intel Software

THE PARALLEL UNIVERSE

インテル® oneAPI を使用したヘテロジニアス・プログラミング

インテル® FPGA 上での圧縮の高速化

ゲームが GPU 依存ではありませんか？

Julia v1.3 の新しいスレッド化機能

Issue 39 2020

目次

編集者からのメッセージ
新しい年が始まりました—インテル® oneAPI 時代の幕開け 3
Henry A. Gabb インテル コーポレーション シニア主席エンジニア

注目記事

インテル® oneAPI を使用したヘテロジニアス・プログラミング 5
複数のアーキテクチャーにわたる多様なワークロードで妥協のないパフォーマンスを提供

インテル® FPGA 上での圧縮の高速化 19
インテル® oneAPI により FPGA がこれまで以上に身近に

ゲームが GPU 依存ではありませんか？ 29
GPU およびデバイス・コンテキスト・キュー解析により容易に判断

Julia v1.3 の新しいスレッド化機能 37
最新 CPU の能力を最大限に引き出す

インテル® Xeon® プロセッサー向け勾配ブースティング木推論の高速化 49
インテル® DAAL の GBT を使用して予測の質とパフォーマンスを向上する方法

第 2 世代インテル® Xeon® スケーラブル・プロセッサーで K 平均法を高速化 61
インテルのビッグデータ向けハードウェアとソフトウェア・スタック

グラフ・アナリティクス・パフォーマンスの測定 71
グラフ・アナリティクスとは何か、またなぜ重要なのか

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。

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。

新しい年が始まりました

インテル® oneAPI 時代の幕開け

本年も The Parallel Universe にお付き合いください。2020 年は興味深い年になりそうです。一部の読者は、私がヘテロジニアス・コンピューティングへの流れを恐れていたことをご存じでしょう。その後、私はそれを避けられないものとして受け入れ、今では積極的に採用しています。ヘテロジニアス (異種性) は利点をもたらすと同時に、困難を伴います。優れたパフォーマンスと効率をもたらしますが、複雑さが増します。いつの日かマシンにすべてを任せられるようになることを願っていますが (「**より多くのソフトウェア開発がマシンに移行している理由**」(英語) を参照)、それまでは**インテル® oneAPI** をはじめとしてこの複雑さを最小限に抑えるための実用的な手段が用いられます。oneAPI は多様な計算アーキテクチャーにわたる単一のソフトウェア抽象化を表すための**オープン仕様** (英語) です。

昨年 11 月に開催された **Intel® HPC Developer Conference** (英語) で、Raja Koduri (インテル コーポレーションのアーキテクチャー、グラフィックス & ソフトウェア・グループのチーフ・アーキテクト兼シニア・バイスプレジデント) によってインテルの oneAPI 実装が発表されました。本号の注目記事「**インテル® oneAPI を使用したヘテロジニアス・プログラミング**」では、ヘテロジニアス・コンピューティングに対するこの統一された標準ベースのアプローチの概要を示します。「**インテル® FPGA 上での圧縮の高速化**」では、**データ並列 C++** (英語) により FPGA を容易に利用する方法を紹介します。ヘテロジニアスに関連して、「**ゲームが GPU 依存ではありませんか ?**」では、**インテル® グラフィックス・パフォーマンス・アナライザー**などの解析ツールを使用してこれを判断する方法を示します。

先号では、Julia プログラミング言語の構成可能なスレッド化について簡単に取り上げました。本号の「**Julia v1.3 の新しいスレッド化機能**」では、Julia Computing, Inc. の Jameson Nash 氏と Jeff Bezanson 氏、Caltech の Kiran Pamnany 氏が、Julia を使用したタスク並列処理のいくつかのコード例を使用して、構成可能なスレッド化について詳しく説明しています。

本号を締めくくるのはデータ解析に関する 3 つの記事です。1 つ目の「**インテル® Xeon® プロセッサー向け勾配ブースティング木推論の高速化**」は、XGBoost* ライブラリーを使用してモデル予測のパフォーマンスを

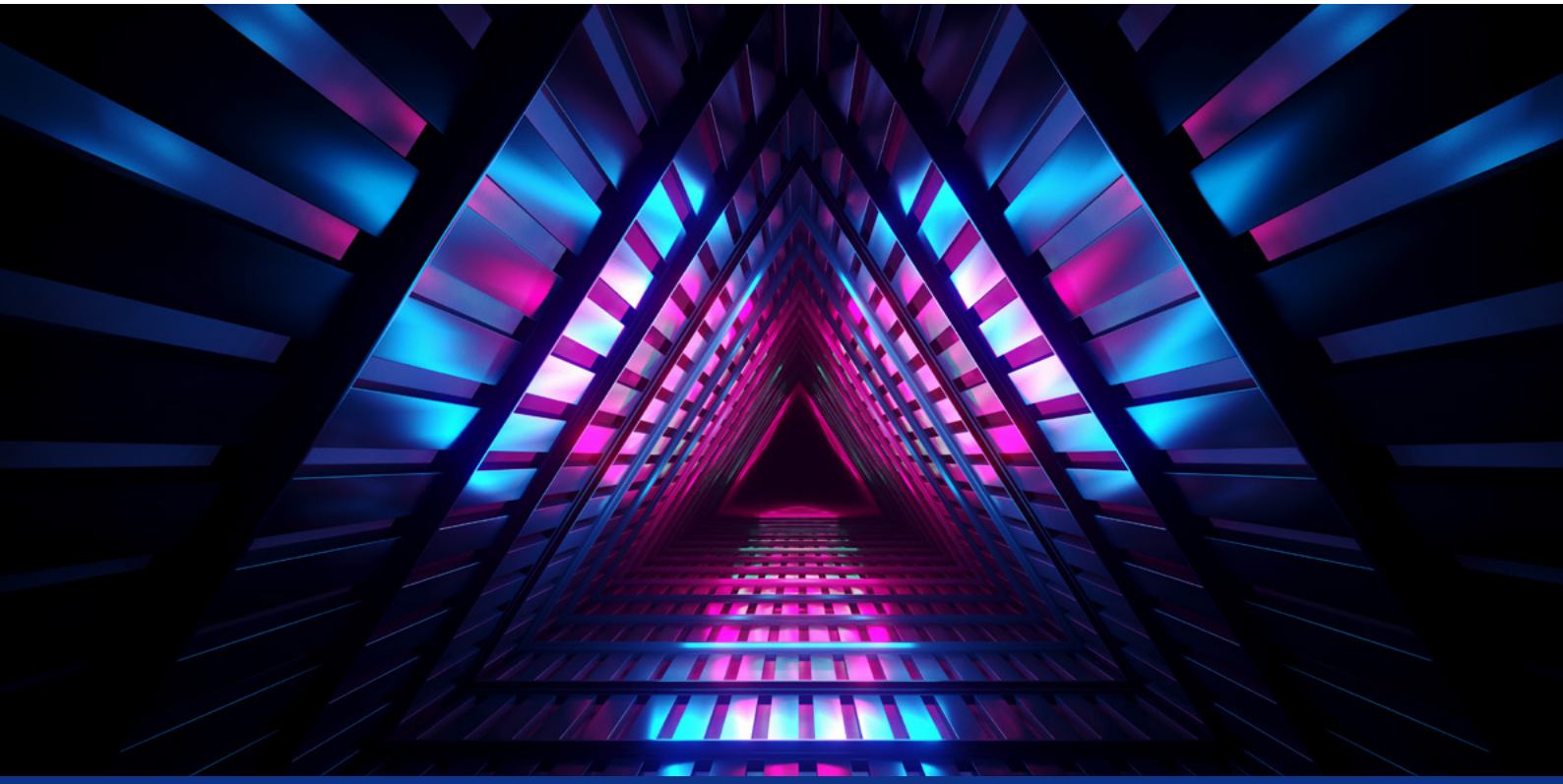
コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

向上する方法を示します。**先号**の注目記事では、XGBoost トレーニングのパフォーマンス向上について取り上げました。2 つ目の「**第 2 世代インテル® Xeon® スケーラブル・プロセッサーで K 平均法を高速化**」は、**Python* 向けインテル® ディストリビューション**と**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)** の最適化を利用して K 平均法を行う方法を紹介します。最後の「**グラフ・アナリティクス・パフォーマンスの測定**」では、グラフ・アナリティクス・ベンチマークの正しい方法と間違った方法について述べます。グラフ・アナリティクスは非常に広範で多様であるため、皆様からのご意見をぜひお寄せください。

今後の号では、インテル® oneAPI に関する多くの記事を掲載する予定です。また、コードの現代化、ビジュアル・コンピューティング、データセンターとクラウド・コンピューティング、データサイエンス、システムと IoT 開発向けのインテルのソリューションの詳細は、**Tech.Decoded** (英語) を参照してください。

Henry A. Gabb

2020 年 1 月



インテル® oneAPI を使用したヘテロジニアス・プログラミング

複数のアーキテクチャーにわたる多様なワークロードで妥協のないパフォーマンスを提供

Nitya Hariharan インテル コーポレーション アプリケーション・エンジニア
Rama Kishan Malladi 同パフォーマンス・モデリング・エンジニア
Amarpal S. Kapoor 同テクニカル・コンサルティング・エンジニア
Kevin P O' Leary 同テクニカル・コンサルティング・エンジニア

今日のハードウェアが達成可能な最大パフォーマンスを引き出すには、ハードウェア機能を適切に使用することと、移植性が高く保守が容易な電力効率の良いコードのバランスをうまく取ることです。これらの要因は必ずしも連携して作用するとは限りません。ユーザーのニーズに応じて優先順位付けする必要があります。ユーザーにとって、アーキテクチャーごとに個別のコードベースを維持することは容易ではありません。スカラー、ベクトル、行列、および空間アーキテクチャーでシームレスに動作する標準化に準拠して簡素化されたプログラミング・モデルは、コードの再利用を高め、習得への投資を軽減することで開発者の生産性を高めます。

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。

oneAPI は、これらの利点を提供するインテル主導の業界イニシアチブです。標準およびオープン仕様に基づいており、**データ並列 C++ (DPC++)** (英語) 言語とドメイン・ライブラリー群が含まれます。oneAPI の目標は、業界のハードウェア・ベンダーがそれぞれの CPU やアクセラレーター向けに独自の互換性のある実装を開発することです。これにより、開発者は複数のアーキテクチャーとベンダーデバイスにわたって、単一の言語とライブラリー API でコーディングすることができます。

インテル製 CPU とアクセラレーター向けのインテルのベータ版 oneAPI 開発ツール実装には、**インテル® oneAPI ベース・ツールキット** (英語) に加えて、異なるユーザー向けのいくつかのドメイン固有のツールキット—**インテル® oneAPI HPC ツールキット** (英語)、**インテル® oneAPI IoT ツールキット** (英語)、**インテル® oneAPI DL フレームワーク・デベロッパー・ツールキット** (英語)、および**インテル® oneAPI レンダリング・ツールキット** (英語)—があります。

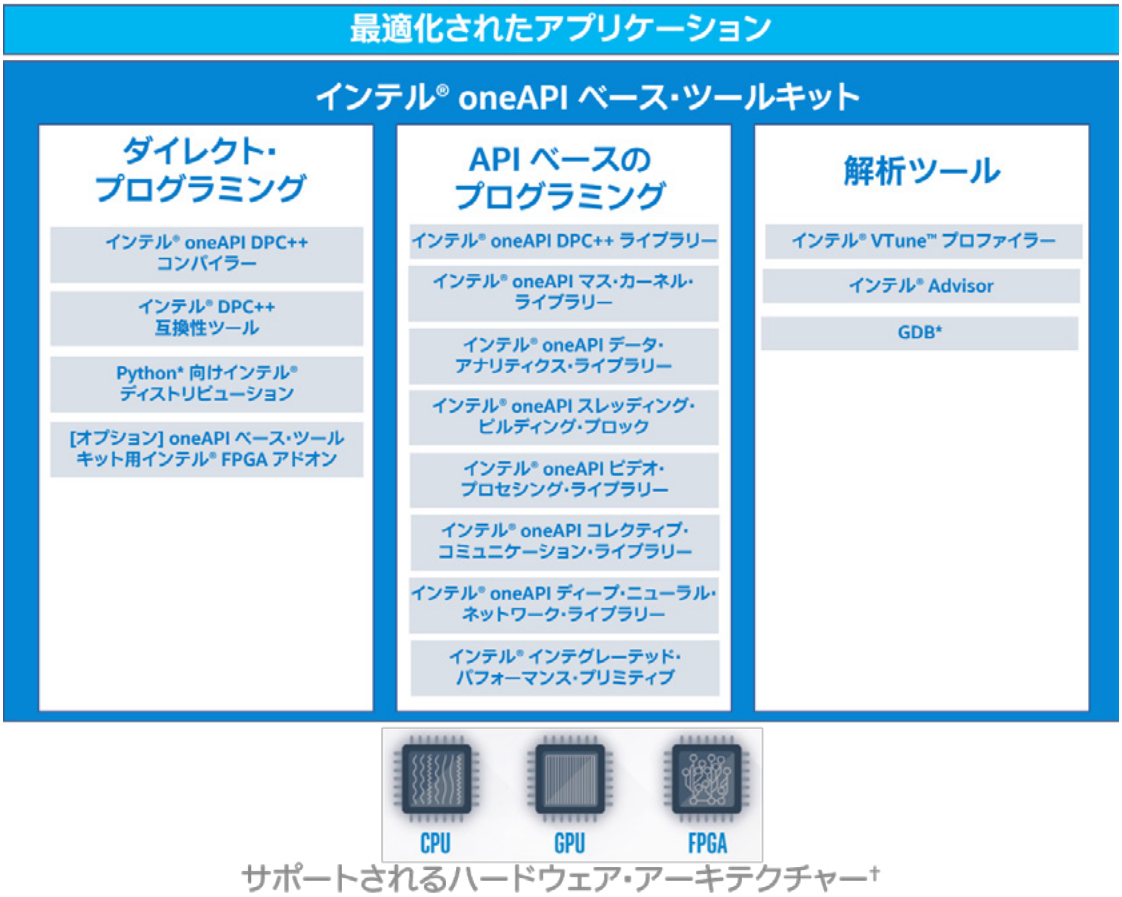
図 1 は、ベータ版**インテル® oneAPI** (英語) 製品とインテル® oneAPI ベース・ツールキットの各種レイヤーを示しています。インテル® oneAPI ベース・ツールキットには、インテル® oneAPI DPC++ コンパイラー、インテル® DPC++ 互換性ツール、複数の最適化ライブラリー、高度な解析およびデバッグツールが含まれます。さまざまなアーキテクチャーにわたる並列性は、Khronos Group の SYCL* をベースとした DPC++ 言語を使用して表現されます。DPC++ 言語は、最新の C++ 機能とインテル固有の拡張によりアーキテクチャーを効率良く使用します。コードは CPU で実行することも、利用可能なアクセラレーターへオフロードすることもできるため、コードの再利用が可能です。フォールバック・プロパティーにより、アクセラレーターが利用できない場合、コードは CPU で実行されます。ホストとアクセラレーターでの実行とメモリー依存関係は明確に定義されます。

インテル® DPC++ 互換性ツールを使用して、CUDA* から DPC++ へコードを移行することもできます。このツールは、開発者が短期間で移行できるように支援し、通常はコードの 80 ~ 90% を自動的に移行します。

DPC++ に加えて、インテル® oneAPI HPC ツールキットは、コードを GPU へオフロードできる OpenMP* 5.0 機能をサポートします。ユーザーは、DPC++ を使用するか、既存の C/C++/Fortran コードのオフロード機能を使用することができます。API ベースのプログラミングは、インテル® GPU 向けに最適化されたライブラリー群 (例えば、**インテル® oneAPI マス・カーネル・ライブラリー** (英語)) によってサポートされます。

ベータ版インテル® oneAPI 製品は、**インテル® VTune™ プロファイラー**[1] と**インテル® Advisor**[2] でも新しい機能を提供しており、アクセラレーターへオフロードしたコードをデバッグしたり、オフロードコードのパフォーマンス・メトリックを確認できます。

インテル® Advisor
現代のハードウェア向けにコードを最適化
詳細



1 **ベータ版インテル® oneAPI ベース・ツールキットのコンポーネント**

この記事では、ヘテロジニアス・プログラミングを推進するベータ版インテル® oneAPI 製品を紹介します。インテル® oneAPI ソフトウェア・モデルについて述べ、コンパイルモデルとバイナリー生成手順を説明します。インテル® oneAPI は、すべてのアーキテクチャーに対応した単一のバイナリーを生成するため、コンパイルとリンク手順が通常のバイナリー生成方法と異なります。最後に、いくつかのサンプルプログラムを検証します。この記事では、「アクセラレーター」、「ターゲット」、「デバイス」という用語を同じ意味で使用していることに注意してください。

インテル® oneAPI ソフトウェア・モデル

SYCL* 仕様ベースのインテル® oneAPI ソフトウェア・モデルは、コード実行とメモリー使用の観点からホストとデバイス間の相互作用を表します。モデルは 4 つの部分で構成されます。

1. **プラットフォーム・モデル**は、ホストとデバイスを指定します。
2. **実行モデル**は、デバイスで実行されるコマンドキューとコマンドを指定します。
3. **メモリーモデル**は、ホストとデバイス間のメモリー使用を指定します。
4. **カーネルモデル**は、計算カーネルのターゲットをデバイスにします。

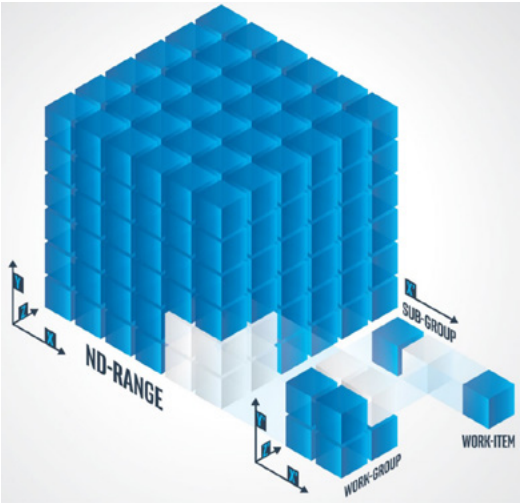
プラットフォーム・モデル

インテル® oneAPI プラットフォーム・モデルは、ホストと、互いにまたはホストと通信する複数のデバイスを指定します。ホストはデバイス上のカーネルの実行を制御し、複数のデバイスが存在する場合はそれらの間で調整します。各デバイスは、複数の計算ユニットを持つことができます。そして、各計算ユニットは、複数の処理要素を持つことができます。プラットフォームがインテル® oneAPI ソフトウェア・モデルの最小要件を満たしていれば、インテル® oneAPI 仕様は GPU、FPGA、ASIC など、複数のデバイスをサポートできます。通常、特定のオペレーティング・システム、特定の GNU* GCC バージョン、デバイスに必要な特定のドライバーがホストにインストールされている必要があります (プラットフォーム要件の詳細は、インテル® oneAPI の各コンポーネントのリリースノートを参照)。

実行モデル

インテル® oneAPI 実行モデルは、ホストとデバイスでのコードの実行方法を指定します。ホスト実行モデルは、カーネルの実行とホストとデバイス間のデータ管理を調整するためコマンドグループを作成します。コマンドグループは、インオーダーまたはアウトオブオーダー・ポリシーで実行可能なキューに送信されます。次のコマンドが実行される前にデバイス上の更新データをホストが利用できるように、キュー内のコマンドを同期できます。

デバイス実行モデルは、アクセラレーターでの計算処理を指定します。実行範囲には、一連の要素 (1 次元または多次元のデータセット) が含まれており、**図 2** の 3 次元の場合に示すように ND-Range、work-group、sub-group、および work-item の階層に分割されます。



2 **ND-Range、work-group、sub-group、および work-item の関係**

これは、インテル拡張である sub-group を除いて SYCL* モデルに似ています。work-item は、カーネル内の最小実行単位です。work-group は、work-item 間のデータ共有方法を決定します。この階層レイアウトは、パフォーマンスを向上するために使用すべきメモリーの種類も決定します。例えば、work-item は通常デバイスメモリーに格納されている一時データを操作し、work-group はグローバルメモリーを操作します。sub-group は、ベクトルユニットを備えたハードウェア・リソースをサポートするために追加されました。これにより要素の並列実行が可能です。

図 2 から ND-Range 内の work-group または work-item の場所が重要であることは明らかです。これは、計算カーネル内で更新されるデータポイントを決定します。各 work-item が操作する ND-Range のインデックスは、nd_item クラスの組込み関数 (**`global_id`**、**`work_group_id`**、**`sub_group_id`**、および **`local_id`**) を使用して決定されます。

メモリーモデル

インテル® oneAPI メモリーモデルは、ホストとデバイスによるメモリー・オブジェクトの処理を定義します。これは、アプリケーションのニーズに応じて、ユーザーがメモリーの割り当て場所を決定するのに役立ちます。メモリー・オブジェクトは、タイプバッファーまたはイメージとして分類されます。メモリー・オブジェクトの場所とアクセスモードは、アクセサーを使用して指定できます。アクセサーは、ホスト上のオブジェクト、デバイス上のグローバルメモリー、デバイスのローカルメモリー、またはホスト上のイメージに対して異なるアクセスターゲットを提供します。アクセスタイプは、読み取り、書き込み、アトミック、または読み書きです。

統合共有メモリーモデルにより、ホストとデバイスは明示的なアクセサーを使用せずにメモリーを共有できます。イベントを使用する同期は、ホストとデバイス間の依存関係を管理します。ユーザーは、ホストまたはデバイスで更新されたデータをいつ再利用できるようにするか制御するイベントを明示的に指定するか、暗黙的にこの決定をランタイムまたはデバイスドライバーに任せることができます。

カーネル・プログラミング・モデル

インテル® oneAPI カーネル・プログラミング・モデルは、ホストとデバイスで実行されるコードを指定します。並列処理は自動では行われません。ユーザーが言語構造を使用して明示的に指定する必要があります。

DPC++ 言語は、ホスト側で C++11 以降の機能をサポートできるコンパイラーを必要とします。しかし、デバイスコードには、C++03 機能と、ラムダ式、可変引数テンプレート、右辺値参照、エイリアス・テンプレートなどの特定の C++11 機能が必要です。**`std::string`**、**`std::vector`**、および **`std::function`** サポートも必要です。仮想関数と仮想継承、例外処理、ランタイム型情報 (RTTI)、および new と delete 演算子を使用するオブジェクト管理を含むデバイスコードには特定の制限があります。

ユーザーは、さまざまな方法でホストコードとデバイスコードを区別できます。ラムダ式は、カーネルコードとホストコードを一致させることができます。ファンクターは、ホストコードを同じソースファイルの別の関数として保持できます。OpenCL* コードを移行するユーザー、またはホストとデバイスコード間に明示的なインターフェイスを必要とするユーザー向けに、カーネルクラスは必要なインターフェイスを提供します。

ユーザーは、3 つの異なる方法で並列処理を実装できます。

- **単一のタスク :** 単一の work-item でカーネル全体を実行します。
- **`parallel_for` 構造 :** 処理要素全体でタスクを分散します。
- **`parallel_for_work_group`:** **`parallel_for_work_group`** 構造は、work-group 全体でタスクを分散します。work-group 内の work-item は、バリアを使用して同期できます。

インテル® oneAPI コンパイルモデル

インテル® oneAPI コンパイルモデルは、ビルドとリンク手順で構成されます。生成されるバイナリーは、複数のアクセラレーターでデバイスコードの実行をサポートする必要があります。つまり、DPC++ コンパイラーとリンカーは、バイナリーを生成するため追加のコマンドを実行する必要があります。通常、ユーザーが意識することはありませんが、これはターゲット固有のバイナリーを生成するのに役立ちます。

ホストコードのコンパイルは、標準の x86 アーキテクチャー向けの方法で行われます。アクセラレーター向けのバイナリーの生成は、1 つまたは複数のアクセラレーターに加えて、各アクセラレーターに固有の最適化をサポートする必要があるため、より複雑です。このアクセラレーター・バイナリーは、ファットバイナリーと呼ばれ、次のものが含まれます。

