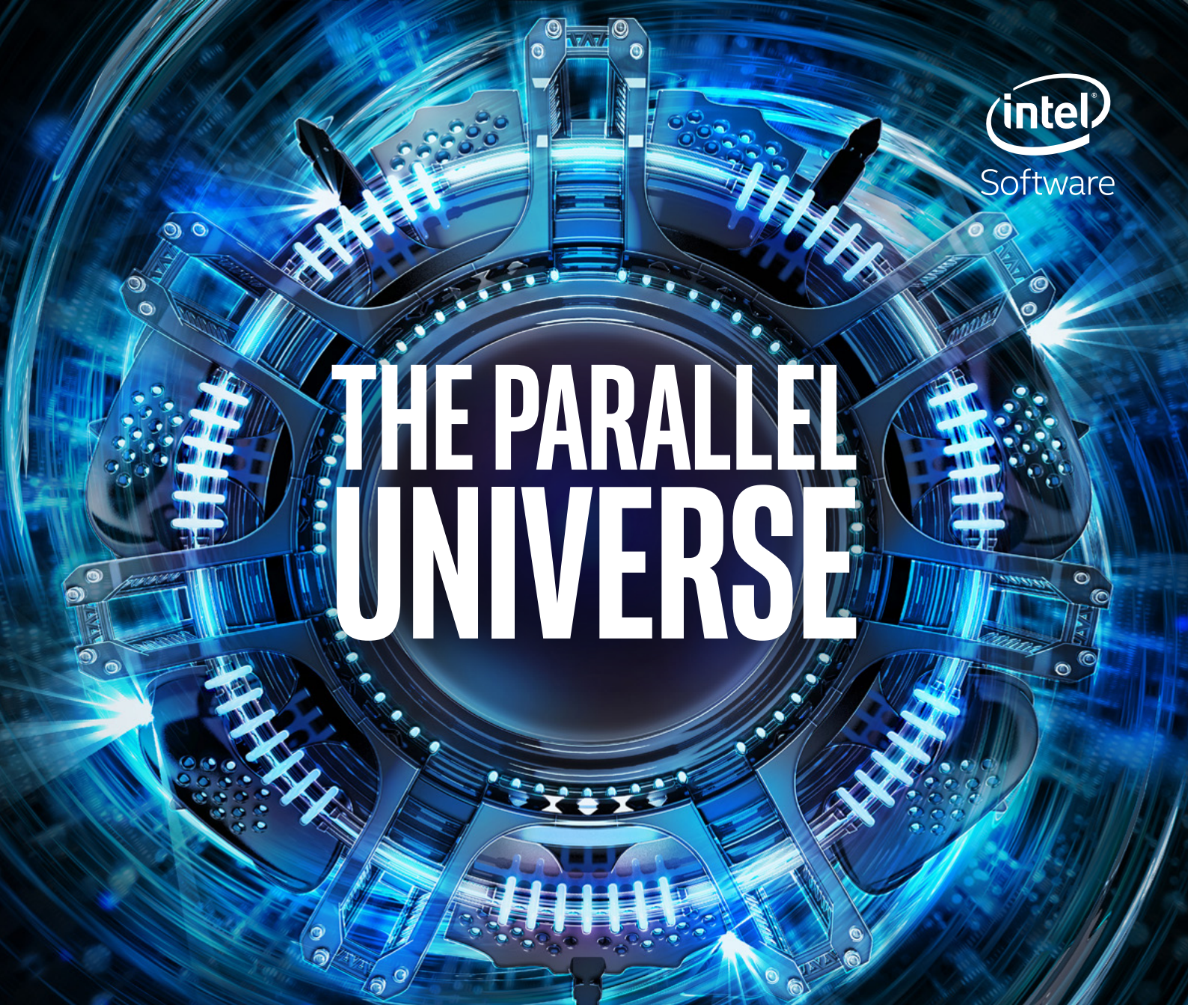


OpenCL* による FPGA プログラミング

自動運転ワークロードに適した Eigen 数学ライブラリーの構築

インテル® MKL のベクトル化された圧縮行列関数による
代数計算の高速化

Issue
31
2018

目次

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 / 共著者であり、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。

2018 年の並列計算

2018 年最初の The Parallel Universe へようこそ。年初にあたり、今後のハードウェアとソフトウェアの傾向についていくつかの予測を行うことを考えましたが、私はコンピューティングの先見者というよりもむしろファストフォロワーです。結局のところ、私の専門はバイオテクノロジーとデータサイエンスであり、コンピューター・サイエンスではありません。前号で、私はヘテロジニアス並列コンピューティングの将来に関して、それほど大胆ではない予測を行いました。私は 2009 年からヘテロジニアス並列処理に関して気にかけていましたが、真の先見者はその何年も前から考えていました。皆さんの周りのあらゆるところで新しい傾向が見られるとしたら、それはもはや予測ではありませんが、FPGA は、ヘテロジニアスへと向かう進化の次のフェーズに達したと言って良いでしょう。

間もなく FPGA はマルチコア・プロセッサーのようにあたりまえになります。しかし、多くは FPGA を効率的にプログラムできていません。そこで、**The Parallel Universe** の創刊者兼編集者であった James Reinders に、FPGA プログラミングに関する記事の執筆を依頼しました。**前号**では、FPGA プログラミングについてソフトウェア開発の視点から述べました。この号では、James とインテルのプログラマブル・ロジック・グループの Tom Hill による、**OpenCL* による FPGA プログラミング**に取り組むための、詳細な解説を紹介します。OpenCL* (Open Computing Language) は、「ヘテロジニアス・システムにおける並列プログラミングのためのオープン規格」です。我々にとって、FPGA プログラミングで最初に取り組むのは OpenCL* です。

2018 年も引き続き、さまざまなプログラミング・ツールとプログラミング・モデルを紹介していきます。この号の 2 つの記事、「**インテル® MKL のベクトル化された圧縮行列関数による代数計算の高速化**」および「**インテル® Advisor の Python* API によりパフォーマンスの詳細を得る**」では、インテル® ソフトウェア開発ツールの新しい機能を取り上げています。1 つ目の記事は、小さな行列の大きなグループを計算するアプリケーションのパフォーマンスを向上するために設計された新しいデータ形式について説明します。2 つ目の記事は、アプリケーション・パフォーマンスのカスタム解析を行ったりカスタム視覚化を作成する、インテル® Advisor データベースに直接アクセスする新しい API について説明します。

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。

Java* は世界で最もポピュラーなプログラミング言語の 1 つですが、The Parallel Universe ではこれまであまり取り上げていませんでした。2018 年は、この状況が変化することになるでしょう。**インテル® Parallel Studio XE** は Java* チューニング･サポートを改良しており､Java* JVM はベクトル計算のサポートを強化しています。「**ビッグデータ・アプリケーションで Java* のパフォーマンスを向上する**」では、後者の拡張を説明します。

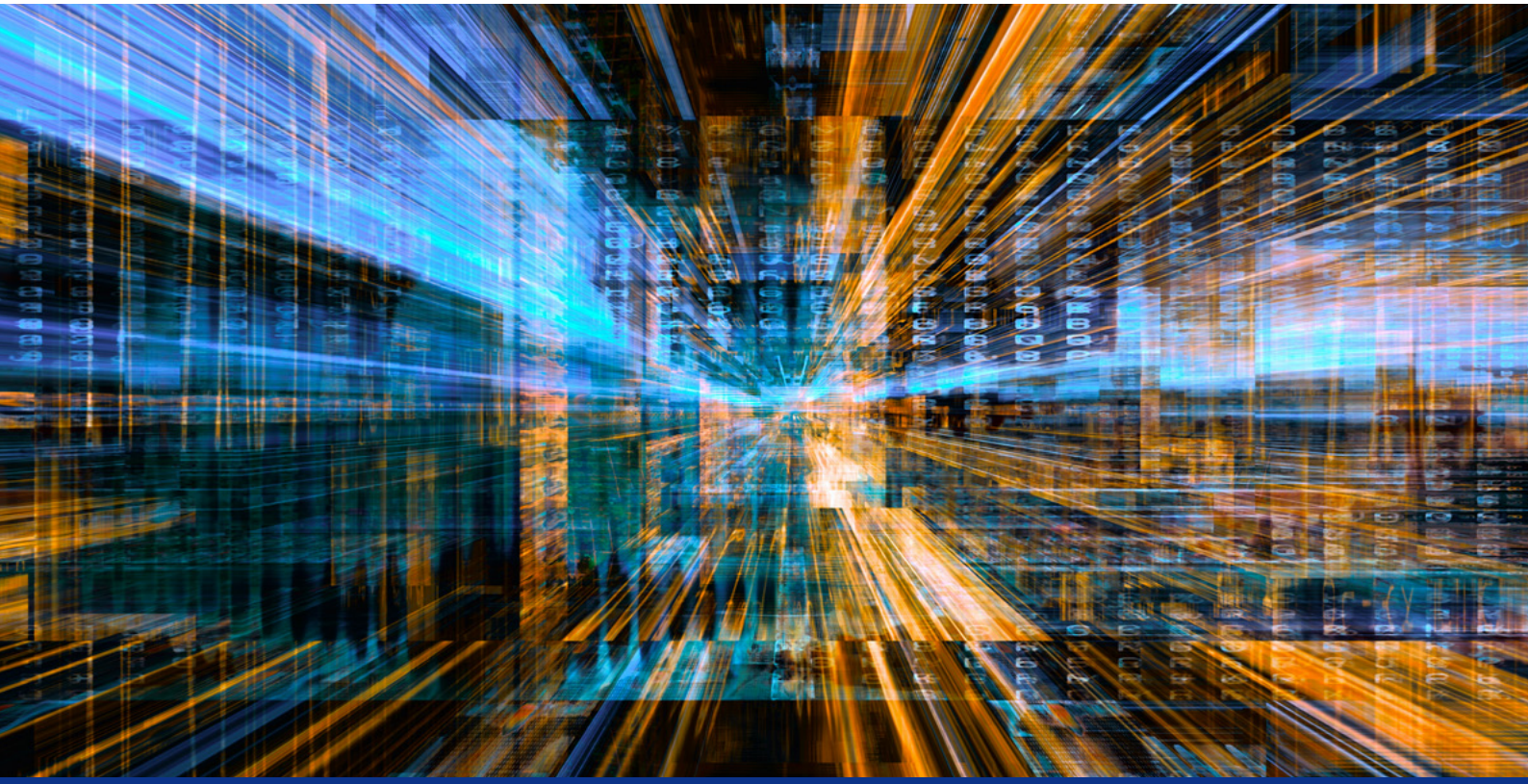
人工知能 (AI) は 2018 年もホットな話題の 1 つです。自動運転は AI の進歩により現実のものとなりましたが、その処理にはハイパフォーマンス･コンピューティングが必須です。「**自動運転ワークロードに適した Eigen 数学ライブラリーの構築**」では、重要な計算カーネルのパフォーマンスを向上する方法を説明します。最後に、「**インテル® AI Academy へようこそ**」では、AI 教育、ツール、テクノロジー向けの、インテルの新しい包括的なプログラムの概要を紹介します。

今後の The Parallel Universe では、新しい不揮発性メモリーの効率的な使用方法、NUMA (Non Uniform Memory Access) アーキテクチャー向けのコードのチューニング、Python* や R のような生産性言語のベスト・プラクティス (および Go* や Julia* の記事) 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。

2018 年も、ソフトウェア開発の将来に関する情報を皆さんにお届けできることを楽しみにしています。

Henry A. Gabb
2018 年 1 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。



OpenCL* による FPGA プログラミング

FPGA のプログラム方法の理解と実践

James Reinders James Reinders Consulting LLC コンサルタント & HPC エンスージアスト
Tom Hill インテル コーポレーション プログラマブル・ロジック・グループ DSP 製品ライン・マネージャー

FPGA が注目されている理由

フィールド・プログラマブル・ゲート・アレイ (FPGA) は、低レイテンシーで高いパフォーマンスと電力効率を実現できる優れたハードウェアです。これらの長所は、FPGA の大規模な並列機能と再構成の容易性により実現されます。FPGA は、次のような処理を行うことができる、再構成可能なチャネルレス型の構造を提供します。

- カスタム・ハードウェア・アクセラレーターを**設計**する。
- 単一アプリケーションに**配置**する。
- 異なるアプリケーションの新しいアクセラレーターとしてデバイスを**素早く再構成**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。

この記事では、独自の設計を行う方法と、それらの設計を使用してアプリケーションを高速化する方法を説明します。ハードウェア・エンジニアは、さまざまなアプリケーションで ASIC の代わりに FPGA を長年使用してきました。これまで、FPGA の構成 (プログラミング) は、Verilog* や VHDL* のような高水準定義言語を利用して行われてきました。これらの設計手法はハードウェア・エンジニアにはよく知られていますが、ソフトウェア開発者にとっては全く新しいものです。

OpenCL* を中心とする新しいツールは、このギャップを埋め、FPGA ハードウェア・プラットフォームの恩恵をソフトウェア開発者にもたらします。これらの新しいツールを使用すると、C のような構文で記述されたカスタム・アルゴリズムを高速で電力効率の良いハードウェアに変換できる、高度に構成可能なデバイスとして FPGA を活用することができます。これらのソリューションの高い柔軟性とパフォーマンスは、最新世代の FPGA デバイスのおかげです。

大規模な並列処理

インテル® Stratix® 10 FPGA は、最大 10 TFLOPS の単精度浮動小数点パフォーマンスを提供します。また、標準数値形式に制限されません。例えば、2 ビットのディープ・ニューラル・ネットワークの実装は困難でしたが、2017 年 11 月にスーパーコンピューター上で行われたデモで紹介された、インテルのディープラーニング・アクセラレーターの構成可能なワークロードを利用すると簡単に実装できます。

インテル® FPGA は、大規模な並列処理とハードウェアの柔軟性によりレイテンシーを軽減します (**図 1**)。FPGA はデータフロー (またはパイプライン) の構成がプログラムされるため、データの移動が最適化され、ホスト CPU と FPGA 間のデータ転送は排除されます。

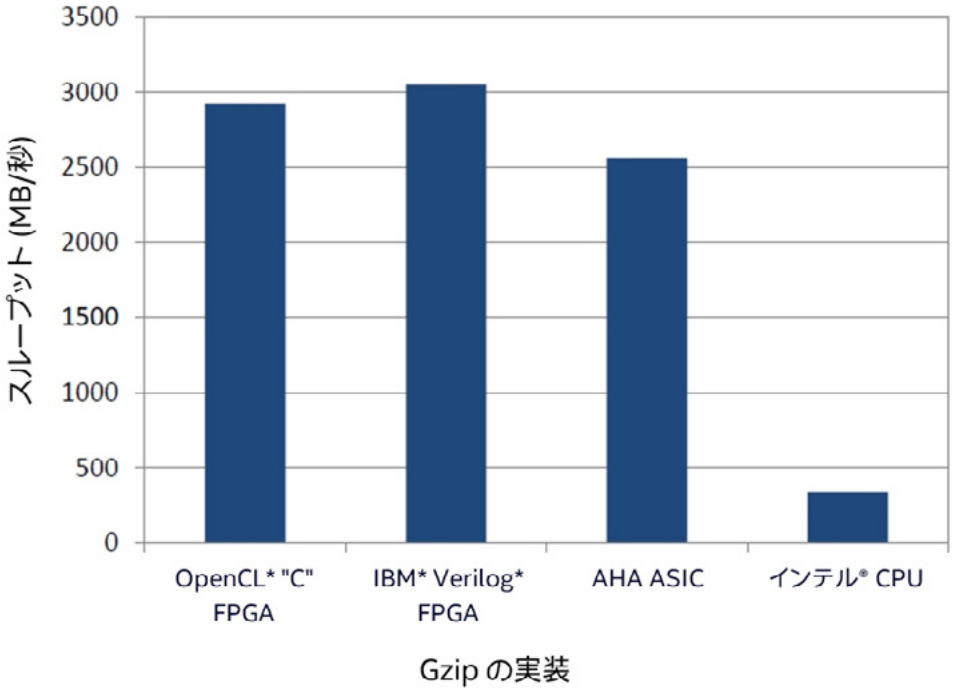
注意すべき点

現在では、C プログラマーが FPGA プログラミングを試してみることを妨げるものは何もありません。ただし、OpenCL* (新しい開発環境と新しい手法を含む) を使用して FPGA のプログラミングを行う場合も、ほかのすべてのプログラミングと同様に、学習曲線が存在します。幸い、オンライン・トレーニングと OpenCL* コーディング・ガイドを利用することで、簡単に FPGA プログラミングに取り組むことができます。

FPGA の開発環境

OpenCL* ソフトウェア開発環境は Eclipse* または Visual Studio* ベースであるため、C プログラマーには馴染み深いものです。この記事では、OpenCL* で最初のプログラムを記述する方法と、そのプログラムを FPGA で実行する方法を説明します。

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 FPGA への対応が進む OpenCL*: 非常に生産的な OpenCL* FPGA プログラミングと、生産性が低く、アプローチの種類が少ない 3 つの従来のプログラミングを使用した結果の比較。(この記事の最後の**関連情報**セクションの「Gzip on a Chip」を参照。)

FPGA の最適化

新しいデバイスとプログラミング手法には、最高のパフォーマンスを達成するさまざまな秘訣があります。これらの最適化は、アルゴリズムのデータ構造組織やクリーン思考などのよく知られているアプローチから、OpenCL* でパイプラインを表現する最も効果的な方法までさまざまです。よくある状況ではありませんが、CPU のインライン・アセンブリーを使用するのと同じように、ハードウェア記述言語プログラミング (Verilog* または VHDL*) も使用可能です。FPGA の最適化は CPU や GPU の最適化とは異なりますが、簡単でも難しくもありません。

CPU や GPU と同様に、FPGA プログラマーは FPGA 向けに記述された (ハードウェア・エンジニアが通常 IP ブロックと呼ぶ) ライブラリーを使用できます。 (FPGA の最適化についてさらに詳しく知りたい方は、この記事の次に、**インテル® FPGA SDK for OpenCL* ベスト・プラクティス・ガイド**をお読みになることを推奨します。)

FPGA プログラミングを試す 3 つのアプローチ

FPGA プログラミングを試す 3 つの方法について説明します。

- **エミュレーション**。FPGA ハードウェアを購入することなく FPGA プログラミングを試すことができます。FPGA プログラムを開発およびデバッグする際に推奨する最初のステップです。SDK は Windows* または Linux* で利用できます。
- **オフラインコンパイル**。エミュレーションを利用してデバッグした後、FPGA なしで OpenCL* 環境を使用してコンパイルを行い、FPGA プログラミング・ファイルを生成します。OpenCL* カーネルの正確なエリアレポートとパフォーマンス・レポートを調べて、FPGA 上で実際に実行する前にコードを改良します。
- **クラウド**。クラウドの **Nimbix*** (英語) システムでブラウザーからツールと FPGA を使用して、エミュレーションから FPGA 上での実行まで、すべての開発を行うことができます。現在の構成は、インテル® ツールのみを含む CPU ベースのマシン、ツールとインテル® Arria® 10 FPGA を含むマシンです。コストは、プログラム開発が 1 時間あたり 36 セントから、FPGA を搭載したマシンの使用が 1 時間あたり 3 ドルからです。Nimbix* システムは Linux* を実行します。

