料
ダウンロード

intel Software

将来に向けた取り組み

技術的なスキルアップ、
エキスパートからの回答、
新しい開発分野への挑戦。
これらすべてにインテルの
無料の技術ウェビナー
(オンデマンド) が役立ちます。

トピックを選択 (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 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

VASP 材料シミュレーション・パフォーマンスの向上

最新のインテル® ソフトウェア開発ツールを利用してハードウェアを効率良く使用する

Fedor Vasilev インテル コーポレーション CRT エンジニア
Dmitry Sivkov, PhD インテル コーポレーション DTC エンジニア
Jeongnim Kim, PhD インテル コーポレーション シニア HPC アプリケーション・エンジニア

材料のモデル化と設計は、科学および工学の非常に重要な分野です。合成材料は、現代の生活の一部であり、コンピューター、自動車、飛行機、道路、住宅、食品の材料は、材料科学の進歩の結果と言えます。ハイパフォーマンス・コンピューティング (HPC) は、現代の材料シミュレーションと設計を可能にします。そして、**インテル® ソフトウェア開発ツール**は、開発者とエンドユーザーがシミュレーションで最高のパフォーマンスを達成できるように支援します。

VASP (英語) は、第一原理から量子材料をモデル化するアプリケーションの 1 つです。計算パフォーマンスを考慮して設計されている HPC アプリケーションで、OpenMP* によりシステム内のすべてのコアを利用し、MPI により計算を HPC クラスター全体に分散します。

この記事では、VASP ワークロードのパフォーマンス・プロファイルとチューニングに、インテル® ソフトウェア開発ツールがどのように役立つかを見ていきます。

VASP のベンチマーク

VASP パフォーマンスの初期情報を取得するため、**インテル® Xeon® スケーラブル・プロセッサー**・ベースのシステムでテストを実施しました。(VASP は、オーストリアのウィーン大学が所有する著作権により保護されたソフトウェアであり、代表は物理学部 Dr. Georg Kresse 教授です。VASP の使用には適切なライセンスが必要です。)

パフォーマンスの評価では、MPI/OpenMP* ハイブリッド並列処理を実装する VASP デベロッパー・バージョン(「**Porting VASP from MPI to MPI+OpenMP [SIMD]** (MPI から MPI+OpenMP* [SIMD] への VASP の移植)」(英語)) で、**HSE 混成密度汎関数理論** (英語) を使用してバルクシリコンの空孔をシミュレーションします。

次のコマンドで VASP を起動します。

```
$ mpiexec.hydra -genv OMP_NUM_THREADS ${NUM_OMP} -ppn ${PPN} -n ${NUM_MPI} ./
vasp_std
```

説明 :

- **${NUM_OMP}** は、OpenMP* スレッドの数です。
- **${PPN}** は、各ノードの MPI ランクの数です。
- **${NUM_MPI}** は、現在のシミュレーションの MPI ランクの合計数です。

ベンチマークには、前処理 / 後処理を除くメインの計算の実行時間を測定する、**LOOP+** 時間の出力を使用します。

デュアルソケットの**インテル® Xeon® Gold 6148 プロセッサー** (2.40GHz、ソケットごとに 20 コア、ノードごとに 40 コア) の初期結果を**表 1** に示します。

表 1. インテル® Xeon® Gold プロセッサー・ベースのクラスター上での初期の LOOP+ と MPI_Allreduce 時間

ノード数	8	16	32
LOOP+ (秒)	194.01	115.77	75.52
MPI_Allreduce (秒)	25.78	5.44	24.81

システム構成 : ハードウェア : インテル® Xeon® Gold 6148 プロセッサー @ 2.40GHz、192GB RAM。インターコネクト : インテル® Omni-Path ホスト・ファブリック・インターフェイス (インテル® OP HFI) アダプター 100 シリーズ [discrete]。ソフトウェア : Red Hat* Enterprise Linux* 7.3、IFS 10.2.0.0.158、Libfabric 1.3.0、インテル® MPI ライブラリー 2018 (I_MPI_FABRICS=shm:ofi)、インテル® MPI Benchmarks 2018 (インテル® C++ コンパイラー 18.0.0 for Linux* でビルド)。ベンチマーク出典 : インテル コーポレーション

このデータをベースライン・パフォーマンスとして使用します。この記事では、**`MPI_Allreduce`** コードパスを使用します。**`MPI_Reduce`** を使用するデフォルトの手法にも同じアプローチを適用できます。