- **SPIR*-V (Standard Portable Intermediate Representation) 中間表現 :** デバイスに依存せず、コンパイル中にデバイス固有のバイナリーを生成します。
- **ターゲット固有バイナリー :** コンパイル時に生成されます。インテル® oneAPI は複数のアクセラレーターをサポートするため、複数のコード形式が作成されます。

clang ドライバー、ホストとデバイスの DPC++ コンパイラー、標準 Linux* (**`ld`**) または Windows* (**`link.exe`**) リンカー、およびファット・オブジェクト・ファイルを生成するツールなど、さまざまなツールがこれらのコード表現を生成します。実行時にインテル® oneAPI ランタイム環境は、ファットバイナリー内にデバイス固有のイメージがないかチェックし、利用可能な場合はそれを実行します。イメージが見つからない場合は、SPIR*-V イメージを使用してターゲット固有のイメージを生成します。

インテル® oneAPI プログラミングの例

このセクションでは、ベータ版インテル® oneAPI DPC++ コンパイラー、OpenMP* デバイスオフロード、インテル® DPC++ 互換性ツールのサンプルコードを紹介します。

DPC++ コードの記述

DPC++ コードを記述するには、DPC++ 言語の API と構文を使用します。**リスト 1** は、C++ (CPU) コードから DPC++ (ホストとアクセラレーター) コードへの変換例を示します。このサンプルコードは、GitHub* で公開されている Högbom CLEAN* アルゴリズムの実装です[4]。このアルゴリズムは、画像内の最大値を見つけて、観測の点広がり関数で畳み込まれたこの点光源の小さなゲインを減算して、最大値がしきい値よりも小さくなるまでこれを繰り返します。実装には 2 つの関数 **`findPeak`** と **`subtractPSF`** があります。これらは **リスト 1** と **2** に示すように、C++ から DPC++ へ移行する必要があります。

リスト 1. `subtractPSF` コードのベースラインと DPC++ 実装

```
subtractPSF C++ code: baseline
for (int y = starty; y <= stopy; ++y) {
  lhsIdx = y * residualWidth + startx;
  rhsIdx = (y - diffy) * psfWidth + (startx - diffx);
  for (int x = startx; x <= stopx; ++x, lhsIdx++, rhsIdx++)
    residual[lhsIdx] -= MUL * psf[rhsIdx];
}
```

```
subtractPSF DPC++ code
sycl::gpu_selector device_selector;
sycl::queue d_queue(device_selector);
sycl::buffer<float, 1>  a_device(psf.data(), psf.size());
sycl::buffer<float, 1>  b_device(residual.data(), residual.size());
unsigned long size_par = (stopy - starty);
sycl::range<1> a_size{size_par};
auto offset = sycl::id<1>(starty);

d_queue.submit([&](sycl::handler &cgh) {
  auto a_in  = a_device.get_access<sycl::access::mode::read>(cgh);
  auto b_res = b_device.get_access<sycl::access::mode::write>(cgh);
  cgh.parallel_for<class T>(a_size,offset,[=](sycl::id<1> idx) {
    int y = idx[0];
    int lhsIdx = y * residualWidth + startx;
    int rhsIdx = (y - diffy) * psfWidth + (startx - diffx);
    for (int x = startx; x < stopx; ++x, lhsIdx++, rhsIdx++) {
      b_res[lhsIdx] -= MUL * a_in[rhsIdx];
    }
  });
});
d_queue.wait_and_throw();
```

C/C++ から DPC++ へ移行するには、次のようなコード変更が必要です。

- **特定のデバイスのデバイスキューを追加** (デバイスセレクター API を使用)
- **デバイス上でバッファーを作成 / アクセス** (**`sycl::buffer/get_access API`** を使用)
- **`parallel _ for` を呼び出して**計算カーネルをスポーン / 実行
- **カーネル実行の完了を待機** (およびオプションで例外をキャッチ)
- **インテル® DPC++ コンパイラーとオプションを使用 : `dpcpp -std=c++11 -O2 -lsycl -lOpenCL`**

リスト 2 に **`findPeak`** 関数のコード変更を示します。ハードウェアの並列処理能力をうまく引き出せるように、DPC++ コードは **`local_work_size`**、**`global_id`**/**`local_id`**、**`workgroup`**、および OpenCL* や OpenMP* の構造に似た多くの API をサポートしています。

リスト 2. `findPeak` コードのベースライン (上) と DPC++ (下) 実装。`clPeak` は値と位置データの構造体です。work-group の同時実行は、グローバル ID とローカル ID、および work-group 内の複数のスレッド (work-item) 間のバリア同期を使用して行います。この parallel_for 実行の結果をレデュースして (リストでは示していません)、work-group 全体の最大値と位置を特定します。

```
findPeak: Baseline
maxVal = 0.0;
maxPos = 0;
const size_t size = image.size();
for (size_t i = 0; i < size; ++i) {
  if (abs(image[i]) > abs(maxVal)) {
    maxVal = image[i];
    maxPos = i;
  }
}
```

```
findPeak: DPC++
auto bufacc = buf.get_access<sycl::access::mode::read>(cgh);
auto resacc = res.get_access<sycl::access::mode::read_write>(cgh);
sycl::accessor<clPeak, 1, sycl::access::mode::read_write,
               sycl::access::target::local>
               local_res(sycl::range<1>(local_size), cgh);
cgh.parallel_for<class ex1>
                (sycl::nd_range<1>(sycl::range<1>(global_size),
                 sycl::range<1>(local_size)),[=](sycl::nd_item<1> item)
{
  size_t global_id = item.get_global_id(0);
  size_t local_id  = item.get_local_id(0);
  size_t local_dim = item.get_local_range(0);
  size_t group_id  = item.get_group(0);

  local_res[local_id].val = 0.0;
  local_res[local_id].pos = 0;

  if(fabs(bufacc[global_id]) > fabs(local_res[local_id].val)){
    local_res[local_id].val = bufacc[global_id];
    local_res[local_id].pos = global_id;
  }

  item.barrier(sycl::access::fence_space::local_space);
  if (local_id == 0) {
    resacc[group_id].val = 0.0;
    resacc[group_id].pos = 0;
    for (int i=0; i  < local_dim; i++) {
      if(fabs(local_res[i].val) > fabs(resacc[group_id].val)){
        resacc[group_id].val = local_res[i].val;
        resacc[group_id].pos = local_res[i].pos;
      }
    }
  }
});
```

OpenMP* オフロードサポート

ベータ版インテル® oneAPI HPC ツールキットは、OpenMP* オフロードサポートを提供します。これにより、ユーザーは OpenMP* デバイスオフロード機能を利用できます。OpenMP* プラグマを使用して C++ で記述されたオープンソースの Jacobi コード[3] を見てみましょう。このサンプルコードは、メインの反復ステップで次の処理を行います。

- Jacobi 更新を計算
- 古い解と新しい解の差異を計算
- 古い解を更新
- 残差を計算

反復コードを**リスト 3** に示します。

リスト 3. OpenMP* プラグマを使用するサンプル Jacobi ソルバー

```
// Iterate M times
#pragma omp parallel private (i, t)
{
#pragma omp for { // Jacobi update on xnew using b and x }
#pragma omp for reduction (+:d) { // Calculate difference d }
#pragma omp for { //overwrite x with x_new }
#pragma omp for reduction (+:r) { // Calculate residual r }
}
```

リスト 4 は、**omp target** 節を使用してデバイス環境に転送するデータを指定する、更新後のコードです。この節は、**to**、**from**、**tofrom**、または **alloc** のいずれかのデータ修飾子とともに使用できます。配列 **b** は変更されないため、**to** を使用します。また、**x** と **xnew** はオフロード・ディレクティブの前に初期化され、デバイス環境内で更新されるため、**tofrom** を使用します。リダクション変数 **d** と **r** も各反復中に設定および更新され、**tofrom map** 節を持ちます。

リスト 4. OpenMP* オフロードプラグマを使用する更新後のサンプル Jacobi ソルバー

```
// Iterate M times
#pragma omp target data map(tofrom:x,xnew) map(to:n,b) map(tofrom:d,r)
{
#pragma omp parallel for { // Jacobi update }
#pragma omp parallel for reduction (+:d) { // Calculate difference d }
#pragma omp parallel for { // Overwrite x_old with x_new }
#pragma omp parallel for reduction (+:r) { // Calculate residual r }
}
```

インテル® oneAPI コンパイラーでオフロード・ターゲット・コードをコンパイルするには、以下を設定する必要があります。

- コンパイラー・パスに関連する**環境変数**
- **関連する**ライブラリー
- **各種**コンポーネント

これらの環境変数へのパスは、ユーザーマシンのインテル® oneAPI 設定に依存します。ここでは、仕様の使いやすさを実証するため、どのマシンでも同様であるコンパイルプロセスを見てみます。コードをコンパイルするには、次のように LLVM ベースの **icx** または **icpc -qnextgen** コンパイラーを使用します。

```
$ icpc -qnextgen -fiopenmp -fopenmp-targets=spir64 -D__STRICT_ANSI__ jacobi.cpp
-o jacobi
```

-D__STRICT__ANSI__ オプションは、GCC 7.x 以上のシステムとの互換性を保証します。**spir64** オプションは、コードのターゲットに依存しない表現を参照して、リンク時または実行時にターゲット固有のコードに移行します。コードを実行するには、次のコマンドを実行します。

```
$ export OMP_TARGET_OFFLOAD=”MANDATORY”
$ export LIBOMPTARGET_DEBUG=1
$./jacobi
```

OMP_TARGET_OFFLOAD の **MANDATORY** は、オフロードを GPU で実行する必要があることを示します。デフォルトでは、オフロードを CPU と GPU で実行できることを示す **DEFAULT** に設定されます。**LIBOMPTARGET_DEBUG** オプションを設定すると、デバッグに役立つオフロードランタイム情報が提供されます。

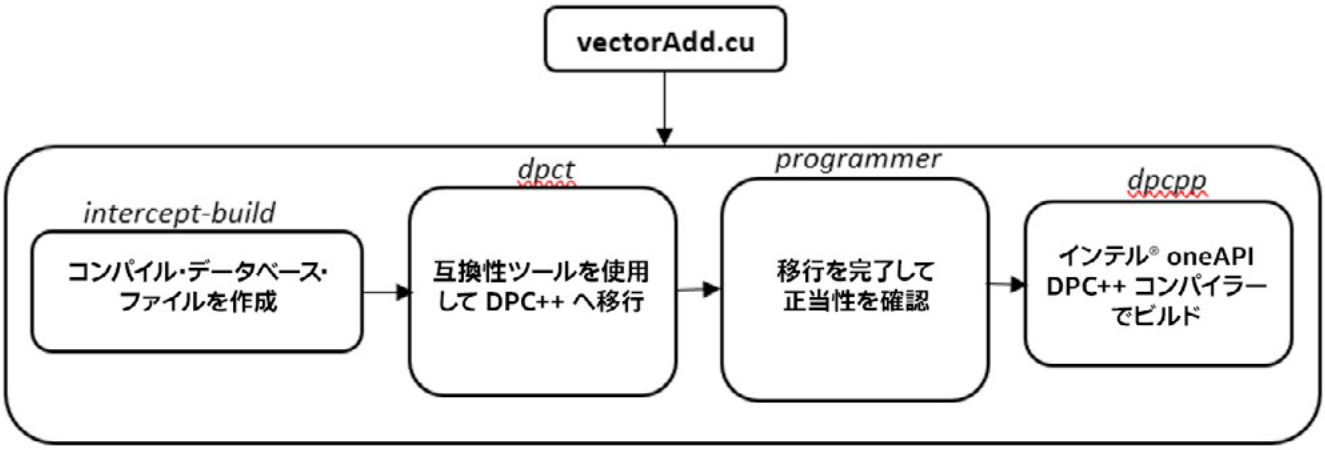
この OpenMP* オフロードサポートの例は C/C++ プログラム向けですが、Fortran オフロードもサポートされています。そのため、Fortran コードベースの HPC ユーザーも GPU 上でコードを実行できます。

インテル® DPC++ 互換性ツール

インテル® DPC++ 互換性ツールは、インテル® oneAPI ベース・ツールキットの一部として提供されているコマンドラインベースのコード移行ツールです。その主な役割は、既存の CUDA* ソースを DPC++ に移行することです。自動移行できないソース位置は適切なエラーと警告で示されます。また、インテル® DPC++ 互換性ツールは、ユーザーの判断が必要なソース位置にコメントを追加します。

図 3 は、CUDA* ソースコードから DPC++ へ移行する際の典型的なワークフローです。インテル® DPC++ 互換性ツールは、現在 Linux* および Windows* オペレーティング・システムをサポートしています。この記事では Linux* 環境を例に説明します。インテル® DPC++ 互換性ツールは、CUDA* SDK に含まれるヘッダーファイルを必要とします。移行プロセスを実証するため、ここでは CUDA* SDK 10.1 の VectorAdd サンプルを使用します。このサンプルは、通常以下の場所にあります。

```
$ ls /usr/local/cuda-10.1/samples/0_Simple/vectorAdd
```



3 **既存の CUDA* アプリケーションを移行するための推奨ワークフロー**

VectorAdd は約 150 行からなる単一ソースのサンプルです。このサンプルの CUDA* カーネル・デバイス・コードは、配列 **A** と **B** のベクトル加算を行い、結果を配列 **C** に格納します。

ここに示すコマンド、パス、および手順は、この記事の執筆時点のものであることに注意してください。製品の最終バージョンでは変更される可能性があります。

インテル® DPC++ 互換性ツールを使用するように環境を初期化するには、次のコマンドを実行します。

```
$ source /opt/intel/inteloneapi/setvars.sh
```

setvars.sh スクリプトは、インテル® DPC++ 互換性ツールだけでなく、インテル® oneAPI ベース・ツールキットのすべてのツールを利用できるように環境を初期化します。

ここでは、**リスト 5** に示す CUDA* Makefile の簡易バージョンを使用します。

リスト 5. CUDA* コードを DPC++ に移行する Makefile

```
$ cat Makefile
CC=/usr/local/cuda-10.1/bin/nvcc
CCFLAGS=-m64 -gencode arch=compute_50,code=sm_50 -gencode
arch=compute_52,code=sm_52 -gencode arch=compute_60,code=sm_60 -gencode
arch=compute_61,code=sm_61 -gencode arch=compute_70,code=sm_70 -gencode
arch=compute_75,code=sm_75 -gencode arch=compute_75,code=compute_75
OBJECT=vectoradd.o
SOURCE=vectoradd.cu
main:
        $(CC) $(CCFLAGS) -o $(OBJECT) -c $(SOURCE)
clean:
        rm -rf vectoradd.o
```

次の手順は、Makefile の実行時に発行されるコマンドをインターセプトして、JSON 形式のコンパイル・データベース・ファイルに格納します。インテル® DPC++ 互換性ツールには、これを行うため **intercept-build** と呼ばれるユーティリティーが用意されています。次のコマンドで起動します。

```
$ intercept-build make
```

そして、実際の変換処理を実行します。

```
$ dpct -p compile_commands.json --in-root=. --out-root=dpct_output vectorAdd.cu
```

--in-root と **--out-root** オプションは、ユーザー・プログラム・ソースの場所と移行後の DPC++ コードの出力先を設定します。この処理は、**./dpct_output/vectorAdd.dp.cpp** を生成します。

ベクトル加算が統合 GPU で実行されるように、デフォルトのキューへ送信する代わりに明示的に GPU キューを指定します。サポートされるプラットフォームの一覧と各プラットフォームのデバイスの一覧は、**get_platforms()** と **platform.get_devices()** を呼び出して取得できます。ターゲットデバイスを検出すると、統合 GPU 用のキューが構築され、このキューにベクトル加算カーネルがディスパッチされます。この手法は、複数の独立したカーネルを同じホスト / ノードに接続されたさまざまなターゲットデバイスへオフロードする際にも使用できます。

次に、変更した DPC++ コードをコンパイルします。

```
$ dpcpp -std=c++11 -I=/usr/local/cuda-10.1/samples/common/inc
vectorAdd.dp.cpp -lOpenCL
```

結果バイナリーを呼び出し、**リスト 6** に示すようにベクトル加算が統合 GPU 上で実行されることを確認します。

リスト 6. 移行した DPC++ コードの統合 GPU での実行結果

```
$ ./a.out
[Vector addition of 5000000 elements]
Platform: Intel(R) OpenCL HD Graphics
 Device: Intel(R) Gen9 HD Graphics NEO
Platform: Intel(R) OpenCL
 Device: Intel(R) Core(TM) i7-7567U CPU @ 3.50GHz
Platform: SYCL host platform
 Device: SYCL host device
Copy input data from the host memory to the device
kernel launch with 19532 blocks of 256 threads
Running on :Intel(R) Gen9 HD Graphics NEO
Copy output data from the device to the host memory
Test PASSED
```

ツールに関する詳細は、次のヘルプオプションで確認できます。

```
$ intercept-build -h
$ dpct -h
```

複数のアーキテクチャーにわたる多様なワークロードで妥協のないパフォーマンスを提供

この記事では、oneAPI とベータ版インテル® oneAPI ツールキットについて紹介し、インテル® oneAPI ベース・ツールキットのコンポーネントの概要を説明しました。ベータ版には、HPC、AI、解析、ディープラーニング、IoT、およびビデオ解析アプリケーションの oneAPI への移行を支援する各種ツールキットが用意されています。『DPC++ プログラミング・ガイド』は、アクセラレーター・パフォーマンスを最適化するさまざまな構造に関する詳細を提供します。記事で紹介した OpenMP* の例は C++ プログラム向けですが、GPU オフロードは C や Fortran でもサポートされます。oneAPI は、コードを移行してさまざまなアクセラレーターで実行するのに必要なソフトウェア・エコシステムを提供します。

参考資料

1. **インテル® VTune™ プロファイラー**
2. **インテル® Advisor**
3. **OpenMP* を使用した Jacobi ソルバー (英語)**
4. **https://github.com/ATNF/askap-benchmarks/tree/master/tHogbomCleanOMP (英語)**

HIGHLIGHTS

oneAPI に関するお客様の声

世界中の企業、大学、機関が oneAPI エコシステムをサポートしており、その輪は拡大しています。

この記事の続きはこちら (英語) でご覧になれます。>

intel
Software

インテル® DevCloud

データセンターからエッジまでさまざまなワークロードに対応した開発サンドボックス

最新のインテル製ハードウェアとソフトウェアを搭載したクラスター上でワークロードを開発、テスト、実行

統合されたインテルの最適化フレームワーク、ツール、ライブラリーを利用することで、プロジェクトに必要なものがすべて揃います。

インテルのさまざまなハードウェアを評価

自然言語処理や時系列解析などの機能を備えた最先端のクラスターと、エッジ・アクセラレーション・ハードウェアを利用して、スキルを高め、さまざまな実験を行うことができます。

インテル® ソフトウェア開発ツールで開発

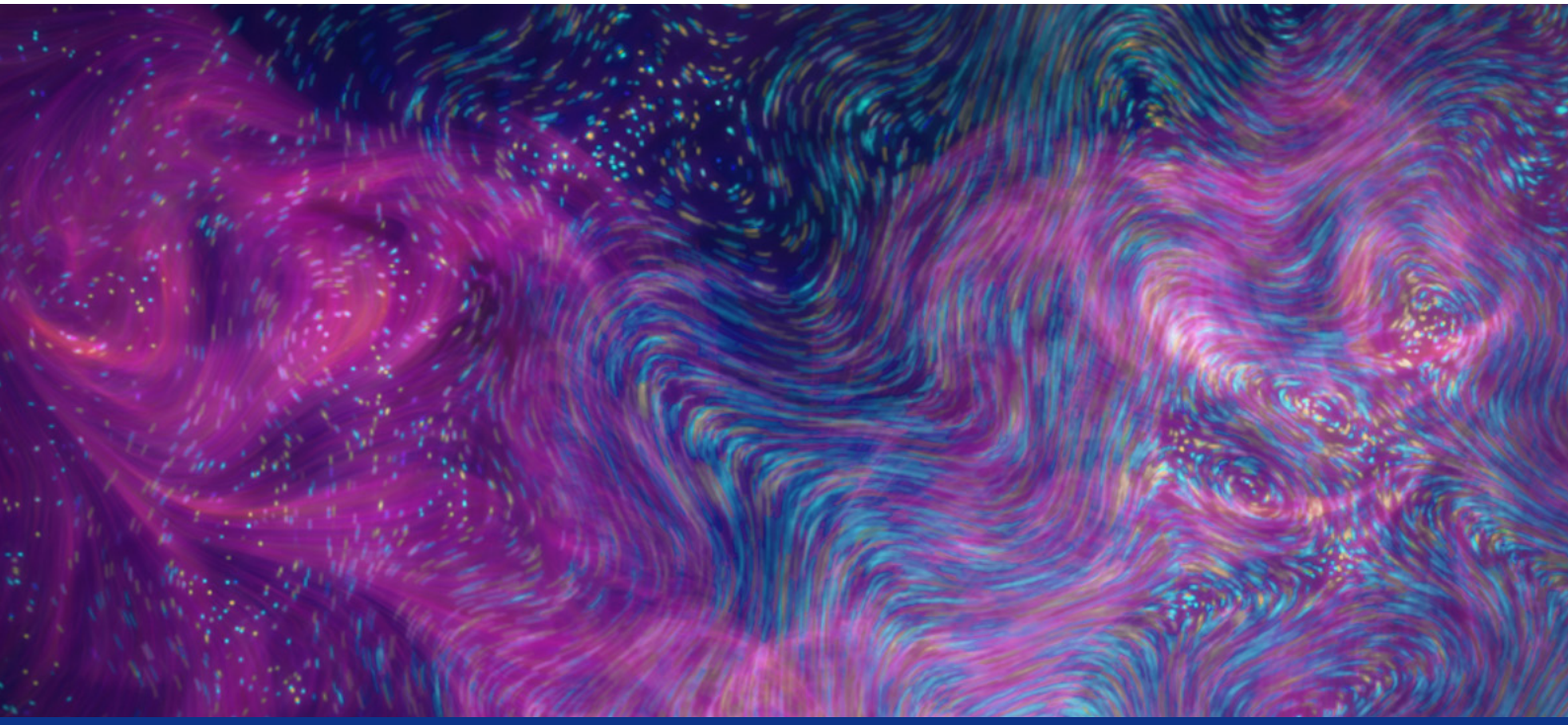
インテルから最新のコンパイラー、パフォーマンス・ライブラリー、ツールをダウンロード、設定、およびインストールせずに、すぐにプロジェクトを開始できます。

ポピュラーな AI フレームワークを使用

トレーニングと推論をすぐに開始できるインテル® Optimized AI Frameworks を利用してアルゴリズムとアプリケーションを高速化できます。

今すぐ開始 (英語) >

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



インテル® FPGA 上での圧縮の高速化

インテル® oneAPI により FPGA がこれまで以上に身近に

Andrei Hagiescu インテル コーポレーション FPGA ソフトウェア・エンジニア
David Cashman インテル コーポレーション FPGA ソフトウェア・エンジニア

フィールド・プログラマブル・ゲート・アレイ (FPGA) は、さまざまなワークロードでハイパフォーマンスを実現できる柔軟なハードウェア・プラットフォームを提供します。この記事では、**インテル® oneAPI** (英語) を利用して実装されたインテルの GZIP 設計例について述べ、インテル® oneAPI によって FPGA を容易に利用できることを示します (インテル® oneAPI の詳細は、本号の注目記事「**インテル® oneAPI を使用したヘテロジニアス・プログラミング**」を参照)。

設計例は、多くのストレージおよびネットワーク・アプリケーションに不可欠な可逆データ圧縮アルゴリズムである DEFLATE を実装します。**SYCL*** (英語) で記述され、**インテル® oneAPI データ並列 C++ (DPC++) コンパイラー** (英語) でコンパイルされたこの設計例は、優れた圧縮率で圧縮時間を大幅に高速化します。

GZIP 設計例は、FPGA で利用可能な空間の超並列処理を利用して、メモリーアクセス、辞書検索、マッチングを並列化することで LZ77 圧縮アルゴリズムを高速化します。インテル® oneAPI を使用して実装されるため、コードはあらゆるコンピュート・テクノロジーをターゲットにすることができますが、ここでは FPGA 向けに最適化します。設計例は、GZIP 互換の圧縮データファイルを生成するため、開発者は標準のソフトウェア・ツールを使用してこの圧縮ファイルを展開できます。