ターゲット・プラットフォームとして FPGA 開発ボードを使用することもできます。最新の FPGA のすべての機能にアクセスできます。1 枚の PCIe* カードのコストは、**インテル® Arria® 10 FPGA** を搭載した場合は約 5,000 US ドル、**インテル® Stratix® 10 FPGA** を搭載した場合は約 11,000 US ドルになります。技術的には、**Cyclone®** FPGA 開発ボードで OpenCL* を使用することも可能です。**DE1-SoC ボード** (英語) はわずか 250 US ドルと非常に魅力的です。しかし、**表 1** を見れば分かるように、このボードは計算の高速化に適しているとはお世辞にも言えません。FPGA 上で OpenCL* を使用した計算の高速化を 250 US ドル以下で試すには、上記のオプション 1、2、3 の組み合わせを推奨します。ここでは、あえて DE1-SoC ボードでインテル® FPGA SDK for OpenCL* を使用しました。問題なく動作しましたが、予想通り計算能力は非常に低いものでした。

表 1. OpenCL* をサポートしているインテル® FPGA プラットフォーム (一部)

プラットフォーム	製品番号	最大ロジック エレメント数 (百万)	最大エンベデッド・メモリー (M ビット)	最大ピーク TFLOPS (単精度、IEEE)
Cyclone® V SoC 開発ボード	**DE1-SoC ボード** (英語) (Cyclone® V - 5CSEMA5F31C6N)	0.085	3.9	0.001 未満
インテル® Arria® 10 FPGA 開発キット (英語)	**DK-DEV-10AX115S-A** (英語)	1.1	53	1.4
インテル® Stratix® 10 FPGA 開発キット (英語)	**DK-DEV-1SGX-H-0A** (英語)	5.5	229	9.2
Nimbix* クラウド	**BittWare A10PL4 ボード** (英語) および **インテル® Arria® 10 GX 1150** (英語)	1.1	53	1.4

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

インテル® Arria® 10 FPGA ファミリーおよびインテル® Stratix® 10 FPGA ファミリーはアプリケーションの高速化を、Cyclone® FPGA ファミリーは低消費電力の組込み機器での利用を、それぞれ想定しています。OpenCL* は、インテル® Arria® 10 FPGA とインテル® Stratix® 10 FPGA の両方でサポートされています。また、一部の Cyclone® V SoC FPGA でもサポートされています。どのボードを選んだ場合でも、インテル® FPGA SDK for OpenCL* が特定のシステム構成を適切にターゲットするために使用する、ボード・サポート・パッケージ (BSP) が必要です。

なぜ OpenCL* なのか

OpenCL* は、ヘテロジニアス・システムのアルゴリズムを高速化する C を拡張した標準規格であり、プログラミング・モデルです。CUDA* のように、OpenCL* の中心となるのは、デバイス (ハードウェア・アクセラレーター) 上で実行するカーネルの概念です。カーネルは C99 のサブセットで記述されています。CUDA* とは異なり、OpenCL* は可搬性がある、ロイヤルティー・フリーのオープンソース標準規格であり、CPU、GPU、および FPGA を含むあらゆる計算デバイスをサポートするように設計されています。

OpenCL* には、ホスト (CPU) が PCIe* 経由でデバイス (今回のケースでは FPGA) と通信するため、またはカーネルがホストを介することなく別のカーネルと通信するためのアプリケーション・プログラミング・インターフェイス (API) も用意されています。これに加えて、インテルは、10Gb イーサネットなどのストリーミング I/O インターフェイスからカーネルへデータを直接送る I/O チャネル API も提供しています。

OpenCL* プログラミング・モデルでは、ユーザーがコマンドキューにタスクをスケジュールします。各デバイスには少なくとも 1 つのコマンドキューがあります。OpenCL* ランタイムは、データ並列タスクを分割して、デバイスのプロセシング・エレメントに送ります。これがホストが任意のデバイスと通信する方法です。ベンダー固有の実装を行わないようにすることは、個々のデバイスベンダーの責任です。**インテル® FPGA SDK for OpenCL*** (英語) は固有の実装を行わず、OpenCL* 1.2 標準規格に準拠しています。

インテル® FPGA SDK for OpenCL*

インテル® FPGA SDK for OpenCL* を使用すると、ユーザーは、従来のハードウェア FPGA 開発フローではなく、より迅速で高レベルのソフトウェア開発フローを利用できます。OpenCL* のアクセラレーター・コードを、x86 ベースのホストで数秒エミュレートして、特定のアルゴリズムのパイプライン依存関係に関する情報を含む詳細な最適化レポートを取得できます。このレポートにより、長いコンパイル時間をかけて FPGA 上で実行する最終的なコードを生成する前に、プログラムをデバッグ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。

インテル® FPGA SDK for OpenCL* は、FPGA 設計の複雑さを排除することにより開発を促進します。ソフトウェア・プログラマーは、ANSI C ベースの言語である OpenCL* C と追加の OpenCL* 構文を使用して、ハードウェア・アクセラレーションを活用したカーネル関数を記述できます。

インテルは、SDK の一部として、通常ソフトウェア・プログラマーが利用する開発フローに近いツールのスイートを提供します。

- **エミュレーター**。x86 CPU 上でコードをステップ実行し、コードが機能的に正しいことを保証します。
- **詳細な最適化レポート**。内部ループのロードとストアの依存関係を理解するのに役立ちます。
- **プロファイラー**。適切なメモリー結合とストールフリーのハードウェア・パイプラインを保証するためカーネルのパフォーマンス情報を提供します。
- **OpenCL* コンパイラー**。カーネルコードに 300 を超える最適化を行い、1 ステップで FPGA イメージ全体を生成できます。

インテル® FPGA SDK for OpenCL* は、**インテル® Xeon® プロセッサー**、SoC の組込み **ARM* Cortex*-A9 プロセッサー・コア** (英語)、および **IBM* Power Systems* プロセッサー** (英語) を含む、さまざまなホスト CPU をサポートしています。複数の FPGA と複数のボードにおけるスケーラブルなソリューションに加えて、DDR SDRAM (シーケンシャル・メモリー・アクセス)、QDR SRAM (ランダム・メモリー・アクセス)、内部 FPGA メモリー (低レイテンシー・メモリー・アクセス) などのさまざまなメモリーターゲットをサポートしています。半精度、単精度、および倍精度浮動小数点演算もサポートしています。

FPGA プログラミングを試す

では、FPGA プログラミングを試す 3 つの手法を順に紹介します。

エミュレーション

エミュレーション手法では FPGA は必要ありません。FPGA 向けの OpenCL* プログラムを記述する場合、エミュレーションは OpenCL* コードを効率的にデバッグするための重要な最初のステップです。FPGA 向けのコンパイルは非常に遅く、デバッグオプションも制限されているため、FPGA で確認する前にエミュレーションを使用してできるだけデバッグしておくことが非常に大切です。

これらの FPGA プログラミングを試す 3 つの手法を説明するため、単純なベクトル加算について考えてみましょう。**インテル® FPGA SDK for OpenCL*** をダウンロードしてインストールした後、**インテル® FPGA ソフトウェア・デザイン・センター** (英語) からベクトル加算のサンプルをダウンロードします。サンプルをダウンロードした後 .zip ファイル (Windows*) または .tar ファイル (Linux*) を展開して、README.html の内容を確認します。SDK およびサンプルコードは無料でダウンロードして使用できます。Windows* と Linux* の両方でサンプルが正しく動作することを確認し、インテル® FPGA SDK for OpenCL* は、**VBox*** (英語) 内で Linux* を実行している macOS* システムで効率良く動作することが分かりました。

コードのパスにスペースを含めないように注意してください。ツールがパスのスペースを正しく処理できなかったために、いくつかの問題が発生しました。FPGA ツールのドキュメントの一部にはこの情報が記載されていますが、OpenCL* にしか興味がなければ読むことはないでしょう。(現在、これらの制限を解決する作業を進めています。)

エミュレーション手法には 3 つのステップがあります。

1. FPGA コードをビルドする。
2. ホストコードをビルドする。
3. プログラムを実行する。

FPGA コードをビルドする

FPGA コードは OpenCL* で記述されたカーネルから構成されます。Altera OpenCL* コンパイラー (aoc) で **-march=emulator** オプションを指定してエミュレーションを使用します。このオプションを指定しない場合、コンパイラーは FPGA のすべての合成、配置、および配線を行うため、コンパイル時間が非常に長くなります。**-march=emulator** オプションを指定すると、コンパイルは短時間で完了し、x86 コードが生成されます。生成されたプログラムは FPGA なしで実行できます。ここでの目標はエミュレーションであるため、このオプションを指定します。**vector_add** ディレクトリーから、README.html の指示に従って、次のコマンドを使用します。

```
aoc -march=emulator device/vector_add.cl -o bin/vector_add.aocx
```

ホストコードをビルドする

ホストコードは C で記述されており、OpenCL* API を呼び出して、FPGA をセットアップし、カーネルをロードして FPGA で使用します。Windows* では、Microsoft* Visual Studio* を利用します (無償の Community エディションで動作することを確認済み)。Linux* では、Makefile を利用して “make” を行います (GNU* C++ コンパイラーを使用)。

プログラムを実行する

ホストプログラムがエミュレートされた FPGA コードを実行するよう、README.html の指示に従って **CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1** に設定します。この指示のほかに、次の操作を行います。

最初に (ほかのステップを行う前に)、インテル® FPGA SDK for OpenCL* の変数を初期化します。

- **Windows*:** `%INTELFPGAOCLSDKROOT%\init_opencl.bat`
  (テストマシンでは、`C:\intelFPGA\17.1\hld\init_opencl.bat` でした。)
- **Linux*:** `$INTELFPGA_ROOTDIR/hld/init_opencl.sh`
  (テストマシンでは、`/opt/intelFPGA_pr/17.1/hld/init_opencl.sh` でした。)

Windows* では、すべての処理が 64 ビットで行われるように Visual Studio* の環境変数も初期化します。(テストシステムでは、この初期化を行わないとコンパイルに失敗しました。)

- `vcvars64.bat` (テストマシンでは、`C:\Program Files (x86)\Microsoft Visual Studio\Shared\14.0\VC\bin\amd64\vcvars64.bat` でした。)

オフラインコンパイル

OpenCL* 環境を使用して FPGA プログラミング・ファイルとレポートを生成するコンパイルを実行できます。これらのレポートは、`aoc` コマンドを実行すると自動的に作成されます。FPGA のパフォーマンスを多少 (10 ~ 20 パーセント) 落としてプログラミング・ファイルの出力とレポートを高速に生成する `-fast-compile` オプションもあります。これらのレポートはカーネルの最適化に役立ちます。レポートはコンパイル中に静的に作成されるため、この出力を得るために FPGA でプログラムを実際に実行する必要はありませんが、特定の FPGA 向けにコンパイルする必要があります (`-march=emulator` オプションはレポートを生成しません)。`vector_add` ディレクトリーから、FPGA コードをビルドしたときと同じコマンドを使用します。

```
aoc -march=emulator device/vector_add.cl -o bin/vector_add.aocx
```

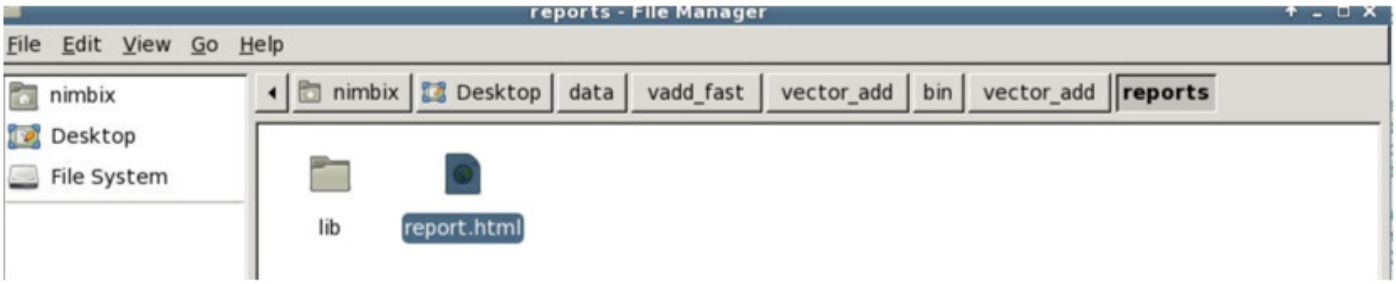
コンパイラーは、次のような使用状況レポートを出力します。

```
aoc: Optimizing and doing static analysis of code...

!===========================================================================
! The report below may be inaccurate. A more comprehensive
! resource usage report can be found at vector_add/reports/report.html
!===========================================================================

+-------------------------------------------------------------------+
; Estimated Resource Usage Summary ;
+----------------------------------------+--------------------------+
; Resource + Usage ;
+----------------------------------------+--------------------------+
; Logic utilization ; 18% ;
; ALUTs ; 9% ;
; Dedicated logic registers ; 9% ;
; Memory blocks ; 9% ;
; DSP blocks ; 0% ;
+----------------------------------------+--------------------------;
Compiling for FPGA. This process may take a long time, please be patient.
```

これらのリソース使用状況は、OpenCL* プログラムが FPGA で利用可能な並列処理を適切に行っているか判断するのに役立ちます。これは重要なチューニング情報であり、長いコンパイル時間をかけることなく取得できます。非常に多くのリソースを使用している場合、そのプログラムは特定の FPGA には適さないことがあります。逆に、ほとんどリソースを使用していない場合でも、コードをチューニングすることでパフォーマンスを向上できる可能性があります。デバイスの (例えば、浮動小数点処理能力の) 10 パーセントしか使用していない場合、最大限のパフォーマンスを引き出していないと理解するのは容易です。

フルコンパイルを行うと、より詳細なレポートが作成されます。これらのレポートはより正確であり、微調整には役立ちますが、作成するには長い時間がかかります。そのため、FPGA がない場合でも、レポートを高速に (約 1/4 の時間で) 作成する高速コンパイルオプションを指定すると時間を節約できます。(レポートの使用方法は、**インテル® FPGA SDK for OpenCL* ベスト・プラクティス・ガイド**を参照してください。)

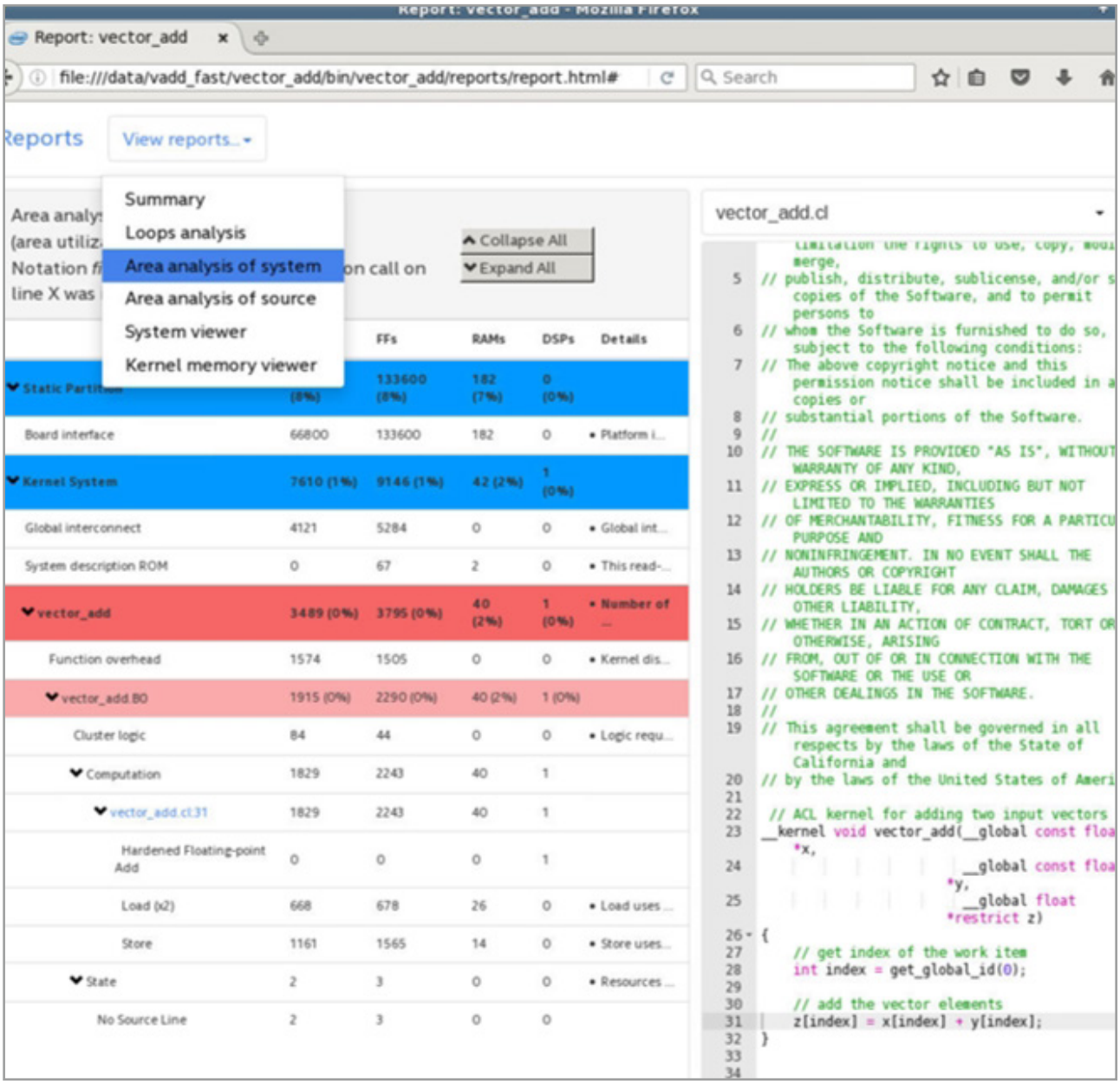
現在は、コンパイラーで複数の処理を自動的に行い、結果をまとめてブラウザーに表示することができます (以前は、手動で処理を行う必要がありました)。コンパイル中に作成された report.html ファイルを見つけることから始めます (**図 2**)。(ファイルの場所は使用状況レポートに含まれています。)



2 レポートを開く

結果ページから複数のレポートを参照できます。**図 3** は、ベクトル加算のサンプルのサマリーを示しています。約 1 時間のコンパイルの後に作成された詳細レポートで、コンパイル開始から数秒でレポートされた使用状況を確認するというのはなかなか面白いことです (ALUT 使用 9 パーセント、専用論理レジスター 9 パーセント、メモリーブロック 9 パーセント、DSP なし)。**図 4** は、エリア解析レポートを示しています。このレポートでは、プログラムにドリルダウンして、ソースコードとリソース使用状況の関係を理解することができます。