それでは、インテル® ソフトウェア開発ツールを利用して、アプリケーション・パフォーマンスを最適化する方法を見ていきましょう。

アプリケーション・パフォーマンス・スナップショットによるパフォーマンス・プロファイル

典型的な HPC アプリケーションは、さまざまなプログラミングおよび実行モデルで構成されており複雑です。現代のクラスター上で最適なパフォーマンスを達成するには、開発者は次のような複数の要因を考慮する必要があります。

- MPI および OpenMP* による並列処理
- メモリーアクセス
- FPU 使用率
- I/O

これらのいずれかのパフォーマンスが悪いと、アプリケーション全体のパフォーマンスが低下します。**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 (APS)** (英語) は、これらのパフォーマンスを素早く確認できる優れた機能です。スタンドアロン・ユーティリティーとして無料で利用することもできます。

「**Getting Started with APS (APS 入門)**」(英語) では、APS で MPI アプリケーションを起動する場合、次のコマンドを実行して MPI アプリケーションに関するデータを収集することを推奨しています。

```
$ <mpi launcher> <mpi parameters> aps <my app> <app parameters>
```

説明 :

- **<mpi launcher>** は、mpirun、srun、aprun などの MPI ジョブランチャーです。
- **<mpi parameters>** は、MPI ランチャーのパラメーターです。

注 :

- **apps** は、最後の **<mpi launcher>** パラメーターでなければなりません。
- **<my app>** は、アプリケーションの場所です。
- **<app parameters>** は、アプリケーションのパラメーターです。

APS は、アプリケーションを起動して、データ収集を実行します。解析が完了すると、**aps_result_<date>** ディレクトリーが作成されます。

インテル® MPI ライブラリーには、スタートアップ·プロセスに外部ツールを統合する 2 つの非常に便利なオプションがあります。

1. **aps**
2. **gtool**

これらについて見てみましょう。

-aps オプション

このオプションを使用すると、統計データを含む新しいフォルダーが生成されます : **aps_result_<date>-<time>**。aps ユーティリティーで収集したデータを解析できます。次に例を示します。

```
$ mpirun -aps -n 2 ./myApp
$ aps aps_result_20171231_235959
```

このオプションは、明示的に **aps** をターゲットとし、我々の目的にも合致しています。

-gtool オプション

このオプションは、**mpiexec.hydra** および **mpirun** コマンドを利用して、指定したプロセスに対してインテル® VTune™ Amplifier、**インテル® Advisor**、Valgrind*、GNU* デバッガー (GDB) などのツールを起動する場合に使用します。このオプションの代わりに、**I_MPI_GTOOL** 環境変数を使用することもできます。

-gtool オプションは、特定のプロセスの解析を容易にし、各引数セット (コロン '**:**' で区切ります) で解析ツールのコマンドを指定しなくても済むようにします。引数セット内で **-gtool** を使用することもできますが、一度に複数のセットで使用したり、2 つの解析手法 (**-gtool** と引数セット) を混在させないでください。

構文 :

```
-gtool "<command line for tool 1>:<ranks set 1>[=launch mode 1][@arch 1];
      <command line for tool 2>:<ranks set 2>[=exclusive][@arch 2]; …
      ;<command line for a tool n>:<ranks set n>[=exclusive][@arch n]"
      <executable>
```

または

```
$ mpirun -n <# of processes>
-gtool "<command line for tool 1>:<ranks set 1>[=launch mode 1][@arch 1]"
-gtool "<command line for a tool 2>:<ranks set 2>[=launch mode 2][@arch 2]" …
-gtool "<command line for a tool n>:<ranks set n>[=launch mode 3][@arch n]"
<executable>
```

表 2 に引数を示します。

表 2. 引数

<arg>	パラメーター
<rank set>	ツールの実行に関連したランクの範囲を指定します。ランクはカンマで区切るか、連続するランクのセットには '-' を使用します。すべてのランクに対してツールを実行するには、**all** 引数を使用します。[注 : 正しくないランク・インデックスを指定すると、警告が出力され、ツールは有効なランクの処理を続行します。]
[=launch mode]	起動モードを指定します (オプション)。利用可能な値は表 3 を参照してください。
[@arch]	ツールを実行するアーキテクチャーを指定します (オプション)。**<rank set>** を指定した場合、この引数を指定すると、指定したアーキテクチャーのホスト上の該当するプロセスに対してのみツールが起動されます。このパラメーターはオプションです。**[@arch]** の利用可能な値は、**I_MPI_PLATFORM** を参照してください。