FPGA の仕組み

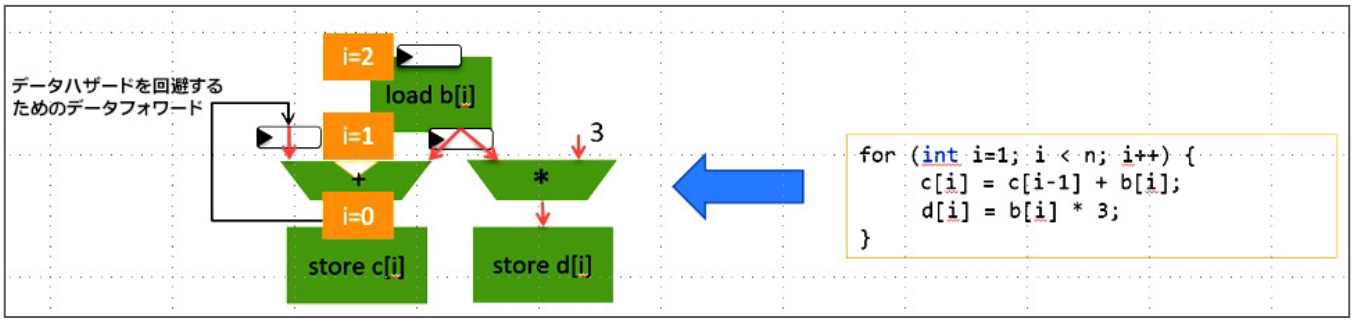
FPGA は、多くの低レベルの計算およびストレージ要素 (加算器 / 乗算器、論理操作、メモリーなど) が 2D アレイとして構成され、再構成可能なルーティングで接続された、再構成可能なデバイスです。これらの要素は、複雑な計算パイプラインと特殊なオンチップ・メモリー・システムを形成することができます。汎用コードを実行するように設計されている従来のアーキテクチャーとは対照的に、FPGA はターゲット・アプリケーションのパフォーマンスを高速化するカスタム・アーキテクチャーを実装するように再構成可能です。例えば、FPGA は、各クロックサイクルでループ反復全体を実行する特殊な計算パイプラインを実装できます。FPGA は、オフチップメモリー (DDR4 など) とホスト CPU 接続 (PCIe* 経由) を備えたアクセラレーション・モデルに最適です。

FPGA 向け DPC++ コンパイル

DPC++ コンパイラーの FPGA バックエンドは、指定されたコード向けに FPGA を再構成するビットストリームを生成します。SYCL* プログラムの各カーネルは、FPGA リソースのサブセットを使用して実装されます。すべての実装カーネルは同時に実行できます。

カーネル内で各ループ本体は、ループ反復全体の処理に必要なすべての機能ユニットを含む、深く特殊なパイプラインに変換されます。条件文は、予測実行としてリファクタリングされます。パイプラインを 1 回横断してループ反復全体を実行します。

コンパイラーは、並列に実行可能な命令を識別して、それらが複数の機能ユニットで並列に実行されるように同じパイプライン・ステージに配置します。さらに、データがフォワードされるようにパイプラインを最適化して、後続のループ反復がデータハザードなしでできるだけ早く (多くの場合は連続したサイクルで) 発行されるようにします (**図 1**)。



1 **ユーザーコードから生成されたパイプライン**

FPGA 向けのコンパイルには、通常数時間かかります。開発サイクルを短縮するため、FPGA バックエンドは、最適化の決定を支援するエミュレーター、スタティック・パフォーマンス・レポート、および動的プロファイラーを備えています。エミュレーションとスタティック・レポートの生成は数分で済み、従来の RTL 開発フローと比較して、FPGA アプリケーション開発の生産性を大幅に向上します。

インテル® oneAPI の利点

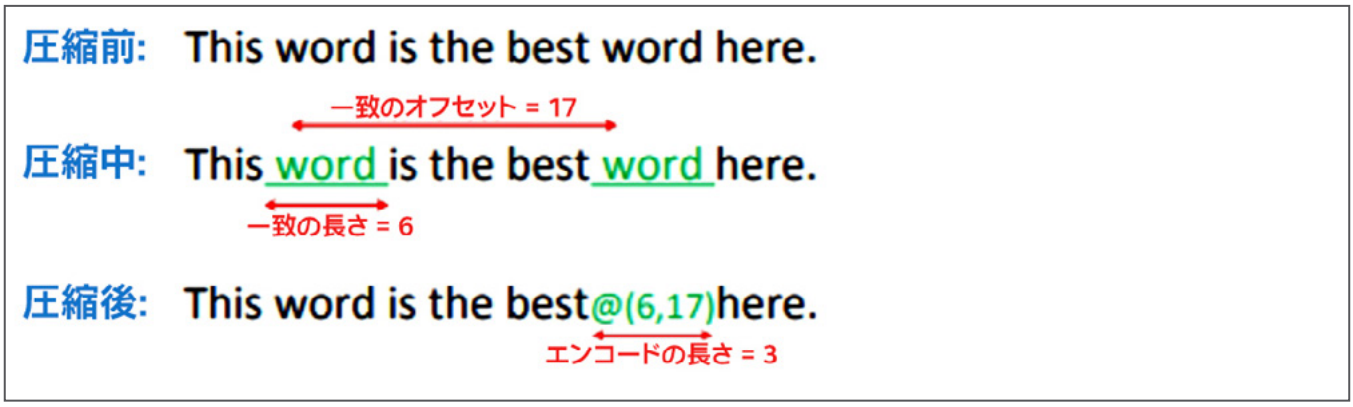
GZIP 設計は、単一ソース設計や複数のアクセラレーターに加えて、アクセラレーターがホットスポット・コードを実行し CPU が非ホットスポット・コードを処理するヘテロジニアス・コンピューティングなど、インテル® oneAPI プログラミング機能を最大限に活用します。GZIP 設計は、インテル® oneAPI を使用して記述されているため、ハードウェア変換はコンパイラー・バックエンドに任せて、アルゴリズムの詳細に集中して C++ で FPGA 向けのコンパイルに適したアルゴリズムを記述できます。アルゴリズムは C++ で表現されるため、容易に動作テストを行うことができます。ハードウェアのパフォーマンスは、ハードウェア・コンパイルを実行する前にレポートを確認することで予測できます。

設計例

GZIP 設計のアーキテクチャーは、DEFLATE のデータフローに従います。処理を行う 3 つのカーネルを作成します。

1. LZ77
2. ハフマン
3. CRC

最初のカーネルは LZ77 データを計算して、ファイルから重複シーケンスを検索して排除します。**図 2** に処理の例を示します。



2 **LZ77 圧縮の例**

重複シーケンスは、前出への相対参照に置き換えられます。参照のエンコードに必要なストレージはオリジナルのテキストよりも少なくて済むため、ファイルサイズが小さくなります。

一致を見つけるには、それまでのすべてのシーケンスを記憶して、最適な一致候補を選択する必要があります。シーケンスを置換した場合は現在の一致の最後から、または一致が見つからなかった場合は次のシンボルから、マッチング処理を継続します。これは容易ではありません。

目標は、各 FPGA クロックサイクルで 16 バイトを処理することです。そのためには、1 クロックサイクルで次の処理を行う必要があります。

- 16 の開始点それぞれで最良の一致を見つけるため、**履歴を確認します。**
- (オーバーラップしない) **最良の 1 つ以上の一致を選択します。**
- 出力に**結果を書き込みます。**
- 読み取った**文字列を辞書に格納します。**

一致はオーバーラップできないため、この処理を並列に実行できるかどうかは明白ではありません。以前の一致にないことが判明するまで、特定のバイトから始まる一致を選択することはできません (次のセクションで詳しく説明します)。

2 つ目のカーネルは、データにハフマン符号化を適用して、最終圧縮結果を生成します。この処理中、すべてのシンボルと参照は可変幅符号化に置き換えられ、頻繁に出現するシンボルのビット数が減り、ファイルサイズがさらに小さくなります。この処理を並列化する方法は簡単です。16 (以上) の独立したハードウェア・ユニットを利用して、それぞれで指定されたシンボルのハフマン符号を決定します。しかし、出力時に注意が必要です。可変長の各出力を出力ストリームの適切な場所に書き込むため、以前のすべての出力の大きさを把握する必要があります。また、ハフマン符号はバイト境界でアライメントされていないため、多くのビット・レベルの操作が必要になります。ここでは詳しく述べませんが、基本的にこの処理は LZ77 ほど複雑ではありません。

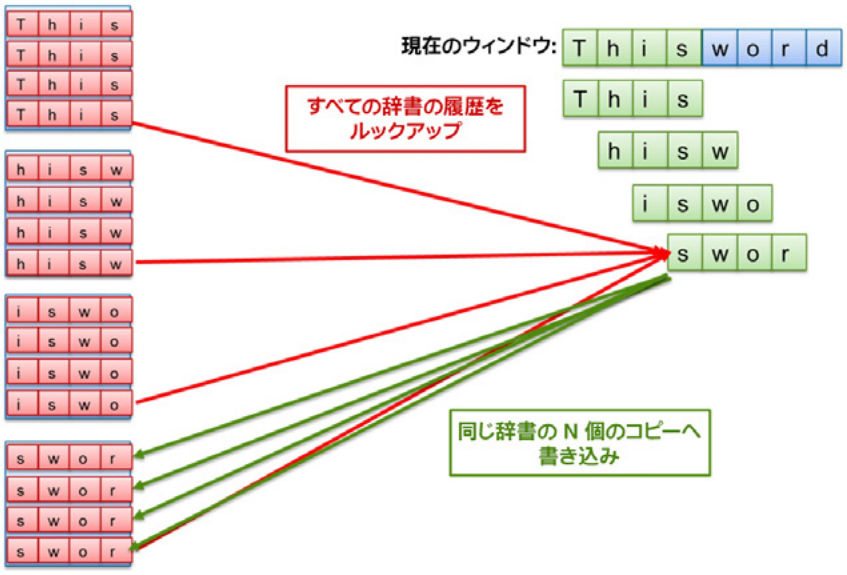
3 つ目のカーネルは入力データの CRC32 を計算します。このカーネルは、ほかの 2 つのカーネルから独立しており、並列に実行できます。また、比較的簡単に FPGA で実装できるため、ここでは詳しく述べません。

FPGA 向けに最適化された LZ77 実装

データ処理のシーケンシャルな性質を示すため、LZ77 エンコーダーは単一のタスクと単一のメインループとして記述されています。コードは、単一スレッドを使用して、ファイル全体でシンボルを 1 つずつ反復します。前出のシーケンスをデータファイルに格納するため、辞書のセットを作成します。これらの辞書には、ハッシュマップと同様に、格納するデータのハッシュを使用してインデックスが付けられます。パフォーマンス上の理由から、より新しい (重複する) エントリーは、同じハッシュキーを持つ前出のエントリーの辞書データを上書きします。辞書は、新しい入力をすべての辞書に書き込むことで更新されます。重複を軽減するため、前出のシーケンスは複数の互いに重複しないセットに分割します。

なぜ複数の辞書が必要なのでしょうか？ サイクルごとに 16 バイトを処理するには、各サイクルで 16 の文字列 (それぞれ各バイトで始まる) を辞書に格納して、16 のハッシュ・ルックアップを実行します。各 FPGA メモリーブロックにはポートが 2 つしかありません。同時アクセスを可能にするには、16 の個別のメモリーシステムを作成して、それぞれ異なる位置に文字列を格納します。これにより履歴が 16 の辞書に分散されるため、16 のハッシュ・ルックアップを 16 の異なる辞書で、つまり合計 256 のルックアップを行う必要があります。これにより問題がより複雑になったように見えるかもしれません。

各辞書のコピーを 16 個、つまり合計 256 個の辞書を作成することで、この問題を解決できます。これらの辞書は FPGA のオンチップメモリーの大部分を使用しますが、各クロックサイクルで 16 のハッシュ・ルックアップを行って書き込むことができます。**図 3** は、クロックサイクルごとに 4 バイトを処理する場合の辞書の構造の例です (ちなみに、辞書専用の FPGA 領域は、1 サイクルあたり 16 バイトを超える処理を妨げる主な要因です)。



3 **4 バイト並列アクセスの辞書の複製**

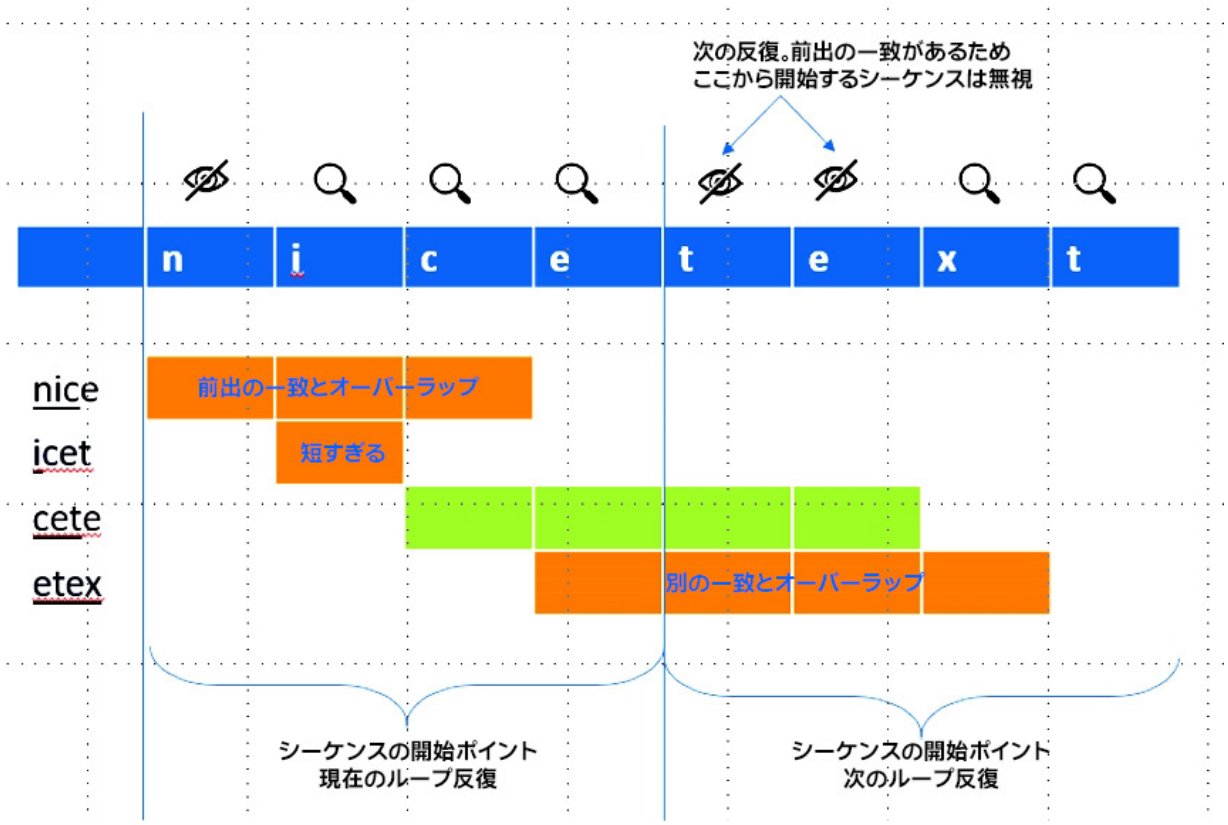
並列処理を表現するのは難しいと思われるかもしれません。しかし、実際には、コンパイラーがルックアップ間のデータ並列性を識別します。そして、コードからすべての辞書ルックアップを同時に実行できる特殊な FPGA ロジックを生成します。コードの **VEC** はサイクルごとに処理されるバイト数で、**LEN** は文字列のサイズです。設計例では、どちらも 16 に設定されています。ここでは、テンプレート・メタプログラミング (アンローラー) を使用して、すべての **i** と **j** に内部関数のコードを複製しています。すべてのルックアップからのデータは、同じパイプラインの一部として、リダクションのような操作で集計されます。

辞書により 1 つの問題は解決しましたが、各クロックサイクルでループ反復全体を完了するためには、さらに多くの処理が必要です。しかし、追加の作業は必要ありません。FPGA コンパイラーは自動的にデータパスをパイプライン化し、ループの反復 0 が辞書からの読み取りを完了すると、反復 1 が開始し、反復 0 は最良の一致の選択を開始します。一致の選択は複数のサイクルにわたる可能性があり、各ステージではループの異なる反復が処理されます。

```
//loop over VEC compare windows, each has a different hash
unroller<0, VEC>::step([&] (int i){
       //loop over all VEC bytes
       unroller<0, LEN>::step{[&] (int j) {

              compare_window[j][0][i] = dictionary_0[hash[1][j];
              compare_window[j][1][i] = dictionary_0[hash[1][j];
              compare_window[j][2][i] = dictionary_0[hash[1][j];
              compare_window[j][3][i] = dictionary_0[hash[1][j];
       }};
}};
```

出力の生成に関する課題の最後の 1 つは、後続の一致への影響を正しく考慮することです。一致が見つかったら、オーバーラップする一致は無視する必要があります。設計例の **single_task** コードでは、ループの後続の反復を続行する前に計算する必要があるループ伝播変数がこれにあたります。計算が複雑すぎると、FPGA のクロック速度を制限したり、各クロックサイクルでループ反復を完了できなくなる恐れがあります。

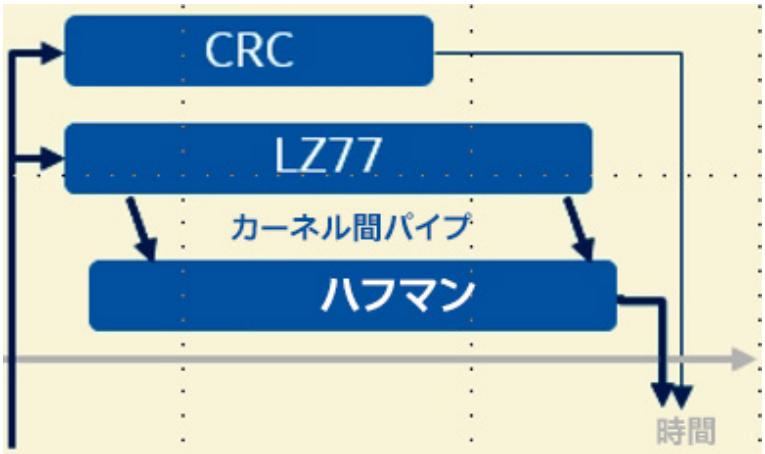
ここでは FPGA をターゲットとするため、ハードウェアをカスタマイズしてこの依存関係の処理を最適化できます。依存関係の処理に関連する計算の量を最小限にすることで、データハザードを単純化します。例えば、ハッシュ・ルックアップを投機的に行い、オーバーラップする一致も含めて、すべての可能な一致を特定します。そして、後でオーバーラップする一致を排除します。FPGA バックエンドは、データハザードを完全に回避するように必要なフォワーディング・ロジックを最適化します。**図 4** は、投機的な一致の排除を示しています。



4 **以前のループ反復で識別された一致、現在のループ反復のオーバーラップする一致、および品質の低い一致を考慮して投機的な一致を排除**

タスクレベルの並列処理

ファイルを圧縮するため、3 つの処理カーネルはホスト CPU によって非同期に送信されます。これらのカーネルは FPGA ハードウェアで同時に実行できます。それぞれの処理時間はわずかに異なり、CRC が最も早く完了します。LZ77 はハフマンカーネルで必要なデータを生成します。追加の遅延とオフチップメモリーを介したデータ転送を回避するため、SYCL* 言語のインテル拡張であるカーネル間通信パイプを使用します。この拡張により、**図 5** に示すように、オフチップメモリーに書き込むことなく、異なるカーネルがデータを順次交換できるようにします。



5 **GZIP のタスク並列処理**

GZIP 設計のビルド

インテル® oneAPI コードサンプル (英語) から **GZIP 設計をダウンロードして実行** (英語) できます。FPGA エミュレーションをターゲットにすることで、正当性と動作の検証が可能です。例えば、**`/bin/emacs-24.3`** を圧縮すると、次のような出力が得られます。

```
make fpga_emu
./gzip.fpga_emu /bin/emacs-24.3 -o=test.gz
Running on device: Intel(R) FPGA Emulation Device
Compression Ratio 34.2421%
PASSED
```

次に、設計のスタティック最適化レポートを生成します。最適化する場合、これらのレポートを精査して、カーネル向けに生成される特殊なパイプラインの構造を理解すべきです。レポートは、**`gzip_report.prj/reports/report.html.make reports`** に生成されます。

最後に、FPGA ハードウェア上で設計をコンパイルして実行します。最適化されたコンパイルには数時間かかります。

```
make fpga
```

パフォーマンス

GZIP を起動するには、次のコマンドを実行します。

```
./gzip.fpga <input_file> [-o=<output_file>]
```

パフォーマンスを評価するため、アプリケーションは圧縮関数を繰り返し呼び出して、全体の実行時間とスループットをレポートします。以下は出力例です。

```
./gzip.fpga_emu /bin/emacs-24.3 -o=test.gz
Running on device: pac_a10 : Intel PAC Platform (pac_f400000)
3.5 GB/s
Compression Ratio 34.2421%
```

FPGA がより身近に

インテル® oneAPI により、FPGA がこれまで以上に利用しやすくなります。空間アーキテクチャーは、多くの場合、驚異的な並列性を備えていないドメインにおいて、著しい高速化の可能性をもたらします。これは、DPC++ コンパイラーとインテル® oneAPI により、容易に利用することができます。

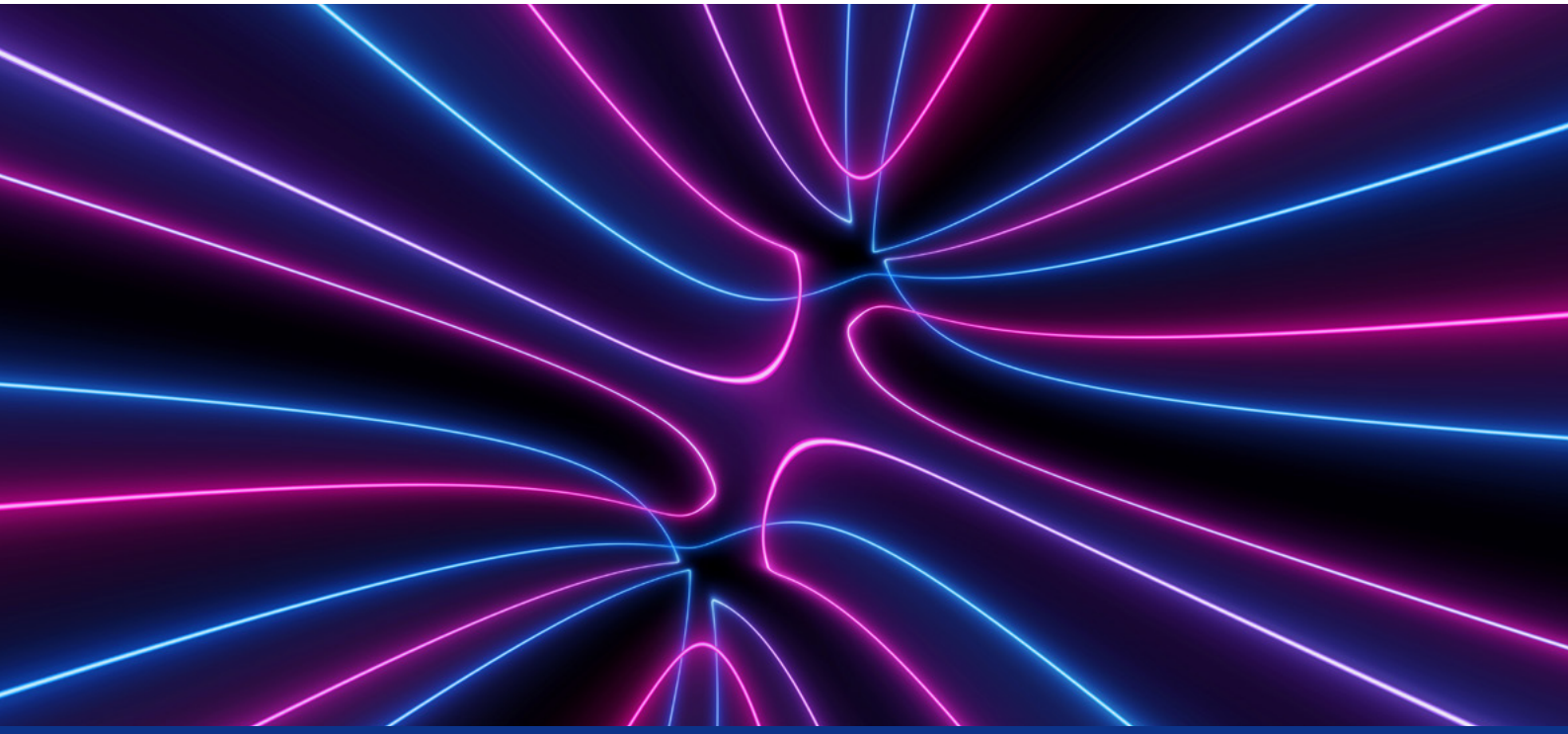
VIDEO HIGHLIGHTS

oneAPI: 効率的なクロスアーキテクチャー開発への道

コンピュート・テクノロジーの進化が加速するにつれて、膨大な量のデータセントリックなワークロードを処理できる多様性を備えた計算ハードウェアへの依存度が世界的に高まっています。インテル コーポレーションのコンピュート・パフォーマンス開発製品グループ本部長の Bill Savage は次のように述べています。「顧客のワークロードのニーズを満たすことこそが、CPU、GPU、FPGA、AI アクセラレーターを含む多様なアーキテクチャーのプログラミングを簡素化するインテル主導のイニシアチブである oneAPI の目的です。」