3 `-fast-compile` を使用したコンパイルのサマリーページ



4 **-fast-compile** を使用したコンパイルのシステムレポートのエリア解析

最後のヒント : "**aocl env aocxfilename.aocx**" コマンドは aocx ファイルがどのようにコンパイルされたか詳細な情報を出力します (**report.html** サマリーの情報よりも詳細です)。

クラウド

クラウド手法では、各自が FPGA を所有する必要はありません。また、ソフトウェアをダウンロードしたりセットアップする必要もありません。代わりに、Nimbix* のアカウントをセットアップし、使用量に応じて料金を払います。macOS*、Windows*、および Linux* ホストでブラウザーから、このアプローチを使用して動作することを確認しました。レートは 1 時間当たり 36 セントからで、秒単位で比例配分して計算されます。ごくわずかな料金で OpenCL* を試すことができ、ハードウェアやソフトウェアのセットアップが必要ないため大幅に時間を節約できました。最終的に、Nimbix* でベクトル加算のサンプルを試すために支払った料金は 5 ドル以下でした。

ほかのクラウドベースの FPGA 開発システムとは異なり、Nimbix* とインテルは、クラウドで実行するためツールの特別バージョンを作成していません。つまり、FPGA コードのビルドに費用をかけたくなければ、手法 1 と 2 の無料のツールを使用して、クラウドで実行するバイナリーをビルドすることができます。FPGA 上で実行できる準備が整うまで、ローカルで開発、デバッグ、およびビルドするのはより自然なことに感じられるでしょう。また、クラウドには最新のツールが用意されています。クラウドで使用されているボードと同じボードを持っていれば、ローカルシステムでもコンパイルできます。

Nimbix* クラウドにはあらゆる環境が事前にセットアップされているため、OpenCL* ツールをインストールする必要はありません (**`init_opencl`** スクリプトを source する必要もありません)。ただし、指示に従ってサンプルを実行する前に、Nimbix* のアカウントを作成し、インスタンスをアクティベートして、サンプルコードをダウンロードする必要があります。

Nimbix* クラウドを利用する手順は次のとおりです。

- **このリンク (英語) を使用してアカウントをセットアップします。**顧客情報と課金情報のみ入力してください。**`PushToCompute`** オプションは選択しないでください。料金が $0.00 になっていることを確認して SUBMIT をクリックします。
- **「Nimbix* クラウドでインテル® FPGA 開発ツールを使用するためのクイック・スタート・ガイド」(英語) を開きます。**このガイドには多くの役立つ情報が含まれていますが、OpenCL* を使用するためには必要ない情報も含まれています。このガイドを読み、マシンの **`/data`** ディレクトリーにストレージがマウントされているため、マシンをアクティベートすることなく読み書きできる点に特に留意します (「Persistent Storage and Data Transference (永続ストレージとデータ転送)」セクションを参照)。

前述のエミュレーションの手順に従って開始することを推奨します。ただし、作業を行うのはクラウド上です。具体的な手順を次に示します。

- **ダッシュボード (https://www.nimbix.net (英語)) で** [Login] をクリックしてシステムにログインします。[Compute] (右側のメニュー) をクリックし、検索フィールドに Quartus と入力して、使用するマシンを絞り込みます。利用可能な最新バージョンの Quartus (ここでは 17.1) および [PAYG] を選択します。
- **次に、[Desktop] を選択し、**CPU とメモリーを選択します。例えば、エミュレーションのコンパイルと実行のみ行う場合は 2 CPU で十分です。非エミュレーション・ビルドを行う場合は、追加のメモリーが必要になるため、8 CPU を推奨します。
- **マシンの準備ができたら、**クリックして接続します。これで、マシンにアクセスして使用できます。マシンを終了 (左下の [N] をクリックし、[Log Out] を選択して [Shutdown] を選択) するまで課金は継続されていることに注意してください。端末ウィンドウを切断してもマシンは終了しないため、課金は継続されたままです。ダッシュボード (**https://www.nimbix.net** (英語)) からマシンを終了 (kill) して、すべてのアクティビティーが停止していることを確認することを推奨します。
- **マシンで、**右クリックしてメニューを表示し、端末ウィンドウを開きます。
- 作業が保存される **`/data`** ディレクトリーで作業を行うことを**推奨します**。そのため、端末ウィンドウで最初に次のコマンドを入力します。

```
cd /data
```

- **OpenCL* ベクトル加算のサンプル (英語) で** tar ファイル (**`.tgz`**) のリンクを見つけます。**`wget`** を使用して tar ファイルをダウンロードします。

```
wget https://www.altera.com/content/dam/altera-www/global/en_US/others/support/
examples/download/exm_opencl_vector_add_x64_linux.tgz
```

- **tar ファイルを展開します。**

```
tar xvf exm_opencl_vector_add_x64_linux.tgz
```

- **README.html** には詳細な説明が含まれています。ここで必要なのは、3 つのコマンド (FPGA コードのビルド、ホストコードのビルド、実行) のみです。これらのコンパイルと実行にかかった時間は約 30 秒でした。

```
cd vector_add
```

```
aoc -march=emulator device/vector_add.cl -o bin/vector_add.aocx
make
```

```
CL_CONTEXT_EMULATOR_DEVICE_ALTERA=1 bin/host -n=10000
```

これで第 1 段階は完了です。この時点では、OpenCL* プログラムはエミュレーションによりクラウド上で実行されています。FPGA 上で実行するには、(FPGA 向けに) リビルドして実行する必要があります。

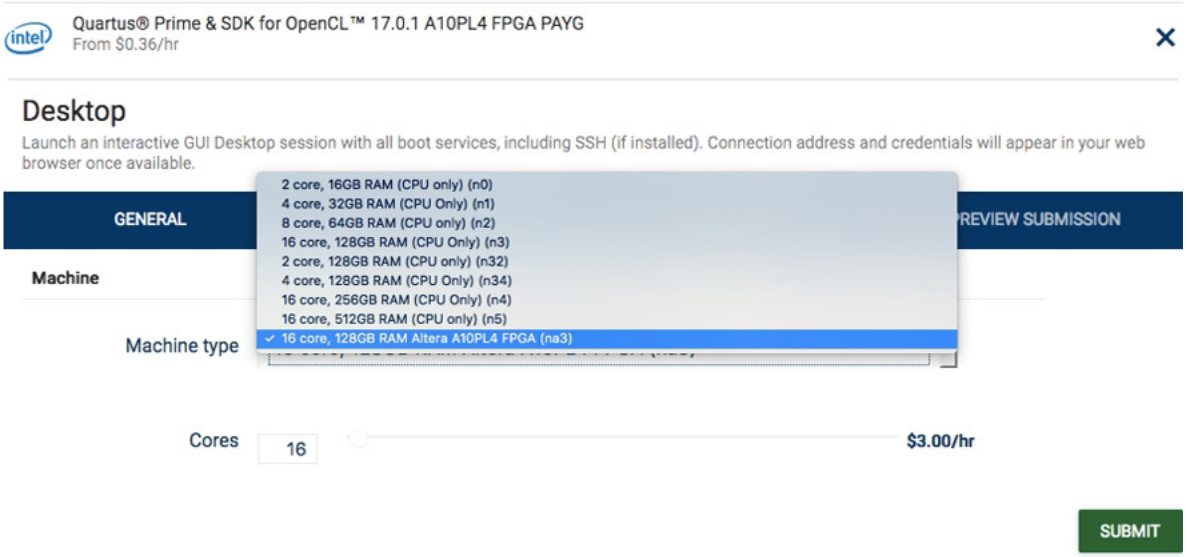
- **リビルド。**`-march=emulator` オプションを指定しないで `aoc` を使用してコンパイルします。次のコマンドを使用して 8 CPU で完了するまで約 3 時間かかりました。

```
aoc device/vector_add.cl -o bin/vector_add.aocx
```

  `-fast-compile` オプションを追加して、次のコマンドを使用した場合、17.1 コンパイラーで aocx ファイルのビルドにかかる時間は 1 時間以下になりました。

```
aocl -fast-compile device/vector_add.cl -o bin/vector_add.aocx
```

- **実行。**FPGA を搭載しているマシン上で実行する必要があります。Nimbix* はインテル® Arria® 10 FPGA を搭載した **BittWare** (英語) の A10PL4 ボードを使用しているため、名前に A10PL4 を含む [Application] を選択します。A10PL4 ボードに実際にアクセスできるマシンタイプを選択する必要があります。プルダウンメニューで、(CPU only) ではなく A10PL4 を含むマシンタイプを選択していることを確認します (**図 5**)。少なくとも 1 台の FPGA が有効なマシンを選択できるはずです。



5 [Machine type] プルダウンメニューから FPGA が有効なマシンを選択している例

- コマンドを使用して (**CL_CONTEXT_EMULATOR_DEVICE_ALTERA** を設定しないで) アプリケーションを実行します。

```
bin/host -n=10000
```

これで第 2 段階も完了です。正常に実行されると、次のようなメッセージが表示されます。

```
Initializing OpenCL
Platform: Intel(R) FPGA SDK for OpenCL(TM)
Using 1 device(s)
a10pl4_dd4gb_gx115 : A10PL4 (acla10pl40)
Using AOCX: vector_add.aocx
Reprogramming device [0] with handle 1
Launching for device 0 (10000 elements)
Time: 0.409 ms
Kernel time (device 0): 0.062 ms
Verification: PASS
```

aocl diagnose コマンドはシステムで利用可能なカードを調べるのに役立ちます。

```
aocl diagnose
aocl diagnose: Running diagnose from /opt/intelFPGA_pro/17.0/hld/board/a10pl4/
linux64/libexec
------------------------- acl0 -------------------------
Vendor: BittWare Inc

Phys Dev Name Status Information
acla10pl40 Passed A10PL4 (acla10pl40)
PCIe dev_id = 2494, bus:slot.func = 02:00.00, Gen3 x8
FPGA temperature = 52.6523 degrees C.
DIAGNOSTIC_PASSED
---------------------------------------------------------
```

2017 年 11 月以降の 17.1 ビルドを使用して新しい **`-fast-compile`** オプションを指定すると、コンパイル時間は大幅に短縮されます。これは、FPGA の非常に長いコンパイル時間を短縮するインテル独自の機能です。**`-fast-compile`** オプションを指定すると、コンパイル時間は大幅に (通常は 1/4 ~ 1/5 に) 短縮されますが、FPGA プログラムの実行効率はわずかに (通常は 10 ~ 20 パーセント) 低下します。複数のカーネルで構成される設計の場合は、変更されたカーネルを自動的に識別して変更点のみを再コンパイルする、インクリメンタル・コンパイル・オプションが利用できます。

OpenCL*: ハイパフォーマンス、オープンソース、広範なサポート

この記事では、OpenCL* による FPGA プログラミングの優れた点について概要を紹介しました。例えば、インテルの OpenCL* は、FPGA 上で実際の実行をプロファイルして結果を調べるためにフックをサポートしています。また、ホストと I/O チャネルを含む FPGA パフォーマンスを広範にサポートしています。チャネルは、I/O またはホストから FPGA カーネルへの直接データ・ストリーミングをサポートするため、インテルにより作成された OpenCL* 言語の拡張です。

この記事をお読みになった皆さんが、OpenCL* による FPGA プログラミングの新しい世界を探求されることを期待しています。インテルと Altera の Web サイトでは、無料のトレーニング・リソースを含む、多くのレポート、ツール、ライブラリー、および拡張を提供しています。

関連情報

- **インテル® FPGA SDK for OpenCL***
- **インテル® FPGA SDK for OpenCL* のドキュメント**
- **インテル® FPGA SDK for OpenCL* ベスト・プラクティス・ガイド** - OpenCL* で FPGA プログラミングを行う際の優れたガイド。レポートとその意味についての情報も含まれています。
- **Nimbix* クラウドでインテル® FPGA 開発ツールを使用するためのクイック・スタート・ガイド** (英語)
- **インテル® FPGA ソフトウェア・デザイン・センター** - コードサンプルをダウンロードできます。
- **インテル® FPGA プラットフォーム** - アクセラレーター・ボードのリンクが含まれています。
- **OpenCL* 標準規格** (英語)
- **インテル® Cyclone® 10 LP ファミリー**、**インテル® Arria® 10 ファミリー**、および**インテル® Stratix® 10 ファミリー**の製品情報
- OpenCL* 2014 の国際ワークショップの OpenCL* + FPGA による gzip アクセラレーションに関する論文 : **Gzip on a Chip: High Performance Lossless Data Compression on FPGAs using OpenCL*** (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を参照してください。

自動運転ワークロードに適した Eigen 数学ライブラリーの構築

インテル® MKL で速度のニーズを満たす

Steena Monteiro インテル コーポレーション 自動運転エンジニアリング
Gaurav Bansal インテル コーポレーション 自動運転エンジニアリング

自動運転のワークロードは、その主要部でさまざまな行列演算を行っています。センサー・フュージョンと位置推定アルゴリズム (異なるバージョンのカルマンフィルターなど) は、自動運転ソフトウェア・パイプラインに不可欠なコンポーネントです。**インテル® マス・カーネル・ライブラリー (インテル® MKL)** には、高速 DGEMM など、さまざまな数学演算向けにチューニングされた高性能なサブプログラムが含まれています。自動運転開発者コミュニティーは一般に、行列演算に C++ 数学ライブラリーの Eigen[1] を使用しています。インテル® MKL とハイパフォーマンス行列 - 行列乗算向けに高度にチューニングされたライブラリーである LIBXSMM [2, 3] を組み合わせることで、行列演算を高速化できる可能性があります。この記事では、行列乗算ベンチマークにおけるネイティブ Eigen のパフォーマンスと拡張カルマンフィルター (EKF)[5] のパフォーマンスを、**インテル® Xeon® プロセッサー**上で GNU* コンパイラーとインテル® コンパイラーでインテル® MKL および LIBXSMM を使用して調査および向上します。[4, 5]

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。

カルマンフィルターにおける高速化のニーズ

自動運転パイプラインは、カメラ、RADAR や LIDAR のようなセンサーから運転環境についての情報を収集する予測から、センサー・フュージョンと位置推定、パス・プランニング、そしてステアリングの角度やスロットルなどの最終的な車両制御まで、一連の計算ブロックを含んでいます。厳しいエンドツーエンドのレイテンシー要件を満たすには、ソフトウェア・パイプライン全体にわたるパフォーマンスの最適化が不可欠です。パイプラインの各コンポーネントには通常、厳格なレイテンシー要件が設定されており、ほぼリアルタイムに処理することが求められています。この記事では、センサー・フュージョンと位置推定の重要なコンポーネントである EKF の高速化について取り上げます。

拡張カルマンフィルター (EKF) アルゴリズム

EKF は、車両の状態 (デカルト位置座標、速度、ヨー角など) に関する予測を行う、単純かつ非常に強力なアルゴリズムです。EKF には、何度も反復される 2 つの連続するステップがあります。

- **予測ステップ**は、時間による値の変化を含むモーションモデルに基づいて現在の変数の値とそれらの不確実性を推定します。
- **更新ステップ**は、センサーから測定の次のセットを受け取ると開始します。このステップは、1 つの重要な要因 (予測した推定と現在の測定の推定の加重平均) に基づいて予測した推定を更新します。重みが高いほど、不確実性は低くなります [6]。

特に、このアルゴリズムは、LIDAR および RADAR センサーの測定値から車両の位置 (**px,py**) と速度 (**vx,vy**) を予測します。RADAR と LIDAR の値を結合した車両位置の推定は、LIDAR および RADAR 単体での推定よりも精度が高くなります。物体の位置を推定する LIDAR 測定は、デカルト座標形式 (**px,py**) で定義されています。RADAR 測定は、通常は極座標形式で、デカルト座標に変換できます。測定値は LIDAR よりも低い解像度で作成されます [6]。

表 1 は、異なる状態と推定を表すために EKF が使用するベクトルと行列を示しています [4, 5]。

表 1. EKF で使用する行列とベクトル

行列 / ベクトル	目的	次元
U	制御ベクトル	4x1
X	状態ベクトル	4x1
Z	観測ベクトル	2x1 または 3x1
F	状態遷移行列	4x4
P	予測共分散推定	4x4
Q	プロセス (ガウス) ノイズの共分散行列	2x3、2x2、3x3、3x2
R	観測 (ガウス) ノイズの共分散行列	2x3、2x2、3x3、3x2
H	観測行列	2x4 または 3x4

予測

x' = F *x + u　　予測状態推定

P' = F * P *FT+ Q　　予測共分散推定

観測の更新

y = z – H' * x　　イノベーションまたは観測残差

S = H * P' * HT+ R　　イノベーション (または残差) の共分散

K = P' * HT * S-1　　準最適カルマンゲイン

x = x' + K * y　　更新された状態推定

P = (I – K * H) * P'　　更新された共分散推定

重要な数学ライブラリー

インテル® MKL は、インテル® プロセッサー・アーキテクチャー上でパフォーマンスを最大限に引き出す、高度に最適化、スレッド化、ベクトル化された数学関数を提供します。さまざまなコンパイラー、言語、オペレーティング・システム、リンク、およびスレッド化モデルと互換性があります。ここで重要なのは、行列 - 行列乗算向けに高度にチューニングされた DGEMM 関数を提供することです[6]。小行列での DGEMM の追加エラーチェックのオーバーヘッドを排除するため、インテル® MKL には、実行時に最も速いコードパスを使用することを保証する **-DMKL_DIRECT_CALL** コンパイラー・オプションが用意されています[7]。

Eigen[1] は、行列演算から幾何学アルゴリズムまで、さまざまな演算を提供する、オープンソースの、使いやすい C++ ライブラリーで、異なるレベルのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) およびインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) でのベクトル化が可能です。Eigen でインテル® MKL を利用するには、**-DEIGEN_USE_MKL_ALL** オプションを指定します。

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。

LIBXSMM は、非常に小さな行列サイズで高速な行列 - 行列乗算を行うためにチューニングされた、オープンソースのハイパフォーマンス・ライブラリーです [2, 3]。LIBXSMM は、インテル® SSE、インテル® AVX、インテル® AVX2、およびインテル® AVX-512 を含むさまざまな命令セット向けに、小さな行列 - 行列乗算カーネルの JIT (Just-in-time) コードを生成します。LIBXSMM は、$(M*N*K)^{1/3}$ が 80 未満の行列に最適です。LIBXSMM は、特に、xGEMM コード生成において個別のフロントエンド (高水準言語とルーチンの選択) とバックエンドを備えたモジュラー設計により、高いパフォーマンスを提供します [2]。LIBXSMM は、わずかな労力でアプリケーションに統合できるよう、S/DGEMM を呼び出す単純なインターフェイスを提供しています。**図 1**、**2**、および **3** は、行列乗算に使用できる LIBXSMM の 3 つのモードを示しています。