ランクセットは、同じ **@arch** パラメーターではオーバーラップできません。**@arch** パラメーターを省略すると、異なるアーキテクチャーと見なされます。そのため、次の構文は有効であると見なされます。

```
-gtool "gdb:0-3=attach;gdb:0-3=attach@hsw;/usr/bin/gdb:0-3=attach@knl"
```

また、一部のツールは連携して動作できない場合や、同時に使用すると不正な結果になる場合があります。**表 3** は、**[=launch mode]** のパラメーター値のリストです。

表 3. [=launch mode] のパラメーター値

[=launch mode]	ツールの起動モード (オプション)。各ツールに対して、複数の値をカンマ ',' 区切りで指定できます。
exclusive	ツールがホストごとに 1 つのランクでのみ起動するようにします。
attach	ツールを **-gtool** から実行ファイルにアタッチします。デバッガーやデバッガーのようにプロセスにアタッチできるほかのツールを使用する場合、この値を指定する必要があります。このモードは、デバッガーでのみテストされています。
node-wide	ツールを **-gtool** から **<rank set>** が指定されているすべてのランク、またはすべてのランクの場合はすべてのノードに適用します。つまり、実行ファイルよりも上位 (**pmi_proxy** デーモン) にツールが適用されます。

これは、非常に強力で複雑な制御オプションです。「**インテル® MPI ライブラリー・デベロッパー・リファレンス**」(英語) に「**gtool options (gtool オプション)**」(英語) に関する章があります。ここでは、説明に必要なものにのみ注目します。コマンドライン・オプション に加えて、環境変数 (**I_MPI_GTOOL**) でも制御できるため、起動コマンドについてはここでは取り上げません。これは、実行前にすべての設定を行う複雑な起動スクリプトを使用する場合に非常に便利です。

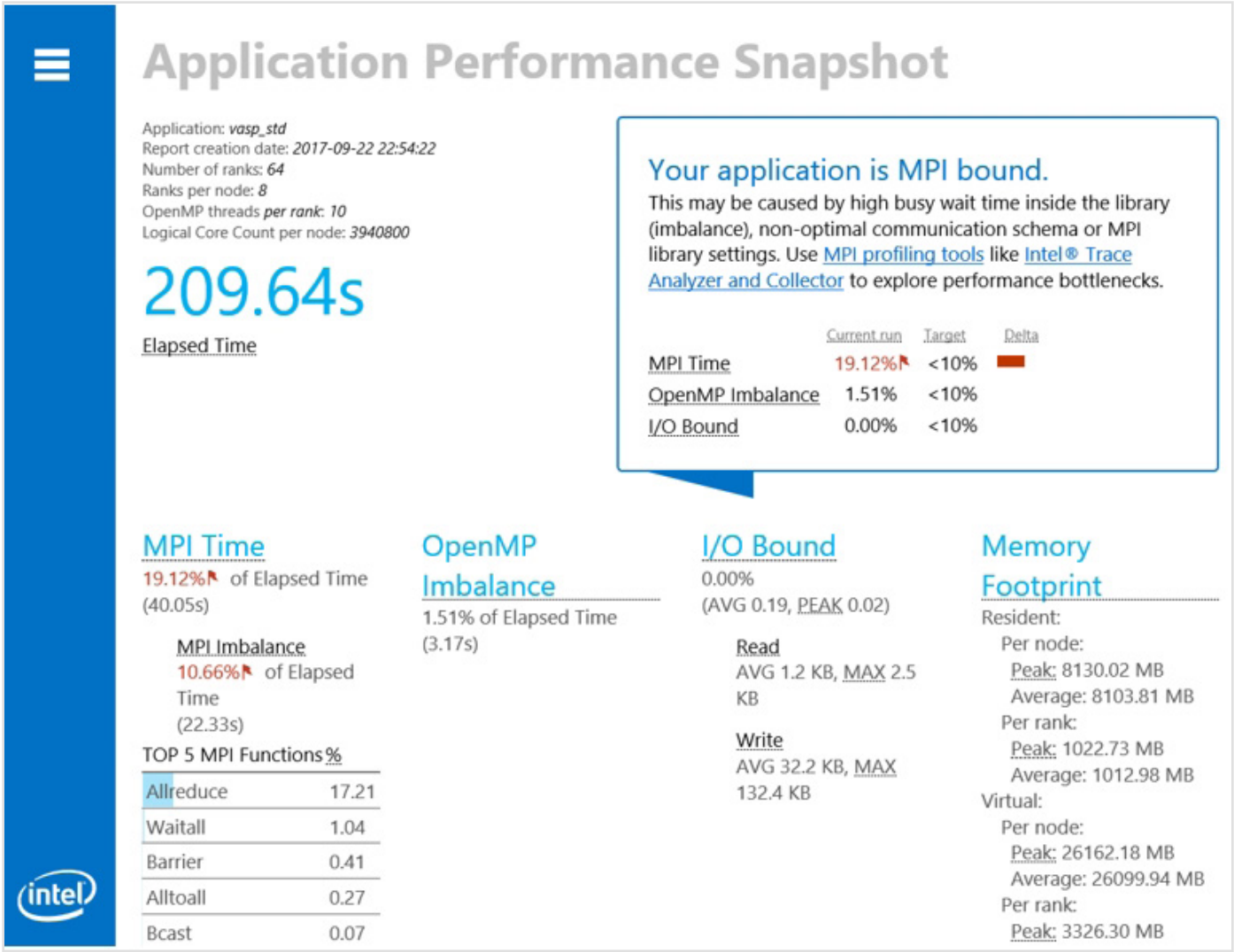
ここで使用する解析は単純なものなので、以下を設定するだけで済みます。

```
$ export I_MPI_GTOOL="aps:all"
```