こちら (英語) からビデオをご覧いただけます。>

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 (software.intel.com/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation またはその子会社の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation



ゲームが GPU 依存ではありませんか？

GPU およびデバイス・コンテキスト・キュー解析により容易に判断

Oleg Fedyaev インテル コーポレーション グラフィックス・ソフトウェア・エンジニア

コンピューター・グラフィックスは、コンピューターでの作業、映画の視聴、スマートフォンの使用、さらには車の運転まで、日常生活において重要な役割を果たしています。グラフィックス・プロセッサーのパフォーマンスは、この 10 年間で劇的に向上しました。そして、その過程におけるビデオゲーム業界の影響は小さくありません。同時に、GPU 機能の継続的な成長は、ゲーム開発者に新たな可能性をもたらし、ハードウェアを最大限に活用する画期的なレンダリング手法やエフェクトの考案を促してきました。しかし、GPU ハードウェア開発者とゲーム開発者のこの競争には課題があります。多くの場合、革新的なレンダリング手法はハードウェアの制限に直面します。

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。

この記事では、高レベルのシステム概要を使用して、ゲームが GPU 依存かどうかを素早く簡単に判断する方法を紹介します。

ゲーム・パフォーマンスの基本

ビデオゲーム制作には多額の費用がかかります。パフォーマンス最適化への投資は、ゲーム・プロジェクトの収益性に影響する重要な工程です。通常、ゲームのジャンル—アクション、アドベンチャー、戦略、その他—によりパフォーマンス要件は異なります。アーティファクトの描画に顕著な停滞や遅延が生じ、ゲームが目に見えて遅い場合は、対処が必要なパフォーマンスの問題が存在します。

ゲームのパフォーマンスを測定するメトリックは、1 秒あたりにレンダリングされるフレーム数 (FPS) を示すフレームレートです。FPS はさまざまなアプリケーションのベンチマークとランク付けに使用され、FPS が高いほうが優れています。ほとんどの場合、このアプローチを使用できます。例えば、FPS が低いと動きが多いアクションゲームは見栄えが良くありません。

最近のゲームは、複数のコンポーネントで構成されており複雑です。

- グラフィックスのレンダリング
- 物理特性の計算
- サウンドの再生
- スクリプトの実行
- ネットワークのホスティング
- その他

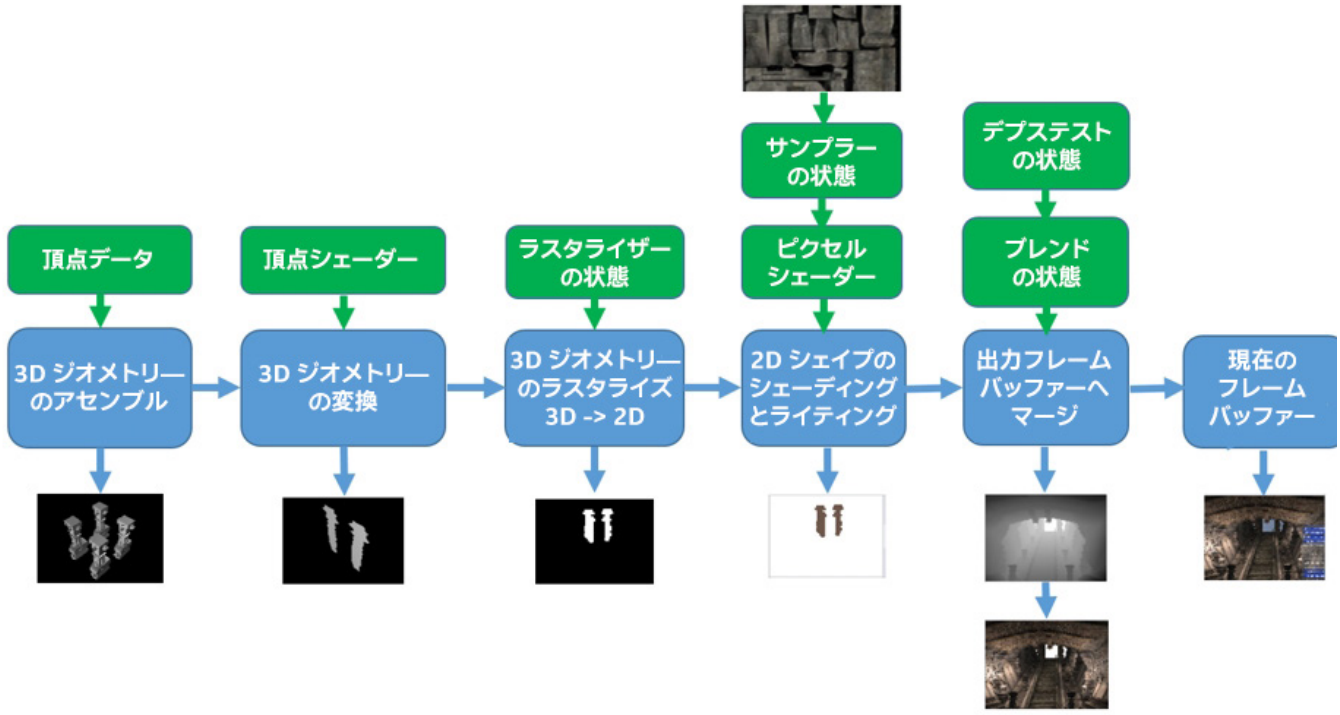
各コンポーネントは、個別にまたは連動して、ゲームのパフォーマンスに影響します。そのため、アプリケーションが GPU 依存か CPU 依存かを特定することは容易ではありません。

レンダリングはゲームを構成するコンポーネントの 1 つですが、グラフィックスはゲームの独特な作風、スピリット、雰囲気を作り出す上で重要であるため、レンダリングから解析を開始することは合理的です。

典型的なレンダリング・パイプライン

グラフィックス・レンダリング・パイプライン、グラフィックス・プログラミング・モデル、およびこの処理におけるグラフィックス・ドライバーの役割を理解せずに、GPU がパフォーマンス・ボトルネックかどうかを正確に判断することはできません。アプリケーション・コードからハードウェアにまで、スタック全体の GPU アクティビティーを徹底的に解析するには、かなりの専門知識が必要です。幸いにも、基本的なパフォーマンス解析を実行すれば、詳しい調査を行わなくても GPU 全体の利用状況を確認できます。

レンダリング・パイプラインの動作にはリソースと状態が伴います。パイプラインにバインドされたリソースは、何を、どこでレンダリングすべきかを指定します。リソースは、適切な形式で記述されたジオメトリー、テクスチャー、およびレンダリング・ターゲットで構成されます。ラスタライズ・パラメーター、デプス条件のテスト、属性のブレンド、およびその他の状態は、画面上にイメージを生成するためこれらのリソースをどのように解釈して処理するかを指定します。レンダリング・リソースと状態は、シェーダーと呼ばれる GPU プログラムと緊密に連携して、レンダリング・パイプラインのさまざまなステージで実行されます (**図 1**)。



1 **レンダリング・パイプライン**

従来のレンダリング・パイプラインは、ソースデータを受け取り、各ステージで順次処理します。最終的な結果が得られるまで、これを繰り返します。レンダリングされたオブジェクトは、最初に仮想空間で変換され、次にスクリーンサーフェスに投影されます。その後、投影の可視部分が色付けされ、フレームバッファー内のほかのレンダリング・オブジェクトとマージされます。

グラフィックス・プログラミング・モデルは単純です。レンダリング・コマンドを正しい順序で送信するため、描画コンテキストを構成して開き、使用します。パイプラインにジョブの実行を指示する描画コマンドを呼び出す前に、必要なリソース、状態、およびプログラムをパイプラインにバインドする必要があります。この処理は、最終的なシーンがフレームバッファーに形成されるまで、オブジェクトのレンダリングに必要な回数だけ繰り返されます。最終的なシーンは、バッファー・スワップ・コマンドでスクリーンに出力されます。OpenGL*、DirectX*、Vulkan*、またはその他のグラフィックス API など、アプリケーションが使用する API に関係なく、この概念は同じです。

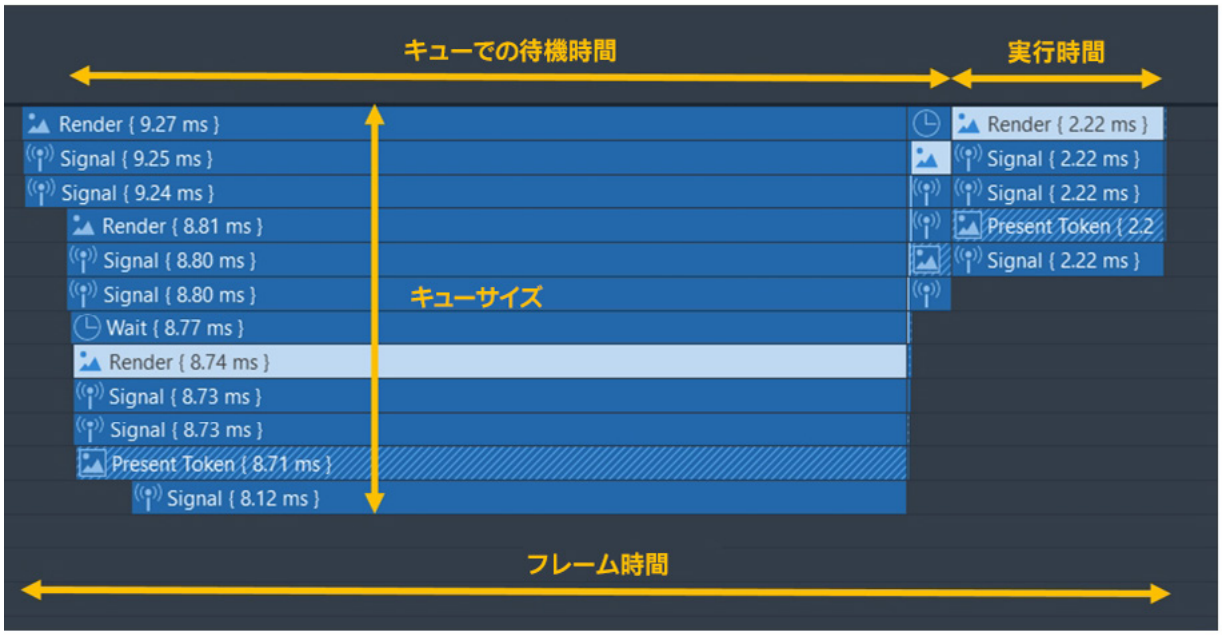
1 つの描画に多くの複雑な処理が含まれていることは明らかです。描画時間には、各処理にかかった時間が含まれます。個々の描画にかかる時間はさまざまで、合計フレーム・レンダリング時間に影響します。長いフレーム時間は GPU 依存を示す可能性があります。これは、グラフィックス・ドライバーのパフォーマンス・メトリックを基に GPU ロードを予測することで確認できます。

グラフィックス・ドライバー・アクティビティー

一般的なグラフィックス・プログラムは、グラフィックス・ドライバーを操作し、直接 GPU を操作することはありません。アプリケーションで描画コンテキストを開くと、ドライバー・デバイス・コンテキストと呼ばれるグラフィックス・ドライバーへのインターフェイスが暗黙的に作成されます。レンダリングを可能にするため、ドライバーは多くの処理を実行する必要があります。

- GPU 上でのメモリーブロックの解放と割り当て
- レンダリングに必要なリソースの CPU から GPU へのアップロード
- GPU 実行ユニットのレジスターの設定
- GPU プログラムのアップロード
- CPU への結果の転送
- その他

アプリケーション・コード内で一連のグラフィックス API を呼び出すと、ドライバーはそれらを GPU が処理可能な一連のコマンドに変換します。コマンドはすぐには GPU で実行されません。代わりに、コマンドバッファーに蓄積されます。ドライバーは、一連のコマンドをパケットにまとめて、実行デバイス・コンテキスト・キューにプッシュして、実行をスケジュールします (**図 2**)。



2 **コマンドバッファー**

デバイス·コンテキスト·キューには、さまざまなコマンドタイプからなるさまざまなパケットタイプが含まれます。パケットタイプは、パケット内で多数のコマンドタイプにより決まります。各パケットは、前のパケットの最後のコマンドが GPU で実行を完了するまでキューで待機します (例えば、**図 2** で選択されている Render パケットを参照)。

デバイス・コンテキスト・キューを調査することで、パフォーマンスに関する有用な情報が得られます。例えば、大きなキューサイズは、多くの場合 GPU に送信されるグラフィックス処理量が多いことを示します。長いパケット実行時間は、計算負荷の高い描画処理が原因の可能性があります。長いパケット待機時間は、非効率なレンダリング·アルゴリズムや同期が原因の可能性があります。

1 つのフレームに関連するすべてのパケットが判明すると、キューに最初のパケットを送信してから、同じフレームについて最後に送信したパケットの最後のコマンドが実行を完了するまでの時間を計算することで、おおよそのフレーム時間を予測できます。

しかし、フレーム時間が長くても、レンダリングを実行するグラフィックス·プロセッサーに関連付けられている GPU ハードウェア·キューを調査するまで、アプリケーションが GPU 依存かどうか判断することはできません。GPU は共有リソースであり、複数のアプリケーションがグラフィックスをレンダリングするため同時に利用できます。長いレンダリング時間は、GPU コンテキストを取得した別のアプリケーションとの同時実行の結果であるかもしれません。

ハードウェア GPU キュー (**図 3**) は、GPU 全体の利用状況を明確にします。このキューを使用して、GPU のビジー状態と、その時点でレンダリングされていたアプリケーションを特定できます。

3 **ハードウェア GPU キュー**

図 3 の GPU キューのスナップショットは、コマンドパケットの色で区別された少なくとも 2 つのアプリケーションが同時にレンダリングしていることを示しています。どちらのアプリケーションも GPU 依存ではありません。青色で示されたアプリケーションのフレーム時間は約 11 ミリ秒であり、これは約 80FPS に相当します。80FPS は通常十分に高い値です。緑色は非常に小さなフレーム (それぞれ約 5 ミリ秒) のバックグラウンド・プロセスのようです。さらに、実行コマンドパケット間に GPU がアイドル状態であった期間を示す複数のギャップがあるため、GPU はそれほどビジーではありません。

ソフトウェアとハードウェア・キューの解析は、信頼性の観点から非常に有望です。また、必要なパフォーマンス・データの収集方法が分かっているため、これらのキューは簡単に構築できます。

システム・イベント・トレース

Windows*、Linux*、macOS* など、使用するオペレーティング・システムに関係なく、システムのイベント・トレース・レイヤーに接続することができます。このレイヤーは、異なるシステムモジュール内の主要な実行ポイントに関連付けられたさまざまなイベントを記録します。一部のイベントは、パフォーマンス解析の対象です。グラフィックス・ドライバーも例外ではありません。ドライバーはコマンドパケットをデバイス・コンテキスト・キューにプッシュしたり、コマンドパケットを GPU へアップロードしたり、またはコマンドパケットの最後のコマンドを実行するたびに、対応するイベントをシステム・トレース・レイヤーへ送信するため、それらを簡単に収集できます。例えば、Windows* でデバイス・コンテキスト・キューと GPU コマンド・パケット・キューを構築する場合、Windows* イベント・トレーシング (ETW) システムの Microsoft-Windows-DxgKrnl プロバイダーからいくつかのイベントを収集する必要があります。

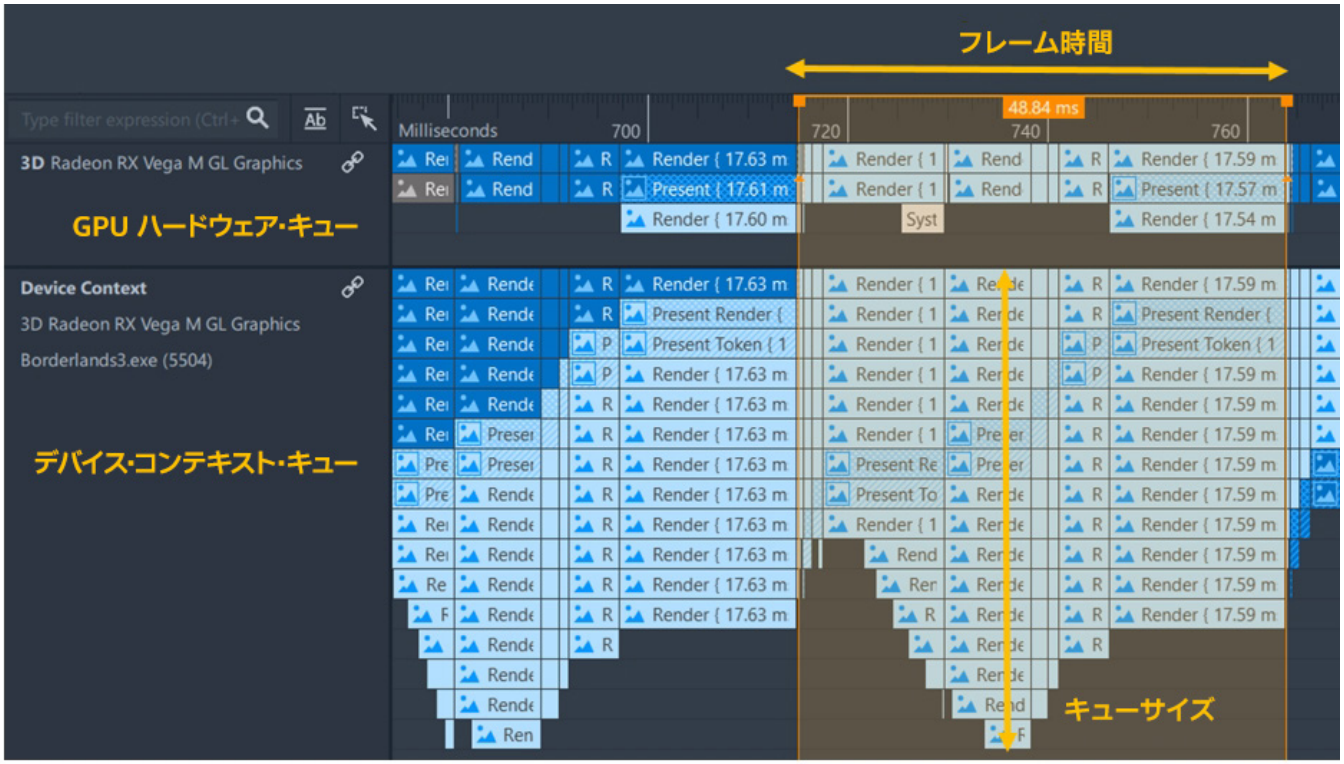
イベントデータにエンコードされているさまざまな属性により、異なるイベントを組み合わせてキュー内の各パケットの現在の状態をいつでも区別することが可能です。

グラフィックス・アプリケーション解析

システム・イベント・トレースは、ドキュメントが充実しており、どのプラットフォームでも使用できます。システム・トレース・データの収集または視覚化には、多くのツールを利用できます。しかし、デバイス・コンテキスト・キューと GPU ハードウェア・キューを適切に、同時に解析できるツールは限られています、**インテル® GPA グラフィックス・トレース・アナライザー** (英語) はその 1 つで、高レベルのシステム解析から単一フレームの描画ごとの解析まで、さまざまな詳細レベルでグラフィックス・アプリケーションのパフォーマンスを解析できるように設計されています。

ここで紹介したことを、実際のグラフィックスを多用するゲームに適用してみましょう。実験を予測可能なものにするため、中程度の性能のグラフィックス・プロセッサーを搭載したワークステーションでテストします。2019 年 9 月にリリースされた大人気のアクション RPG シューターゲームの最新作である **Borderlands* 3** (英語) を使用します。これをインテル® HD グラフィックス 630 内蔵グラフィックスと AMD Radeon* RX Vega M GL ディスクリート・グラフィックスの 2 つのグラフィックス・プロセッサーが搭載された **Windows* 10 搭載ミニ PC、インテル® NUC エンスージアスト - NUC8i7HVKVA** で実行します。このゲームをワイドスクリーン・モニターに適した 2560x1440 解像度で実行し、ゲームのすべてのグラフィックス・オプションをハイプロファイルに設定すると、ゲームエンジンはレンダリングに最適なグラフィックス・プロセッサーを選択します。このデバイスでは、Radeon* Vega が選択されました。

5 分間ゲームをプレイして最初に気になったことは、マウスやゲームパッドなどの入力デバイスの状態が変わってから、スクリーンのシーンが切り替わるまでの遅延です。一部の移動オブジェクトのアニメーションがやや不自然に見えます。インテル® GPA グラフィックス・トレース・アナライザーでトレースを収集して開くと、デバイス・コンテキスト・キューと GPU ハードウェア・キューのタイムライン・トラックから GPU 依存シナリオのすべての属性が一目で確認できます (**図 4**)。



4 **デバイス・コンテキスト・キューと GPU ハードウェア・キュー**

GPU キューにはギャップ (アイドル時間) がありません。完全にビジーで、ゲームから送信されたコマンドを継続的に実行しています。デバイス・コンテキスト・キューのサイズは十分に大きく、多くのグラフィックス処理が準備され、レンダリングを待機しています。単一フレーム内の GPU で実行されたすべてのコマンドパケットを選択すると、フレーム時間が分かります。フレーム時間は約 48.8 ミリ秒 (約 21FPS) であり、このアクションゲームには十分ではありません。このようなゲームで最適なゲーム体験を実現するには、通常 60FPS 以上が必要です。

高レベルのシステム概要を使用して GPU 依存シナリオを解析し、ソフトウェアとハードウェア・キューの詳細を調査することでいくつかの利点が得られます。この解析は簡単で正確です。グラフィックスの専門知識を必要とせず、レンダリングに使用されるグラフィックス API のタイプに依存しません。また、どのプラットフォームでも実行することができ、システムトレースから対応するパフォーマンス・イベントを収集できます。

関連情報

- **インテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA)**
- **インテル® グラフィックス・トレース・アナライザー (英語)**
- **インテル® グラフィックス・パフォーマンス・アナライザー・クックブック (英語)**

intel
Software

テクノロジーの未来を紐解く

開発者向けのナレッジハブ、Tech.Decoded へようこそ

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

エキスパートの考察を得る

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

さらに深く掘り下げる

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

概念のコード化

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

Tech.Decoded を見る (英語) >

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

Julia v1.3 の新しいスレッド化機能

最新 CPU の能力を最大限に引き出す

Jameson Nash、Jeff Bezanson Julia Computing, Inc.
Kiran Pamnany Caltech (カリフォルニア工科大学)