インストール中に、LIBXSMM は次の値を明示的に構築します。

- 特定の M、N、および K 値
- M、N、および K 値と異なるリーディング・ディメンジョン値
- α および β の特定の値

```
void libxsmm smm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm dmm(int m, int n, int k, const double* a, const double* b, double* c);
```

1 自動的にディスパッチされた LIBXSMM[2] の行列乗算 API

```
void libxsmm simm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm dimm(int m, int n, int k, const double* a, const double* b, double* c);
```

2 ディスパッチされていない LIBXSMM[2] の行列乗算 API

```
void libxsmm sblasmm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm dblasmm(int m, int n, int k, const double* a, const double* b, double*
c);
```

3 BLAS[9] を使用した行列乗算の LIBXSMM API

インテル® MKL と LIBXSMM の有効化

オリジナル形式では、Eigen は小さな行列の乗算にインテル® MKL を使用しません (特に、M+N+K が 20 未満の場合)。そこで、Eigen がすべての行列サイズでインテル® MKL の DGEMM を呼び出すようにするため、Eigen のソースコードを変更して M+N+K<20 ヒューリスティックを削除しました。

次に、Eigen で LIBXSMM を有効にするため、Eigen のネイティブ行列 - 行列乗算実装を **`libxsmm_dgemm`** の呼び出しに変更しました。

テスト環境のセットアップ

Eigen を使用する 2 つのワークロードのパフォーマンスを調べました。

1. 正方倍精度行列のセットに DGEMM を実装する**単純な DGEMM ベンチマーク**。
2. 合成的に生成された RADAR と LIDAR データを処理する **EKF の実装**。

ネイティブ Eigen、Eigen とインテル® MKL、および Eigen と LIBXSMM を使用してテストしました。すべてのベンチマークはシリアルで実行しました。

表 2 は、ライブラリーとコンパイラーのバージョンおよびハードウェア仕様の詳細です。

表 2. ライブラリー、コンパイラー、ハードウェアの仕様

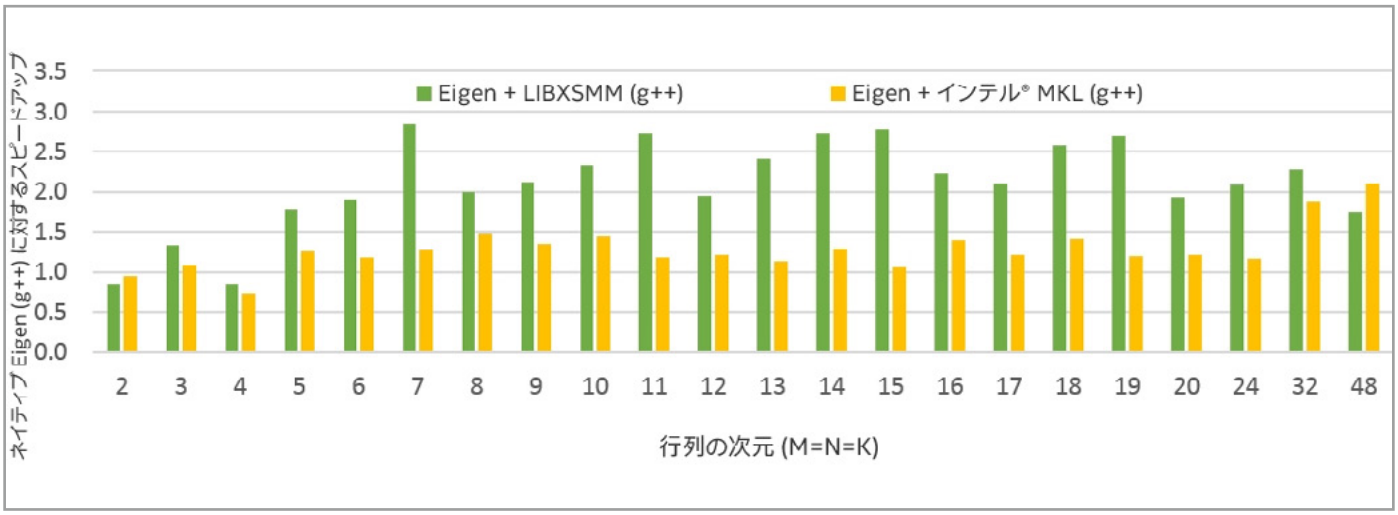
ソケットごとのコア数	20
ソケット	2
L1 キャッシュ	32K
L2 キャッシュ	1,024K
L3 キャッシュ	28,160K
クロック	2.40GHz (インテル® ターボ・ブーストが有効な場合は 3.70GHz)
インテル® C++ コンパイラーのバージョン	17.0.4 20170411
GNU* G++ コンパイラーのバージョン	7.2
Eigen のバージョン	3.3.3
インテル® MKL のバージョン	インテル® MKL 2017 Update 3

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

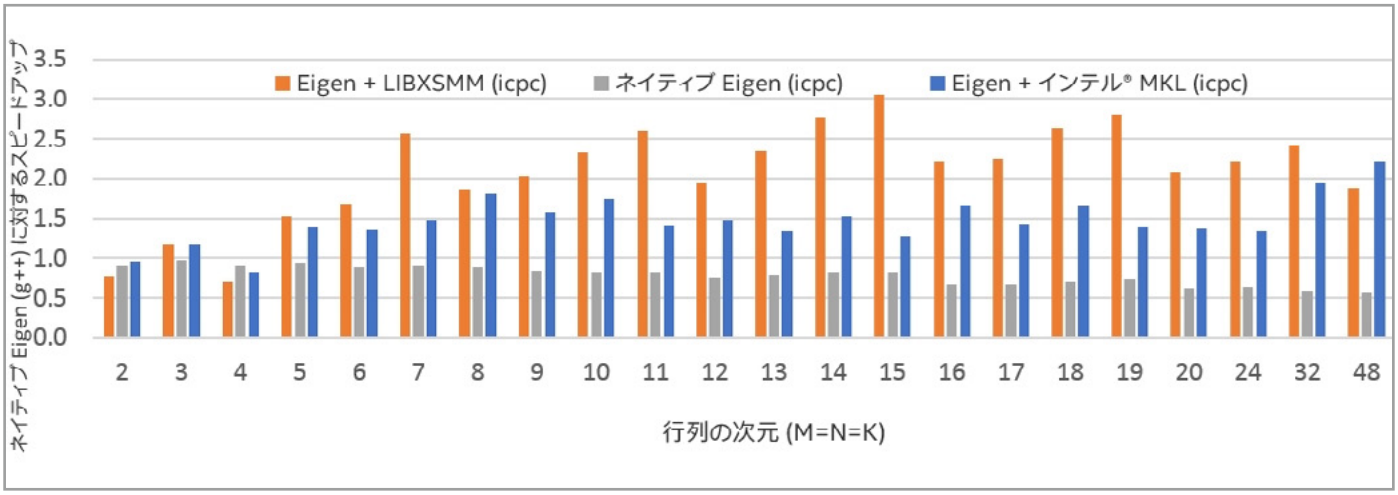
インテル® Xeon® プロセッサー上での DGEMM ベンチマーク

この DGEMM ベンチマークの性能指数は、ネイティブ Eigen と g++ に対するパフォーマンス (GFLOPS) の向上です。行列サイズ 2 と 4 を除いて、Eigen とインテル® MKL および Eigen と LIBXSMM はどちらも、すべての行列クラスでネイティブ Eigen と比較してスピードアップしました。GNU* コンパイラーとインテル® コンパイラーのどちらでコンパイルした場合でも、ネイティブ Eigen のパフォーマンスが最も低いことに注目してください (**図 4** および **5**)。パフォーマンスの向上という点から見ると、全体的な傾向は次のようになります。

- **`Eigen + LIBXSMM`** は、ほぼすべての行列で最もパフォーマンスが高い
- **`Eigen + LIBXSMM (g++)`** は、サイズ 13 以下の行列で最もスピードアップしている
- **`Eigen + LIBXSMM (icpc)`** は、サイズ 14 以上の行列で、g++ とインテル® C++ コンパイラーのすべてのケースで最もスピードアップしている



4 g++ 7.2 を使用した場合のネイティブ Eigen と比較したインテル® MKL および LIBXSMM のスピードアップ

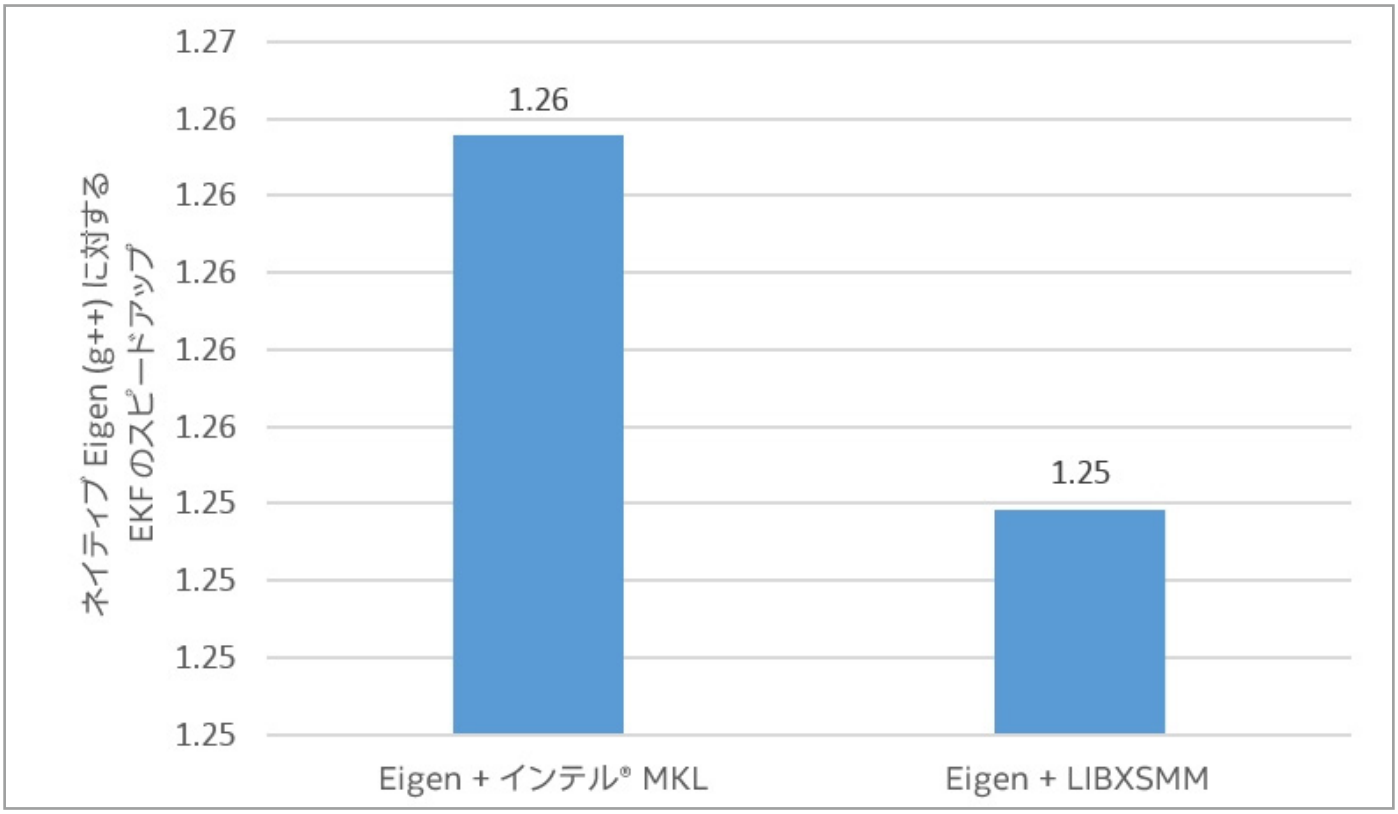


5 インテル® C++ コンパイラーを使用した場合のネイティブ Eigen と比較したインテル® MKL および LIBXSMM のスピードアップ

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。

拡張カルマンフィルターの測定とスピードアップ

ネイティブ Eigen、Eigen とインテル® MKL、および Eigen と LIBXSMM を使用して EKF を測定しました。DGEMM ベンチマークでは、サイズ 13 以下の行列では g++ のパフォーマンスが最も高くなっていました。EKF は小さなサイズの行列でも動作するため、g++ を使用した EFK のスピードアップを測定しました。スピードアップ測定のベースラインは、ネイティブ Eigen を使用した EKF です。性能指数は、各センサー測定値の予測および更新の中央値です (合計 10,000 のセンサー測定値を処理)。**図 6** に示すように、インテル® MKL または LIBXSMM を組み合わせることにより、EFK は約 1.2 倍にスピードアップしています。



6 Eigen とインテル® MKL および Eigen と LIBXSMM を使用した場合の拡張カルマンフィルターのスピードアップ

パフォーマンスの向上

この記事では、センサー・フュージョンと位置推定に使用される一般的な自動運転のワークロードである、EKF のパフォーマンスを向上することに専念しました。インテル® Xeon® プロセッサー上でのパフォーマンスの向上を 2 つの方法で測定しました。

- インテル® MKL と LIBXSMM を使用した場合の**ネイティブ Eigen と比較した行列 - 行列乗算カーネルのスピードアップ**
- EKF ワークロードの**パフォーマンスの向上**

インテル® C++ コンパイラーで Eigen + LIBXSMM を使用した場合、ネイティブ Eigen と比較して最大 3.1 倍にスピードアップしました。インテル® MKL および LIBXSMM を使用することにより、EKF のパフォーマンスは約 1.2 倍になりました。

参考資料

1. Eigen: **http://eigen.tuxfamily.org** (英語)
2. LIBXSMM: **http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post137s2-file2.pdf** (英語)
3. LIBXSMM コード・リポジトリー : **https://github.com/hfp/libxsmm** (英語)
4. 拡張カルマンフィルター : **https://en.wikipedia.org/wiki/Extended_Kalman_filter** (英語)
5. 拡張カルマンフィルターのチュートリアル : **https://www.cse.sc.edu/~terejanu/files/tutorialEKF.pdf** (英語)
   拡張カルマンフィルターを使用したオブジェクト・トラッキングとセンサー測定の融合 : **https://medium.com/@mithi/object-tracking-and-fusing-sensor-measurements-using-the-extended-kalman-filter-algorithm-part-1-f2158ef1e4f0** (英語)
6. インテル® MKL ベンチマーク : **https://software.intel.com/en-us/mkl/features/benchmarks** (英語)
7. MKL_DIRECT_CALL を使用してサイズの小さな問題でのインテル® MKL のパフォーマンスを向上 : **https://www.isus.jp/products/mkl/improve-mkl-performance-for-small-problems/**
8. Basic Linear Algebra Subprograms (BLAS): **http://www.netlib.org/blas/** (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を参照してください。

インテル® MKL のベクトル化された圧縮行列関数による代数計算の高速化

圧縮データレイアウトのパフォーマンスの利点を最大化

Kirana Bergstrom 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
Eugene Chereshnev 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
Timothy B. Costa インテル コーポレーション HPC アプリケーション・エンジニア

多くのハイパフォーマンス・コンピューティング・アプリケーションは、非常に小さな行列の大きなグループを扱う行列演算に依存しています。最新バージョンの**インテル® マス・カーネル・ライブラリー (インテル® MKL)** は、このような問題向けにベクトル化ベースの最適化を含む新しい圧縮関数を提供します。

インテル® MKL の圧縮関数は、SIMD (Single Instruction, Multiple Data) 行列演算を活用しており、スカラーカーネルとして抽象的に表現されたカーネルによって行列のサブグループを操作し、クロス行列のベクトル化によってレジスターに値をロードします。これらの関数は、マルチスレッドを利用して標準のデータ形式向けに記述されたカーネルに依存するバッチ手法よりも、はるかに優れたパフォーマンスをもたらします。

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

(インテル® MKL のバッチ汎用行列 - 行列乗算の詳細は、**こちら** (英語) を参照してください。) 効率良いベクトル化によるパフォーマンスの向上に加えて、サブグループの計算をスレッド化することで複数のレベルでの並列化が可能なため、インテル® MKL の圧縮関数はバッチ手法と同じ並列化の原則による利点も得られます。

最新バージョンのインテル® MKL では、6 つの圧縮関数が追加されています。

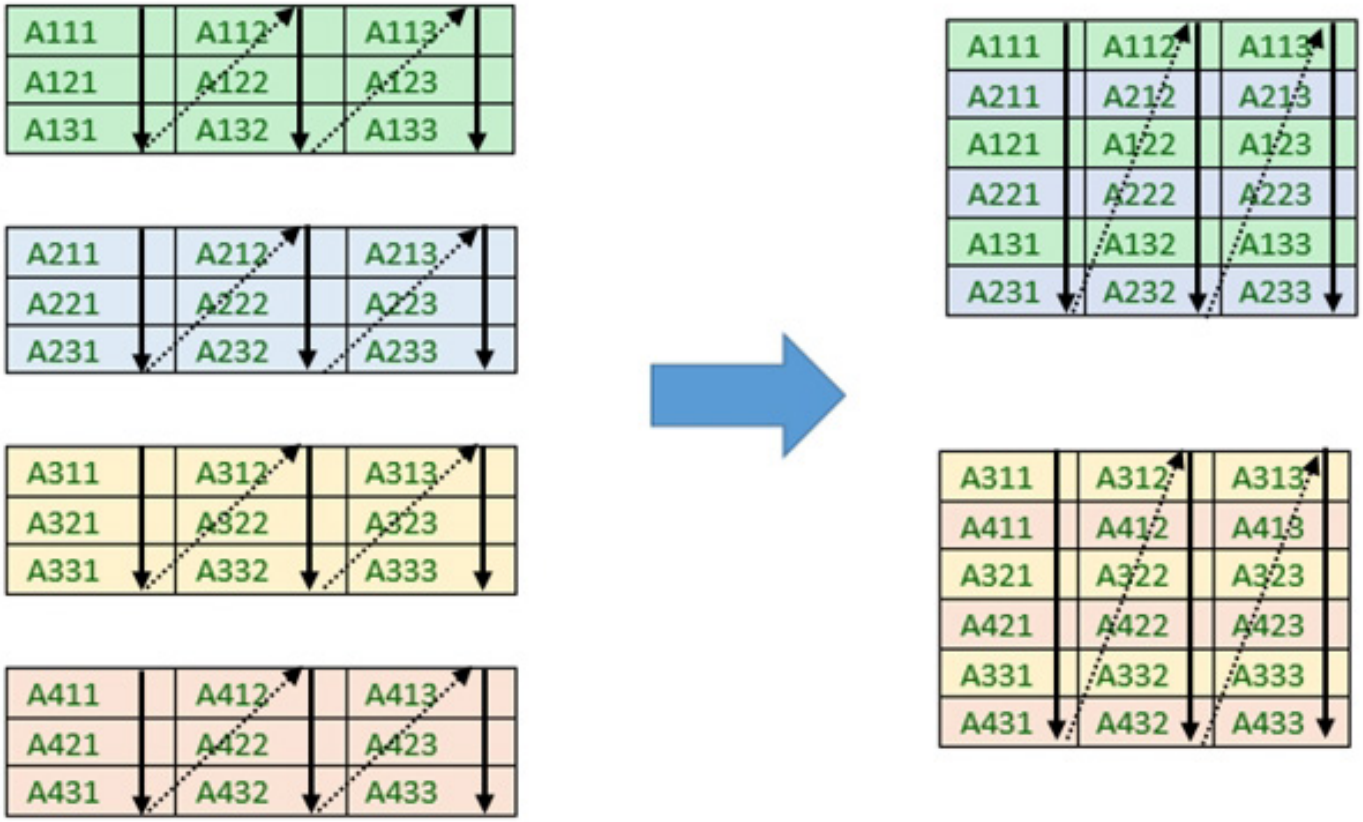
1. 汎用行列 - 行列乗算
2. 三角行列を含む方程式ソルバー
3. LU 分解 (ピボットを用いない)
4. (ピボットを用いない LU 分解からの) 逆行列の計算
5. コレスキー分解
6. QR 分解

