intel software

THE PARALLEL UNIVERSE

データ並列 C++ のリダクション操作のパフォーマンスを解析する

ヘテロジニアスな処理にはデータ並列化が必要：
SYCL* と DPC++ から始めよう

OpenMP* のアクセラレーター・オフロード

Issue
45
2021

目次

編集者からのメッセージ 3

ヘテロジニアスな処理にはデータ並列化が必要：SYCL* と DPC++ から始めよう 5

注目記事

データ並列 C++ のリダクション操作のパフォーマンスを解析する 10
一般的なリダクション・パターンのさらなるチューニング

OpenMP* のアクセラレーター・オフロード 22
ヘテロジニアス・アーキテクチャー全体にわたる移植性

インテル® oneAPI ツールキットを使用した分散型 AI トレーニングの最適化 30
インクリメンタルなチューニングによりパフォーマンスを大幅に向上

ディープラーニングにおける CPU の役割と可能性 38

SigOpt と Ray Tune を使用した MiniNAS ニューラル・アーキテクチャー検索 41
SigOpt を使用してモデルのアーキテクチャーを系統的に検索

エンドツーエンドの AI パイプラインのパフォーマンス最適化 51
インテル® プロセッサー向けに最適化されたフレームワークとライブラリー

CatBoost のパフォーマンスを最大 4 倍に最適化 58
マシンラーニングのトレーニングのパフォーマンスを向上するコツ

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。

ISC 2021 で大きな発表がありました！

インテル コーポレーションの副社長兼ハイパフォーマンス・コンピューティング事業本部長の Trish Damkroger は、2021 International Supercomputing Conference（ISC）の基調講演において、いくつかの新しい製品と製品拡張を発表しました。ここではそのすべてを取り上げることはできませんが（詳細は、「**HPC および AI 向けの新しいインテルの XPU 革新**」（英語）を参照）、その中から私が特に関心を持っている 2 つを紹介したいと思います。1 つ目は、インテルの Xe-HPC ベースのディスクリート GPU（開発コード名 Ponte Vecchio、略称 PVC）が実働テスト段階に入りました。現在、私は統合 GPU で oneAPI の実験を行っていますが、PVC でプロダクション・スケールのワークロードを実行できるのを心待ちにしています。2 つ目は、統合高帯域幅メモリー（HBM）とインテル® Advanced Matrix Extensions（インテル® AMX）が、次世代のインテル® Xeon® スケーラブル・プロセッサー（開発コード名 Sapphire Rapids、略称 SPR）に搭載される予定です。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）によるベクトル化は、すでにインテル® Xeon® スケーラブル・プロセッサーにおいて優れたアクセラレーションを提供していますが、HBM とインテル® AMX により、さらに効果的なインテル® Xeon® スケーラブル・プロセッサー・ベースのアクセラレーションを実現します。

本号では、データ並列 C++ での低レベルのパフォーマンス・チューニングから、AI パフォーマンスを向上させるシンプルかつ効果的な手法まで、さまざまなトピックを紹介します。oneAPI とヘテロジニアスな並列処理に関しては、注目記事「**データ並列 C++ のリダクション操作のパフォーマンスを解析する**」で前号の記事「**データ並列 C++ のリダクション操作**」をさらに深く掘り下げます。名誉編集長の James Reinders による寄稿記事「**ヘテロジニアスな処理にはデータ並列化が必要**」もあります。「**OpenMP* のアクセラレーター・オフロード**」では、OpenACC* から移植性に優れた OpenMP* 標準へ移行する方法と OpenMP* オフロードのパフォーマンスを向上するヒントを紹介します。

データ・アナリティクスに関しては、インド工科大学ルーキー校の Sparsh Mittal 教授による寄稿記事「**ディープラーニングにおける CPU の役割と可能性**」を掲載しています。そして、ニューラル・ネットワークのトレーニングのパフォーマンスに関する 2 つの記事「**インテル® oneAPI ツールキットを使用した分散型 AI トレーニングの最適化**」と「**SigOpt と Ray Tune を使用した MiniNAS ニューラル・アーキテクチャー検索**」が続きます。最後に、**インテル® oneAPI AI アナリティクス・ツールキット**（英語）とインテルの最適化を含む既製のパッケージを使用して、古典的なマシンラーニングのパフォーマンスを大幅に向上する方法を紹介する 2 つの記事、「**エンドツーエンドの AI パイプラインのパフォーマンス最適化**」と「**CatBoost のパフォーマンスを最大 4 倍に最適化**」で本号を締めくくります。

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。

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、oneAPI を利用したヘテロジニアス並列コンピューティング向けのインテル・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
2021 年 7 月

ヘテロジニアスな処理にはデータ並列化が必要：SYCL* と DPC++ から始めよう

James Reinders The Parallel Universe 創刊編集長および名誉編集長

[注：この記事は、**The New Stack**（英語）で掲載されている James Reinders の記事を転載したものです。]

これからは XPU の時代です。

ハードウェアの革新により、CPU、GPU、FPGA、DSP、ASIC などが爆発的に増えていますが、これらを XPU と総称することができます。XPU とは、あらゆる種類の「処理ユニット」であり、アプリケーションの計算を支援するあらゆるハードウェアを指します。

XPU の急増は、より多様な処理ユニットに対応したコードが求められるため、開発者にとって大きな課題です。新しいアーキテクチャーでアプリケーションのパフォーマンスを向上するには、コードを書き換えたり、テストしたりする時間やコストを考慮しなければなりません。コードの保守性を維持するため、コードをできるだけ多くの XPU に対応させることが最も重要です。アプリケーション開発をクロスアーキテクチャー・モデルに移行することで、

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。

時間とコストを大幅に削減できることが明らかになっていますが、ヘテロジニアス・コンピューティングの普及に伴い、これは差し迫った課題となっています。

現在、私たちの世界は XPU の世界へと急速に変化しており、最終的にはすべてのコンピューティングを変革することになるため、再考が必要です。

XPU：高速な計算を行うソフトウェアの再発明

広く利用されているソフトウェア・プログラミング・システムである CUDA* は、NVIDIA の GPU 向けに設計されており、効果を発揮しています。OpenCL* はオープンなアプローチで、マルチベンダー・サポートを実現していますが、C 言語中心で C++ のニーズに対応できないという欠点があります。

CUDA* と OpenCL* は、その目的を十分に果たしてきましたが、今後は XPU の可能性を引き出すため、真にオープンでマルチベンダーなアプローチが必要になります。

SYCL* とデータ並列 C++（DPC++）が最善である理由

CUDA* と OpenCL* から得た経験を基に、ヘテロジニアス・システム向けの C++ ベースのデータ並列化を実現する、真にオープンなソリューションが登場しました。それが SYCL* です。SYCL* は、複数のハードウェア・アクセラレーター上でプログラミングの生産性を向上させる高度なプログラミング・モデルです。SYCL* は、さまざまなベンダーによるサポート、幅広い関心、そして複数のコンパイラー・プロジェクトによるサポートを短期間で達成しました。

ヘテロジニアス化が進む世界で効率良くプログラミングを行うには、すべての XPU でパフォーマンスを引き出す必要があるため、SYCL* は重要です。真にオープンなアプローチだけがそれを可能にします。