そして、VASP のプロファイルを収集します。プロファイル名は **<name>** で、次のコマンドで実行できます。

```
$ aps -report=<name>
```

図 1 のような結果を示す HTML ファイルが生成されます。



1 VASP の APS 画面

APS は、パフォーマンス・ボトルネックを特定するためのアクション可能な情報を提供します。VASP は、MPI 依存であることが分かります (実行時間の 19.12% が MPI で費やされています)。詳細は、[MPI Time] セクションで確認できます。[MPI Imbalance] が経過時間の 10.66% を占めています。[MPI Imbalance] は、ほかのランクが **`MPI_Allreduce`** 呼び出しに到達するのを待機する時間など、MPI によって費やされた非生産的な時間です。[TOP 5 MPI Functions] セクションは、最も時間を費やしている上位 5 つの関数を示します。この実行では、**`MPI_Allreduce`** がトップで、パフォーマンスの最適化で最初にターゲットにすべきであることを示しています。

ここでは、最新のインテル® MPI ライブラリー 2019 とインテル® Performance Scaled Messaging 2 (インテル® PSM2) のマルチエンドポイント (Multi-EP) 機能を使用して **`MPI_Allreduce`** のスピードアップを試みます。さあ、作業に取り掛かりましょう。

インテル® MPI ライブラリーとインテル® Omni-Path アーキテクチャー

前述のシステム構成から、ここでは**インテル® Omni-Path アーキテクチャー (インテル® OPA)** (英語) を使用していることが分かります。インテル® MPI ライブラリーでインテル® Omni-Path アーキテクチャーのすべての機能を利用する最良の方法は、インテル® MPI ライブラリー 2019 用の Open Fabrics Interface (OFI) を使用することです。インテル® MPI ライブラリー 2019 テクニカルプレビューは、インテル® MPI ライブラリー 2018 Update 1 にバンドルされており、以下にインストールされます。

```
<install_dir>/compilers_and_libraries_2018.1.163/linux/mpi_2019/
```

TMI と OFI はともに、インテル® OPA を使用する最良のオプションとして、インテル® PSM2 プロバイダーをサポートしています。

```
$ export I_MPI_FABRICS=shm:ofi
$ export I_MPI_OFI_PROVIDER=psm2
```

インテル® PSM2 は、インテル® PSM の後継製品で、膨大な数の MPI ランクを持つ大規模なクラスター向けの API とライブラリーを提供します。「New Features and Capabilities of PSM2 (インテル® PSM2 の新機能)」で概要が紹介されています。