マルチコア・プロセッサーを活用することは、すべてのプログラミング言語において非常に重要です。また、プログラマーはアプリケーションのパフォーマンスを最大限に引き出すことに努力を費やしています。この記事では、Julia v1.3 の新しいマルチスレッド・ランタイムを利用して最小限の労力で最新 CPU を最大限に活用する方法を説明します。

重要なことは、プログラマーの負担を減らすことです。Julia には、構成を効率良く行うように設計された、さまざまな最新のプリミティブが用意されています。プログラマーの考えを単純化しようとすると、いくつかのトレードオフが発生します。Julia の重要な設計方針は、一般的なタスクを簡単にすること、そして難しいタスクを実行可能にすることです。この方針は、次のように、言語のさまざまな点で実証されています。

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。

- 自動メモリー管理
- 関数、オブジェクト、およびテンプレートを 1 つのディスパッチ・メカニズムに組み合わせ
- パフォーマンス向け型推論 (オプション)

ここでは、この方針を並列処理に拡張します。既存の並列メカニズムをベースに、既存のコードのシングルスレッド実行の (相対的な) 単純性を保ちつつ、新しいコードではマルチスレッド実行が可能な並列機能を追加します。この作業は、**スレッディング・ビルディング・ブロック** (英語) などの並列プログラミング・システムから着想を得たものです。

このパラダイムでは、プログラムの任意の部分を並列実行の対象としてマークできます。タスクはマークされたコードを利用可能なスレッドで自動的に実行します。動的スケジューラーは、いつ、どこでタスクを開始するか決定します。この並列処理モデルには多くの便利な特性があり、ガベージ・コレクションと似ています。ガベージ・コレクションでは、いつ、どのようにオブジェクトが解放されるか心配することなく、オブジェクトを自由に割り当てることができます。タスク並列処理では、いつ、どこでタスクが実行されるか心配することなく、(場合によっては数百万の) タスクを自由にスポーンすることができます。

モデルは移植性に優れており、低レベルの制御を行う必要はありません。プログラマーはスレッドを管理する必要はなく、利用可能なプロセッサー数やスレッド数を知る必要もありません。モデルは入れ子可能で、構成可能です。並列タスクを開始するライブラリー関数を呼び出す並列タスクを開始しても、問題なく動作します。この特性は、多くの作業がライブラリー関数により行われる高水準言語では非常に重要です。プログラマーは、ライブラリーがどのように実装されているか心配することなく、シリアルコードや並列コードを記述できます。このモデルは Julia ライブラリーに制限されません。FFTW のようなネイティブ・ライブラリーにも対応しています。現在は、OpenBLAS に対応する作業を行っています。

Julia でスレッドを実行する

Julia v1.3 を使用して複数のスレッドを起動する例を見てみましょう。最初に、**https://julialang.org/downloads** (英語) から Julia の最新リリース (現在は v1.3.0) をダウンロードします。**`JULIA_NUM_THREADS`** 環境変数に使用するスレッド数を設定して **`./julia`** を実行します。あるいは、Julia をインストールした後に、**http://docs.junolab.org/latest/man/installation/** (英語) の手順に従って Juno IDE をインストールします。利用可能なプロセッサー・コア数に基づいてスレッド数が自動的に設定されます。スレッド数を変更するグラフィカル・インターフェイスも提供されます。

スレッド数と現在のスレッドの ID を照会して、動作しているスレッドを確認できます。

```
julia> Threads.nthreads()
4

julia> Threads.threadid()
1
```

タスクとスレッド

スレッドが動作していることをビジュアルに実証する方法は、スケジューラーが、セミランダムで、インターリーブ順にワークをピックアップしているのを示すことです。Julia には、範囲を分割して静的なスケジュールにより各スレッドでアイテムを実行する '**`@threads for`**' マクロが以前のバージョンから用意されています。次の例では、スレッド 1 がアイテム 1 と 2 を実行し、スレッド 2 がアイテム 3 と 4 を実行します (以下同様)。

```
bash$> JULIA_NUM_THREADS=8 julia <<EOF
Threads.@threads for i = 1:12
    println(i, " on thread ", Threads.threadid())
end
EOF
1 on thread 1
3 on thread 2
12 on thread 8
9 on thread 5
7 on thread 4
2 on thread 1
4 on thread 2
5 on thread 3
8 on thread 4
11 on thread 7
10 on thread 6
6 on thread 3
```

インテル® TBB
効率良い並列プログラミングへのショートカット
無料
ダウンロード

新しいバージョンでは、同じプログラムを完全に動的なスケジュールで実行できるようになりました。新しい '**@spawn**' マクロと既存の '**@sync**' マクロを使用してワークアイテムを表現します。

```
bash$> JULIA_NUM_THREADS=8 julia <<EOF
@sync for i = 1:12
    Threads.@spawn println(i, " on thread ", Threads.threadid())
end
EOF
2 on thread 5
3 on thread 4
8 on thread 7
6 on thread 5
12 on thread 7
7 on thread 6
9 on thread 8
10 on thread 5
4 on thread 3
1 on thread 2
5 on thread 1
11 on thread 1
```

いくつかの例を見てみましょう。

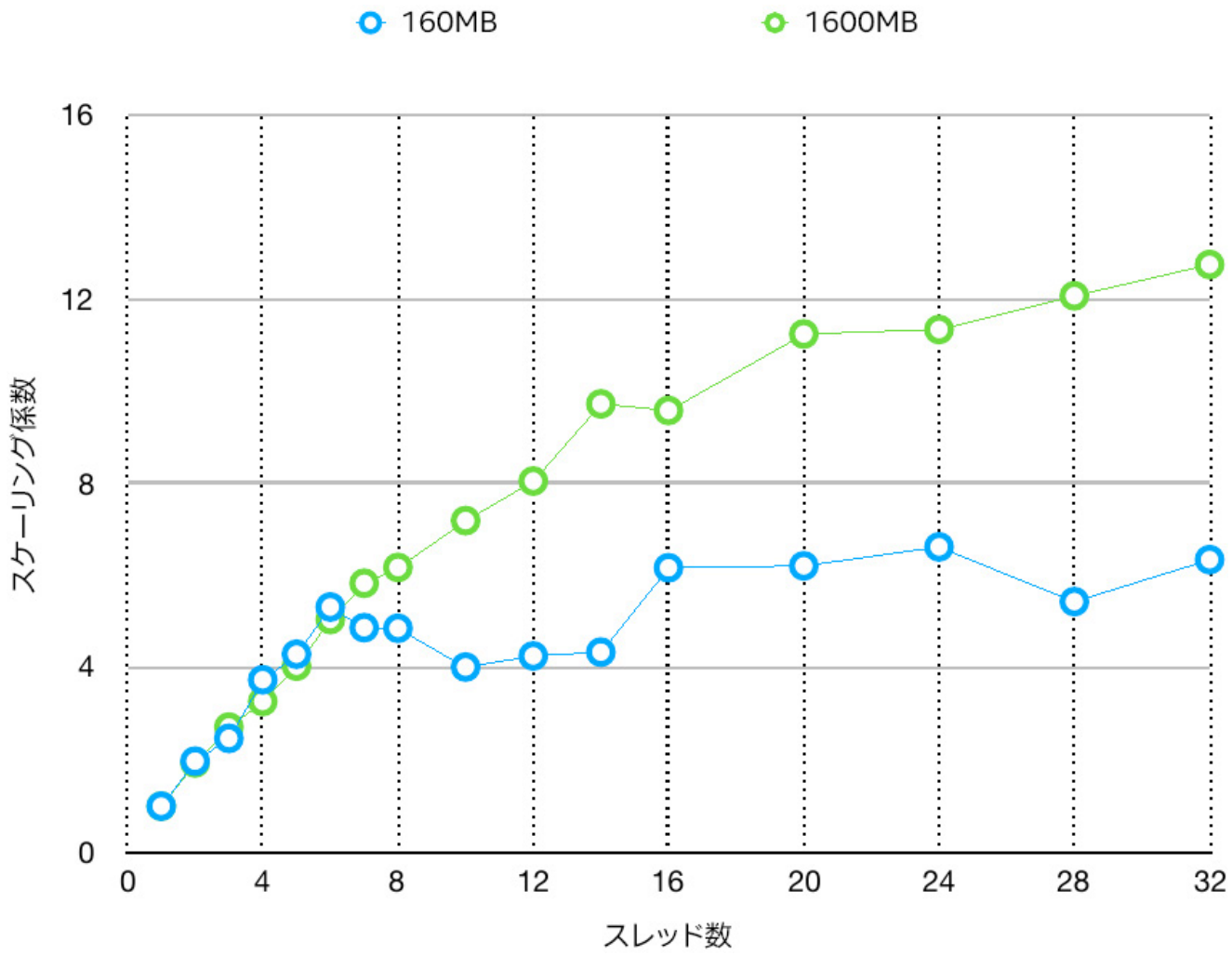
並列マージソート

典型的なマージ・ソート・アルゴリズムは、複数のスレッドを使用することで優れたパフォーマンスを示します。この関数は、配列の独立した部分を入力の最終的なコピーにマージする前にソートする、**O(n)** サブタスクを生成します。ここでは、**fetch** で値を返す各タスクの機能を使用します。

```
# perform a merge sort on `v` using parallel threads
function psort(v::AbstractVector)
    hi = length(v)
    if hi < 100_000 # below some cutoff, run in serial
        return sort(v, alg = MergeSort)
    end

    # split the range and sort the halves in parallel recursively
    mid = (1 + hi) >>> 1
    half = Threads.@spawn psort(view(v, 1:mid))
    right = psort(view(v, (mid + 1):hi))
    left = fetch(half)::typeof(right)

    # perform the merge on the result
    out = similar(v)
    pmerge!(out, left, right)
    return out
end

function merge!(out, left, right)
    ll, lr = length(left), length(right)
    @assert ll + lr == length(out)
    i, il, ir = 1, 1, 1
    @inbounds while il <= ll && ir <= lr
        l, r = left[il], right[ir]
        if isless(r, l)
            out[i] = r
            ir += 1
        else
            out[i] = l
            il += 1
        end
        i += 1
    end
    @inbounds while il <= ll
        out[i] = left[il]
        il += 1
        i += 1
    end
```

```
 @inbounds while ir <= lr
        out[i] = right[ir]
        ir += 1
        i += 1
    end
    return out
end

function pmerge!(out, left, right)
    ll, lr = length(left), length(right)
    @assert ll + lr == length(out)
    if length(out) < 100_000
        # below some threshold, just do the merge
        merge!(out, left, right)
    else
        # split the larger chunk in half, then binary search the smaller half to split it
        if ll > lr
            jl = ll ÷ 2
            # stable sort: find the last entry in right strictly smaller than l
            jr = searchsortedfirst(right, left[jl]) - 1
        else
            jr = lr ÷ 2
            # stable sort: find the last entry in left not bigger than r
            jl = searchsortedlast(left, right[jr])
        end
        @sync begin
            let left = view(left, 1:jl),
                right = view(right, 1:jr),
                out = view(out, 1:(jl + jr))
                Threads.@spawn pmerge!(out, left, right)
            end
            let left = view(left, (jl + 1):ll),
                right = view(right, (jr + 1):lr),
                out = view(out, (jl + jr + 1):length(out))
                pmerge!(out, left, right)
           end
        end
    end
    nothing
end
```

図 1 は、スレッド数の増加がスケーリングに与える影響を示しています。進行中のスレッドを使用したため、インプレースで入力を変化させてワークバッファーを再利用することにより、さらにパフォーマンスを最適化することができました。



1 **40 ハイパースレッドのサーバー (2 x インテル® Xeon® Silver 4114 プロセッサー @ 2.20GHz) の `psort` のスケーリング**

ここでは示されていませんが、**`fetch`** は子タスクの例外も自動的に継承します。

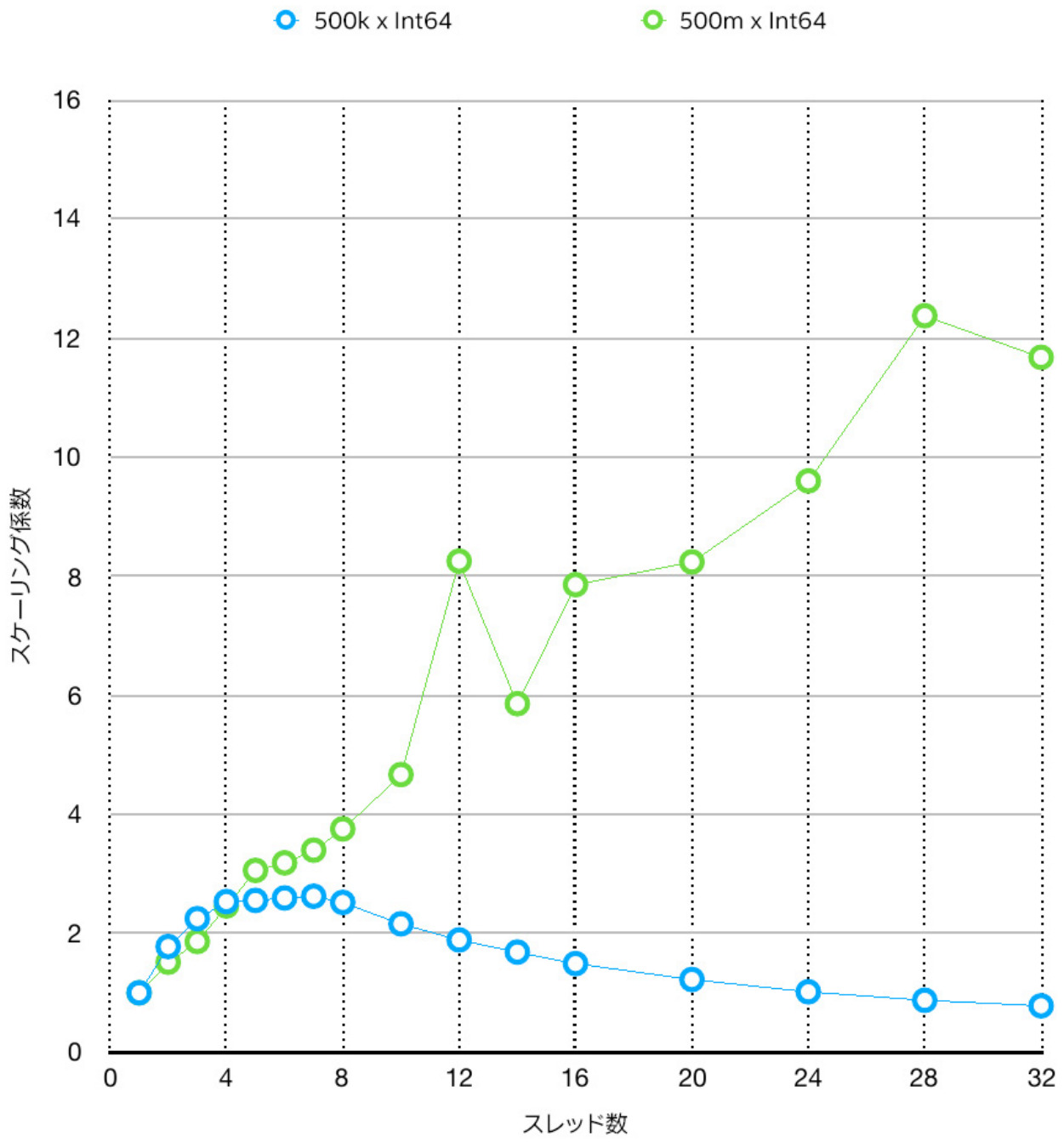
並列スキャン

`prefix sum` (「スキャン」とも呼ばれる) も、複数のスレッドを使用することで利点が得られる典型的な問題です。アルゴリズムの並列バージョンは、ワークを 2 つの木に配置して部分和を計算します。次の実装は、マシンで利用可能なすべてのコアと SIMD ユニットを活用します。

```
using .Threads: @threads
function prefix_threads!(⊕, y::AbstractVector)
    l = length(y)
    k = ceil(Int, log2(l))
    # do reduce phase
    for j = 1:k
        @threads for i = 2^j:2^j:min(l, 2^k)
            @inbounds y[i] = y[i - 2^(j - 1)] ⊕ y[i]
        end
    end
    # do expand phase
    for j = (k - 1):-1:1
        @threads for i = 3*2^(j - 1):2^j:min(l, 2^k)
            @inbounds y[i] = y[i - 2^(j - 1)] ⊕ y[i]
        end
    end
    return y
end

A = fill(1, 500_000)
prefix_threads!(+, A)
```

図 2 は、スレッド数の増加がスケーリングに与える影響を示しています。

Julia のいくつかの機能を組み合わせて、この単純かつパフォーマンスに優れた実装を簡単に表現できます。システムは、異なる種類の引数向けに最適化された関数のバージョンを自動的にコンパイルします。コンパイラーは、事前および JIT の両方で、特定の CPU モデル向けに関数を自動的に調整します。Julia には、さまざまな CPU 向けにプリコンパイルされたコードの「システムイメージ」が用意されていますが、実行時に使用されるプロセッサーが別の機能セットもサポートしている場合、コンパイラーはそのプロセッサー向けのコードを自動的に生成します。一方、スレッディング・システムは、ワークを動的にスケジュールして利用可能なコアに対応します。



2 **40 ハイパースレッドのサーバー (2 x インテル® Xeon® Silver 4114 プロセッサー @ 2.20GHz) の `prefix_threads` のスケーリング**

並列処理を考慮した API

アプリケーション・コードの操作を並列で安全に実行するには、スレッドを考慮した次のような API を使用する必要があります。

- **基本並列操作 :** `task` および関連する関数 (`schedule`、`yield`、`wait` を含む)
- **mutex:** `ReentrantLock` および `Condition` 変数 (`lock`、`unlock`、`wait` を含む)
- **同期プリミティブ :** `Channel`、`Event`、`AsyncEvent`、`Semaphore`
- **I/O およびその他のブロッキング操作 :** `read`、`write`、`open`、`close`、`sleep` を含む
- **試験的スレッドモジュール :** さまざまなビルディング・ブロックおよびアトミック操作

スケジューラーのデザイン

`partr` スケジューラーのプロトタイプ実装は、2016 年の終わりに Kiran Pamnany により C で最初に記述され[1]、キャッシュの効率的なスケジューリングの研究[2] に従っています。この作業の目標は、グローバルな深さ優先の作業順を備えたスレッド化ライブラリーを構成することでした。`partr` では、タスクを起動するスレッドのスレッド ID で優先度を設定する近似優先度キューを使用して、この機能を実装しています。

外部ライブラリー

この作業の目的は、キャッシュ・スラッシングや頻繁なコンテキスト・スイッチによりパフォーマンスを低下させる、CPU オーバーサブスクリプションが発生しないようにマルチスレッド・ライブラリーをサポートすることでした。以前は、Julia を N スレッドに制限することをユーザーが事前に決定し、(`libfftw` や `libblas` などの) スレッド化ライブラリーに M ÷ N コアを使用するように伝えることが唯一の方法でした。最も一般的な選択は 1 と M で、複数のコアを活用できるのはアプリケーションの一部のみです。しかし、スレッドプールのワークアイテムを素早く作成して実行できるようになったことから、外部ライブラリーとスレッドプールを統合する方法を調査しています。これは調査中の機能であり、さまざまなライブラリーのパフォーマンスと API のニーズについてのフィードバックを受け取っています。

FFTW をオリジナルのランタイム (pthreads ベースのワークプール) の代わりに新しいスレッドランタイム上で動作するように対応させる作業はわずか数時間で完了しました (ライブラリー開発者の協力のおかげです)。パフォーマンス・チューニングを (まだ) 行っていない状態でも、優れたパフォーマンス結果が得られました。スケジューラーのレイテンシーを最適化するための重要なポイントを学び、現在は正確なパフォーマンス・パリティーを達成する作業を行っています。しかし、多少のオーバーヘッドが発生する場合でも、スレッドを考慮してユーザーを構成し、`partr` スケジューラーからリソース・スケジューリングを達成できることは、プログラム操作の全面的な改良に役立つと期待しています。

今後の動向

Julia のマルチスレッドに対するアプローチは、新しいフレームワークに多くの既知のアイデアを組み合わせるものです。アイデアは単体でも優れたものであり、それらを組み合わせることでさらに優れたものになると考えています (多くの場合はそうなります)。この先、Julia エコシステムでさまざまな構成可能な並列ライブラリーが発展することを楽しみにしています。

参考資料 (英語)

1. **https://github.com/kpamnany/partr**
2. Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling Threads for Constructive Cache Sharing on cmps. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA ’ 07, pages 105–115, New York, NY, USA, 2007. ACM.