SYCL* は、ヘテロジニアスなハードウェア（XPU）を対象としたシングルソースの C++ データ並列プログラミングを実現するオープン・スタンダードです。SYCL* は、ホストに C++ を使用し、デバイスにドメイン固有のカーネル言語を使用するのではなく、システム上の複数のデバイスを対象に C++ でシングルソース・コンパイルを可能にします。

SYCL* は、C++ にカーネルスタイルのプログラミングと、システム内のアクセラレーターを検索、照会、使用するためのメカニズムをもたらします。カーネルベースのプログラミングは、データ並列性を活用する重要なプログラミング・スタイルであり、OpenCL* や CUDA* でもサポートされています。標準的な方法でアクセラレーターを列挙してアクセスする機能は、OpenCL* でも導入されています。

また、DPC++ は、LLVM コミュニティーにオープンな実装を提供しており、すべてを LLVM C++ コンパイラーに反映することを目指しています。DPC++ は、いくつかの拡張を加えて SYCL* を実装することを目的としており、**SYCL* 2020**（英語）標準規格がリリースされる前から、その多くの機能の実装に取り組んできました。現在、SYCL* 2020 仕様全体（**日本語版の仕様はこちら**）との整合性をとるための作業を行っており、すべての作業は非常に活発な**オープンソース・リポジトリー**（英語）で確認できます。DPC++ は、**インテル**がインテル® CPU、GPU、FPGA ターゲットに使用しています。**Codeplay**（英語）も NVIDIA の GPU ターゲットに使用しています。別の SYCL* コンパイラーである **hipSYCL**（英語）は、AMD の HIP/ROCm* と接続することで、AMD の CPU や GPU をサポートしています。SYCL* 用の複数のオープンソース・コンパイラーが存在することは、コミュニティーにとって素晴らしいことであり、SYCL* が幅広く、多様で、オープンなサポートを受けていることを示しています。

2019 年から 2020 年にかけて、私はチームと協力して、SYCL* と DPC++ に関する初めての書籍を出版しました。この書籍は、**Apress のウェブサイト**（英語）から無料でダウンロードできます。書籍の出版後まもなく、Khronos Group が **SYCL* 2020**（英語）の最終仕様（**日本語版の仕様はこちら**）を発表しました。

SYCL* 2020 の仕様がリリースされたことは、重要なマイルストーンと言えます。この仕様は、業界の多くの熱心な人々による長年の仕様策定の成果であり、将来性のあるオープンな仕様です。C++17 をベースにした SYCL* 2020 は、標準的な C++ アプリケーションの高速化を可能にし、ISO C++ のロードマップとの連携を強化しています。Khronos Group は SYCL* 2020 の発表の中で、統合共有メモリー（USM）のサポート、ビルトイン・リダクション、CTAD の広範な使用、C++ 標準に沿ったアトミック操作など、SYCL* 2020 の特徴を強調しています。

多様性とプログラミングの健全性のため XPU のオープン性を維持しよう

SYCL* と DPC++ は、XPU を効率良く利用する手助けとなります。これらは、ライブラリーやさまざまなソフトウェア開発ツールを含む、XPU をサポートするための広範な推進活動の一部です。これこそが、私が情熱を持って取り組んでいる **oneAPI 業界イニシアチブ**（英語）の原点であり、インテルに復帰した際には、その一員となることを楽しみにしていました。すべての XPU をオープンに使用する課題を軽減する、という目的への支援が SYCL* と oneAPI への関心を高めています。その一例が、oneAPI ディープ・ニューラル・ネットワーク・ライブラリー（oneDNN）です。当初、oneDNN はインテル® プロセッサー向けに高度に最適化されていましたが、世界最速のコンピューターである日本の「富岳」（ARM のプロセッサーを搭載）を加速させました。その結果、oneDNN は現在、ARM のプロセッサーも強力にサポートしています。オープンな SYCL* と oneAPI のライブラリーとツールは、オープン性とパフォーマンスを備えた、すべての XPU 向けのプログラミングを実現する新しい時代の到来を後押ししています。

ソフトウェア開発者のコミュニティーが一丸となって、業界全体に貢献する SYCL* をはじめとする標準規格を策定し、ヘテロジニアス・プログラミング（XPU）や、並列処理を取り入れた最新の C++ の採用を強力にサポート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。

SYCL* は、幅広いサポート、多くの参加者、複数のオープンソースの実装、そして無限の可能性を秘めたオープン・スタンダードです。DPC++ は、オープンな LLVM ベースのコンパイラーを提供し、SYCL* をサポートする労力を軽減し、XPU 間の強力な互換性を促進します。oneAPI は、ソフトウェア開発のあらゆる側面において、XPU のオープンでハイパフォーマンスなアクセスについて議論し、推進するためのフォーラムを提供しています。

XPU 時代に備え、SYCL*、DPC++、oneAPI について理解を深めていただければと思います。私たちは一丸となってオープンに XPU をサポートし、XPU の多様性から得られるメリットを享受し、効率良くプログラミングを行いましょう。

intel software

コードの境界を乗り越える

インテル® DevCloud for oneAPI でクロスアーキテクチャー・プログラミングのパワーを体験してください。

デモ

Mandelbrot デモを異なるアーキテクチャーで実行して、クロスアーキテクチャーのパフォーマンスを確認できます。

学習

サンプルコードを含む 25 の Jupyter* Notebook でデータ並列 C++ を実際に体験できます。

開発

最新のインテルの CPU、GPU、FPGA で将来に対応したアプリケーションを計画してテストできます。

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、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
© Intel Corporation

データ並列 C++ のリダクション操作のパフォーマンスを解析する

一般的なリダクション・パターンのさらなるチューニング

Ramesh Peri インテル コーポレーション シニア主席エンジニア

前号の記事「**データ並列 C++ のリダクション操作**」では、総和演算子を使用して 1,000 万要素の配列を単一の値にレデュースするいくつかのカーネルを調査しました。この記事では、マルチブロック・インターリーブ・リダクションと呼ばれる別のリダクション手法を紹介します。第 9 世代と第 12 世代のインテル GPU でインテル® VTune™ プロファイラーを使用してすべてのリダクション操作を比較し、カーネル間でパフォーマンスに差が生じる理由を説明します。

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。

マルチブロック・インターリーブ・リダクション

データ並列 C++（DPC++）では、ショートベクトルをロード / ストアや算術演算子の操作が定義された基本データ型として定義しています。これらのショートベクトルを使用して別のレベルのブロックを追加し、サポートするアーキテクチャー向けに非常に長いベクトル演算をコンパイラーに生成させることができます。**vec<int, 8>** データ型は、8 つの整数からなるベクトルです。このデータ型を使用して、**図 1** に示すリダクション操作を実装します。図に示すように、アクセスパターンはベクトルサイズが 2、サブグループ・サイズが 4 で、ワーク項目ごとに入力ベクトルの 4 要素を処理します。