また、行列のグループを簡単にパック / アンパックできるサービス関数も追加されています。(インテル® MKL 2018 の圧縮 API については、**インテル® MKL デベロッパー・リファレンス** (英語) を参照してください。)

圧縮形式

圧縮関数は、圧縮形式と呼ばれる、インターリーブ・データ・レイアウトの連続するメモリーセグメントにパックされた行列を操作します[1]。圧縮形式では、行列は長さ V にパックされます。V は、アーキテクチャーのレジスター長と行列要素のサイズに依存します。各パックは 3D テンソルで、行列インデックスは最も速くインクリメントされます。これらの圧縮パックは、レジスターにロードされ、SIMD 命令を使用して操作されます。

図 1 は、4 つの 3x3 実数行列のパッキング処理を示しています。この例では、パック長が V=2 であるため、2 つの圧縮パックが生成されます。

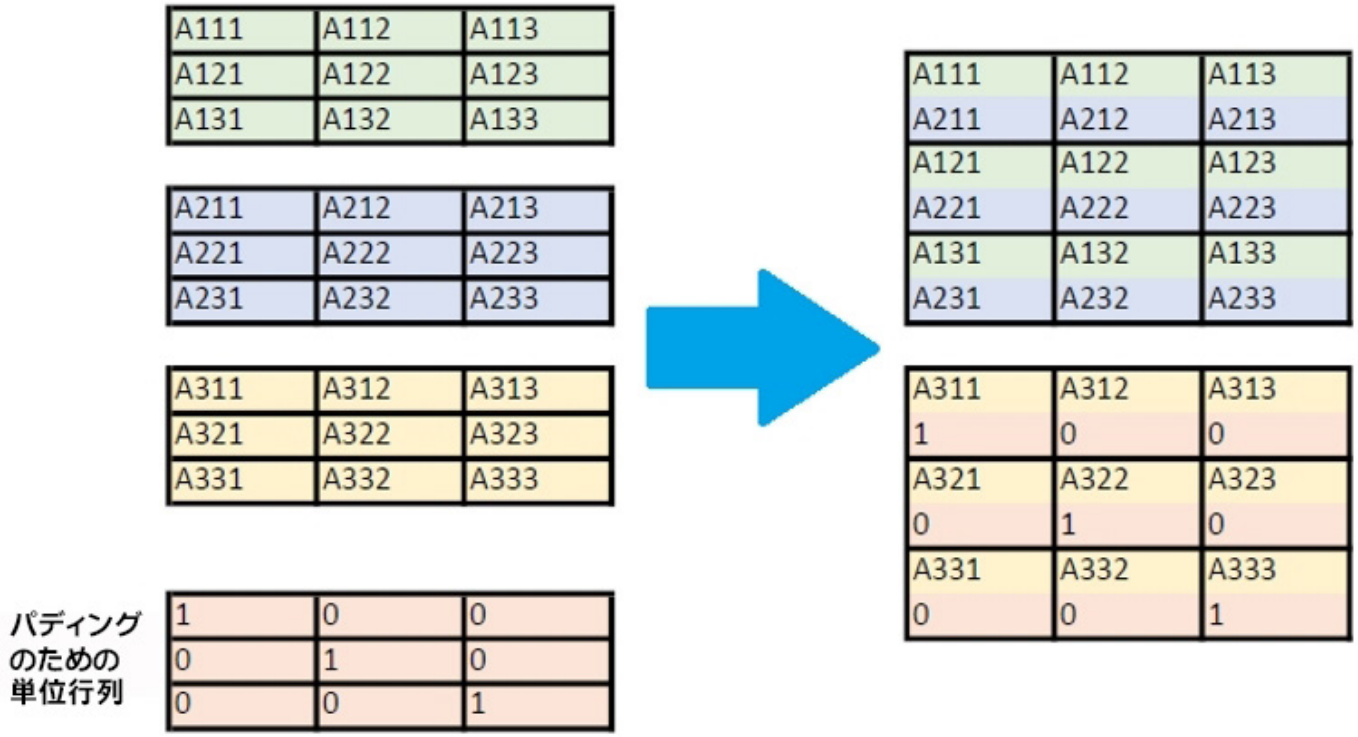


1 4 つの 3x3 実数行列の圧縮形式 (パック長 V=2)

複素数行列では、実部と虚部が別々にパックされます。実部と虚部のパックはメモリーで交互に使用されるため、複素数行列を扱う問題のパックの数は、実数行列を扱う同じ問題の 2 倍になります。複素数要素を扱うほとんどの算術演算は、追加の操作なしで (つまり、シャッフルしなくても) レジスター型変数で表現できるため、このパック形式は複素数の圧縮関数に適しています。

パック長 V は、SIMD ベクトルビット長と行列要素のビットサイズ (あるいは、複素数の場合は対応する実数型のビットサイズ) の商です。例えば、SIMD ベクトル長が 256 ビットのアーキテクチャーで倍精度行列を操作する場合 (V=4)、ベクトル化された命令を使用することで圧縮カーネルは 4 つの行列を同時に操作できます。

圧縮カーネルは、個別のパックを操作します。パックする行列の数がパック長の倍数でない場合、最後のパックは完全に埋まっていないため、組込みベースのカーネルでは処理できません。パフォーマンスを最適化するには、最後のパックが完全に埋まるように追加データでパディングすべきです。ゼロ除算などの数値問題を回避するため、単位行列を使用します。インテル® MKL の圧縮関数は、単位行列の操作では数値エラーになりません。**図 2** は、パック長が 2 の場合の 3 つの行列のパディング処理を示しています。

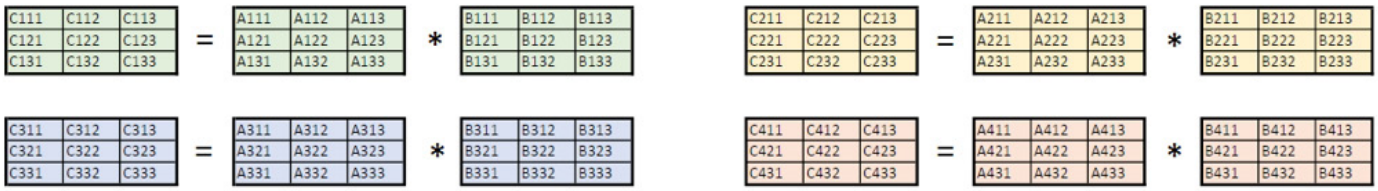


2 単位行列でパディング

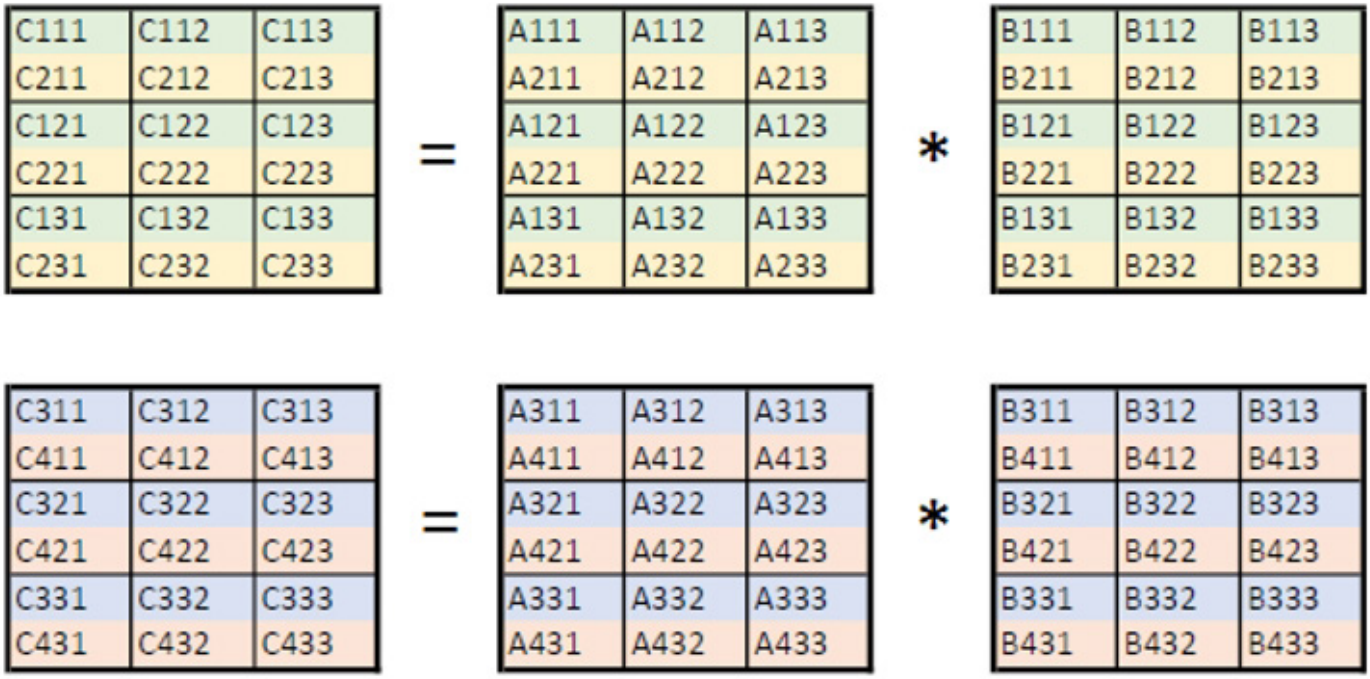
図 3 は、4 つの 3x3 実数行列に対して **C=A*B** を実行する行列 - 行列乗算の単純な圧縮バージョンです。汎用 (またはバッチ) 関数は、この問題では 4 つの行列 - 行列乗算を実行する必要があります。

すでに行列がパック長 V=2 で圧縮形式にパックされていると仮定すると、**図 4** に示すように 2 つの行列 - 行列乗算が行われます。

この 2 つの乗算に使用される行列の要素はベクトル長 V で、レジスターにロードされ、通常の行列 - 行列乗算のスカラー要素のように扱われます。



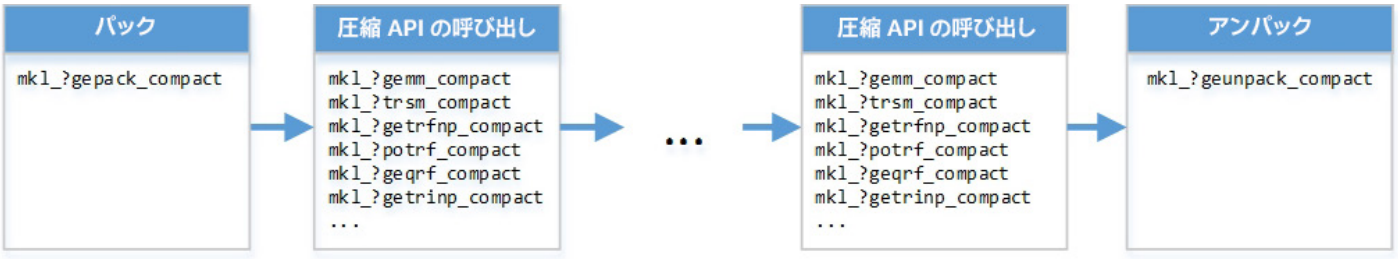
3 4 つの 3x3 行列の汎用 GEMM 操作



4 4 つの 3x3 行列の圧縮 GEMM 操作

圧縮 API

圧縮関数を呼び出す前に、入力データを圧縮形式にパックする必要があります。そして、最初の圧縮関数の直後に別の圧縮関数が呼び出される場合を除いて、実行後に出力データをこの圧縮形式からアンパックします。**図 5** は、インテル® MKL の圧縮関数の通常の呼び出しフローです (**?** は、精度指定子に置き換えます : **s**、**d**、**c**、**z**)。



5 圧縮 API の呼び出しフロー

インテル® MKL は、行列を圧縮形式にパック / アンパックするサービス関数を提供しています。2 つのサービス関数 **mkl_get_format_compact** と **mkl_?get_size_compact** は、アーキテクチャーに最適な形式と圧縮配列の格納に必要な圧縮バッファーサイズを計算します。ユーザーは、これらの関数の戻り値をパック関数 (**mkl_?gepack_compact**)/ アンパック関数 (**mkl_?geunpack_compact**) にパラメーターとして渡します。

行列のパック / アンパックは、計算のオーバーヘッドを増やします。パックド行列に対して複数の圧縮関数を呼び出す場合、最初の圧縮関数呼び出しの前に行列をパックして、最後の圧縮関数の後に行列をアンパックすることで、パフォーマンスの利点を最大化できます。

インテル® MKL の圧縮関数は、128/256/512 ビット SIMD レジスター向けに最適化されています。圧縮パックは、圧縮カーネルによって個別に処理されます。圧縮カーネルには、参照と組込みベースの 2 種類があります。参照カーネルは、C で実装されており、特定の CPU 向けに最適化されていません。一方、組込みベースのカーネルは、ベクトル組込み関数を基に特定の命令セット向けに最適化されています。インテル® MKL の圧縮関数では、次の 2 つのケースで参照カーネルが呼び出されます。

1. **アーキテクチャー**がインテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2) 以降をサポートしていない場合
2. アーキテクチャーの**ネイティブ形式**がパックド行列の形式と互換性がない場合

どちらのケースでも、圧縮関数は正しく計算を実行しますが、参照カーネルのパフォーマンスは組込みベースのカーネルを下回ることがあります。

個々の圧縮パックの計算は、本質的に独立しています。そのため、圧縮関数は簡単にスレッド化できます。インテル® MKL の圧縮関数は、内部でこの単純な並列化を行い、複数のスレッドで圧縮関数を呼び出す場合、パフォーマンスを向上します。**図 6** は、多数の行列について逆行列の計算を実行する 2 つのアプローチの比較です。左のコードでは、すべての行列に対して OpenMP* ループを利用して計算が並列化されています。一方、右のコードは、圧縮 API によるベクトル化と並列化を利用しています。

```
/* OpenMP* ループ */
#pragma omp parallel for
for (i = 0; i < number_of_matrices;
i++) {
    /* LU 分解を実行 */
    call mkl_dgetrfnp

    /* 逆行列の計算を実行 */
    call mkl_dgetrinp
}
```

```
/* ポインターへのポインターから圧縮形式へパック */
call mkl_dgepack_compact

/* 圧縮 LU 分解を実行 */
call mkl_dgetrfnp_compact

/* 圧縮逆行列の計算を実行 */
call mkl_dgetrinp_compact

/* 圧縮形式からポインターへのポインターへアンパック */
call mkl_dgeunpack_compact
```

6 逆行列の計算 : すべての行列に対する OpenMP* 並列ループ (左) と圧縮関数の使用 (右)

アプリケーション

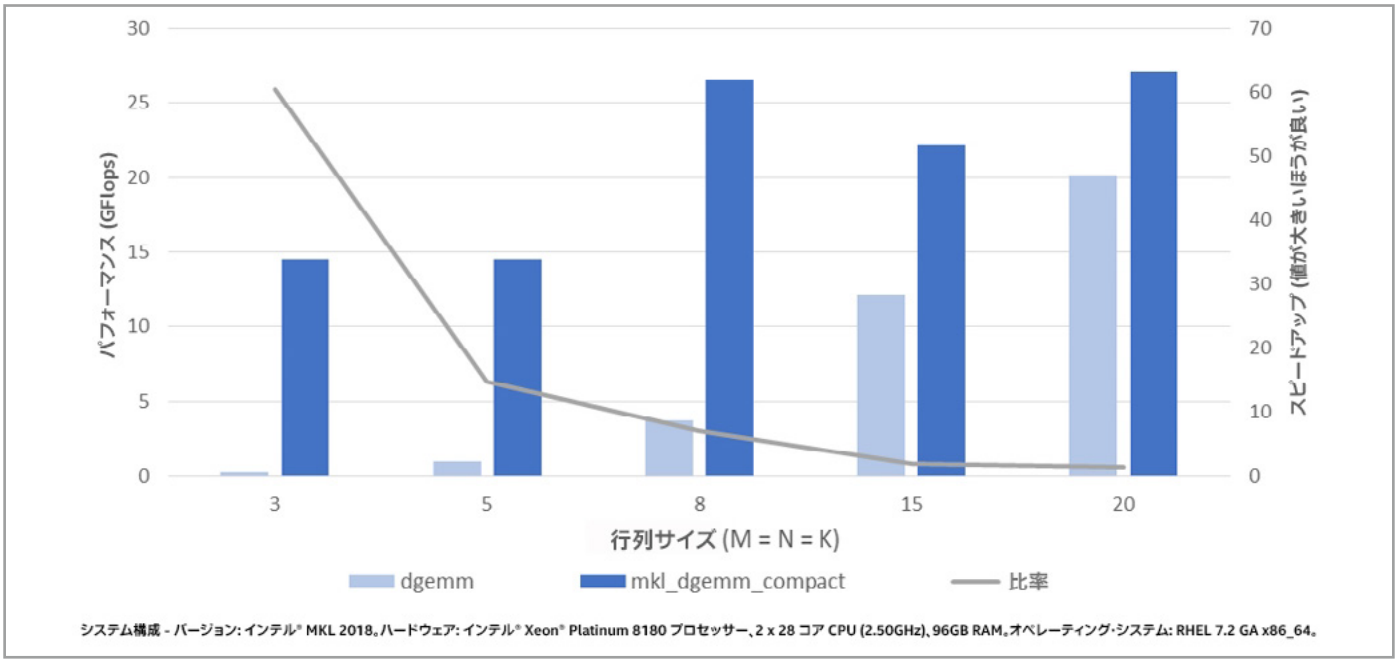
圧縮 BLAS (Basic Linear Algebra Subprograms) と LAPACK (Linear Algebra Package) 関数を利用できるアプリケーションは多数あります。コンピューター・ビジョンでは、非常に小さな行列の大きなグループに対する線形代数演算が必要になります。例えば、画像の異常検出には、コレスキー分解を使用して数千の密線形方程式を同時に解く必要があります。これらの分解処理は独立しているため、圧縮関数を利用することにより、スピードアップが見込めます。