VIDEO HIGHLIGHTS

データ並列 C++ でハードウェアの境界をなくす

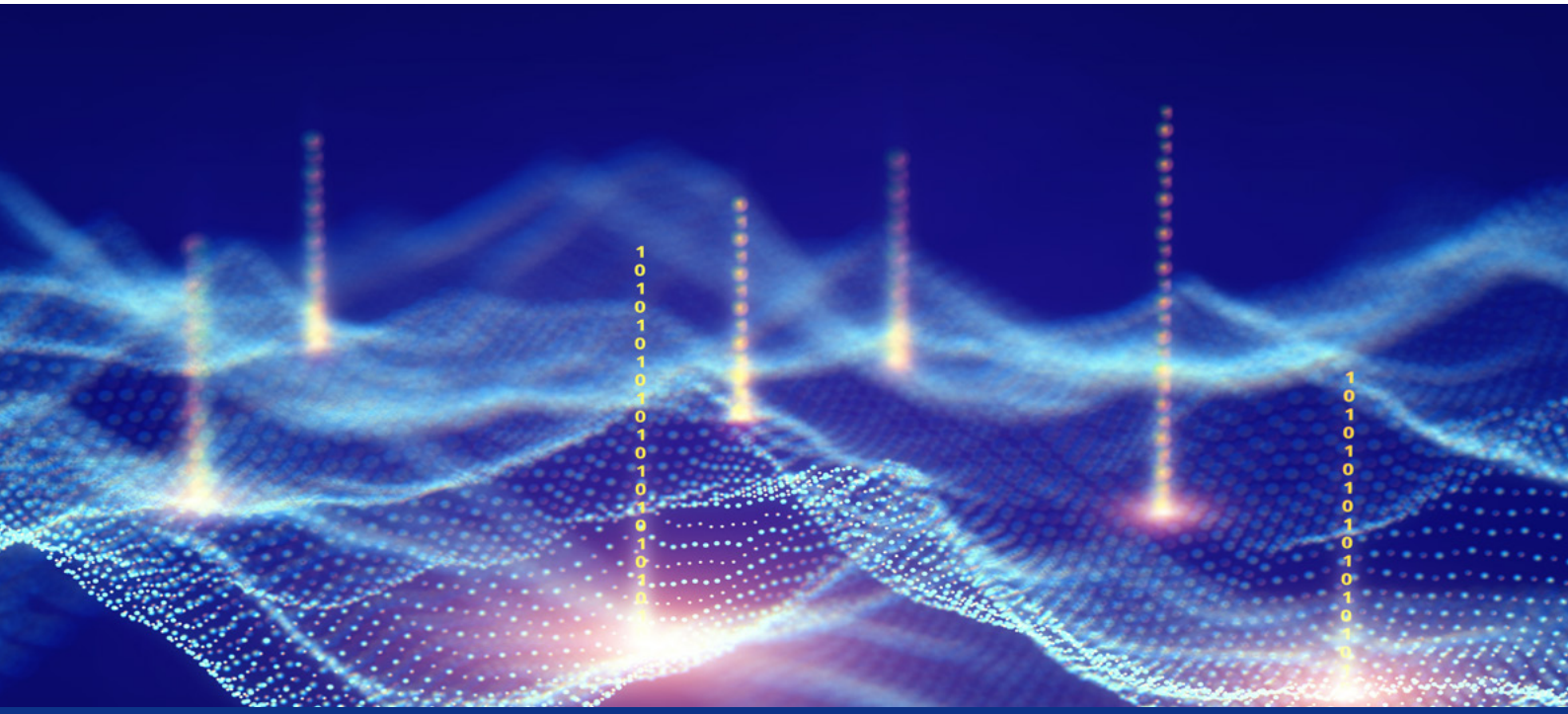
データ並列 C++ (DPC++) を利用すると、開発者はさまざまなハードウェア・ターゲット (CPU やアクセラレーター) 間でコードを再利用し、特定のアクセラレーター向けのカスタム・チューニングを行うことができます。DPC++ は、C++ と SYCL* ベースの、単一アーキテクチャー専用アプローチの代わりとなるオープンなプログラミング言語で、開発者が専用のワークロード要件を満たすソリューションを作成できるように支援します。インテルのバイスプレジデント Alice Chan による、プログラミングの柔軟性における変化の解説をご覧ください。

こちら (英語) からビデオをご覧いただけます。 >

oneAPI で複数のアーキテクチャーにわたる多様なワークロードに妥協のないパフォーマンスを提供

詳細 (英語) >

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



インテル® Xeon® プロセッサー向け 勾配ブースティング木推論の高速化

インテル® DAAL の GBT を使用して予測の質とパフォーマンスを向上する方法

Kirill Shvets インテル コーポレーション マシンラーニング・エンジニア
Egor Smirnov インテル コーポレーション ソフトウェア・エンジニアリング・マネージャー

勾配ブースティング木 (GBT)[1] は、分類と回帰タスク向けの正確で効率的なマシンラーニング (ML) アルゴリズムです。多くの GBT 実装がありますが、最も一般的なのは XGBoost[2] ライブラリーでしょう。「インテル® Xeon® プロセッサー向け XGBoost の高速化」(The Parallel Universe **38 号**)[3] では、XGBoost の CPU ベースのトレーニングの大幅な高速化を紹介しました。また、**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)**[4] のパフォーマンスはさらに高いことを示しました。エンドユーザーはモデルの予測ができるだけ速く必要なため、ML 推論はトレーニングと同じように重要です。この記事では、この点に焦点を当てて説明します。

パフォーマンスの比較

この記事では、次の GBT 実装の推論パフォーマンスを比較します。

- **フォレスト推論ライブラリー (FIL)**[5]、ブースティング決定木モデル向けの GPU アクセラレーション推論を提供する RAPIDS cuML[6] のオープンソース・ライブラリー
- **XGBoost** CPU/GPU 推論 (master ブランチ)
- **インテル® DAAL** (バージョン 2019 Update 5)、インテル® プラットフォーム向けに高度に最適化されたオープンソース・ライブラリー

パフォーマンス解析には、Bosch、Epsilon、および Mortgage (Q1' 2000 サブセット) データセットを使用しました。各データセットをシャッフルし、予測段階で次のサブセットをランダムに選択しました。

- **Bosch**[7] (968 特徴、10 万観測点、~110 万予測)
- **Epsilon**[8] (2,000 特徴、40 万観測点、10 万予測)
- **Mortgage**[9] Q1' 2000 (45 特徴、~900 万観測点、100 万予測)

選択したデータセットで最高の精度を得るのに十分なモデルサイズを選択しました (付録)。パフォーマンス測定には、AWS* EC2* インスタンスを使用しました。

- **CPU:** c5.metal (第 2 世代**インテル® Xeon® スケーラブル・プロセッサー**、2 ソケット、ソケットあたり 24 コア)
- **GPU:** p3.2xlarge (NVIDIA* Tesla* V100)

詳細は、「構成」セクションを参照してください。

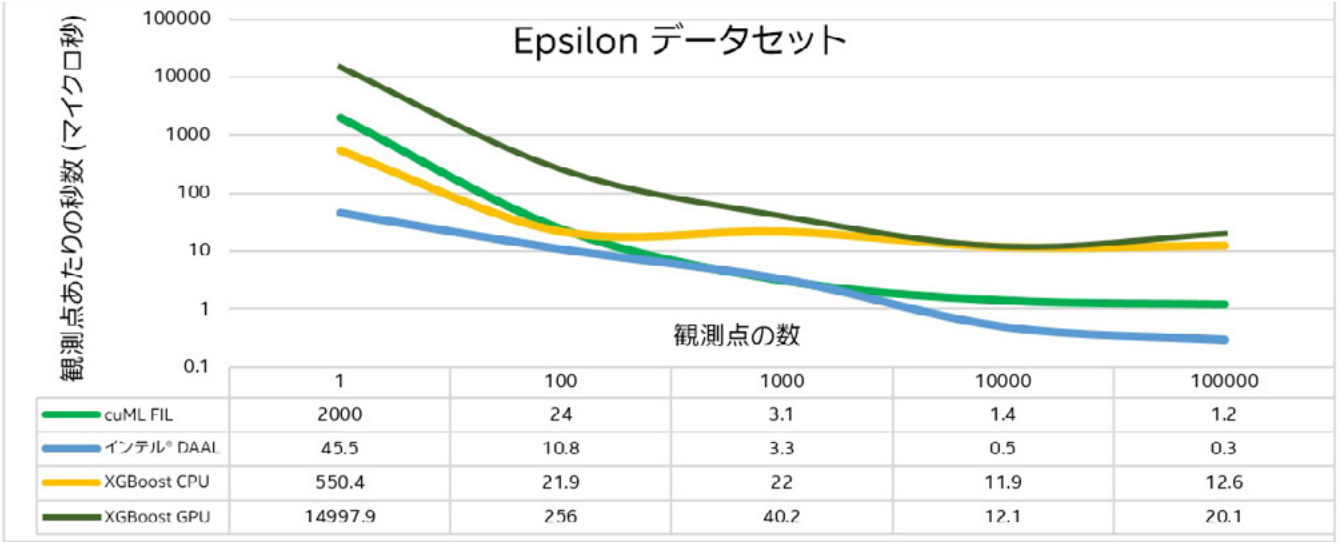
ML では、一度に 1 つのデータ観測点に推論モデルを適用する (バッチサイズを 1 に設定する) ことが一般的です。GBT 推論もこの方法で適用できますが、包括的にするため、ここではバッチサイズを変更します。

図 1、**図 2**、および**図 3** は次のことを示しています。

- **XGBoost の CPU 実装** (黄色の線) の推論パフォーマンスは GPU 実装 (深緑の線) とほぼ同じで、小さなバッチではパフォーマンスが高い。
- **FIL (緑の線) は多くのケース (中および大きなバッチサイズ) で XGBoost CPU/GPU よりパフォーマンスが高く**、インテル® DAAL はさらにパフォーマンスが高い。
- **インテル® DAAL 実装 (青の線) は 1 行のケースで FIL (緑の線) より大幅にパフォーマンスが高い。**
- **インテル® DAAL 実装 (青の線) は中および大きなバッチサイズでパフォーマンスが高い。**

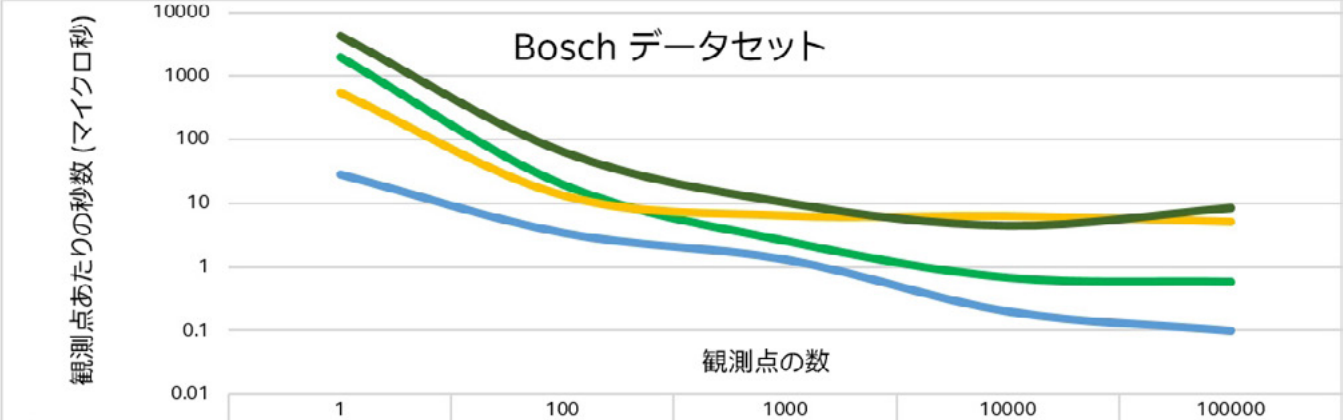
XGBoost GPU および CPU のパフォーマンス測定では、入力データはネイティブ形式 (XGBoost DMatrix) で提供されました。インテル® DAAL の GBT の予測では、連続した NumPy* 配列を使用しました。

内部的な実装の理由により、cuDF データ形式は FIL ではパフォーマンス指向ではありません。そのため、外部ロード (CuPy ライブラリーなど) よりデバイスのデータをロードする効率的で高速な方法である、連続した NumPy* 配列を入力として提供する必要があります。しかし、GPU の実際のエンドツーエンド ML ワークロードでは、cuDF で ETL を行い、データ変換なし (追加のオーバーヘッドなし) で cuDF DataFrame を入力にするほうが一般的です。**図 4** は、cuDF 形式の FIL 推論は、特にインテル® DAAL と比較して、NumPy* データ形式の FIL より非常に遅いことを示しています。エンドツーエンドのマシンラーニング・アプリケーションでは、想定されたパフォーマンスが得られます。



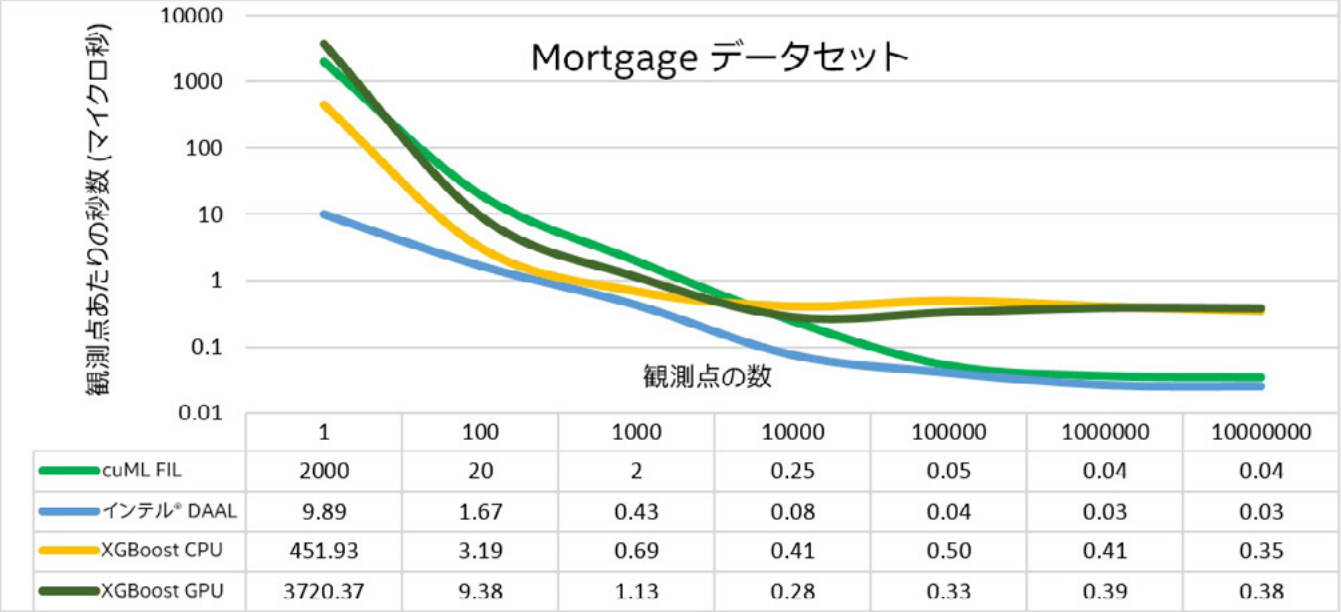
	1	100	1000	10000	100000
cuML FIL	2000	24	3.1	1.4	1.2
インテル® DAAL	45.5	10.8	3.3	0.5	0.3
XGBoost CPU	550.4	21.9	22	11.9	12.6
XGBoost GPU	14997.9	256	40.2	12.1	20.1

1 **Epsilon データセットの GBT 推論パフォーマンス (μ 秒値が低いほど高性能)**



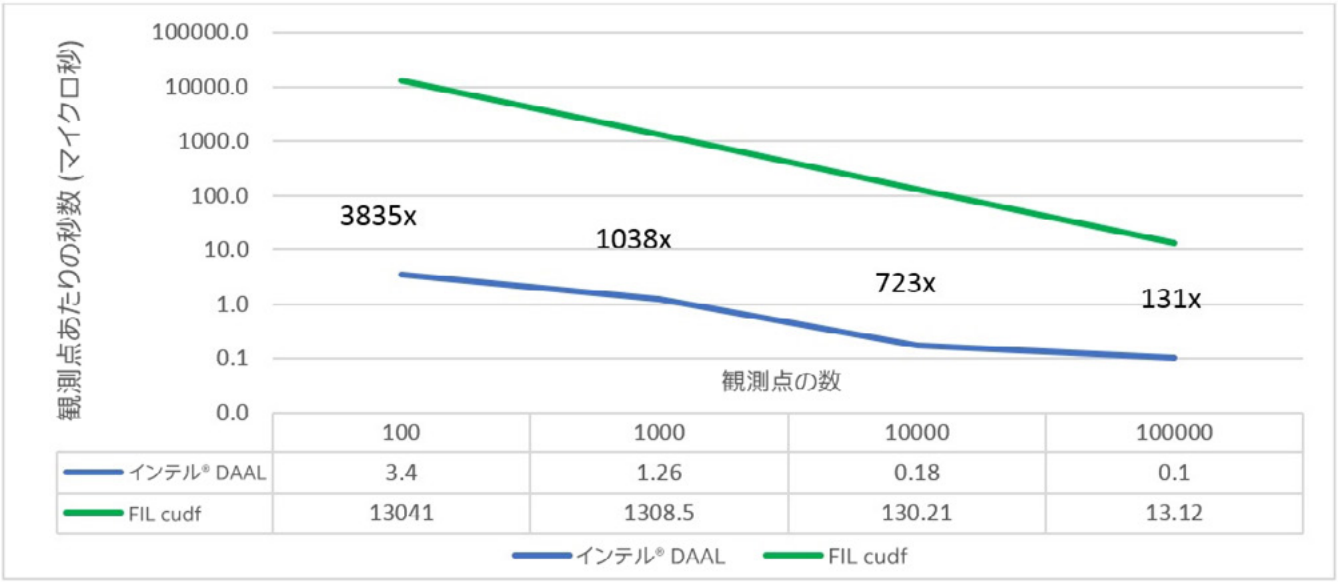
	1	100	1000	10000	100000
cuML FIL	2000	20	2.6	0.69	0.59
インテル® DAAL	28	3.4	1.3	0.2	0.1
XGBoost CPU	550.1	13.3	6.4	6.31	5.1
XGBoost GPU	4200.6	64.4	10.3	4.5	8.5

2 **Bosch データセットの GBT 推論パフォーマンス (μ 秒値が低いほど高性能)**



	1	100	1000	10000	100000	1000000	10000000
cuML FIL	2000	20	2	0.25	0.05	0.04	0.04
インテル® DAAL	9.89	1.67	0.43	0.08	0.04	0.03	0.03
XGBoost CPU	451.93	3.19	0.69	0.41	0.50	0.41	0.35
XGBoost GPU	3720.37	9.38	1.13	0.28	0.33	0.39	0.38

3 **Mortgage データセットの GBT 推論パフォーマンス (μ 秒値が低いほど高性能)**



	100	1000	10000	100000
インテル® DAAL	3.4	1.26	0.18	0.1
FIL cudf	13041	1308.5	130.21	13.12

4 **FIL cuDF およびインテル® DAAL の GBT 推論のスループット、Bosch (μ 秒値が低いほど高性能)**

パフォーマンス測定の結果は次のようにまとめられます。

- **インテル® DAAL 推論**は最も重要なユースケースの 1 つであるバッチサイズが 1 の場合にほかの実装より非常に速い。
- **インテル® DAAL 推論**は大きなバッチサイズやほかのサイズのモデルでも速い。
- **XGBoost CPU と GPU 推論**はすべてのデータセットでほぼ同じである。
- **cuDF 形式** (エンドツーエンド・アプリケーションで一般的に使用される) の FIL 推論はインテル® DAAL 推論より非常に遅い。

コストの比較

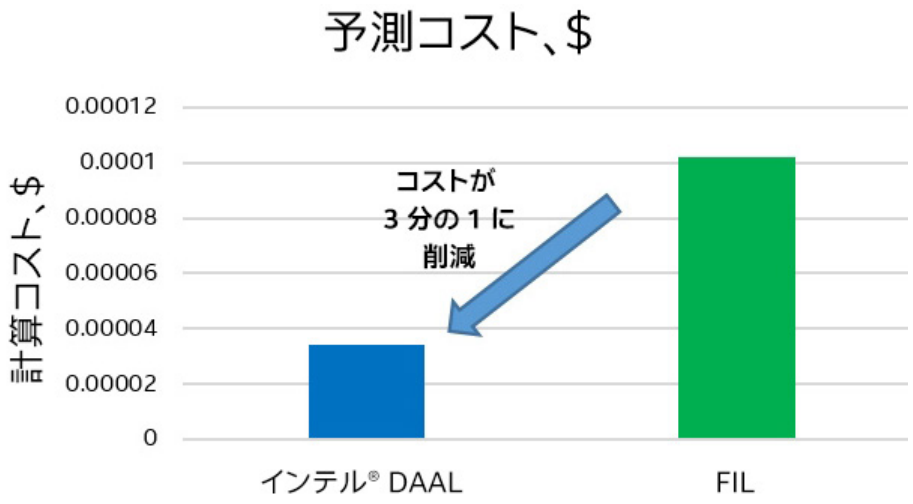
インテル® DAAL と cuML FIL の予測コストを比較するため、最も効率的な CPU および GPU 実装を選択しました。テストでは、次の AWS* EC2* インスタンスを使用しました (2019 年 12 月時点のコスト[10])。

- **CPU:** c5.metal、1 時間あたり $4.08
- **GPU:** p3.2xlarge、1 時間あたり $3.06

表 1 は、Epsilon データセット (10 万観測点) の予測コスト計算の例を示しています。
図 5 は、インテル® DAAL と FIL の予測コストの比較を示しています。

表 1. Epsilon データセットの予測コスト

ライブラリー	時間、ミリ秒	インスタンスのコスト、$/時間	推論コスト (時間 x インスタンスのコスト)、$
cuML FIL	120	3.06	$10.2 * 10^{-5}$
インテル® DAAL	30	4.08	$3.4 * 10^{-5}$



5 **インテル® DAAL と FIL の予測コストの比較**

ほかのデータセットでのコスト計算は次のようになりました。

- **Bosch データセット :** インテル® DAAL の予測コストは cuML の 4.4 倍向上
- **Mortgage データセット :** インテル® DAAL は cuML と同等

GPU 実装と比較して、CPU でのインテル® DAAL の GBT 推論は高速かつ低コストです。

機能の比較

XGBoost、インテル® DAAL、および FIL で利用可能な機能を比較して**表 2** にまとめました。

表 2. XGBoost、インテル® DAAL、FIL の機能

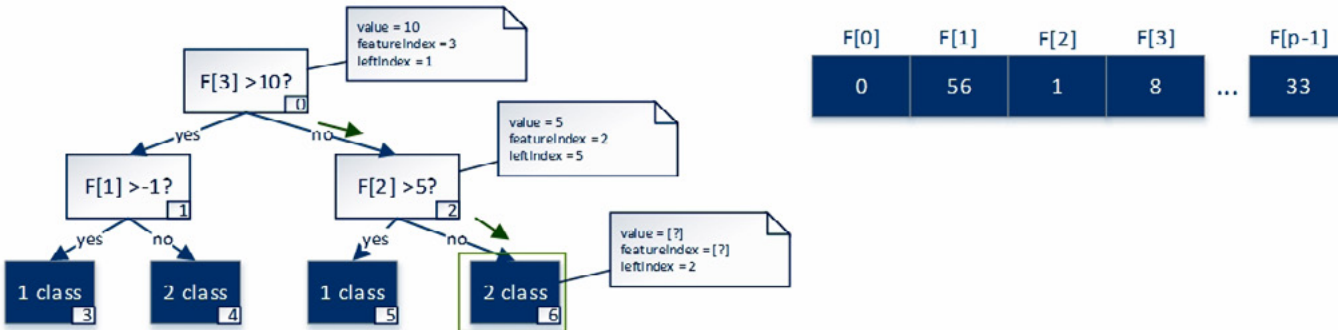
サポート	XGBoost	インテル® DAAL	FIL
2 項分類	○	○	○
多クラス分類	○	○	×
回帰	○	○	○
Float64 入力形式	○	○	×
Float32 入力形式	○	○	○

FIL の制限により、すべてのパフォーマンス比較は単精度浮動小数点形式で提供され、2 項分類と回帰データセットを使用しました。

インテル® DAAL が高速である理由

最新の**インテル® Xeon® プロセッサー**を最大限に活用するため、インテル® DAAL は GBT 推論に**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)** ベクトル命令セットを使用しています (**図 6**)。GBT 推論の主な操作は次のとおりです。

- 各分割ノードの条件を評価して**データを比較**します。
- 予測される観測点の木ノードと 1 つの特徴に**ランダム・メモリー・アクセス**します (読み取り)。



6 **GBT 推論**

1 つの木で 1 つの観測点を処理するように、葉ノードに達するまでこの手法を使用します。木の現在のノードで、観測点 '**`F[node.featureIndex]`**' の特定の特徴値とノードの分割値 '**`node.value`**' を比較します。次のノードのインデックスは以下のように定義します。

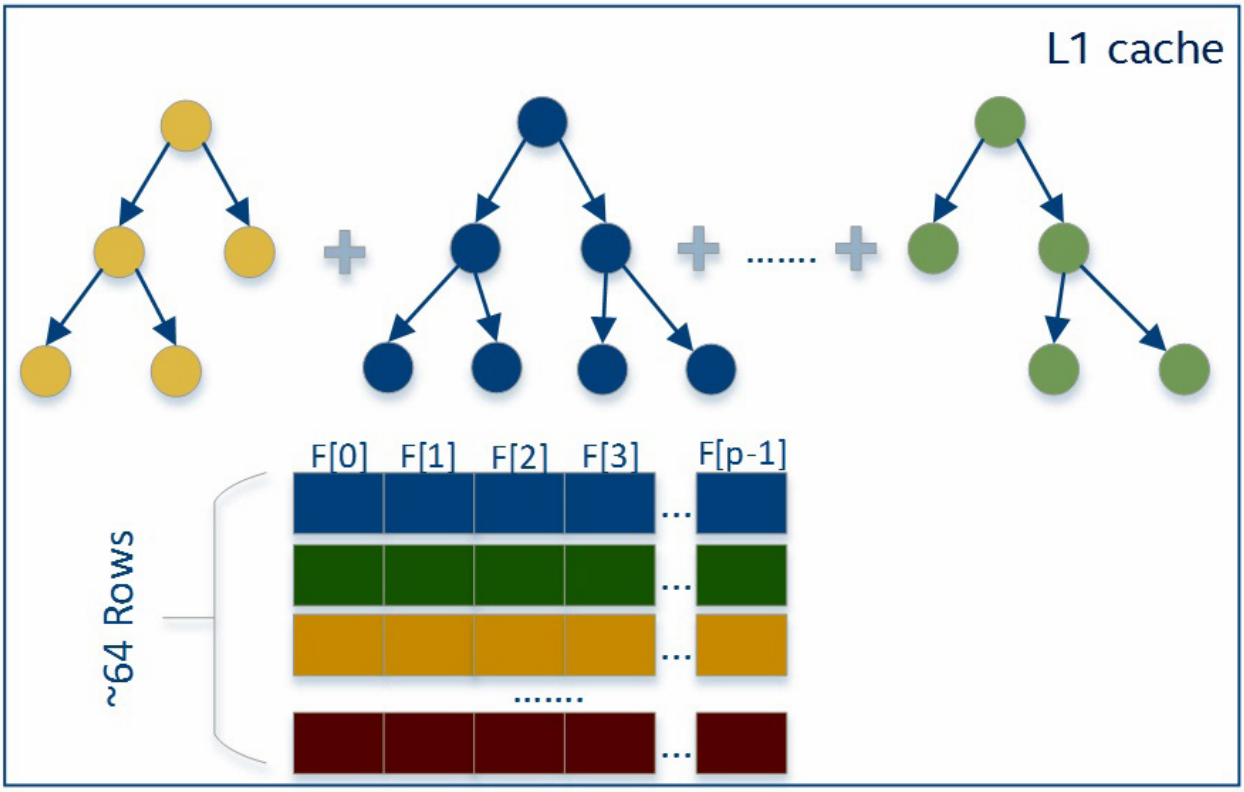
$$node.leftIndex + intercept, \text{where } intercept = \begin{cases} 0, if\ F[node.featureIndex] > node.value \\ 1, if\ F[node.featureIndex] \leq node.value \end{cases}$$