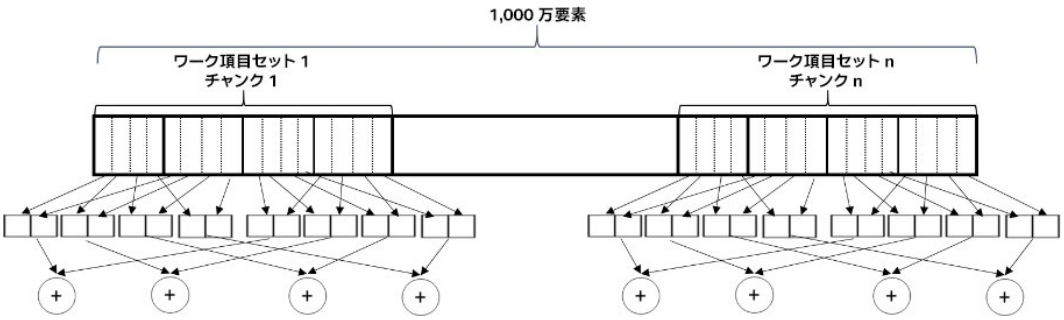

図 1. 要素のベクトルを読み込み、その要素に対してベクトル・リダクション操作を行い、最終結果のベクトルをレデュース

以下のコードは、ベクトルサイズ 8、サブグループ・サイズ 16 で、ワーク項目ごとに入力ベクトルの 256 要素を処理する、上記のメモリー・アクセス・パターンのリダクション操作を実装します。

```
void multiBlockInterleavedReduction(sycl::queue &q,
                    sycl::buffer<int> inbuf,
                    int &res) {
  const size_t data_size = inbuf.get_size()/sizeof(int);
  int work_group_size =
      q.get_device().get_info<sycl::info::device::max_work_group_size>();
  int elements_per_work_item = 256;
  int num_work_items = data_size / elements_per_work_item;
  int num_work_groups = num_work_items / work_group_size;
  sycl::buffer<int> sum_buf(&res, 1);

  q.submit([&](auto &h) {
      const sycl::accessor buf_acc(inbuf, h);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);
      sycl::accessor<sycl::vec<int, 8>, 1, sycl::access::mode::read_write,
                     sycl::access::target::local>
          scratch(work_group_size, h);
      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
                [=](sycl::nd_item<1> item)
                [[intel::reqd_sub_group_size(16)]] {
         size_t glob_id = item.get_global_id(0);
         size_t group_id = item.get_group(0);
         size_t loc_id = item.get_local_id(0);
```

```
            sycl::ONEAPI::sub_group sg = item.get_sub_group();
            size_t sg_size = sg.get_local_range()[0];
            size_t sg_id = sg.get_group_id()[0];
            sycl::vec<int, 8> sum{0, 0, 0, 0, 0, 0, 0, 0};
            using global_ptr =
               sycl::multi_ptr<int,sycl::access::address_space::global_space>;
            int base = (group_id * work_group_size + sg_id * sg_size)
                             * elements_per_work_item;
            for (size_t i = 0; i < elements_per_work_item / 8; i++)
              sum += sg.load<8>(global_ptr(&buf_acc[base + i * 8 * sg_size]));
            scratch[loc_id] = sum;
            for (int i = work_group_size / 2; i > 0; i >>= 1) {
              item.barrier(sycl::access::fence_space::local_space);
              if (loc_id < i)
                 scratch[loc_id] += scratch[loc_id + i];
            }
            if (loc_id == 0) {
               int sum=0;
               for (int i = 0; i < 8; i++)
                  sum += scratch[0][i];
               auto v = sycl::ONEAPI::atomic_ref<int,
                           sycl::ONEAPI::memory_order::relaxed,
                           sycl::ONEAPI::memory_scope::device,
                           sycl::access::address_space::global_space>(
                           sum_acc[0]);
               v.fetch_add(sum);
            }
         });
      });
}
```

このカーネルは、明示的にアドレスを計算する代わりに、ベクトルロード操作を使用してエンコードすることもできます。また、各ワーク項目がロードしたベクトルを最初にローカルでレデュースするように変更することもできます (この実装のアクセスパターンを**図 2** に示します)。

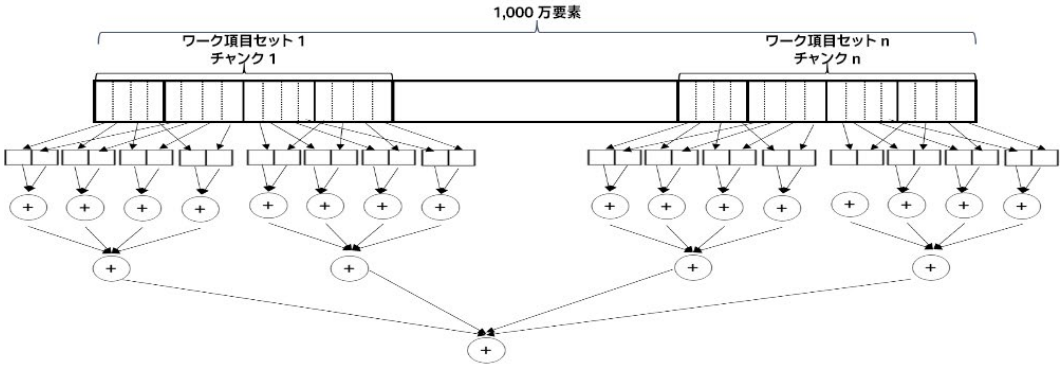

図 2. 要素のベクトルを読み込み、ベクトルを単一の結果にレデュースしてからリダクションを行う

```
void multiBlockInterleavedReductionVector(sycl::queue &q,
                    sycl::buffer<int> inbuf,
                    int &res) {
  const size_t data_size = inbuf.get_size()/sizeof(int);
  int work_group_size =
      q.get_device().get_info<sycl::info::device::max_work_group_size>();
  int elements_per_work_item = 256;
  int num_work_items = data_size / 4;
  int num_work_groups = num_work_items / work_group_size;
  sycl::buffer<int> sum_buf(&res, 1);

  q.submit([&](auto &h) {
      const sycl::accessor buf_acc(inbuf, h);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);
      sycl::accessor<int, 1, sycl::access::mode::read_write,
                     sycl::access::target::local>
          scratch(1, h);
      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
                [=](sycl::nd_item<1> item)
                [[intel::reqd_sub_group_size(16)]] {
         size_t glob_id = item.get_global_id(0);
         size_t group_id = item.get_group(0);
         size_t loc_id = item.get_local_id(0);
         if (loc_id==0)
            scratch[0]=0;
         sycl::vec<int, 4> val;
         val.load(glob_id,buf_acc);
         int sum=val[0]+val[1]+val[2]+val[3];
         item.barrier(sycl::access::fence_space::local_space);
         auto vl = sycl::ONEAPI::atomic_ref<int,
                         sycl::ONEAPI::memory_order::relaxed,
                         sycl::ONEAPI::memory_scope::work_group,
                         sycl::access::address_space::local_space>(
                         scratch[0]);
         vl.fetch_add(sum);
         item.barrier(sycl::access::fence_space::local_space);
         if (loc_id==0) {
               auto v = sycl::ONEAPI::atomic_ref<int,
                         sycl::ONEAPI::memory_order::relaxed,
                         sycl::ONEAPI::memory_scope::device,
                         sycl::access::address_space::global_space>(
                         sum_acc[0]);
               v.fetch_add(scratch[0]);
       }
      });
  });
}
```

リダクション・カーネルのパフォーマンス解析

これらのカーネルのパフォーマンスを評価するため、2 つの異なるインテル GPU で実行します。

1. インテル® HD グラフィックス 630（第 9 世代統合グラフィックス、略称 Gen9)。この GPU には 24 の実行ユニット（EU）が搭載されており、それぞれ 7 スレッドを利用できます。
2. インテル® Iris® Xᵉ グラフィックス（第 12 世代統合グラフィックス、略称 Gen12)。この GPU には 96 の実行ユニット（EU）が搭載されており、それぞれ 7 スレッドを利用できます。