偏微分方程式 (PDE) ベースのシミュレーションは、各ブロックがメッシュ･エンティティー (ノード、エッジ、フェース、セルの中心など) である、ブロック疎行列として表現されたメッシュを離散化します。例えば、理想的な気体モデルを用いた 3D 圧縮性流体力学モデルは、5x5 ブロックを使用します。典型的な PDE ベースのアプリケーションでは、反復線形ソルバーが一連の行列 - ベクトルと行列 - 行列の積を実行します。行列に対する操作および行列内の操作は、並列に実行できます。「Designing Vector-Friendly Compact BLAS and LAPACK Kernels」(Kim et al., 2017) では、圧縮行列 - 行列乗算、三角ソルバー、LU 分解を使用する圧縮流体力学のシミュレーションにおいて、線形ソルバーが最大 6 倍のスピードアップを達成しています。

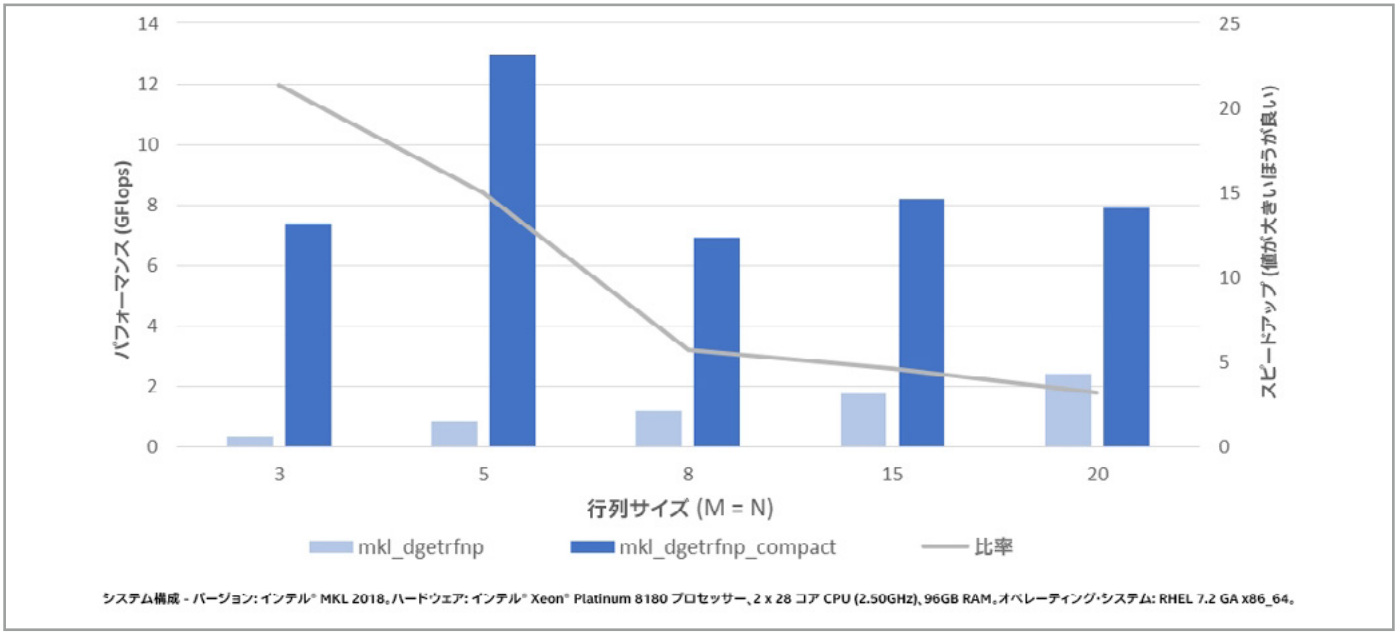
コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

パフォーマンス結果

図 7 と**図 8** は、汎用行列 - 行列乗算 (GEMM) と汎用行列のピボットを用いない LU 分解 (GETRFNP) のパフォーマンスの向上を示しています。結果は、汎用インテル® MKL 関数の呼び出しを測定しています。

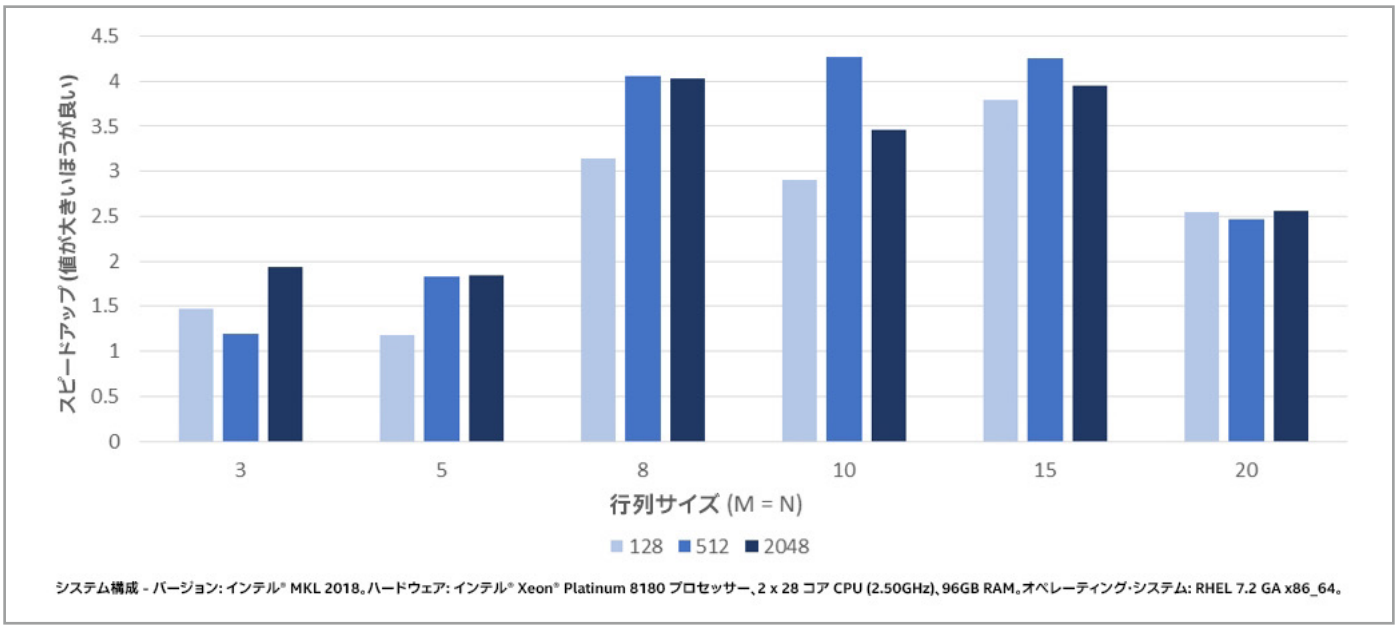


7 インテル® MKL の **`mkl_dgemm_compact`** を使用した 512 の小さな行列乗算



8 インテル® MKL の **`mkl_dgetrfnp_compact`** を使用した 512 の小さな LU 分解

関数のパック / アンパックによりオーバーヘッドが生じますが、パックとアンパック呼び出しの間に複数の圧縮関数を呼び出すことでこれを軽減できます。典型的なユースケースは、ピボットを用いない LU 因数分解から逆行列を計算します。**図 9** に示すように、オーバーヘッドが生じても、インテル® MKL の圧縮関数は、一貫して優れたスピードアップをもたらし、一部のサイズや行列の数では、汎用のインテル® MKL 関数の呼び出しと比較して、最大 4 倍高速です。



9 128、512、2,048 の小さな行列の LU 分解 + 逆行列の計算 (ピボットを用いない) のスピードアップ (オーバーヘッドを含む)

代数計算の高速化

クロス行列のベクトル化が可能な非標準レイアウトでデータを格納することで、小さなサイズの行列で BLAS 関数と LAPACK 関数を大幅にスピードアップできます。インテル® MKL 2018 の新しい圧縮関数は、圧縮データレイアウトのパフォーマンスの利点を最大限に得られるように支援します。

[1]K.Kim, T. C. (2017)."Designing Vector-Friendly Compact BLAS and LAPACK Kernels."Proceedings of SC'17 Conference.

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

intel Software

システム & IoT アプリケーションの
イノベーション

インテル® System Studio 2018 販売開始

システムの立ち上げを容易にし、インテル® プラットフォームで最大限の
パフォーマンスを引き出す統合ツールスイートの最新バージョンで
IoT デバイスとアプリケーションの開発を強力に支援。
Java* および Intel Atom® プラットフォームのサポート、
エッジ側のデータ処理を高速化するライブラリー、クラウドコネクターと
400 を超えるセンサーへのアクセス、拡張デバッガー、
自動トレースを含む新しい機能をお試しください。

無料の 90 日間商用ライセンス >

intel SYSTEM STUDIO

#PurePerformance

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、Intel Atom 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。
© 2018 Intel Corporation.

ビッグデータ・アプリケーションで Java* のパフォーマンスを向上する

新しい拡張による数値計算の高速化と向上

Kumar Shiv インテル コーポレーション 主席エンジニア
Rahul Kandu 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

20 年以上前に登場して以来、Java* は広く使用されるエンタープライズ言語になりました。当初は、1 つのコードでどこでも実行できるという特性により採用されていましたが、その後は、既存の Java* プロジェクトを統合して新しいアプリケーションを比較的簡単にビルドできることから採用が加速しました。Apache Hadoop* などのオープンソースの Java* アプリケーション・スタックの成長により、Java* の採用はさらに広まりました。基本的に、開発者は Java* を生産性を向上するものとして見ています。

Java* プログラマーは、1 つのコードでどこでも実行できるようにするため、Java* Virtual Machine (JVM) に依存しています。JVM もハイパフォーマンスを提供します。一方、C/C++ 開発者はコンパイラーに依存しています。C/C++ 言語では、コンパイルは実行パフォーマンスに影響せず、あまり頻繁に行われないため、コンパイラー

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。

は詳細な解析を行い、特殊な最適化を提供できます。それに対して JVM は、JIT (Just-in-Time) コンパイルモデルを使用します。コンパイルは動的に、実行するたびに行われるため、軽量でできるだけ洗練されている必要があります。JIT コンパイラーの最適化については長年にわたってさまざまな取り組みが行われ、一般的なビジネス処理における Java* のパフォーマンスは、C/C++ と遜色のないものになりました。

今日のビジネス・アプリケーションでは、例えば解析やディープラーニングなどの分野において大規模な数値計算が必要になります。これらのユースケースはすべて、大量の行列演算を実行しながら膨大な量のデータを扱います。Java* アプリケーションは、そのようなデータを整理して配布するのに適しています。ビッグデータは JVM 上で実行されているというのは、誇張表現ではありません。しかし、浮動小数点演算を多用する場合には、C やアセンブリーで記述され手動でチューニングされたコードが適しています。数値計算に対応できるように Java* エコシステムを拡張することは、開発者の生産性の向上につながります。

ビッグデータ・アプリケーション、分散型ディープラーニング・プログラム、人工知能 (AI) ソリューションは、既存の Apache Spark* や Apache Hadoop* クラスターで直接実行して、効率良くスケールアウトできます。Apache Spark* フレームワークで動作するマシンラーニング・アルゴリズム、および JVM 上のデータサイエンスの現在の変革により、Java* における SIMD (Single Instruction, Multiple Data) サポートの強化が求められています。SIMD サポートは、ハイパフォーマンス・コンピューティング (HPC)、線形代数ベースのマシンラーニング (ML) アルゴリズム、ディープラーニング (DL) アルゴリズム、人工知能 (AI) などの分野において新たな可能性を切り拓くでしょう。

例として、**OpenJDK* 9** (英語) の新しい Math FMA API について考察し、**インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX)** 対応**インテル® Xeon Phi ™ プロセッサー**でいくつかの ML アルゴリズムの Java* パフォーマンスを検証してみましょう。

FMA (Fused Multiply Add) 演算

最近のインテル® プロセッサーは、FMA 演算 (**`A=A*B+C`**) を含む SIMD 演算を実行するインテル® AVX 命令を備えています。FMA は、線形代数ベースの ML アルゴリズム、ディープラーニングおよびニューラル・ネットワーク (ドット積、行列乗算)、金融および統計計算モデル、多項式評価に非常に役立ちます。FMA 命令は、IEEE-754-2008 標準形式の浮動小数点値に対してベクトル化された **`a*b+c`** 演算を実行します。ここで、**`a*b`** の乗算は無限精度で実行されます。加算の最終結果は、任意の精度へ丸められます。FMA 命令は、第 2 世代**インテル® Core™ プロセッサー**以降で利用できます。SIMD 演算で FMA 命令を使用するには、JVM JIT コンパイラーは、Java* で記述された FMA 演算を CPU プラットフォームで利用可能なインテル® AVX の FMA 拡張にマップする必要があります。

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。

Java* 9 で利用可能な FMA API

OpenJDK* 9 は、java.lang.math パッケージの一部として FMA API を Java* 開発者に提供しています。OpenJDK* 9 には、インテル® AVX の FMA 拡張向けコンパイラー組込み関数が含まれており、最近の CPU (例えば、インテル® Xeon Phi™ プロセッサーや**インテル® Xeon® Platinum 8180 プロセッサー**) では、FMA Java* ルーチンを CPU 命令に直接マップします。そのため、開発者は何もする必要がありません。ただし、FMA 命令を使用する Java* アルゴリズムでは、FMA でないパックド浮動小数点乗算命令と加算命令のシーケンスは、FMA とは異なる結果になる可能性が高いことを考慮すべきです。収束基準を定式化する場合、予期しない最終結果にならないように、中間結果の精度の違いを考慮することが重要です。Java* では、**a*b+c** を通常の浮動小数点式として評価する場合、乗算と加算でそれぞれ一度ずつ丸め誤差が発生します。

Java* では、FMA 演算は Math FMA API でサポートされます。FMA ルーチンは、3 つの引数の FMA 結果を返します。最初の 2 つの引数の正確な積を 3 つ目の引数と合計して、その結果を最も近い倍精度値に一度だけ丸めたものを返します。FMA 演算は、**java.math.BigDecimal** クラスを使用して実行されます。無限大および NaN の演算入力値は、**BigDecimal** でサポートされません。これらの入力値は、正しい結果を計算するため 2 回丸められます。

FMA API は、浮動小数点型の入力値 **a**、**b**、**c** を受け取り、浮動小数点型の結果を返します。単精度と倍精度の両方がサポートされます。FMA 演算は倍精度で実行されます (後述します)。実装は、最初に **BigDecimal** でサポートされない非有限入力値をスクリーニングして処理します。すべての入力が有限範囲内にある場合、次の式は浮動小数点型の入力値 **a** と **b** を **BigDecimal** オブジェクトへ明示的にキャストして、その積を計算します。

```
BigDecimal product = (new BigDecimal (a)).multiply (new BigDecimal (b));
```

3 つ目の浮動小数点型の入力値 **c** がゼロである特殊なケースでは、API は **a*b** の積もゼロの場合を考慮して、ゼロの符号を慎重に扱います。最終結果がゼロの場合、その符号は次の浮動小数点式に従って計算されます。

```
if (a == 0.0 || b == 0.0) {
  return a * b + c;
}
```

c がゼロで、積が非ゼロの場合、**BigDecimal** の積の **doubleValue()** が返されます。

```
else {
  return product.doubleValue ( );
}
```

a、**b**、**c** がすべて非ゼロの場合、次の値が返されます。

```
else {
  return product.add (new BigDecimal (c)). doubleValue ( );
}
```

次の例は、OpenJDK* 9 の FMA API を使用して **A[i]=A[i]+C*B[i]** 操作を行います。

```
for (int i = 0; i < A.length && i < B.length; i++)
   A[i] = Math.fma(C, B[i], A[i]);
```

単精度および倍精度浮動小数点の FMA は、どちらも倍精度で計算され、結果は戻り型に応じて float または double 型で格納されます。double 型は float 型の 2 倍を超える精度であるため、**a*b** の積は正確です。積に **c** を加算するときに一度の丸め誤差が発生します。さらに、float 型の精度ビットは **p** であるのに対し、double 型は (**2p+2**) ビットであるため、**a*b+c** の二度の丸め操作 (1 つは double 型への丸め操作、もう 1 つは float 型への丸め操作) は、中間結果を直接 float に丸めた場合と等しくなります。FMA メソッドの呼び出しは、さらにパフォーマンスを向上するため、利用可能な場合 CPU FMA 命令にマップされます。

最新の OpenJDK* 9 リリースとソースビルドで Java* アプリケーションを実行する場合、JVM は FMA 命令が利用可能なハードウェア (第 2 世代インテル® Core™ プロセッサー以降のインテル® プロセッサー) 向けにハードウェア・ベースの FMA 組込み関数を有効にします。FMA 組込み関数は、**a*b+c** 式の値を計算する JDK9 の **java.lang.Math.fma(a,b,c)** メソッド向けに生成されます。

JVM オプション **-XX:+PrintIntrinsics** を使用して、FMA 組込み関数の使用を確認できます。

```
@ 6   java.lang.Math::fma (12 bytes)   (intrinsic)
```

BLAS マシンラーニング・アルゴリズムにおける FMA パフォーマンス

Java* プログラムでは、Math FMA を使用して計算カーネルを実装できます。Math FMA は、最近の CPU に搭載されている FMA ハードウェア拡張を利用します。最新の OpenJDK* 10 ソースビルドでは、インテル® Xeon Phi™ プロセッサー上で **Math.fma** を使用して BLAS I DDOT (ドット積) のパフォーマンスを最大 3.5 倍向上できます。JVM JIT コンパイラーは、FMA メソッド呼び出しをハードウェア命令にマップして、自動ベクトル化やスーパーワード最適化により SIMD 演算へベクトル化できます。BLAS-I DAXPY のパフォーマンスは、最新の OpenJDK* ソースビルドで Math FMA を使用することで、インテル® Xeon Phi™ プロセッサー上で最大 2.3 倍向上できます。

オリジナルの Java* DAXPY 実装を以下に示します。

```
int _r = n % 4;
int _n = n - _r;
for (i = 0; i < _n && i < dx.length; i+=4) {
    dy[i   +dy_off] = dy[i   +dy_off] + da*dx[i   +dx_off];
    dy[i+1 +dy_off] = dy[i+1 +dy_off] + da*dx[i+1 +dx_off];
    dy[i+2 +dy_off] = dy[i+2 +dy_off] + da*dx[i+2 +dx_off];
    dy[i+3 +dy_off] = dy[i+3 +dy_off] + da*dx[i+3 +dx_off];
  }
for (i = _n; i < n; i++)
  dy[i +dy_off] = dy[i +dy_off] + da*dx[i +dx_off]
```

以下は、DAXPY アルゴリズムの **Math.fma** 実装です。

```
for (i = 0; i < dx.length; i++)
    dy[i] = Math.fma (da, dx[i], dy[i]);
```