メインの推論操作、比較、およびランダム・メモリー・アクセスは、インテル® AVX-512 ベクトル命令 **`vpgatherd{d,q}`** と **`vcmpp{s,d}`** を使用して効率良く実装できます。この命令を適用するため、各木で同時に観測点 (行のブロック) について木を検索します。

- 行ブロックの各観測点について情報 (特定の木ノードの分割値 '**`value`**' および特徴インデックス '**`featureIndex`**') を**収集**します。
- 観測点の '**`value`**' および **`F[featureIndex]`** に**ベクトル比較を適用**します。
- 比較に基づいて各観測点について次のノードの**インデックスを計算**します。

木の葉ノードに達するまで、各観測点についてこの操作を行います。

アルゴリズムのパフォーマンスはメモリーアクセスの効率とメモリー帯域幅に依存します。インテル® DAAL のスマート・データ・ブロッキングを木構造に使用して、テンポラルなキャッシュ局所性、木のサブセット、L1 データキャッシュのサイズに収まる観測点のブロックを改善します。その後、木のサブセットで必要な観測点の予測を実装します。これにより、ほとんどのデータアクセスが、最も帯域幅が広い L1 データキャッシュで行われるようになります (**図 7**)。



7 メモリー帯域幅

パフォーマンス比較の結果

パフォーマンス比較の結果をまとめると、CPU でインテル® DAAL の GBT を使用した場合、ほかの実装より優れたパフォーマンスで同じ予測品質を達成できることが分かります。インテル® DAAL の推論はオンライン推論 (バッチサイズが 1 の場合) で非常に高速です。ユーザーにより提供されるデータ処理と出力の予測に必要な時間を最小限に抑えることは重要であり、大きなデータのバッチを集計する可能性はないため、これはアプライド ML における重要なユースケースであると言えます。

データが大きなバッチで集計される一般的なケースでも、CPU のインテル® DAAL は FIL GPU と XGBoost よりパフォーマンスが高くなります。利用可能な最大バッチサイズでは、インテル® DAAL の FIL に対する相対スピードアップは、Epsilon で 4 倍、Bosch で 5.9 倍、Mortgage で 1.4 倍になります。

優れたパフォーマンスに加えて、インテル® DAAL にはユーザーフレンドリーな Python* インターフェイスと、単純で短い推論向けのコードが用意されています。

同じペイロードでは、インテル® DAAL を使用した CPU ベースの推論の計算コストは、FIL の計算コストより低くなります (平均で 2.85 倍[10])。

コード例

XGBoost:

```
import xgboost as xgb

# x_train, labels, x_test = ...

# Train model with XGBoost
model_xgb = xgb.train(..., xgb.DMatrix(x_train))

# Generate predictions (as a NumPy array)
predicts = model_xgb.predict(xgb.DMatrix(x_test))
```

インテル® DAAL:

```
import daal4py as d4p

# x_train, labels, x_test, n_classes = ...

# Train model with Intel® DAAL
train_algo = d4p.gbt_classification_training(n_classes, ...)
train_result = train_algo.compute(x_train, labels)

# Generate predictions (as a NumPy array)
pred_alg = d4p.gbt_classification_prediction(n_classes)
predicts = pred_alg.compute(x_test, train_result.model)
```

FIL:

```
from cuml import ForestInference

# Load the classifier previously saved with xgboost model_save()
model_path = "xgb.model"
fm = ForestInference.load(model_path, output_class=True)

# Generate predictions (as a gpu array)
fil_preds_gpu = fm.predict(X_test.astype('float32'))
```

参考資料

1. Tianqi Chen and Carlos Guestrin. **XGBoost: A Scalable Tree Boosting System** (英語). In 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016.
2. **XGBoost ライブラリー (英語)**
3. **The Parallel Universe 38 号**
4. **インテル® DAAL**
5. **Forest Inference ライブラリー (英語)**
6. **cuML (英語)**
7. **Bosch データセット (英語)**
8. **Epsilon データセット (英語)**
9. **Mortgage データセット (英語)**
10. **Amazon EC2* 料金表**

構成

アルゴリズムのパラメーター

木の数と最大の深さは各データセットで異なりますが、ほかのパラメーターはすべて同じです (**表 3**)。パフォーマンス・データは 2019 年 11 月に収集されました。

表 3. XGBoost およびインテル® DAAL のテストのパラメーター

XGBoost	インテル® DAAL
Fptype: float32	Fptype: float32
'max_bin': 256	'maxBins': 256
'learning_rate': 0.1	'shrinkage': 0.1
'reg_lambda': 1	'lambda_': 1

FIL 推論には、事前トレーニング済みの XGBoost モデルを使用しました。

FIL は現在倍精度形式をサポートしていないため、パフォーマンスは単精度浮動小数点形式 (float32) で測定しました。

ハードウェア構成

- **CPU 構成 :** c5.metal AWS* EC2* インスタンス : 第 2 世代インテル® Xeon® スケーラブル・プロセッサー、2 ソケット、ソケットあたり 24 コア、インテル® ハイパースレッディング・テクノロジー有効、インテル® ターボ・ブースト・テクノロジー有効、OS: Ubuntu* 18.04.2 LTS、合計メモリー 193GB (12 スロット /16GB/2933MHz)。
- **GPU 構成 :** p3.2xlarge AWS* EC2* インスタンス : インテル® Xeon® E5-2686 v4 プロセッサー @ 2.30GHz、1 ソケット、4 コア、インテル® ハイパースレッディング・テクノロジー有効、インテル® ターボ・ブースト・テクノロジー有効、GPU: Tesla* V100-SXM2-16G (ドライバーバージョン : 410.104、CUDA* バージョン : 10.0)、OS: Ubuntu* 18.04.2 LTS、合計メモリー 61GB (4/13312MB)。

ソフトウェア構成

- RAPIDS FIL (Forest Inference ライブラリー) – バージョン 0.9
- インテル® DAAL - バージョン 2019 Update 5
- XGBoost – master (ef9af33a000f09dbc5c6b09aee133e38a6d2e1ff)

ほかのソフトウェア : Python* 3.6、Numpy* 1.16.4、pandas* 0.25、scikit-lean 0.21.2。

付録

テストしたサブセットの精度は、一般に 2 つの主要なモデル・パラメーター (木推定器の数と深さ) に関連するため、分類データセットで適切な精度を得るのに十分なモデルサイズを選択しました (**表 4** および**表 5**)。

表 4. Epsilon (10 万行)

木の数/最大の深さ	100-4	100-8	100-12	...	900-12	1000-4	1000-6	1000-8	1000-12
分類の精度	0.81504	0.83103	0.8258	...	0.8751	0.87624	0.87431	0.8726	0.86559

表 5. Bosch (10 万行)

木の数/最大の深さ	100-4	100-8	100-12	200-4	200-8	...	1000-4	1000-8	1000-12
分類の精度	0.99418	0.99467	0.99469	0.99421	0.99451	...	0.99441	0.99429	0.99387

選択したモデル (Epsilon “1000-4”) の推論精度は、インテル® DAAL と FIL および XGBoost CPU/GPU でほぼ同じ (0.874400) でした。もう一方のモデル (Bosch “100-12”) でも、精度はインテル® DAAL と FIL および XGBoost CPU/GPU でほぼ同じ (0.99444) でした。

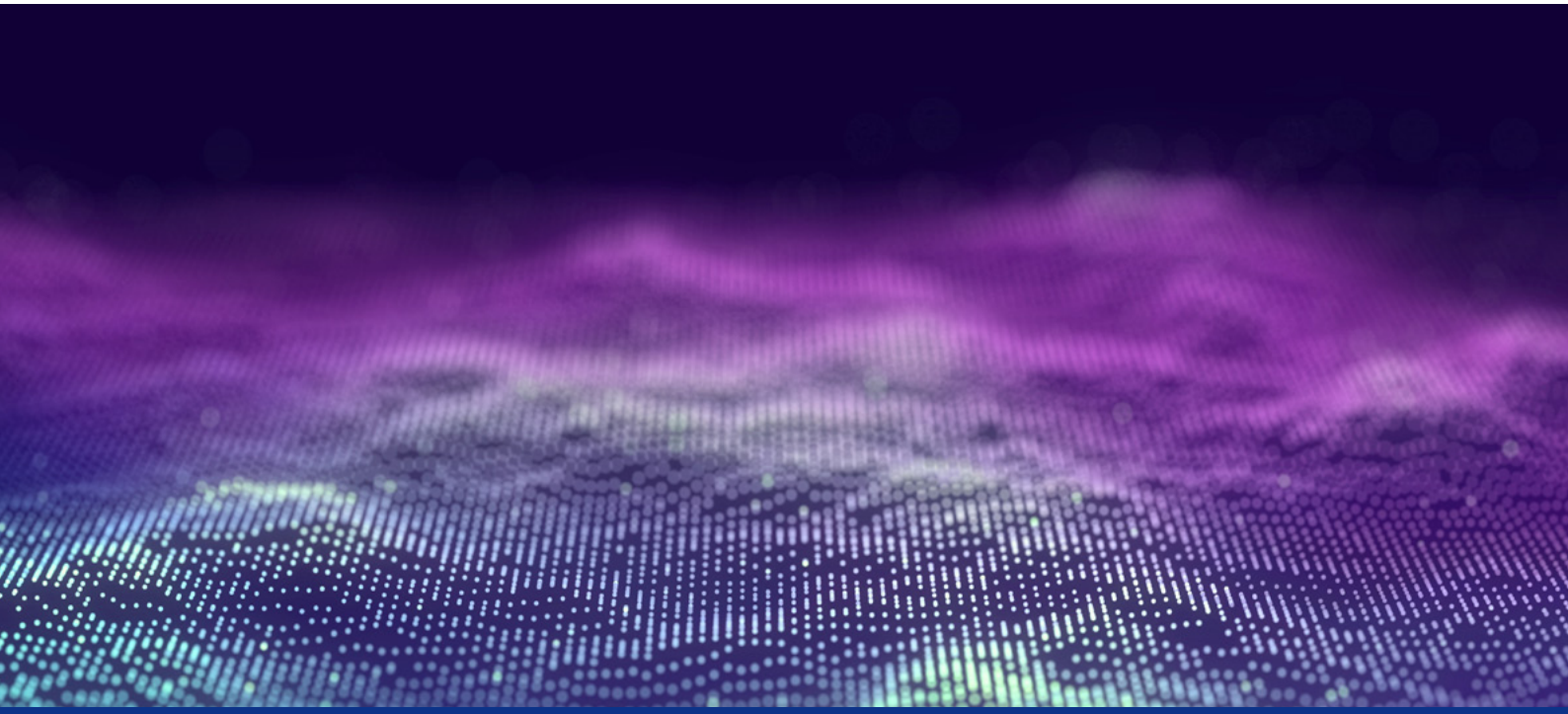
Mortgage データセットの回帰推論には、NVIDIA が RAPIDS **ベンチマーク** (英語) で使用しているパラメーター (トレーニング・モデルの木推定器の数は 100、最大の深さは 8) を使用しました。

VIDEO HIGHLIGHTS

インテル® DevCloud とは

インテル® DevCloud は、CPU、GPU、および FPGA で構成されるクラスターで、複数のインテル® oneAPI ツールキットがプリインストールされています。このビデオは、インテル® DevCloud バッチシステムでサンプル oneAPI アプリケーションをセットアップおよび実行する方法について詳しく説明したガイドです。

こちら (英語) からビデオをご覧いただけます。>



第 2 世代インテル® Xeon® スケーラブル・プロセッサーで K 平均法を高速化

インテルのビッグデータ向けハードウェアとソフトウェア・スタック

Alexander Andreev インテル コーポレーション マシンラーニング・エンジニア
Egor Smirnov インテル コーポレーション ソフトウェア・エンジニアリング・マネージャー

私たちが日々生成するデータの量は急激に増加しており、その膨大なデータを処理して活用できる、ハイパフォーマンスでスケーラブルなアルゴリズムが求められています。インテルは、ハードウェアとソフトウェアの両方を最適化して、インテル® プラットフォーム向けの効率良いデータ分析ツールを作成することに労力を注いでいます。この数年で、インテルは、最適化されたフレームワーク[1] とライブラリー[2] を含む、強力なデータ分析およびマシンラーニング (ML) ソフトウェアとハードウェア・スタックを構築しました。

ハードウェアのハイパフォーマンスとスケーラビリティーは、**インテル® Xeon® スケーラブル・プロセッサー**[3] (第 2 世代は**インテル® ディープラーニング・ブースト [インテル® DL ブースト] テクノロジー** (英語)、**インテル® Nervana™ ニューラル・ネットワーク・プロセッサー [インテル® Nervana™ NNP]** (英語)、**インテル® FPGA**、および**インテル® Movidius™ ニューラル・コンピュート・スティック**[4] (英語) を含む) により提供されます。

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。

インテルにより最適化された scikit-learn とインテル® DAAL

インテルにより最適化された有名な ML アルゴリズムのフレームワーク、scikit-learn[5] は、**Python* 向けインテル® ディストリビューション** (IDP)[6] の一部です。IDP scikit-learn 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)**[7] を使用してインテル® アーキテクチャー上で高いパフォーマンスを実現します。インテル® DAAL は、モバイル (**Intel Atom® プロセッサー**) からデータセンター (**インテル® Xeon® プロセッサー**) まで、広範なインテル® アーキテクチャー向けに最適化されたオープンソース[8] のデータ分析ライブラリーです。インテル® DAAL は、C++、Java*、および Python* API を提供します。異なるアーキテクチャー (CPU、GPU、その他) での効率的な開発のための統合プログラミング・モデル、**ベータ版インテル® oneAPI ツールキット**[9] (英語) では、**データ並列 C++ (DPC++)** (英語) API も提供されます。

K 平均法は IDP scikit-learn で高速化された ML アルゴリズムの 1 つです。

K 平均法アルゴリズム

ポピュラーなクラスタリング・アルゴリズムである K 平均法は、天文学、医学、市場細分化などの多くの分野で使用されています。インテル® DAAL に実装されている高速化された ML アルゴリズムの 1 つであり、単純かつ強力です。K 平均法の目的は、一連の N の観測点を K のクラスターに分割することです。具体的には、クラスター内の 2 乗誤差 (観測点からそのクラスターの中心 [セントロイド] までのユークリッド距離の 2 乗の総和) を最小化します。反復アルゴリズムで、各反復には次の 2 つのステップが含まれます。

1. **各観測点で**、各セントロイドまでの距離を計算した後、各観測点を最も近いセントロイドを含むクラスターに再割り当てします。
2. **各クラスターで**、セントロイドをそのクラスターに割り当てられている観測点の平均として計算します。

次のいずれかになるまで、2 つのステップを繰り返します。

1. **反復数**が事前に定義された最大値を超える。
2. **アルゴリズムが収束**する (クラスター内の 2 乗誤差が事前に定義されたしきい値未満になる)。

最初の反復の初期セントロイドは、異なる初期化手法を使用して求めます。ランダムな観測点を初期セントロイドとして選択するか、K-means++[10] などの複雑な手法を使用します。

インテル® DAAL と RAPIDS cuML の分散型 K 平均法の比較

インテル® DAAL で K 平均法がどのくらい高速化されるか確認するため、CPU ベースの実装より 90 倍速いと言われる RAPIDS cuML[11] と比較しました。さらに、cuML と (インテル® DAAL を使用している) IDP の scikit-learn を比較しました。

パフォーマンスの測定には、次の Amazon Web Services* Elastic Compute Cloud*[12] (AWS* EC2*) インスタンス (ノード) を使用しました。

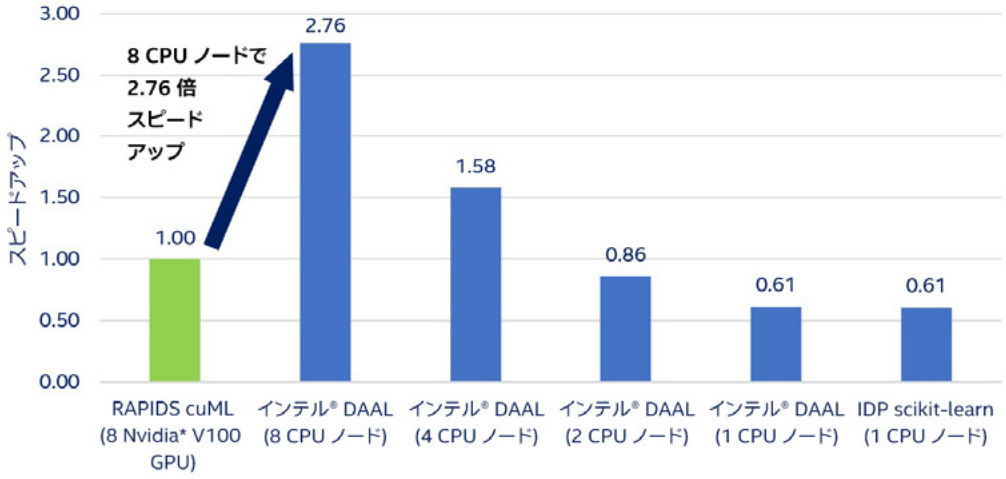
- **複数 (最大 8) の AWS* EC2* `c5.24xlarge` インスタンス**、第 2 世代インテル® Xeon® スケーラブル・プロセッサー
- **1 つの AWS* EC2* `p3dn.24xlarge` インスタンス**、8 NVIDIA* V100 GPU

8 つの **`c5.24xlarge`** インスタンスの時間あたりのコストが 1 つの **`p3dn.24xlarge`** インスタンスの時間当たりのコストとほぼ同じであるため、**`c5.24xlarge`** インスタンスの最大数を 8 にしました。

V100 のメモリーが格納できるサイズと同じ、2 億の観測点、50 列、10 クラスターの合成データセットを使用しました。ちなみに、データセットの観測点を 2 億から 3 億に増やすと、RAPIDS メモリーエラー (**`RMM_ERROR_OUT_OF_MEMORY`**) が発生しました。CPU ベースのシステムは通常、はるかに大きなデータセットを処理できます。データセット生成のコードは、この記事の「コード例」セクションを参照してください。

同一条件で比較するため、測定したすべてのケースで同じ初期化手法を選択しました。また、十分な精度が得られてメモリーに収まる、float32 データ型を選択しました。

図 1 および **表 1** は、4 つ以上のインテル® Xeon® スケーラブル・プロセッサー・ノードでは、インテル® DAAL のパフォーマンスが 8 つの V100 GPU の cuML より高いことを示しています。1 つの CPU ノードでも、8 つの V100 GPU より 40% 遅いだけです。また、1 つの CPU ノードでもデータセット全体を保持して処理できますが、8 つ未満の V100 GPU では処理できません。1 つの CPU ノードの IDP scikit-learn とインテル® DAAL の結果はほぼ同じです。トレーニング時間のわずかな差はインテル® DAAL 関数呼び出しのオーバーヘッドによるものです。



1 **インテル® DAAL を使用した K 平均法トレーニングのスピードアップ**

表 1. K 平均法のトレーニング時間 (秒) とスピードアップ : IDP scikit-learn、インテル® DAAL、RAPIDS cuML の比較

ライブラリー	トレーニング時間、秒	スピードアップ
IDP scikit-learn (1 CPU ノード)	20.40	0.61
インテル® DAAL (1 CPU ノード)	20.21	0.61
インテル® DAAL (2 CPU ノード)	14.41	0.86
インテル® DAAL (4 CPU ノード)	7.84	1.58
インテル® DAAL (8 CPU ノード)	4.49	2.76
RAPIDS cuML (8 Nvidia* V100 GPU)	12.41	1.00

K 平均法アルゴリズムはすべての測定ケースで同じ結果に収束します。クラスター内の 2 乗誤差は 4×10^8 です。

IDP scikit-learn およびインテル® DAAL の例は、「コード例」セクションを参照してください。

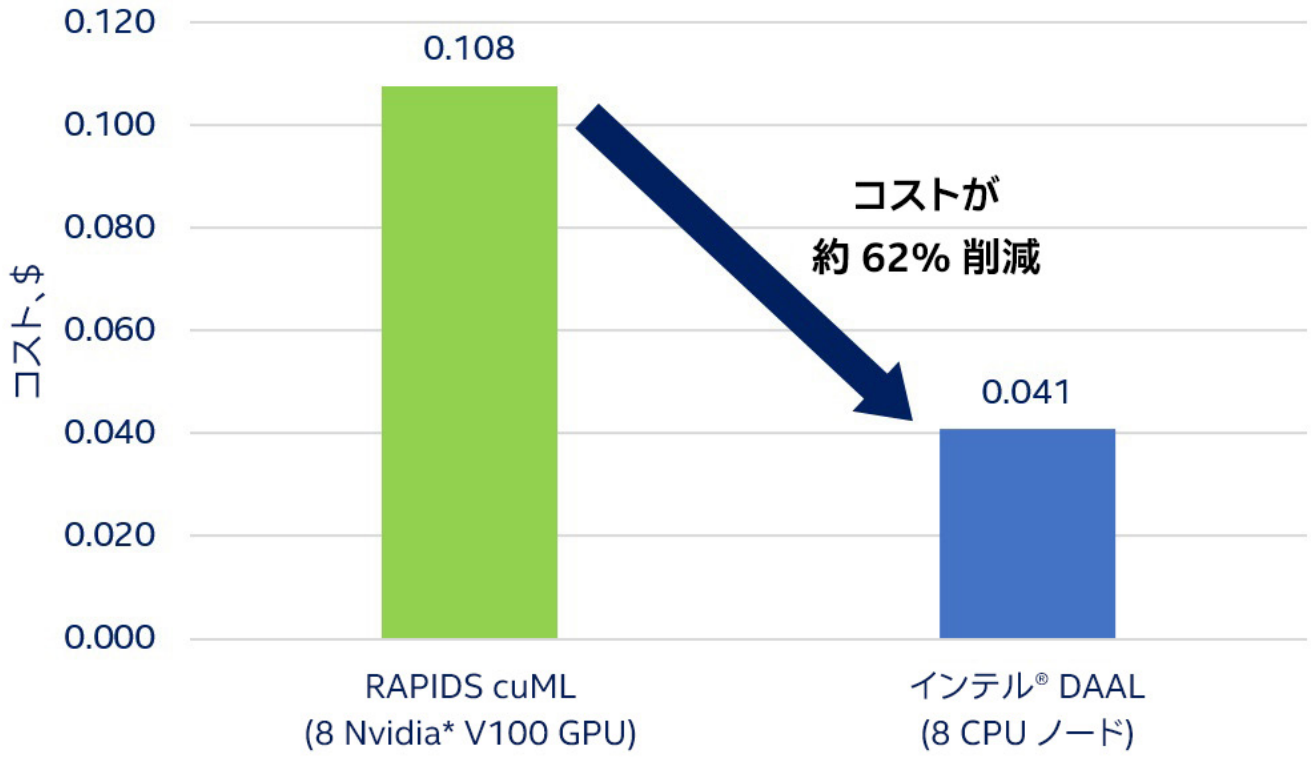
次に、AWS* EC2* インスタンスのコストと 8 つのインテル® Xeon® スケーラブル・プロセッサー・ノードおよび 8 つの V100 GPU のトレーニング時間から、K 平均法のトレーニング・コストを計算しました。

$$\text{K 平均法のトレーニング・コスト (\$)} = \frac{\text{AWS* EC2* インスタンスのコスト (\$/時間)}}{\text{K 平均法のトレーニング時間 (時間)}}$$