インテル® VTune™ プロファイラーを使用してカーネルのパフォーマンスを解析します。また、カーネルの実行時間を長くして適切なプロファイル・データを収集できるように、大規模なリダクション（1,000 万要素の代わりに、5.12 億要素)を実行します。各カーネルのパフォーマンスを**表 1** に示します。これは、各カーネルを 16 回実行して、平均パフォーマンスを記録したものです。データの収集には、インテル® oneAPI ベース・ツールキット（v2021.2.0）を使用しました。

カーネル	インテル® HD グラフィックス 630	インテル® Iris® Xᵉ グラフィックス
reductionAtomics1	146	49
reductionAtomics2	258	141
reductionAtomics3	111	38
ツリー・リダクション	288	115
ビルトイン・リダクション演算子	429	162
multiblockinterleavedreduction	83	37
multiblockinterleavedreductionVector	67	34

表 1. 異なるリダクション実装のパフォーマンス（ミリ秒単位)

reductionAtomics1

このカーネルは、ハードウェアが実行可能なアトミック更新の数によって制限されます（**図 3**)。

Computing Task	Work Size		Computing Task					EU Array			EU Threads Occupancy	Computing Threads Started	L3 Bandwi... GB/sec	GPU Memory Bandw...		GPU Atomics
	Global	Local	Total Time ▼	Average Time	Instance ...	Sl...	S...	Active	Stalled	Idle				Read	Write	
reductionAtomics1(cl::sycl::queue&, cl::sycl::buffer<int, (int)1, cl::sycl::detail::aligned_allocator<	536870912	512	0.832s	0.049s	17	32		28.2%	71.7%	0.1%	92.2%	142,606,336	65.786	44.186	0.000	285212672

図 3. インテル® Iris® Xᵉ グラフィックス上での reductionAtomics1 の統計値

上の図で Global Work Size カラムは、カーネルの合計ワーク項目数を示します。これは、問題のサイズ（5.12 億要素）です。Instance カラムは、カーネルが呼び出された回数を示しており、ここでは 17 です。SIMD Width カラムは 32 で、これはこのカーネルに対してコンパイラーが選択したベクトルサイズです。Computing Threads Started カラムは、このカーネルで実際に実行された個別のスレッドの数です。これは、Global Work Size を SIMD Width で割って、Instance 数を掛けた値に等しいです。最後に、GPU Atomics カラムは、このカーネルで実行されたアトミック操作の数を示します。**`reductionAtomics1`** カーネルでは、各スレッドが 2 つのアトミック操作を発行するため、スレッド数の 2 倍になります（Gen12 の SIMD32 は 2 つの SIMD16 命令としてエンコードされます)。

インテル® VTune™ プロファイラーのアーキテクチャー・ダイアグラムは、Gen9 ではメモリー帯域幅を大幅に改善できることを示しています（ピークが 32GB/ 秒であるのに対して、**`reductionAtomics1`** は 15.3GB/ 秒しか達成できていません）（**図 4**）。Gen12 でこのカーネルは、より高いメモリー帯域幅を達成しています。これは、インテル® Iris® Xe グラフィックス（Gen12）のほうが、インテル® HD グラフィックス（Gen9）よりも多くのアトミックメモリー更新を処理できるためです（**図 5**）。