以下は、JVM JIT コンパイラーによって生成されるマシンコードです。Java* アルゴリズムに対して FMA ベクトル SIMD 命令が生成され、最新の OpenJDK* ではインテル® Xeon Phi™ プロセッサー上の SIMD **vfmadd231pd** 命令にマップされていることが分かります。

```
vmovdqu64 0x10(%rbx,%r13,8),%zmm2{%k1}{z}
vfmadd231pd 0x10(%r10,%r13,8),%zmm1,%zmm2{%k1}{z}
                        ;*invokestatic fma {reexecute=0 rethrow=0
return_oop=0}
                        ; - SmallDoubleDot::daxpy@146 (line 82
```

Math FMA インターフェイスのアプリケーション

Math FMA アプリケーション・インターフェイスは、基本線形代数計算を実行するプログラムで非常に役立ちます。例えば、密 DGEMM (倍精度行列乗算) の最内計算カーネルは、**Math.fma** を使用して次のように変更できます。

```
for (j = 0; j < N; j++) {
  for (l = 0; l < k; l++) {
      if (b[j + l * ldb + _b_offset] != 0.0)  {
        temp = alpha * b[j + l * ldb + _b_offset];
        for (i = 0; i < m; i++) {
          c[i + j * Ldc + _c_offset] = Math.fma (temp,
                                          a[i + l * lda + _a_offset],
                                          c[i + j * Ldc + _c_offset]);
        }
      }
   }
```

疎行列計算は、次のように変更できます。

```
for (int steps = 0; steps < NUM_STEPS; steps++) {
  for (int l = 0; l < ALPHA; l++) {
    double Total = 0.0;
    int rowBegin = Rows[l];
    int rowEnd = Rows[l+1];
    for (int j=rowBegin; j<rowEnd; j++) {
      Total = Math.fma (A[Cols[j],Value[j],Total);
    }
       y[l] = Total;
  }
}
```

2 項オプション価格決定モデル (一般的な金融アルゴリズム) では、**stepsArray[k]=pdByr*stepsArray[k+1]+puByr*stepsArray[k]** 操作を、FMA を使用して次のように記述することができます。

```
void BinomialOptions (double[] stepsArray, int STEPS_CACHE_SIZE,
                          double vsdt, double x, double s, int numSteps,
                          int NUM_STEPS_ROUND, double pdByr,
                          double puByr) {
    for (int j = 0; j < STEPS_CACHE_SIZE; j++) {
       double profit = s * Math.exp (vsdt * (2.0D * j - numSteps)) - x;
       stepsArray[j] = profit > 0.0D? profit: 0.0D;
    }
    for (int j = 0; j < numSteps; j++) {
       for (int k = 0; k < NUM_STEPS_ROUND; ++k) {
            stepsArray[k] = Math.fma (puByr,
                             stepsArray[k],
                             (pdByr * stepsArray[k+1]));
       }
    }

}
```

Java* パフォーマンスの向上

Java* の新しい拡張により、数値計算の高速化と向上が可能です。JVM は、**`Math.fma()`** の実装でインテル® AVX の FMA 命令を採用しています。これにより、行列乗算、HPC、AI アプリケーションのパフォーマンスが大幅に向上します。

BLOG HIGHLIGHTS

Java* とインテルのテクノロジー : 未来を築く

MICHAEL G. INTEL CORPORATION

2017 年 10 月 2 日に開催された JavaOne 2017 の基調講演において、私は Java* を利用してデータの未来を築くことに関して述べました。日々生成されるデータ量は増大し続けていますが、IoT は間もなく人々によって生成される接続とデータを上回るでしょう。

スマート・コネクテッド・デバイスとセンサーによるインターネットの利用は急速に増えており、膨大な量のデータが生成されています。そのままでは、データはそれほど興味深いものではありません。データから詳細な情報を迅速に抽出することが重要です。

企業にとって、情報取得のスピードは新たな競争上の優位性となります。ビジネスを強化し、コスト削減を促進し、顧客の理解を深め、新しい革新を創出し、新たな戦略を立て、今日のデジタルエコノミーに素早く対応するのに役立ちます。ビジネスは根本的に変わりつつあります。

この記事の続きはこちら (英語) でご覧になれます。>

intel Software

コードに限界なし

素晴らしいコードで無限の可能性を追求しましょう。インテル® Parallel Studio XE を利用して、ベクトル化とスレッド化、強力な新しい解析ツールとプロファイル・ツールなどによりパフォーマンスをスピードアップできます。

評価する ›

#PurePerformance

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 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® Advisor の Python* API によりパフォーマンスの詳細を得る

コードのチューニングに関して適切な決定を下すためのデータの取得

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
Egor Kazachkov インテル コーポレーション シニア・ソフトウェア・デベロッパー

良い設計の判断は、適切なデータを基に下されます。

- 最初にスレッド化 / ベクトル化すべきループは？
- 労力に見合うパフォーマンス・ゲインが得られるか？
- スレッドのパフォーマンスはコア数に応じてスケーリングするか？
- ループにベクトル化の妨げとなる依存性があるか？
- トリップカウントとメモリー・アクセス・パターンは？
- 最新のインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) 命令を利用して効率良くベクトル化されているか？あるいは、以前の SIMD 命令を使用しているか？

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

インテル® Advisor は、コードの開発とモダン化を支援するインテルの統合ツールスイートである、**インテル® Parallel Studio XE** に含まれる動的解析ツールです。インテル® Advisor は、これらの質問とその他多くの疑問に対する答えを提供します。アプリケーションのベクトル化とメモリー・プロファイルに関する詳細なプログラムメトリックを収集できます。さらに、GUI とコマンドラインから利用可能なカスタムレポートに加えて、Python* を使用して収集したデータベースを検索し、強力な新しいレポートを生成できるようになりました。

インテル® Advisor を実行すると、すべての収集データが Python* API によりアクセス可能な独自のデータベースに格納されます。そのため、プログラムメトリックに関するカスタムレポートを生成することができます。この記事では、この新機能の使い方を説明します。

はじめに

最初に、インテル® Advisor 環境をセットアップする必要があります。(ここでは、すべてのスクリプトを Linux* 上で実行していますが、インテル® Advisor の Python* API は Windows* もサポートしています。)

```
source advixe-vars.sh
```

次に、インテル® Advisor のデータをセットアップするため、収集を実行します。一部のプログラムメトリックには、トリップカウント、メモリー・アクセス・パターン、依存性などの追加の解析が必要になります。

```
advixe-cl --collect survey --project-dir ./your_project -- <your-
executable-with-parameters>

advixe-cl --collect tripcounts -flops-and-masks -callstack-flops --project-
dir ./your_project -- <your-executable-with-parameters>
```

マップまたは依存性の収集を実行するには、解析するループを指定します。この情報は、インテル® Advisor の GUI またはコマンドライン・レポートから見つけることができます。

```
advixe-cl --collect map -mark-up-list=1,2,3,4 --project-dir ./your_project
-- <your-executable-with-parameters>

advixe-cl --collect dependencies -mark-up-list=1,2,3,4 --project-dir ./
your_project -- <your-executable-with-parameters>
```

最後に、インテル® Advisor のサンプルをテスト領域にコピーします。

```
cp -r /opt/intel/advisor_2018/pythonapi/examples .
```

この記事では、すべてのスクリプトの実行に、現在インテル® Advisor for Linux* に同梱されている Python* を使用します。標準の Python* ディストリビューションでも同様に動作するはずです。

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。

インテル® Advisor の Python* API を使用する

サンプルは、この新機能により生成可能なレポートのほんの一部です。`columns.py` サンプルを使用して、利用可能なデータフィールドのリストを取得できます。例えば、基本サーベイ収集を実行後、**表 1** のメトリックを確認できます。

表 1. サーベイメトリックの例

機能	コード
コンパイラー・バージョン	`compiler_version`
ベクトル化状況	`is_vectorized`
マングル関数 / ループ	`mangled_name`
バイナリー・スタティック解析により提供されるデータ型	`data_types`
コンパイラーにより適用されたループアンロール係数	`unroll_factor`

インテル® Advisor の Python* API を実装する

単純な例を使用して、インテル® Advisor の Python* API でいくつかの強力なメトリックを収集する方法を示します。最初に、インテル® Advisor のライブラリー・パッケージをインポートします。

```
import advisor
```

次に、収集結果を含むインテル® Advisor プロジェクトを開きます。

```
project = advisor.open_project(sys.argv[1])
```

プロジェクトを作成して収集を実行することもできます。(以下の例では、**open_project** を使用してプロジェクトを開きます。) この例では、メモリー・アクセス・パターン (MAP) 収集の結果データにアクセスします。これは、次のコード行で行います。

```
data = project.load(advisor.MAP)
```

このデータをロードしたら、テーブルをループして、キャッシュ効率に関する統計を収集できます。そして、収集したデータを出力します。

```
import sys
try:
# 最初にインテル® Advisor のライブラリーをインポートする
    import advisor
except ImportError:
    sys.exit(1)
# インテル® Advisor プロジェクトを開く
     project = advisor.open_project(sys.argv[1])
# メモリー・アクセス・パターン (MAP) データをロードする
     data = project.load(advisor.MAP)
# MAP データをループしてキャッシュ効率に関する情報を収集する
for site in data.map:
    site_id = site['site_id']
    cachesim = data.get_cachesim_info(site_id)
    print(indent * 2 + 'Average utilization'.ljust(width) + ' =
{:.2f}%'.format(cachesim.utilization))
```

インテル® Advisor の Python* API に関する上級者向けトピック

インテル® Advisor の Python* API とともに提供されるサンプルは、スクリプトを記述するためのベースとなります。**表 2** にいくつかの高度な機能を示します。

表 2. インテル® Advisor の Python* API の高度な機能

機能	コード
インテル® Advisor プロジェクトの作成	`compiler_version project = advisor.create_project(project_dir)`
インテル® Advisor のサーベイ収集の実行	`data = project.collect(advisor.SURVEY)`
インテル® Advisor のトリップカウント収集の実行	`data = project.collect(advisor.TRIPCOUNTS`
インテル® Advisor のトリップカウント収集の実行と FLOPS データの収集	`data = project.collect(advisor.TRIPCOUNTS, collection_args=['flop'])`
インテル® Advisor のトリップカウント収集の実行、ただしトリップカウントは収集せずに FLOPS データのみ収集	`data = project.collect(advisor.TRIPCOUNTS, collection_args=['flop', 'no-trip-counts'])`
インテル® Advisor のルーフライン収集の実行	`data = project.collect(advisor.ROOFLINE)`
インテル® Advisor のメモリー・アクセス・パターン (MAP) 収集の実行	`data = project.collect(advisor.MAP)`
インテル® Advisor の依存性収集の実行	`data = project.collect(advisor.DEPENENCIES)`

以下に、サンプルの一部を示します。サンプルのリストは定期的に更新されます。

次のコードは、すべての収集データを示す複合レポートを生成します。

```
project = advisor.create_project(project_dir)
```

次のコードは、html レポートを生成します。

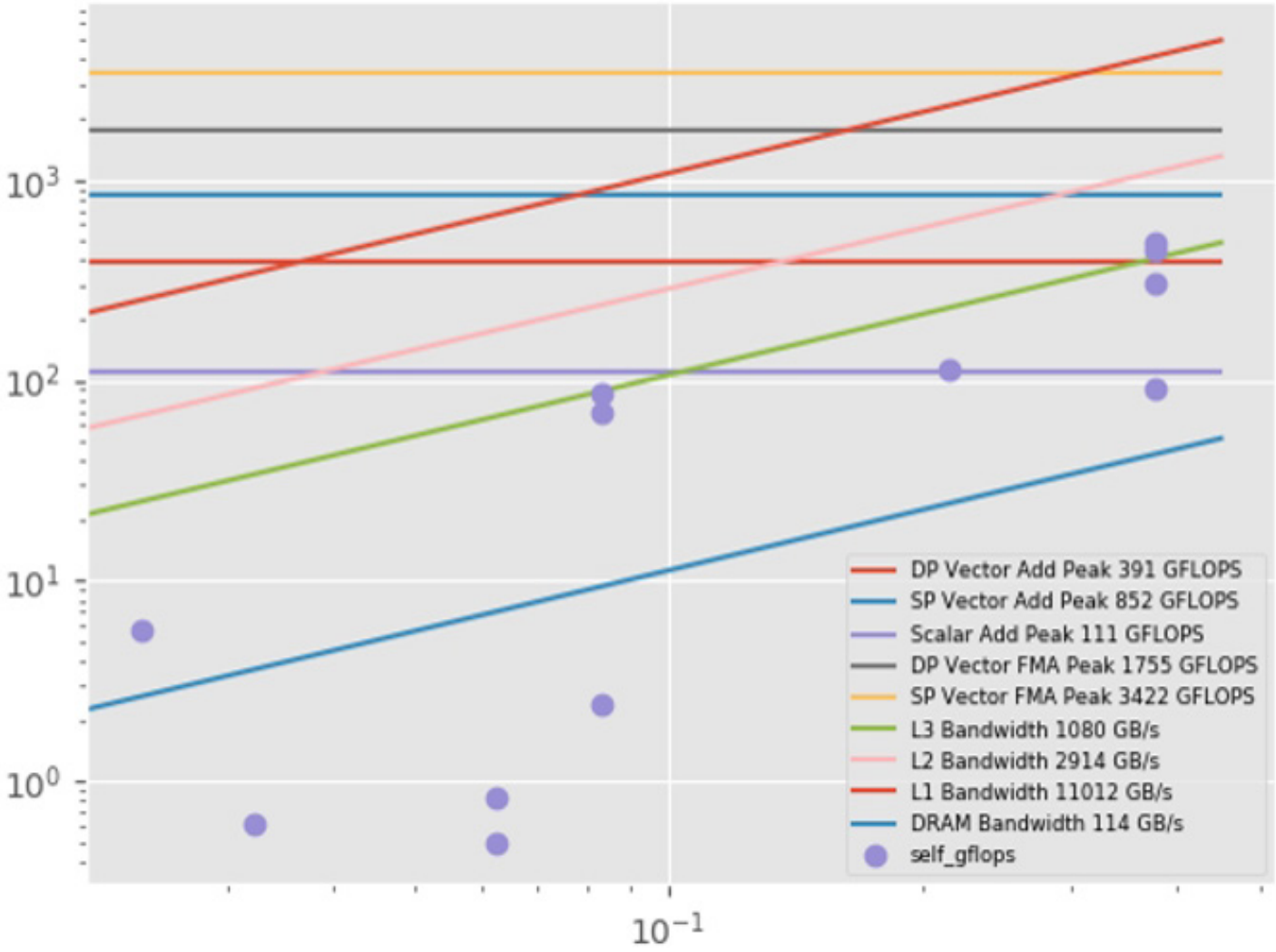
```
advixe-python to_html.py ./your_project
```

次のコードは、ルーフライン HTML グラフ **(図 1)** を生成します。

```
python roofline.py ./your_project
```

roofline.py スクリプトは、**advixe-python** ではなく、外部の Python* コマンドで実行する必要があります。現在は、Linux* でのみ動作します。また、追加のライブラリー NumPy*、pandas*、matplotlib がインストールされている必要があります。キャッシュ・シミュレーション統計を生成するには、次のコードを使用します。

```
advixe-python cache.py ./your_project
```



1 ルーフライン HTML グラフ

キャッシュモデルの結果を**表 3** に示します。

表 3. キャッシュモデルの結果

機能	結果
書き込み	46
読み取り	92
読み取りミス	50
退避されたキャッシュラインの平均使用率	6.25%
使用された退避されたキャッシュラインのバイト数	4
退避された行数	48

ケーススタディー : ベクトル化の比較

このケーススタディーでは、ループを異なるコンパイラー・オプションでコンパイルして、ループのベクトル化を比較する Python* スクリプトを作成します。

ステップ 1: 異なる最適化オプションでコードをコンパイル

最初に、異なるオプションでアプリケーションをコンパイルします。この例では、**インテル® C++ コンパイラー**を使用します (ただし、インテル® Advisor はバイナリーレベルで動作するため、任意のコンパイラーで動作するはずです)。最初のケースでは、**-O0** コンパイラー・オプションを使用して、最適化なしでコンパイルします。2 番目のケースでは、最大限の最適化 (**-O3**) を使用します。

```
icc loops1.cpp -O0 -g -debug inline-debug-info -qopt-report=5 -ipo- -o loops1-
no-opt
icc loops1.cpp -O3 -g -debug inline-debug-info -qopt-report=5 -ipo- -o loops1
```

ステップ 2: Python* コード

スクリプトは非常に単純です。最初にコマンドラインから引数を取得します。インテル® Advisor のプロジェクトが渡された場合、プロジェクトに含まれるデータが使用されます。そうでない場合は、インテル® Advisor のサーベイ実行を行います。サーベイ実行が完了したら、ループのアセンブリーをデコードして、2 つのループで使用された命令を並べて出力します。Python* コードのメイン関数は **`get_formatted_asm`** です。この関数は、インテル® Advisor のデータベースにアクセスして、ループのアセンブリーをデコードします。また、アセンブリー・コードでベクトル命令が使用されているか、およびループの実行時間を確認します。

```
import sys
import itertools

import advisor

# 2 つ目の形式は解析前にデータ収集を許可する
# 1 つ目の形式は収集済みデータを解析するだけ

if len(sys.argv) < 3 or len(sys.argv) > 6:
    print('''

使用法:
advixe-python {} path_to_project_dir loop1 [loop2]

または:
advixe-python {} path_to_project_dir loop1 [loop2] executable1 executable2

loop1 と loop2 は "ソース:行" 形式


''').format(__file__, __file__))
    sys.exit(1)


project_dir = sys.argv[1]
project_dir1 = project_dir + ".1"
project_dir2 = project_dir + ".2"

loop1 = sys.argv[2]
# (スクリプト名を含め) 引数の数が奇数の場合、loop2 は loop1 と同じ
loop2 = sys.argv[3] if len(sys.argv)%2 == 0 else loop1

binary1 = ''
binary2 = ''


# 2 つ目の形式では、最後の 2 つの引数は実行ファイル
if 4 < len(sys.argv) < 7:
binary1 = sys.argv[-2]
binary2 = sys.argv[-1]

# プロジェクトを開くか、作成し、必要に応じて収集を実行
# 指定されたループの書式付きアセンブリー・リストを返す (ベクトル化された命令は "VEC" でマーク)
def get_formatted_asm(project_dir, binary, loop):
    asm = []


    try:
        project  = advisor.open_project(project_dir)
    except:
        project = advisor.create_project(project_dir)


    if binary:
        project.collect(advisor.SURVEY, binary)

    data = project.load(advisor.SURVEY)
    for entry in data.bottomup:
        if loop in entry['function_call_sites_and_loops']:
            asm += ["{:54.54} ".format(entry['function_call_sites_and_loops']),
                    "{:54.54} ".format("Self time: " + entry['self_time']),
                    " "*54]
            for instruction in entry.assembly:
                isVectorized = "VEC" if "VECTORIZED" in instruction['instruction_type'] else ""
                asm.append("{:4.4}{:50.50} ".format(isVectorized, instruction['asm']))
            asm.append("")
    return asm

asm1 = get_formatted_asm(project_dir1, binary1, loop1)
asm2 = get_formatted_asm(project_dir2, binary2, loop2)

# 比較のためアセンブリー・リストと一緒に出力
for (a1,a2) in itertools.izip_longest(asm1, asm2, fillvalue = ' '*40):
    print("{}{}".format(a1,a2))
```