インテル® PSM2 マルチエンドポイント (Multi-EP) と インテル® MPI ライブラリー 2019

インテル® PSM2 (およびインテル® PSM) の重要な概念は、エンドポイントです。インテル® PSM2 は、ほかのエンドポイントとのメッセージの送受信をサポートするため、インスタンス化されたオブジェクト (またはハンドル) としてエンドポイントが定義される、エンドポイント通信モデルを採用しています。(詳細は**こちら** (英語) を参照してください。) デフォルトでは、各プロセスは 1 つのエンドポイントのみ使用できます。ただし、インテル® PSM2 の最新バージョンでは、マルチエンドポイント (Multi-EP) 機能により、マルチスレッド・アプリケーションでエンドポイントを利用できるようになりました。インテル® MPI ライブラリー 2019 テクニカルプレビューは、開発者がインテル® PSM2 マルチエンドポイントを効率良く利用できるようにします。マルチエンドポイントをサポートするため、MPI 標準スレッド化モデルが **`MPI_THREAD_SPLIT`** プログラミング・モデルで拡張されました。これにより、プログラムで MPI とマルチスレッドを効率良く利用して、ほとんどの同期を排除し、パフォーマンスを向上できます。

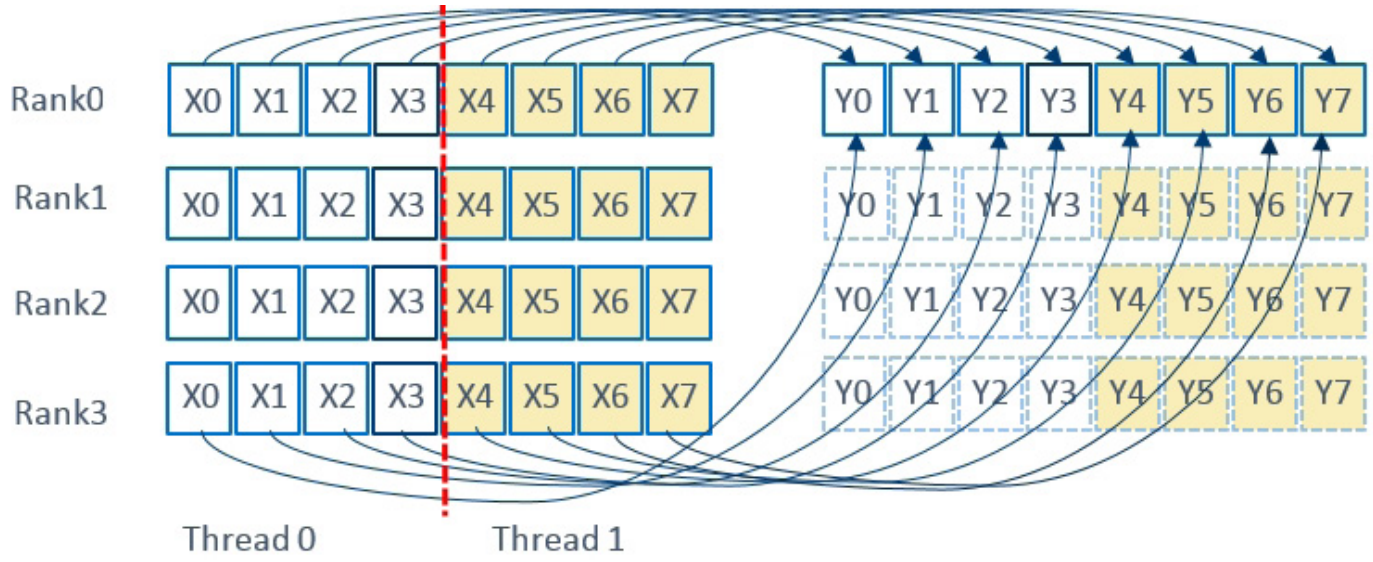
「**インテル® MPI ライブラリー・デベロッパー・リファレンス for Linux***」(英語) の説明にあるように、**MPI_THREAD_SPLIT** 準拠のプログラムは、少なくともスレッド対応の MPI プログラム (**MPI_THREAD_MULTIPLE** スレッドレベルをサポート) でなければなりません。さらに、以下の規則が適用されます。

- **同じプロセスの異なるスレッド**は、同じコミュニケーターを同時に使用できません。
- **スレッドが作成した要求**にほかのスレッドはアクセスできません。つまり、すべての非ブロッキング操作は、同じスレッドにより完了されるか、完了を確認されるか、調査されなければなりません。
- スレッドから呼び出される **MPI_Wait()** や **MPI_Test()** などの操作の進行状況を示す**通信完了呼び出し**は、ほかのスレッドの進行状況は保証しません。