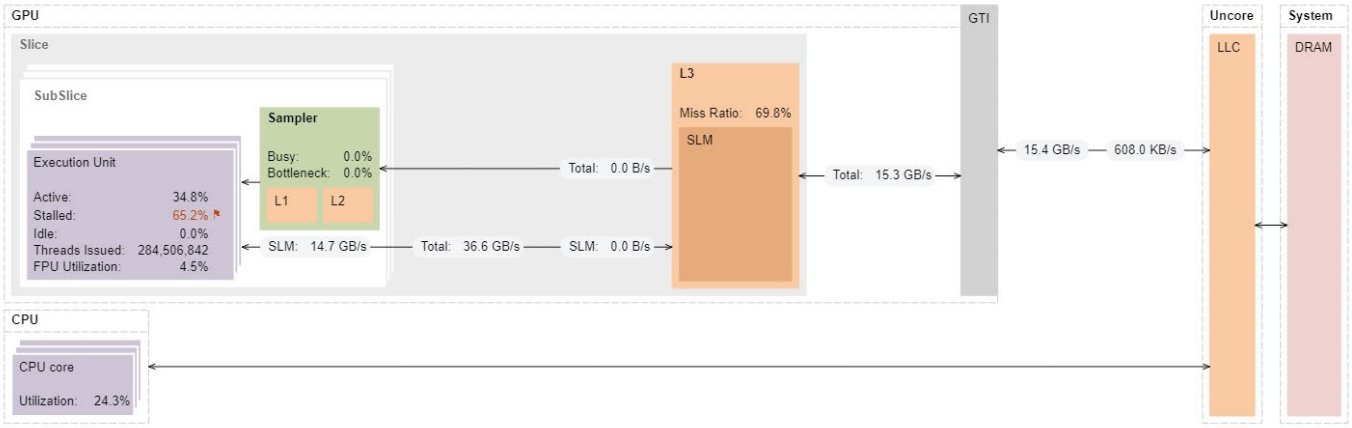

図 4. インテル® HD グラフィックス 630 上での reductionAtomics1 のアーキテクチャー・ダイアグラム

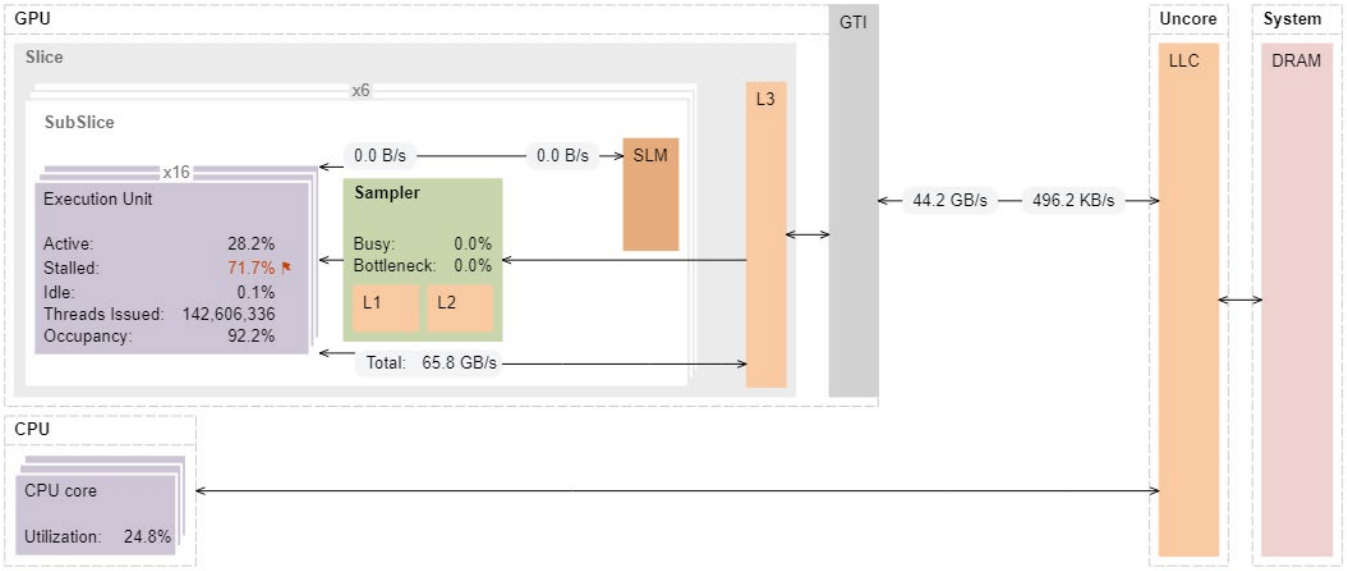

図 5. インテル® Iris® Xe グラフィックス上での reductionAtomics1 のアーキテクチャー・ダイアグラム

reductionAtomics2

このカーネルは、Gen9 と Gen12 の両方でパフォーマンスが低いことが分かります。このカーネルのメモリー・アクセス・パターンでは、コンパイラーが 16 の異なるキャッシュラインそれぞれの 1 要素にのみアクセスするロード命令を生成します。そのため、最初のアクセスではキャッシュを 16 回ミスするため長いレイテンシーが生じ、ほかの 15 の参照はすべてキャッシュにヒットします。キャッシュヒット率と L3 からの帯域幅は良好ですが、全体的なパフォーマンスは 16 回のキャッシュミスが発生する最初のメモリー参照のレイテンシーにより制限されます。**`reductionAtomics1`** と比較すると、キャッシュミス率が非常に低く、L3 メモリー帯域幅が高いものの、全体的なパフォーマンスはどちらのプラットフォームでも低いままです（**図 6**）。

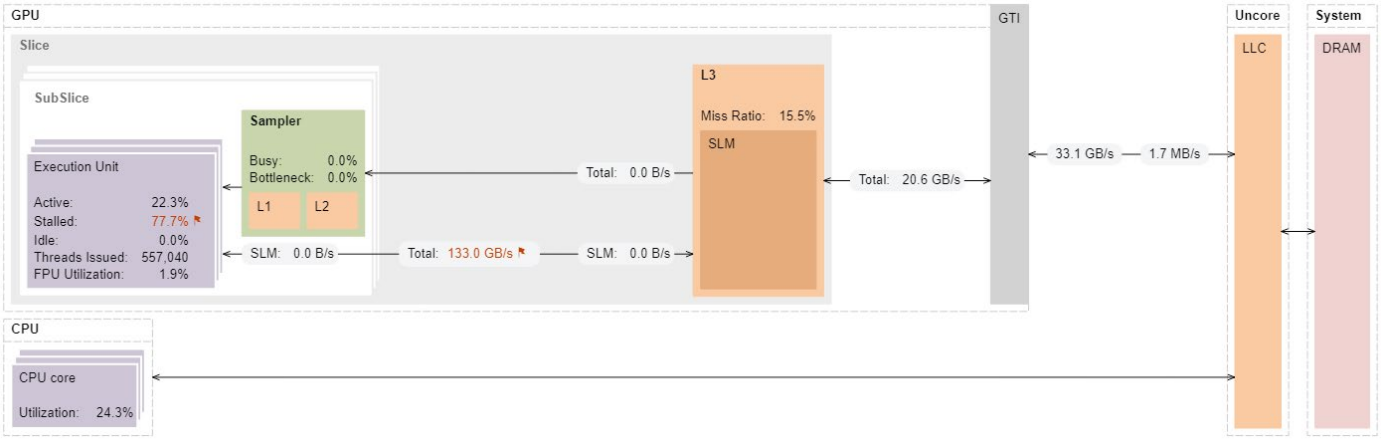

図 6. インテル® HD グラフィックス 630 上での reductionAtomics2 のアーキテクチャー・ダイアグラム

reductionAtomics3

このカーネルのメモリー・アクセス・パターンでは、ベクトルロード命令が 1 つのキャッシュラインのすべての要素を一度にロードします。これにより、キャッシュミス率は 100% になりますが、複数のスレッドが同時に処理できるため、レイテンシーの影響が緩和されます。これは、**`reductionAtomics1`** および **`reductionAtomics2`** と比較して、このカーネルはキャッシュミス率が高く（**`reductionAtomics2`** では 15.5%、**`reductionAtomics1`** では 69.8%）、L3 メモリー帯域幅が低いにもかかわらず、パフォーマンスが優れていることからも分かります（**表 1**）。

Gen12 上での **reductionAtomics3** のアーキテクチャー・ダイアグラムは、メインメモリーからの帯域幅が L3 帯域幅とほぼ同じであることを示しています（L3 帯域幅が 56.5GB/ 秒であるのに対して、メモリー帯域幅は 58.5GB/ 秒）（**図 7**）。

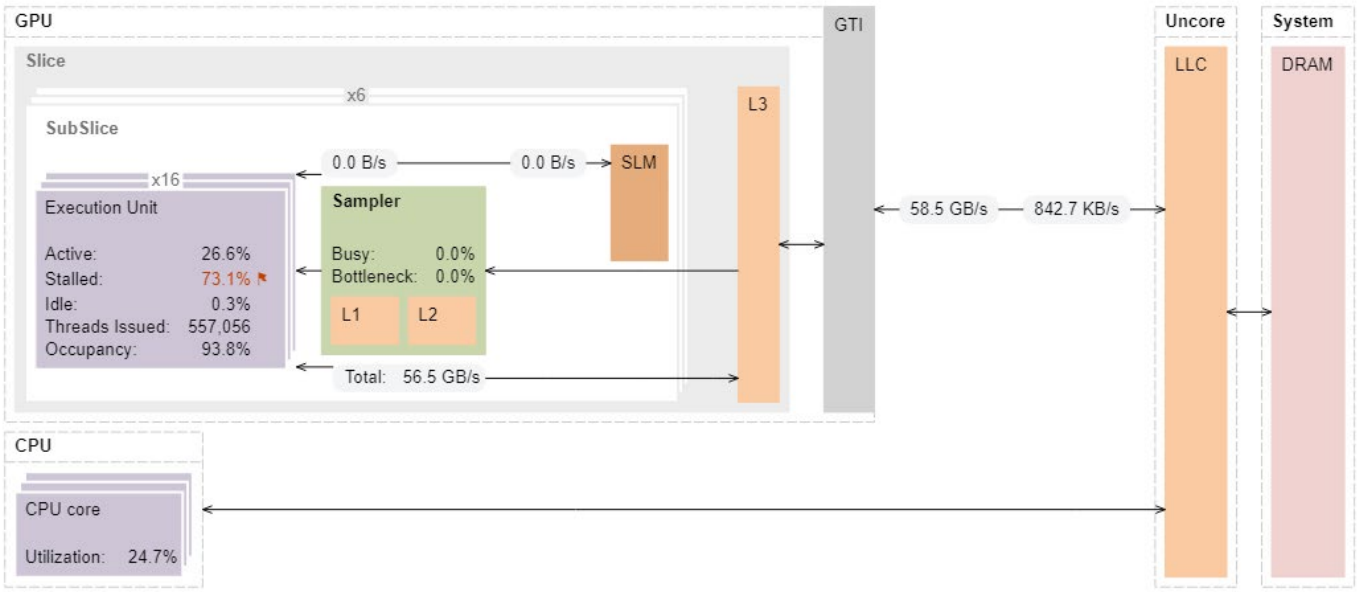

図 7. インテル® Iris® Xe グラフィックス上での reductionAtomic3 のアーキテクチャー・ダイアグラム

ツリー・リダクション

ツリー・リダクションは一般的な手法ですが、Gen9 と Gen12 のどちらでもパフォーマンスが良くありません。リダクション・ツリーの各レベルで半分の EU をアイドル状態にしてしまう、アルゴリズムの本質的な不均衡が効率を悪くしています（**図 8**）。