ステップ 3: Python* スクリプトの実行

```
advixe-python compare_asm.py /home/work/projects/loops-compare loops1.
cpp:34 /home/work/tests/loops/loops1-no-opt /home/work/tests/loops/loops1

[loop in main at loops1.cpp:34]
Self time: 45.5463

    Block 1
    movl  -0xbc(%rbp), %eax
    movsxd %eax, %rax
    imul $0x8, %rax, %rax
    addq  -0x88(%rbp), %rax
    movl  -0xac(%rbp), %edx
    imull  -0xac(%rbp), %edx
    mov $0x1, %ecx
    addl  -0xac(%rbp), %ecx
    movq  %rax, -0x28(%rbp)
    mov %edx, %eax
    cdq
    idiv %ecx
    cvtsi2sd %eax, %xmm0
    movl  -0xc0(%rbp), %eax
    cvtsi2sd %eax, %xmm1
    movsdq  0x555(%rip), %xmm2
    divsd %xmm2, %xmm1
    addsd %xmm1, %xmm0
    movq  -0x28(%rbp), %rax
    movsdq  (%rax), %xmm1
    subsd %xmm0, %xmm1
    movl  -0xbc(%rbp), %eax
    movsxd %eax, %rax
    imul $0x8, %rax, %rax
    addq  -0x88(%rbp), %rax
    movsdq  %xmm1, (%rax)
    mov $0x1, %eax
    addl  -0xac(%rbp), %eax
    movl  %eax, -0xac(%rbp)
    movl  -0xac(%rbp), %eax
    cmp $0x64, %eax
    jl 0x401020 <Block 1>
```

```
[loop in main at loops1.cpp:34]
Self time: 4.62404

    Block 1
VEC movdqa %xmm11, %xmm2
VEC movdqa %xmm11, %xmm0
VEC psrlq $0x20, %xmm2
VEC movdqa %xmm10, %xmm1
VEC pmuludq %xmm2, %xmm2
VEC pmuludq %xmm11, %xmm0
VEC psllq $0x20, %xmm2
VEC pand %xmm13, %xmm0
VEC por %xmm2, %xmm0
    callq  0x401770 <__svml_idiv4>
    Block 2
VEC cvtdq2pd %xmm0, %xmm2
VEC punpckhqdq %xmm0, %xmm0
    add $0x4, %r15b
VEC cvtdq2pd %xmm0, %xmm3
VEC addpd %xmm14, %xmm2
VEC addpd %xmm14, %xmm3
VEC subpd %xmm2, %xmm12
VEC subpd %xmm3, %xmm8
VEC paddd %xmm15, %xmm11
VEC paddd %xmm15, %xmm10
    cmp $0x64, %r15b
    jb 0x401397 <Block 1>
```

ステップ 4: インテル® AVX2 のベクトル化を利用して再コンパイル

さらに最適化してみましょう。プロセッサーがインテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2) 命令セットをサポートしているため、インテル® AVX2 命令を生成するようにコンパイラーに指示します。(デフォルトでは、コンパイラーはインテル® AVX2 命令を生成しません。)

```
icc loops1.cpp -O3 -xCORE-AVX2 -g -debug inline-debug-info -qopt-report=5
-ipo- -o loops1-avx2
```

ステップ 5: 比較の再実行

```
advixe-python compare_asm.py /home/work/projects/loops-compare-opt
loops1.cpp:34 /home/work/tests/loops/loops1 /home/work/tests/loops/
loops1-avx2

[loop in main at loops1.cpp:34]
Self time: 4.81401

    Block 1
VEC movdqa %xmm11, %xmm2
VEC movdqa %xmm11, %xmm0
VEC psrlq $0x20, %xmm2
VEC movdqa %xmm10, %xmm1
VEC pmuludq %xmm2, %xmm2
VEC pmuludq %xmm11, %xmm0
%xmm2
VEC psllq $0x20, %xmm2
VEC pand %xmm13, %xmm0
VEC por %xmm2, %xmm0
    callq  0x401770 <__svml_idiv4>
    Block 2
VEC cvtdq2pd %xmm0, %xmm2
VEC punpckhqdq %xmm0, %xmm0
    add $0x4, %r15b
VEC cvtdq2pd %xmm0, %xmm3
VEC addpd %xmm14, %xmm2
VEC addpd %xmm14, %xmm3
VEC subpd %xmm2, %xmm12
VEC subpd %xmm3, %xmm8
VEC paddd %xmm15, %xmm11
VEC paddd %xmm15, %xmm10
    cmp $0x64, %r15b
    jb 0x401397 <Block 1>

[loop in main at loops1.cpp:34]
Self time: 1.97998

    Block 1
VEC vpmulld %ymm15, %ymm15, %ymm0
VEC vmovdqa %ymm14, %ymm1
    callq  0x4018f0 <__svml_idiv8>
    Block 2
    add $0x8, %r15b
VEC vextracti128 $0x1, %ymm0,

VEC vcvtdq2pd %xmm0, %ymm3
VEC vpaddd %ymm13, %ymm15, %ymm15
VEC vcvtdq2pd %xmm2, %ymm5
VEC vpaddd %ymm13, %ymm14, %ymm14
VEC vaddpd %ymm3, %ymm10, %ymm4
VEC vaddpd %ymm5, %ymm10, %ymm6
VEC vsubpd %ymm4, %ymm8, %ymm8
VEC vsubpd %ymm6, %ymm9, %ymm9
    cmp $0x60, %r15b
    jb 0x4015a9 <Block 1>
```

上記の出力から、アセンブリー・コードで XMM レジスターの代わりに YMM レジスターが使用され、ベクトル長が 2 倍になり、2 倍のスピードアップを達成していることが分かります。

結果

最適化と最新のベクトル命令セットを使用することで、パフォーマンスが大幅に向上しました。

- **最適化なし (-O0):** 45.148 秒
- **最適化あり (-O3):** 4.403 秒
- **最適化とインテル® AVX2 (-O3 -AVX2):** 2.056 秒

システム・パフォーマンスの最大化

現代のプロセッサーの潜在的なパフォーマンスを最大限に引き出すためには、ソフトウェアのベクトル化とスレッド化の両方を行うことが重要です。インテル® Parallel Studio XE に含まれる新しいインテル® Advisor の Python* API は、システムのパフォーマンスを最大限に発揮できるように支援するプログラム統計とレポート生成機能を提供します。ここでは、例を用いてこの新しいインターフェイスを紹介しました。これらの例は、特定のニーズに応じてカスタマイズ / 拡張することができます。インテルは、インテル® Advisor の Python* API に関するフィードバックを収集しています。ご意見･ご要望がある場合は、**mvector_advisor@intel.com** まで英語でご連絡ください。

インテル® Advisor
ベクトル化の最適化とスレッドのプロトタイプ生成
詳細

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/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 の商標です。
* その他の名称、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® AI Academy へようこそ

すべてを対象とした AI 教育

Niven Singh インテル コーポレーション ソフトウェア・プログラム・マネージャー

人工知能 (AI) の登場により、最先端の革新に取り組む開発者や学術研究者を支援するため、インテルが明確なビジョンとアプローチを持っていることは驚くべきことではありません。**インテル® AI Academy** (英語) は、初心者とエキスパートに、将来の AI の形成、創造、構築、開発を支援する、教材、ツール、テクノロジーを提供します。

インテルの AI に対する取り組み

インテルは、コンシューマーからエンタープライズまで、すべてのレベルの AI ソリューションの開発を支援する、エンドツーエンドの AI 製品を提供しています。これには、マシンラーニングとディープラーニング専用のハードウェア・ポートフォリオ (**図 1**)、および革新的な AI アプリケーションとソリューションを実現する完全に最適化されたソフトウェア・スタックが含まれます。

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。

1 インテルの AI ポートフォリオ

ライブラリー・レベルでは、**インテル® マス・カーネル・ライブラリー (インテル® MKL)、インテル® 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)、インテル® Distribution for Python*** を含む、さまざまなマシンラーニングおよびディープラーニング・フレームワークで使用されているプリミティブ関数を最適化済みです。

フレームワーク・レベルでは、開発者が任意のフレームワークを使用できるように、最も広く使用されている解析、マシンラーニング、ディープラーニング・フレームワークの最適化に取り組んでいます。

ツールに関しては、ディープラーニングの訓練とデプロイメントを高速化する**インテル® ディープラーニング SDK** (英語)、推論システム用の**インテル® Saffron™ プラットフォーム** (英語) を提供しており、古典的なマシンラーニングとデータ解析向けのオープンソース・プラットフォーム **Trusted Analytics Platform** (英語) の主要コンポーネントです。

インテル® AI Academy へようこそ

インテル® AI Academy は、開発者、データ・サイエンティスト、学生、教員に、将来の AI ソリューションを実験、設計し、インテル® テクノロジーによる実現のために必要なツールを提供する、メンバーシップ・プログラムです。インテルが提供するハードウェアからプラットフォーム、知識まで、あらゆるものを利用できます。

このプログラムは、4 つの重要な柱に基づいています。

1. 学習
2. 開発
3. 共有
4. 教育

Academy のメンバーは、教材やツールを使用して、常に AI 分野の最先端の開発に取り組むことができます。これらのソリューションは**インテル® クラウド・テクノロジー**で実行でき、さらに AI プロジェクトに関わる開発者やエキスパートからのフィードバックとサポートを得ることができます。

インテルは、革新の推進にはテクノロジーとコミュニティーへのアクセスが重要であると考えており、インテル® AI Academy でこれらを提供しています。**図 2** は、インテル® AI Academy により提供される主なものを示しています。

2 インテル® AI Academy により提供される主なもの

インテル® AI Academy は、以下のものを提供します。

- インテルおよびパートナーによる**オンライン・トレーニングとチュートリアル**
- **精選された**学習パス
- インテル® AI Academy コミュニティーのエキスパートによる**技術的なコンテンツ**
- **インテル® AI DevCloud** (英語) への**アクセス**

インテル® AI Academy に参加すると、**インテル® Xeon® スケーラブル・プロセッサー**・ベースのインテル® AI DevCloud への無料アクセスを申請できます。最適化されたフレームワークを利用し、マシンラーニングとディープラーニングの訓練と推論に必要な計算能力に対応できます。

コミュニティー : 開発者、イノベーター、スチューデント・アンバサダー

広範な訓練と開発リソースに加えて、インテル® AI Academy は、革新的で魅力的な AI プロジェクトに取り組んでいる開発者、データ・サイエンティスト、学生、学術研究者から成る強固なコミュニティーへのアクセスを提供します。多種多様なグループと連携することで、インテルは従来の開発者だけでなく、インテル独自のイノベーターやスチューデント・アンバサダーと協力して、さまざまな業界や分野にわたって革新を創造し、促進する機会を得ました。

NASA Frontier Development Lab

インテルは、**インテル® Nervana™ プラットフォーム** (英語) のディープラーニング・テクノロジーを使用して、月極域の詳細なマップの構築に挑戦していた NASA Frontier Development Lab (FDL) (**図 3**) チームの研究者をサポートし、協調してプロジェクトに取り組みました。チームは、ディープラーニングが専門家と同じ結果をより高速に達成できることを実証し、このプロセスを使用して太陽系のすべての岩石オブジェクトの詳細なマップを自動化できることを示唆しました。NASA FDL チームは、インテル® Nervana™ プラットフォームと Neon フレームワークを利用することで、これらの結果を得ることができました。インテルと NASA FDL の共同作業については、**こちら** (英語) を参照してください。

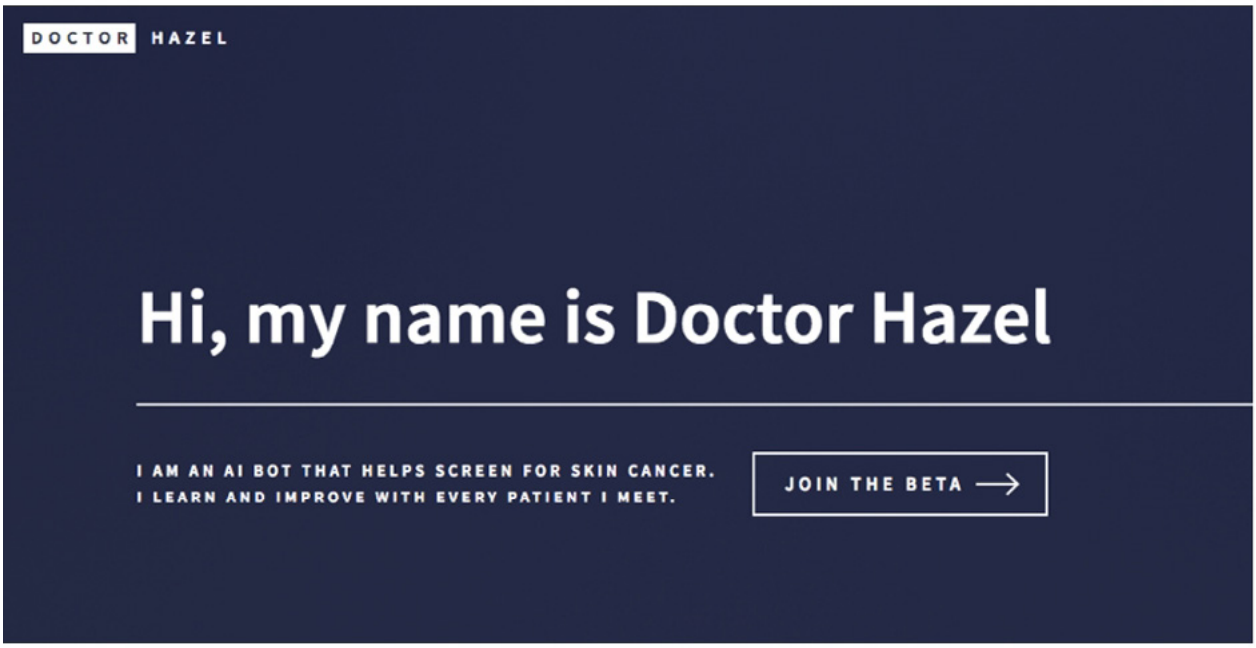
3 NASA Frontier Development Lab

インテルのソフトウェア・イノベーター Peter Ma は、人工知能を利用して癌性のほくろを識別する取り組みを行っています (**図 4**)。**インテル® Movidius™ ニューラル・コンピュート・スティック** (英語) を利用するプラットフォームは、現在 8,000 個のイメージ変数により、次の 4 つの可能性の 1 つを特定します。

1. 異常なし
2. ほくろ
3. メラノーマ
4. その他の癌

Web サイトにほくろやしみのイメージをアップロードすると、数秒で結果が表示されます。AI が癌性であると判断した場合、医療機関での検査を推奨します。処理するイメージの量が増えれば、モデルがさらに洗練され、予測の正解率が向上します。

インテルのソフトウェア・イノベーター Peter Ma と Hazel 博士については、**こちら** (英語) を参照してください。



4 Hazel 博士による皮膚癌検出

ディープラーニングと暗号通貨

インテルのスチューデント・アンバサダー Teju Tadi は、暗号通貨トレーダーの市場心理の検出の開発に対して、ディープラーニングの新しいユースケースの可能性を探っています。このユースケースでは、再帰的ニューラル・テンソル・ネットワーク (RNTN) を使用して、株式市場における価格変動を決定するニュースのヘッドラインを利用した市場心理の解析の概念を、比較的新しい暗号通貨市場に持ち込みます。

インテルのスチューデント・アンバサダー Teju Tadi の暗号通貨プロジェクトの詳細は、**こちら** (英語) を参照してください。

学生と教授にとっての利点

学術的には、インテル® AI Academy は、学生と教授が実践的な経験を得られるように設計されています。

インテル® AI Academy for Students

インテル® AI Academy for Students は、学生がインテルと直接対話できる多数の機会とタッチポイントを提供します。次のような利点があります。

- 最新の最適化フレームワークとツールに関する**インタラクティブなライブ・トレーニング**
- **無料の**トレーニング・コース
- データサイエンスのコンテストやハッカソンへの**参加**
- スチューデント・アンバサダーになる**チャンス**

インテル® Student Ambassador Program

このプログラムは、世界中のトップクラスの大学院生を対象に、インテル® アーキテクチャーの使用により研究を促進し、彼らの専門知識、インスピレーション、イノベーションを紹介します。また、オンラインと業界イベントの両方で学生の取り組みを評価し、インテルとの正式な提携と、エンジニアおよび主題の専門家による指導と支援を提供します。

インテル® AI Academy for Professors

インテル® AI Academy は、学生が AI に簡単に取り組むことができるツールを介して、教授や学術研究者に多数の利点を提供します。専門家主導のレッスンや教材、ビデオ、学習課題に加えて、最新のライブラリー、フレームワーク、ツール、環境などのリソースへの独占的なリモートアクセスを利用できます。これには、学習と教育目的でのインテルのコンピューティング・リソースへのアクセスも含まれます。さらに、助成金や業界の取り組みに興味がある場合は、インテルと協力して指導用コンテンツを作成することもできます。

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。

AI への第 1 歩 : インテル® AI Academy への参加

学習したい場合も、開発したい場合も、作業を共有したい場合も、インテル® AI Academy のメンバーとして行うことができます。次のリソースが役立ちます。

- **インテル® AI DevCloud。** 新しい分野に挑戦するとき、実際に飛び込んでやってみることが 1 番です。AI DevCloud への**アクセスを申請** (英語) して、AI ソリューションのサンドボックス、開発、テスト、最適化を行いましょう。
- **AI Student Kits。** 無料のセルフガイドの AI Student Kits により、AI 理論を理解し、実践演習を使用して学ぶことができます。レッスンでは、PC とサーバー・ワークステーションにおいて、インテル® プロセッサーの利点を活かすことができるツールと最適化ライブラリーを紹介します。**マシンラーニング 101 またはディープラーニング 101** (英語) で今すぐ始めましょう。
- **インテル® Developer Mesh。** **オンライン・デベロッパー・コミュニティー** (英語) で注目プロジェクトの参照、アイデアの提供、作業の共有、関心のある開発トピック (最新の動向、チュートリアル、サポートを含む) のチェックを行うことができます。

インテル® AI Academy
すべてを対象とした AI 教育
詳細 (英語)

Software

THE PARALLEL UNIVERSE

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 Inside、Intel Inside ロゴ、Arria、Arria ロゴ、Intel Atom、Intel Core、Cyclone、Stratix、Xeon、Intel Xeon Phi、Movidius、Intel Nervana、Saffron 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。

JPN/1802/PDF/XL/SSG/SS