また、すべてのスレッドに **thread_id** 識別子が必要です。MPI ランク間の通信は同じ **thread_id** のスレッドに対してのみ行うことができます。

集合操作でのマルチエンドポイント : MPI_Allreduce

VASP にはすでに MPI/OpenMP* ハイブリッド・バージョンがあるため、MPI マルチエンドポイントを容易に適合させることができます。MPI ランクごとに、**MPI_Comm_dup** ルーチンを使用して OpenMP* スレッドの数と同数の MPI コミュニケーターを作成します。そうすると、**図 2** に示すように、各 MPI バッファーが OpenMP* スレッド間で分割されます。



2 スレッド間での MPI バッファーの分割

すべてのコード処理をスキップするため、**PMPI_** インターフェイスを使用して、**MPI_Allreduce** ルーチンに必要な変更を加えます。

```
/*
*Copyright 2018 Intel Corporation.
*This software and the related documents are Intel copyrighted materials, and
your use of them is governed by the express license under which they were
provided to you (License).
*Unless the License provides otherwise, you may not use, modify, copy, publish,
distribute, disclose or transmit this software or the related documents without
Intel's prior written permission.
*This software and the related documents are provided as is, with no express or
implied warranties, other than those that are expressly stated in the License.
*/

#include <stdio.h>
#include <mpi.h>
#include <omp.h>
#include <unistd.h>

int comm_flag=0;
int new_comm=0;
int first_comm=0;
int init_flag=0;
int mep_enable=0;
int mep_num_threads=1;
int mep_allreduce_threshold=1000;
int mep_bcast_threshold=1000;
int k_comm=0;

void mep_init() {
   if(init_flag == 0) {
      mep_enable=atoi(getenv("MEP_ENABLE"));
      mep_num_threads=atoi(getenv("MEP_NUM_THREADS"));
      mep_allreduce_threshold=atoi(getenv("MEP_ALLREDUCE_THRESHOLD"));
      mep_bcast_threshold=atoi(getenv("MEP_BCAST_THRESHOLD"));
   }
   init_flag=1;
}
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) {
   int err;
   int count_thread;
   int rest=0;
   int num_threads=1;
   int thread_num=0;
   MPI_Aint lb,extent;
   size_t datatype_size;
   if (init_flag == 0) mep_init();
```

```
   MPI_Type_get_extent(datatype, &lb, &extent);
   datatype_size=extent;

   // MPI_Allreduce が MPI_IN_PLACE を使用する場合
   if (&sendbuf[0] == MPI_IN_PLACE) {
      if(mep_enable == 1) {
         if(count >= mep_allreduce_threshold) {
            comm_duplicate(comm,mep_num_threads);
            count_thread=count/mep_num_threads;
            rest=count%mep_num_threads;

#pragma omp parallel num_threads(mep_num_threads)
                     private (num_threads, thread_num) {
   num_threads=omp_get_num_threads();
   thread_num=omp_get_thread_num();
   if(thread_num == num_threads-1) {
      err = PMPI_Allreduce(MPI_IN_PLACE,
                           recvbuf+thread_num*count_thread*datatype_size,
                           count_thread+rest,
                           datatype, op, comms.dup_comm[k_comm][thread_num]);
   }
   else {
      err = PMPI_Allreduce(MPI_IN_PLACE,
                           recvbuf+thread_num*count_thread*datatype_size,
                           count_thread,
                           datatype, op, comms.dup_comm[k_comm][thread_num]);
   }
}
   }
   else {
      err = PMPI_Allreduce (MPI_IN_PLACE, recvbuf, count, datatype, op, comm);
   }
}
else {
   err = PMPI_Allreduce (MPI_IN_PLACE, recvbuf, count, datatype, op, comm);
}
   }

   // そうでない場合、MPI_Allreduce は MPI_IN_PLACE を使用しない
   else {
      if(mep_enable == 1) {
         if(count >= mep_allreduce_threshold) {
            comm_duplicate(comm,mep_num_threads);
            count_thread=count/mep_num_threads;
            rest=count%mep_num_threads;
#pragma omp parallel num_threads(mep_num_threads)
                     private (num_threads,thread_num) {
   num_threads=omp_get_num_threads();
   thread_num=omp_get_thread_num();
   if(thread_num == num_threads-1) {
```

```
      err = PMPI_Allreduce(sendbuf+thread_num*count_thread*datatype_size,
                           recvbuf+thread_num*count_thread*datatype_size,
                           count_thread+rest,
                           datatype, op, comms.dup_comm[k_comm][thread_num]);
   }
   else {
      err = PMPI_Allreduce(sendbuf+thread_num*count_thread*datatype_size,
                           recvbuf+thread_num*count_thread*datatype_size,
                           count_thread,
                           datatype, op, comms.dup_comm[k_comm][thread_num]);
   }
}
   }
   else {
      err = PMPI_Allreduce (sendbuf, recvbuf, count, datatype, op, comm);
   }
}
else {
   err = PMPI_Allreduce (sendbuf, recvbuf, count, datatype, op, comm);
}
}
return err;
     }
```