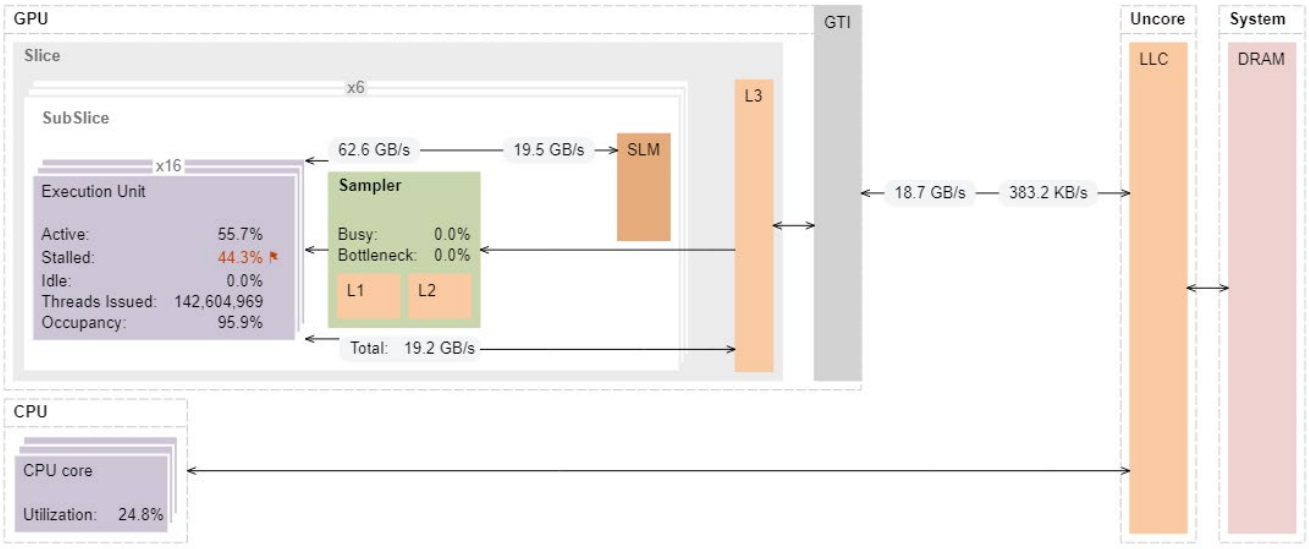

図 8. インテル® Iris® Xe グラフィックス上でのツリー・リダクションのアーキテクチャー・ダイアグラム

コンパイラーのビルトイン・リダクション

コンパイラーのビルトイン・リダクション演算子は現在開発中であり、この記事のほかの手法のパフォーマンスを達成するには追加のチューニングが必要です。インテル® VTune™ プロファイラーによって報告されたアトミックの数と開始された計算スレッドの数のメトリックを確認して、ツリー・リダクションと比較すると、ビルトイン演算子の実装には共有ローカルメモリー（SLM）上のツリー・リダクションが使用されていることが分かります。また、この実装では、リダクション演算子を適用する前に、メインメモリーから共有ローカルメモリーへデータをコピーしているように見えます。これは、インテル® VTune™ プロファイラーのプラットフォーム・ビューの GPU Shared Local Memory レーンのアクティビティーからも分かります（**図 9**）。

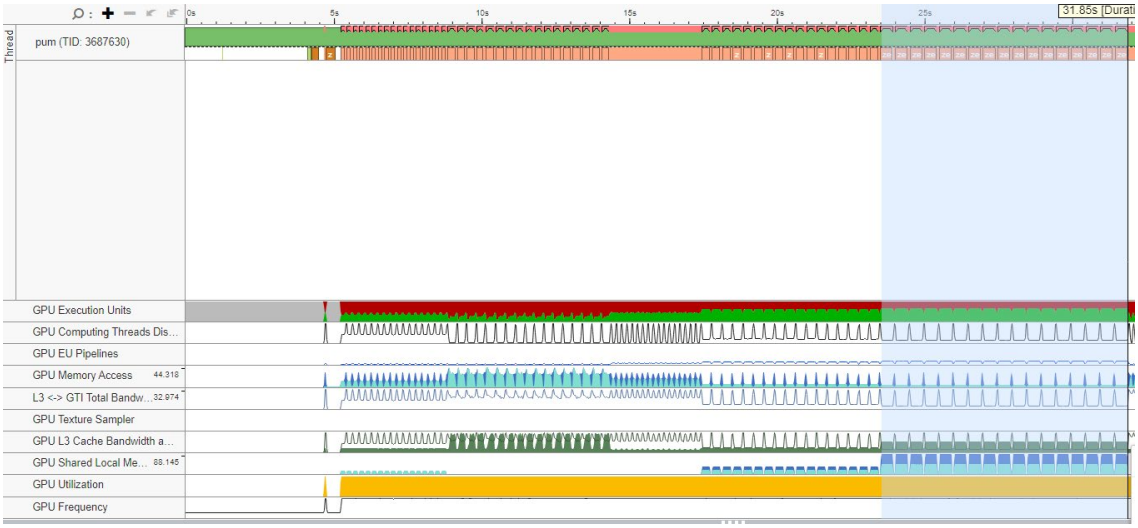

図 9. インテル® HD グラフィックス 630 上でのコンパイラーのビルトイン・リダクション演算子のプラットフォーム・ビュー

ツリー・リダクションとコンパイラーのビルトイン・リダクション演算子のアーキテクチャー・ダイアグラムを比較すると、後者の SLM の使用のほうが高くなっています（読み取りが 116GB/ 秒と 62.6GB/ 秒、書き込みが 59.7GB/ 秒と 19.5GB/ 秒）（**図 8** と **10**）。これは、各スレッドがリダクション操作の前に使用するデータをコピーするためです。ツリー・リダクション実装では、SLM へのデータのコピーがありません。SLM は、各ワークグループが生成する中間値にのみ使用されます。したがって、同じアルゴリズムを使用していても、ツリー・リダクション実装のほうがビルトイン・リダクション演算子よりもパフォーマンスが優れています。