図 2 は、8 CPU ノードのインテル® DAAL を使用した場合、K 平均法のトレーニング・コストが約 62% 削減されることを示しています。

高速で低コスト

AWS* EC2* での分散型 K 平均法の計算は、8 つのインテル® Xeon® スケーラブル・プロセッサー・ノードのインテル® DAAL が 8 つの V100 の RAPIDS cuML より 2.76 倍高速で 60% 以上低コストです。また、インテル® Xeon® プロセッサー・ベースのインスタンスは大きなデータセットを処理できます。例えば、8 つの V100 で「メモリーエラー」になるデータを、1 つのインテル® Xeon® プロセッサー・ベースのノードで処理できます。インテル® Optane™ DC パーシステント・メモリー[13] を使用すると、メモリー容量はソケットあたり 4.5TB (2 ソケットのインスタンスあたり 9TB) まで増加しますが、NVIDIA* DGX-2 の GPU メモリーは 512GB です。



2 **K 平均法のトレーニング・コスト[19]: 8 GPU の RAPIDS cuML と 8 CPU ノードのインテル® DAAL の比較。AWS* EC2* (N. Virginia) インスタンスの価格 : `c5.24xlarge` (インテル® DAAL) - 1 時間あたり $4.08 (8 ノードで 1 時間あたり $32.64)、`p3dn.24xlarge` (RAPIDS cuML) - 1 時間あたり $31.212。**

インテルは、oneAPI プログラミング・モデル[9] で、CPU だけでなく、その他のアーキテクチャー (ディスクリート GPU、FPGA、その他のアクセラレーター) でも高いパフォーマンスを実現します。oneAPI は、さまざまなアーキテクチャーの開発者に、パフォーマンスを損なうことなく、統一されたオープンなプログラミング体験を提供します。インテル® DAAL はベータ版インテル® oneAPI ツールキットに含まれています。インテル® DevCloud 開発サンドボックス[14] で試すことができます。

インテル® DAAL の K 平均法の実装と最適化の詳細

K 平均法アルゴリズムは、クラスター割り当てと目的関数 (`inertia`) の計算で使用される、距離の計算に基づいています。観測点 `s = (s1, s2, ⋯, sd)` とセントロイド `c = (c1, c2, ⋯, cd)` 間の `d` 次元ユークリッド空間の距離は、次のように表現できます。

$$D = \sqrt{\sum_{i=0}^{d} (s_i - c_i)^2}$$

これは次の表現と数学的に等価です。

$$= \sqrt{\sum_{i=0}^{d} (s_i^2 - 2s_i c_i + c_i^2)}$$

つまり、2 乗距離は次のように表現できます。

$$D^2 = \sum_{i=0}^{d} (s_i^2 - 2s_i c_i + c_i^2)$$

インテル® DAAL の K 平均法の実装では、観測点を固定サイズのブロックに分割した後、**インテル® スレッディング・ビルディング・ブロック**[15] を使用して並列で 2 乗距離を計算しています。

成分 $\sum_{i=0}^{d} 2s_i c_i$ は行列 **M = 2 S × C** の要素 $\mathbf{m}_{jn}$ (**j** はブロックの観測点のインデックス、**n** はセントロイドのインデックス、**S** と **C** はそれぞれ観測点とセントロイドのブロックの行列) として表現できます。インテル® DAAL は、**インテル® MKL**[16] の行列乗算を使用して、この距離成分の行列を計算しています。

最初の 2 乗距離成分 $\sum_{i=0}^{d} s_i^2$ は定数で、各観測点につき 1 回のみ計算できます。ほかの 2 つ ($\sum_{i=0}^{d} 2s_i c_i$ および $\sum_{i=0}^{d} c_i^2$) は各反復で再計算します。

K 平均法のすべての計算ステージで、行列乗算、セントロイド割り当て、セントロイド再計算を計算するときに、インテル® DAAL は第 2 世代インテル® Xeon® スケーラブル・プロセッサーの**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)**[17] ベクトル命令を使用します。

高度なソフトウェア最適化手法を使用してハードウェアの機能を活用することにより、インテル® DAAL はハイパフォーマンスな K 平均法クラスタリングを実現します。

IDP scikit-learn と daal4py のインストール

IDP の scikit-learn または daal4py (インテル® DAAL 向け Python* インターフェイス)[18] をインストールする最適な方法は、新しい conda 環境を作成することです。

```
conda create -n idp -c intel daal4py scikit-learn pandas
```

コード例

scikit-learn でデータセットを生成 :

```
from sklearn.datasets.samples_generator import make_blobs
import numpy as np

x, y = make_blobs(n_samples=2*10**8, n_features=50, centers=10,
    cluster_std=0.2, center_box=(-10.0, 10.0), random_state=777)
np.savetxt("kmeans_data.csv", x, fmt="%f", delimiter=",")
```

IDP の scikit-learn で K 平均法を実行 :

```
import numpy as np
import pandas as pd
import sklearn as skl

# set config to prevent long finiteness check of input data
skl.set_config(assume_finite=True)

# fast csv reading with pandas
train_data = pd.read_csv("kmeans_data.csv", dtype=np.float32)

alg = skl.cluster.KMeans(n_clusters=10, init="random",
    tol=0, algorithm="full", max_iter=50)
alg.fit(train_data)
```

daal4py で K 平均法を実行 :

```
import numpy as np
import pandas as pd
import daal4py as d4p

d4p.daalinit()
# fast csv reading with pandas
data = pd.read_csv("local_kmeans_data.csv", header=None, dtype=np.float32)

centroids = d4p.kmeans_init(nClusters=10, fptype="float",
    method="randomDense", distributed=True)
alg = d4p.kmeans(nClusters=10, maxIterations=50, fptype="float",
    accuracyThreshold=0, assignFlag=False, distributed=True)
result = alg.compute(data, centroids)
```

K 平均法トレーニングの構成

- **CPU 構成 :** c5.24xlarge AWS* EC2* インスタンス : 第 2 世代インテル® Xeon® スケーラブル・プロセッサー、2 ソケット、ソケットあたり 24 コア、インテル® ハイパースレッディング・テクノロジー有効、インテル® ターボ・ブースト・テクノロジー有効、RAM: 192GB (12 スロット /16GB/2933MHz)、BIOS: 1.0 Amazon EC2* (ucode: 0x500002c)、OS: Ubuntu* 18.04.2 LTS。
- **GPU 構成 :** p3dn.24xlarge AWS* EC2* インスタンス : インテル® Xeon® Platinum 8175M プロセッサー、2 ソケット、ソケットあたり 24 コア、インテル® ハイパースレッディング・テクノロジー有効、インテル® ターボ・ブースト・テクノロジー有効、RAM: 768GB、GPU: 8 Tesla* V100-SXM2-32G、GPU メモリー : 32GB、BIOS: 1.0 Amazon EC2* (ucode: 0x2000065)、OS: Ubuntu* 18.04.2 LTS。
- **ソフトウェア :** Python* 3.6、NumPy* 1.16.4、scikit-learn 0.21.3、daal4py 2019.5、インテル® DAAL 2019.5、インテル® MPI ライブラリー 2019.5、RAPIDS cuML 0.10、RAPIDS cuDF 0.10、CUDA* 10.1、NVIDIA* GPU ドライバー 418.87.01、dask 2.6.0。
- **アルゴリズムのパラメーター :** 単精度 (float32)、反復数 = 50、しきい値 = 0、ランダム初期セントロイド。

参考資料

1. インテルにより最適化されたフレームワーク (英語)
2. インテルにより最適化されたライブラリー (英語)
3. インテル® Xeon® スケーラブル・プロセッサー
4. インテルの AI 向けハードウェア・スタック (英語)
5. scikit-learn (英語)
6. Python* 向けインテル® ディストリビューション
7. インテル® DAAL
8. GitHub* のインテル® DAAL (英語)
9. ファクトシート : oneAPI
10. k-means++ (英語)
11. RAPIDS 0.9: A Model Built to Scale, section “A Model Built to Scale: cuML” (英語)
12. Amazon EC2*
13. インテル® Optane™ テクノロジー
14. インテル® DevCloud (英語)
15. インテル® TBB
16. インテル® MKL
17. インテル® アドバンスト・ベクトル・エクステンション 512
18. daal4py: インテル® DAAL 向け Python* API (英語)
19. Amazon EC2* 料金表

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。

どのように実現したのだろう？

世界中の開発者が、インテル® ソフトウェア開発ツールを利用して、アプリケーションのパフォーマンス、スケーラビリティー、移植性を向上しています。
そして、その経験をほかの開発者が役立てられるように共有しています。

詳細（英語）>

intel Software

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

グラフ・アナリティクス・パフォーマンスの測定

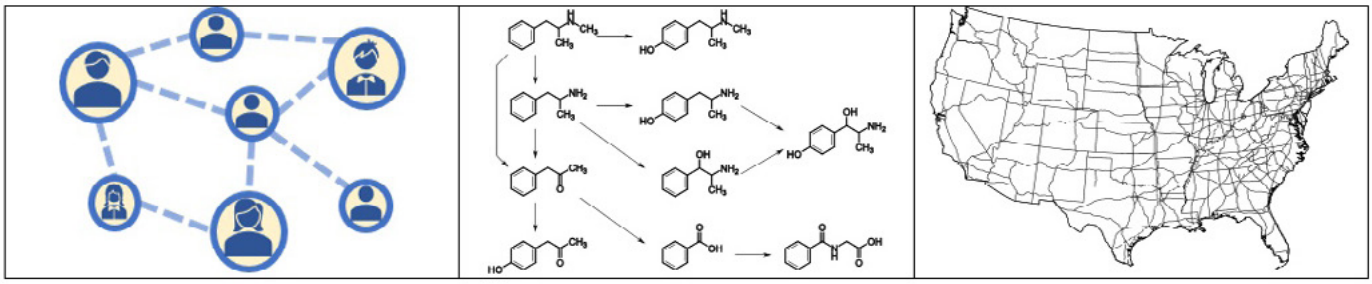
グラフ・アナリティクスとは何か、またなぜ重要なのか

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および The Parallel Universe 編集長

グラフは一連のオブジェクトとそれらの関係を表現する優れた方法です (**図 1**)。グラフ・アナリティクスは、要素間の接続から情報を取り出す一連の手法です。

例えば、次のような用途に使用できます。

- ソーシャル・ネットワークの人の**関係を表す**
- 通信ネットワークや配電網の**切断地点を調べる**
- 生化学的なプロセスに対する**薬の影響を判断する**
- 通信ネットワークの**ロボコールを検出する**
- 地図の場所間の**最適な経路を見つける**



1 **グラフはあらゆるところに存在する**

ガートナーが 2019 年のデータ / アナリティクス・テクノロジー・トレンドのトップ 10 の 1 つに取り上げたように、グラフ・アナリティクスは大きな注目を集めています。

> 「グラフ処理とグラフ DBMS (データベース管理システム) のアプリケーションは、2022 年まで年間 100% 成長し、データ準備を継続的に促進して、より複雑でアダプティブなデータ・サイエンスを実現する見込みです。」(出典 : **ガートナー、2019 年のデータ / アナリティクス・テクノロジ・トレンドのトップ 10 を発表**)

インテルは、グラフ・アナリティクスの分野でも長い間先頭に立って作業を進めてきました。例えば、インテルは、疎線形代数などのグラフ問題を公式化する **GraphBLAS 仕様** (英語) の共同策定者です。**GraphBLAS API** (英語) は 2017 年に公開されましたが、初期の企画書やマニフェストは 10 年以上前に公開されたものです。現在は、業界および学術分野の同じパートナーが、グラフ・アルゴリズムのライブラリー向けの **LAGraph 仕様** (英語) を共同で策定しています。インテルは、国防高等研究計画局 (DARPA) の大規模なグラフ・データセットを処理する新しいプロセッサーを開発するプロジェクトにも協力しています (「**DARPA Taps Intel for Graph Analytics Chip Project**」(英語) を参照)。インテルは、今後も引き続きグラフ・アナリティクスの革新および推進に力を注いでいく予定です。

グラフ・アナリティクス・パフォーマンスのベンチマーク測定

グラフ・アナリティクスは大規模で変化に富んでいます。**図 1** の単純な例でさえ、異なる特性が含まれています。例えば、接続数が多いネットワークもあれば、接続数が少ないネットワークもあります。Twitter* ユーザーのネットワークはウェブページのネットワークとは接続性が異なり、数百万のフォロワーがいるユーザーもいますが、それほどフォロワーが多くないユーザーがほとんどです。グラフ・アルゴリズム、グラフトポロジー、グラフサイズの 1 つの組み合わせでネットワーク全体を表現することはできません。

そのため、カリフォルニア大学バークレー校の **GAP Benchmark Suite** (英語) を使用してグラフ・アナリティクス・パフォーマンスを測定しました。GAP は、6 つの広く利用されているアルゴリズム (**表 1**) と 5 つの小規模から中規模のサイズのグラフ (**表 2**) を使用します。各グラフは異なる特性を備えています。1 つのトポロジーグラフで適切に動作する最適化がほかのトポロジーグラフでは適切に動作しないこともあるため、この点は重要です。例えば、Road グラフは相対的に小さなグラフですが、径が大きいと一部のアルゴリズムで問題が発生します。このため、30 の GAP データ項目で、グラフ・アナリティクス全体を網羅しています。

表 1. GAP は 6 つの一般的なグラフ・アナリティクス・アルゴリズムのパフォーマンスを測定

グラフのアルゴリズム	ベンチマークの定義
幅優先探索 (BFS)	ランダムな 64 の頂点から開始した 64 の走査の平均時間
単一始点最短経路 (SSSP)	64 のランダムな始点とほかのすべての到達可能な頂点間の最短経路の計算の平均時間
連結成分 (CC)	16 の CC 計算の平均時間
ページランク (PR)	16 の PR 計算の平均時間 (許容範囲 = 0.0001、最大反復数 = 1,000)
三角形構造数 (TC)	3 つの TC 計算の平均時間
媒介中心性 (BC)	4 つのランダムな頂点の 16 回の BC 計算の平均時間

表 2. GAP は 5 つのさまざまなサイズとトポロジーのグラフを使用してグラフ・アナリティクス・パフォーマンスの全体像を表示

グラフ	頂点の数 (100 万)	辺の数 (100 万)	種類	トポロジーの特性
Twitter	61.6	1,468.4	リアル	偏った次数分布
Web	50.6	1,949.4	リアル	高平均次数
Road	23.9	58.3	リアル	高径
Kron	134.2	2,111.6	合成	スケールフリー Kronecker 積グラフ
Urand	134.2	2,147.4	合成	非スケールフリー Erdős-Rényi グラフ

GAP は、明確に定義された、客観的な、既製のベンチマークであるという長所もあります。特別なハードウェア構成やソフトウェア構成を必要としないため、簡単に実行できます。

次の手順でベンチマークを実行しました。

1. **GAP パッケージ** (英語) をダウンロードします。
2. **`gapbs-master`** サブディレクトリーで '**`make`**' を実行して、6 つのグラフ・アナリティクス・カーネルのリファレンス実装を構築します。デフォルトの GAP ビルド・パラメーター (GNU* C++ コンパイラー : **`-std=c++11 -O3 -Wall -fopenmp`** オプション) を使用しました。システムにインストールしていたコンパイラーは、GNU* v7.4.0 です。
3. 同じディレクトリーで '**`make -f benchmark/bench.mk bench graphs`**' を実行して、5 つのベンチマーク・グラフをダウンロードまたは生成し、効率的な入力形式に変換します。

4. GAP リファレンス実装は OpenMP* で並列化されているため、スレッド数を設定することが重要です。スレッド数を設定しないと、GAP は並列処理の効率に関係なく、利用可能なコアをすべて使用します。例えば、OpenMP* スレッドの数を **32** に設定するには、'**`export OMP_NUM_THREADS=32`**' コマンドを使用します。
5. 最後に、'**`make -f benchmark/bench.mk bench run`**' を実行してベンチマークを起動し、結果を生成します (**表 3**)。

表 3. 2 ソケットのインテル® Xeon® プロセッサー・ベースのシステム[1] で実行した GAP の計算時間 (秒、値が小さいほど良い)。GAP は、1、8、16、24、32、48、64、および 96 スレッドで実行。各テストの最高値を表示。

グラフ	グラフのアルゴリズム					
	BFS	SSSP	CC	PR	TC	BC
Twitter	0.2	1.8	0.2	7.4	39.0	7.5
Web	0.3	0.8	0.2	4.4	14.9	2.5
Road	0.3	1.0	0.04	1.0	0.02	2.4
Kron	0.3	3.6	0.6	10.6	183.3	17.3
Urand	0.4	5.6	1.1	16.0	12.2	24.8

NVIDIA* V100 で比較データを生成しようとしたところ、いくつかの技術的な問題に直面しました。

1. 1 つの V100 のメモリーで処理できたのは 1 つの GAP グラフ (Road) のみでした。
2. 複数の V100 のメモリーを結合して使用できたのは RAPIDS cuGraph の 1 つの GAP アルゴリズム (PR) のみでした。
3. cuGraph の現在のバージョンは BC 実装をサポートしていません。
4. 特定の実装の詳細やアルゴリズム・パラメーターは cuGraph API およびドキュメントに記述されていません。例えば、cuGraph のマルチ V100 PR 実装には収束許容範囲パラメーターが用意されていないため、GAP の結果を同一条件で比較することが困難です。

結果的に、30 の GAP テストのうち実行できたのは 9 つのみでした (**表 4**)。両方のアーキテクチャーで実行できた GAP の結果を比較すると、グラフが GPU メモリーに収まる場合 (Road) やアルゴリズムが複数の GPU を使用できる場合 (PageRank) でも、ほとんどのテストでインテル® Xeon® プロセッサーのほうが V100 よりパフォーマンスが高くなっています。また、総保有コスト (TCO) の点から見ても、グラフ・アナリティクスにはインテル® Xeon® プロセッサーのほうが適していることは明らかです (**表 5**)。

表 4. AWS* EC2* `p3.16xlarge` インスタンス (8 つの V100 GPU を NVLink* で接続) で実行した cuGraph の計算時間 (秒、値が小さいほど良い)。Road テストは 1 つの V100 を使用。Twitter および Web テストは 8 つの V100 を使用。cuGraph のマルチ V100 PR 実装には収束許容範囲パラメーターが用意されていないため、これらのテストにはデフォルトのパラメーターを使用。Kron および Urand の PR テストは "`thrust::system::system_error`" エラーにより失敗。DNR = メモリー不足のため実行不可。NA = cuGraph v0.9.0 では利用不可。

グラフ	グラフのアルゴリズム					
	BFS	SSSP	CC	PR	TC	BC
Twitter	DNR	DNR	DNR	14.8	DNR	NA
Web	DNR	DNR	DNR	26.8	DNR	NA
Road	0.5	25.7	94.1	0.2	0.01	NA
Kron	DNR	DNR	DNR	失敗	DNR	NA
Urand	DNR	DNR	DNR	失敗	DNR	NA

表 5. インテル® Xeon® プロセッサーと V100 ベンチマーク・システムの相対 TCO。インテル® Xeon® プロセッサー・ベースのベンチマーク・システムと完全に一致する EC2* インスタンスは存在せず、類似の `m4.16xlarge` インスタンスで計算しているため、AWS* 価格の比較は概算であることに注意。

メトリック	インテル® Xeon® プロセッサー・ベースのシステム	V100 ベースのシステム
プロセッサー	2 x インテル® Xeon® Gold 6252 プロセッサー	8 x NVIDIA* Tesla* V100 SXM2 (16GB HBM2)
熱設計電力 (TDP)	2 x 150W = 300W	8 x 300W = 2,400W
価格	2 x $3,655 = $7,310	8 x $10,664 = $85,312
AWS* EC2* インスタンスの価格	m4.16xlarge: $3.20/時間	p3.16xlarge: $24.48/時間

包括的、客観的、かつ再現可能

インテル® Xeon® プロセッサー・プラットフォームの GAP の結果 (**表 3**) は、インテル® コンパイラーを使用して積極的な最適化とベクトル化を行い、OpenMP* **`OMP_PROC_BIND/OMP_PLACES`** 環境変数を微調整し、**`numactl`** ユーティリティーでテストして、ページサイズなどを調整すればさらに向上できると思われますが、この記事の目的とは異なるためここでは取り上げません。この記事の目的は、あくまでも、曖昧な表現やベンチマークのトリックに頼ることなく、包括的、客観的、かつ再現可能なグラフ・アナリティクス・パフォーマンス・データを簡単に得る方法を示すことです。

参考資料

1. プロセッサー : インテル® Xeon® Gold 6252 プロセッサー (2.10GHz、24 コア)、インテル® ハイパースレッディング・テクノロジー有効 (ソケットあたり 48 仮想コア)。メモリー : 384GB Micron DDR4-2666。オペレーティング・システム : Ubuntu* Linux* リリース 4.15.0-29、カーネル 31.x86_64。ソフトウェア : GAP Benchmark Suite (2019 年 9 月にダウンロードして実行)。プロセッサー : NVIDIA* Tesla* V100。メモリー : 16GB。オペレーティング・システム : Ubuntu* Linux* リリース 4.15.0-1047-aws、カーネル 49.x86_64。ソフトウェア : RAPIDS v0.9.0 (2019 年 9 月にダウンロードして実行)。
2. 出 典 : https://www.microway.com/hpc-tech-tips/nvidia-tesla-v100-price-analysis/ (英語) (2019 年 9 月にアクセス) および https://ark.intel.com/content/www/us/en/ark/products/192447/intel-xeon-gold-6252-processor-35-75m-cache-2-10-ghz.html (英語) (2019 年 9 月にアクセス)
3. 出典 : https://aws.amazon.com/ec2/pricing/on-demand/ (英語) (2019 年 9 月にアクセス)
4. GAP のチューニングに興味のある方は、「グラフ解析ワークロードのパフォーマンスの向上」(The Parallel Universe 36 号) を参照してください。

Software

THE PARALLEL UNIVERSE

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。性能やベンチマーク結果について、さらに詳しい情報をお知りになりたい場合は、https://www.intel.com/benchmarks/ (英語) を参照してください。システム構成 : 該当記事のワークロード構成の詳細を参照してください。性能の測定結果は 2019 年 3 月 25 日時点のテストに基づいています。また、現在公開中のすべてのセキュリティー・アップデートが適用されているとは限りません。詳細については、公開されている構成情報を参照してください。絶対的なセキュリティーを提供できる製品はありません。* 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。

テストでは、特定のシステムでの個々のテストにおけるコンポーネントの性能を文書化しています。ハードウェア、ソフトウェア、システム構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。購入を検討される場合は、ほかの情報も参考にして、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。性能やベンチマーク結果について、さらに詳しい情報をお知りになりたい場合は、http://www.intel.com/benchmarks/ (英語) を参照してください。

インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、またはサービスの有効化が必要となる場合があります。実際の性能はシステム構成によって異なります。

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® アドバンスト・ベクトル・エクステンション (インテル® AVX)* は、特定のプロセッサー演算で高いスループットを示します。プロセッサーの電力特性の変動により、AVX 命令を利用すると、a) 一部の部品が定格周波数未満で動作する、b) インテル® ターボ・ブースト・テクノロジー 2.0 を使用する一部の部品が任意または最大のターボ周波数に達しない可能性があります。実際の性能はハードウェア、ソフトウェア、システム構成によって異なります。詳細については、http://www.intel.co.jp/jp/technology/turboboost/ 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・データまたはウェブサイト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ウェブサイトにアクセスし、本資料で参照しているデータが正確かどうかを確認してください。

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

本資料で説明されている製品およびサービスには、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

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

© 2020 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Atom、Intel Optane、Movidius、Nervana、Xeon、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 またはその子会社の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

JPN/2002/PDF/XL/MKTG/SU