comm_duplicate(comm,mep_num_threads) は、MPI コミュニケーターを複製する関数です。**LD_PRELOAD** とともに使用して、デフォルトの **MPI_Allreduce** を置き換えます。

マルチエンドポイントでの実行

変更後のコマンドは、次のようになります。

```
$ export I_MPI_THREAD_SPLIT=1
$ export I_MPI_THREAD_RUNTIME=openmp

$ export MEP_ALLREDUCE_THRESHOLD=1000
$ export MEP_ENABLE=1
$ export OMP_NUM_THREADS=2
$ export MEP_NUM_THREADS=2

$ LD_PRELOAD= lib_mep.so
$ mpiexec.hydra -ppn ${PPN} -n ${NUM_MPI} ./vasp_std
```

表 4 にパフォーマンス結果を示します。

表 4. インテル® Xeon® Gold プロセッサー・ベースのクラスターにおける最終の LOOP+ と MPI_Allreduce 時間 (OpenMP* x MPI の最良の組み合わせ) と初期データ (OpenMP* x MPI の最良の組み合わせ) の比較

ノード数	8	16	32
LOOP+ (秒)	181.55	102.07	65.35
MPI_Allreduce (秒)	14.71	10.88	13.45
LOOP+ スピードアップ	1.07	1.13	1.16
MPI_Allreduce スピードアップ	1.75	2.34	1.84

パフォーマンスの向上

APS を使用することで、**`MPI_Allreduce`** がアプリケーションのパフォーマンス "hotspot" であることが分かりました。新しいインテル® MPI ライブラリー 2019 テクニカルプレビューとインテル® OPA により、システム・ハードウェアを効率良く利用して、**`MPI_Allreduce`** をスピードアップし、アプリケーション全体のパフォーマンスを向上できました。ソフトウェアの変更のみによってもたらされた最大スピードアップは、**`MPI_Allreduce`** のみで 2.34 倍、**`LOOP+`** では 1.16 倍でした。

BLOG HIGHLIGHTS

デバイス・ソフトウェア開発者がプロトタイプから素早く製品化できるようにするには

DINYAR D. (INTEL CORPORATION)

組込みデバイスの開発は、センサー、計算処理、メモリー、および通信テクノロジーの急速な進化に伴い、複雑さを増しています。この記事では、タイムラインの短縮、リソース不足、品質とパフォーマンスの最適化に対する厳しい要件に対応するため、デバイス向けソフトウェア開発のいくつかの課題に注目します。

これらの課題に取り組むため、インテルのエッジツークラウド・エコシステムに支えられた強力なソリューションを利用できます。

この記事の続きはこちら (英語) でご覧になれます。>

Software

THE PARALLEL UNIVERSE

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

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

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いただくか、http://www.intel.co.jp/ を参照してください。

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

インテルは、本資料で参照しているサードパーティーのベンチマーク・データまたは Web サイトの設計や実装について管理や監査を行っていません。本資料で参照している Web サイトまたは類似の性能ベンチマーク・データが報告されているほかの Web サイトも参照して、本資料で参照しているベンチマーク・データが購入可能なシステムの性能を正確に表しているかを確認されるようお勧めします。

この文書および情報は、インテルのお客様向けの参考情報として記載されているものであり、現状のまま提供され、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、知的財産権の非侵害性への保証を含みますが、これらに限定されるものではありません。本資料は、本資料に記述、表示、または記載された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。インテル製品は、医療、救命、延命措置、重要な制御または安全システム、核施設などの目的に使用することを前提としたものではありません。

© 2018 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Core、Xeon、Intel Xeon Phi、VTune 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 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/1805/PDF/XL/SSG/SS