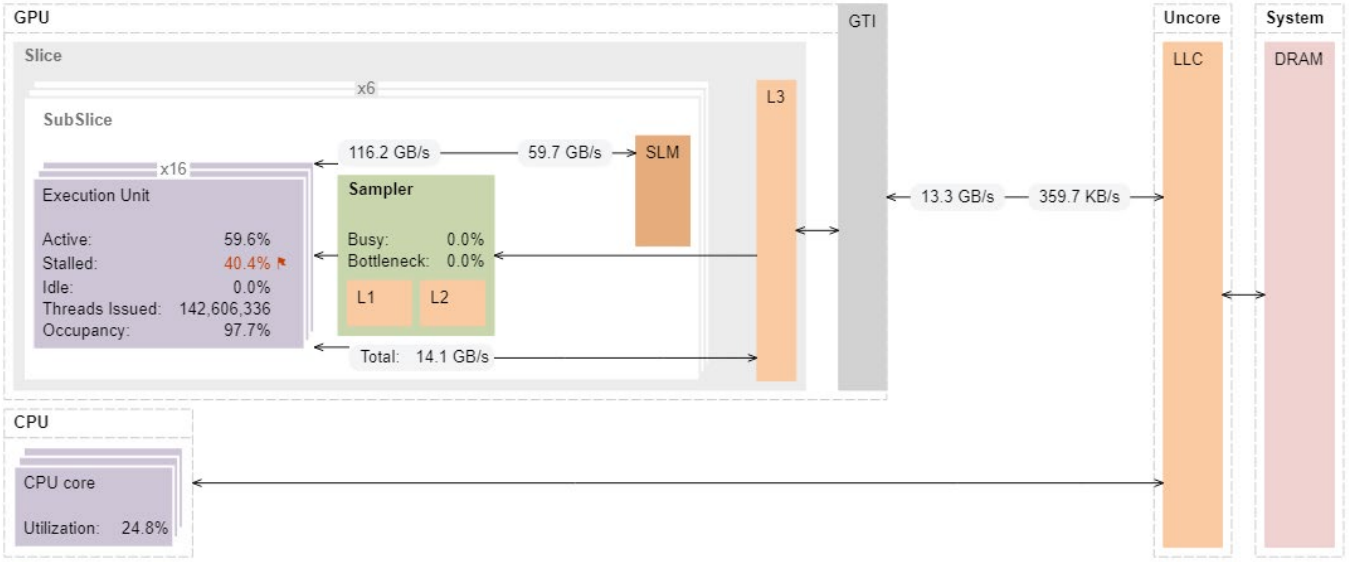

図 10. インテル® Iris® Xᵉ グラフィックス上でのコンパイラーのビルトイン・リダクション演算子のアーキテクチャー・ダイアグラム

リダクション・アルゴリズムのパフォーマンスは、アーキテクチャーによって大きく異なることを覚えておいてください。ビルトイン・リダクション演算子のパフォーマンスは、将来の oneAPI コンパイラーで向上する予定です。

MultiBlockInterleaved

このカーネルのメモリー・アクセス・パターンは、コンパイラーがスレッドごとに 128 要素をロードするブロックロード操作を生成できるように注意深く設計されており、ほかのカーネルよりも高い帯域幅を達成できます。

このカーネルのサブグループのロード操作は、コンパイラーによって 4 つの SIMD16 ロード命令に変換され、それぞれ 4 つのレジスターに要素をロードします（**図 11**）。

```
(W) send.dc0 (16|M0) r44 r60 null 0x0 0x02484400 {@5, $0} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
(W) send.dc0 (16|M0) r48 r61 null 0x0 0x02484400 {@4, $1} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
(W) send.dc0 (16|M0) r56 r63 null 0x0 0x02484400 {@2, $2} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
(W) send.dc0 (16|M0) r52 r62 null 0x0 0x02484400 {@1, $3} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
```

図 11. カーネルの sg.load 操作に対してコンパイラーが生成したアセンブリー・コード

MultiBlockInterleavedVector

最後の **`MultiBlockInterleavedVector`** カーネルは DPC++ のビルトインベクトルを使用して、SIMD32 命令とブロック読み取りにより **`MultiBlockInterleaved`** カーネルよりも優れたパフォーマンスを実現します。**`MultiBlockInterleavedVector`** カーネルは、Gen9 では 32.5GB/ 秒（**図 12**）、Gen12 では 62GB/ 秒（**図 13**）のピークメモリー帯域幅を達成します。

図 12. インテル® HD グラフィックス 630 上での MultiBlockInterleavedVector のアーキテクチャー・ダイアグラム

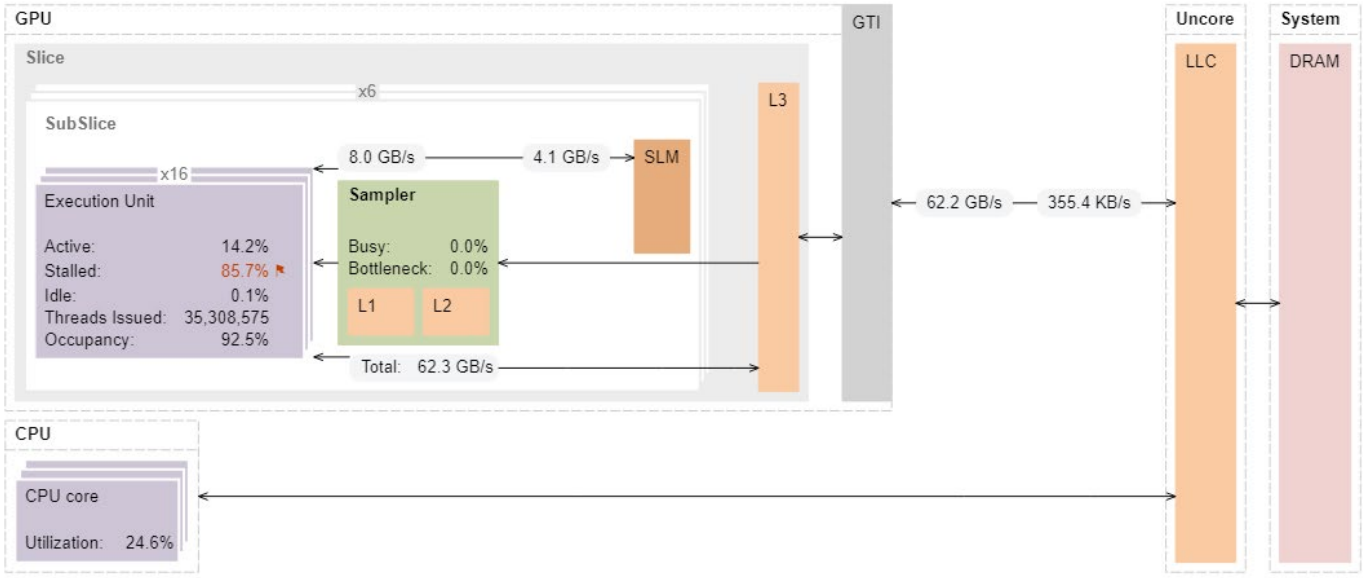

図 13. インテル® Iris® Xe グラフィックス上での MultiBlockInterleavedVector のアーキテクチャー・ダイアグラム

コンパイラーによって生成されるアセンブリー・コードは、スレッドごとに 128 要素をロードします。以下に示すように、SIMD32 ロード命令は 128 要素を 16 のレジスターにロードします（**図 14**）。そのため、このプラットフォームでは、メモリーから GPU へのデータのフェッチでピーク帯域幅を達成します。

```
send.dc1 (16|M0) r25 r21 null 0x0 0x04805000 {@2, $1} [, msg-length:2, resp-length:8, header:no, func-control:5000]
send.dc1 (16|M16) r33 r23 null 0x0 0x04805000 {@1, $2} [, msg-length:2, resp-length:8, header:no, func-control:5000]
```

図 14. インテル® Iris® Xe グラフィックス上で 128 要素をロードする SIMD32 ベクトルロード命令に対して コンパイラーが生成したアセンブリー・コード

まとめ

リダクションは並列プログラミングにおいて重要な操作であり、多くのアプリケーションで使用されています。この全 2 パートのシリーズでは、DPC++ でリダクション操作をコーディングするさまざまな方法を示し、2 つのインテルの統合 GPU でパフォーマンスを評価しました。この記事で使用したカーネルのソースコードは、**https://github.com/rvperi/DPCPP-Reduction**（英語）から利用できます。

関連情報

1. **データ並列 C++ リソースページ**（英語 ）
2. **データ並列 C++ サンプルコード**

OpenMP* のアクセラレーター・オフロード

ヘテロジニアス・アーキテクチャー全体にわたる移植性

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 インテル コーポレーション パフォーマンス・モデリング・エンジニア

OpenMP* 標準は、バージョン 4.0 からアクセラレーター・オフロードをサポートしています。これらのプラグマは、エンドユーザーがデータと計算を GPU などのデバイスへオフロードできるようにします。これにより、移植性に優れたヘテロジニアス並列コードを容易に記述できます。この記事では、いくつかの OpenMP* オフロードプラグマと、その使用方法をサンプルコードを用いて説明します。また、OpenACC* から OpenMP* への移行例も紹介します。

OpenACC* から OpenMP* への移行

OpenACC* は、NVIDIA GPU 向けのプラグマベースのプログラミング手法ですが、ほかのベンダーによりサポートされていないため、1 つのプラットフォームに限定されています。一方、OpenMP* オフロードは、oneAPI フレームワーク、NVIDIA の HPC SDK、AMD の ROCm* スタック、IBM の XL コンパイラー・スイートなど、業界で幅広く

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。

サポートされています。OpenACC* プラグマは OpenMP* へほぼ 1 対 1 でマッピングできます(**表 1**)。そのため、通常は既存の OpenACC* コードから OpenMP* へ簡単に移行できます。**表 1** は、一般的な OpenACC* プラグマと同等の OpenMP* プラグマです。

OpenACC* プラグマ	OpenMP* プラグマ	用途
`#pragma acc parallel`	`#pragma omp target teams`	GPU 上でスレッドのチームを作成して、各チームのマスタースレッドが領域を実行します。
`#pragma acc parallel loop gang worker vector collapse(2)`	`#pragma omp target teams distribute parallel for simd collapse(2)`	GPU ハードウェア・スレッドに作業を分散して、計算を並列に実行します。
`#pragma acc kernels loop reduction(+:norm)`	`#pragma omp parallel for reduction(+:norm)`	リダクションを並列に実行します。
`#pragma acc data copy(A[0:Sz])`	`#pragma omp target data map(tofrom: A[0:Sz])`	ターゲットデバイスとの間でデータをコピーします。
`#pragma acc update host(A[0:Sz])`	`#pragma omp target update from(A[0:Sz])`	デバイスからホスト上のデータを更新します。
`#pragma acc data copyin(A[0:Sz])`	`#pragma omp target data map(alloc:A[0:Sz])`	デバイス上にメモリーを割り当てます。
`#pragma acc update device(X)`	`#pragma omp target update to(X[0:Sz])`	ホストからデバイス上のデータを更新します。
`#pragma acc loop vector`	`#pragma omp simd`	ベクトル化します。
`#pragma acc atomic update`	`#pragma omp atomic update`	メモリー位置をアトミックに更新します。

表 1. 一般的な OpenACC* プラグマと同等の OpenMP* プラグマ

図 1a と **1b** は、OpenACC* から OpenMP* へ移行されたコードスニペットです。これは、電波天文パッケージ **tConvolveACC**（英語）のカーネルです。OpenACC* プラグマ **`#pragma acc parallel loop`** は、OpenMP* オフロードプラグマ **`#pragma omp target parallel for`** とターゲットデバイスとの間の明示的なデータ転送プラグマに置き換えられます。OpenACC* 実装では、暗黙のコピーや統合共有メモリー割り当てを使用してデータ転送を管理している可能性があります。

```
degridKernelACC(...)
{
  ...
  #pragma acc parallel loop
  for (dind = 0; dind < d_size; ++dind) {
    ...
  }
}
...
gridKernelACC(...)
{
  ...
  #pragma acc parallel loop
  ...
        #pragma acc atomic update
        gptr_re[0] = gptr_re[0] + cval.real();
  ...
}
```

図 1a. OpenACC* で記述された tConvolveACC 実装からのサンプルカーネル

```
degridKernelOmpOffload(...)
{
  ...
  #pragma omp target parallel for                  \
          map(tofrom:d_data[0:d_size])             \
          map(to:d_grid[:grid.size()])             \
          map(to:d_C[:C.size()])                   ...
  for (dind = 0; dind < d_size; ++dind) {
    ...
  }
}
...
gridKernelOmpOffload(...)
{
  ...
  #pragma omp target teams distribute parallel for     \
          map(tofrom:d_grid[:grid.size()])             ...
  ...
        #pragma omp atomic update
        gptr_re[0] = gptr_re[0] + cval.real();
  ...
}
```

図 1b. OpenMP* で記述された tConvolveACC 実装からのサンプルカーネル

インテル® プラットフォーム上での OpenMP* オフロード

オフロードコードをビルドして実行するために必要な手順を見てみましょう。**インテル® oneAPI ベース・ツールキット** 2021.2.0 で以下のコンパイラー・オプションを使用して、OpenMP* オフロードコードをテストします。

```
-fiopenmp -fopenmp-targets=spir64="-mllvm \
-vpo-paropt-enable-64bit-opencl-atomics=true \
-fp-model=precise"
```

-fiopenmp と **-fopenmp-targets=spir64** の 2 つの新しいオプションは、GPU 向けのファットバイナリーを生成するようにコンパイラーに指示します。**-vpo-paropt-enable-64bit-opencl-atomics=true** コンパイラー・オプションは、アトミック操作とリダクション操作を有効にします。詳細は、**オンライン・ドキュメント**を参照してください。

GPU 上で OpenMP* オフロードコードを実行するため、ユーザーは **OMP_TARGET_OFFLOAD** 環境変数を設定する必要があります（GPU が利用できない場合、ランタイムエラーになります）。また、ユーザーは**レベルゼロ**（英語）または OpenCL* バックエンドを選択できます。

```
export OMP_TARGET_OFFLOAD = MANDATORY
export LIBOMPTARGET_PLUGIN = {LEVEL0|OPENCL}
```

LIBOMPTARGET_DEBUG（英語）環境変数を 1 以上に設定すると、GPU オフロードのデバッグ情報を取得できます。**図 2a** では、**tConvolveACC** の OpenMP* オフロードカーネルをレベルゼロプラグインで実行した場合のデバッグ情報をハイライト表示しています。2 つのオフロード領域は、**Benchmark** クラスの **gridKernelACC** 関数と **degridKernelACC** 関数にあります。**図 2b** は、**map** 節でターゲットデバイスへ転送される変数を示します。**図 2c** は、ホストからターゲットデバイスへ転送されるデータを示します。計算に必要なすべてのデータがデバイスで利用可能になると、**図 2a** の下部に示すように、カーネルが実行されます。

```
Target LEVEL0 RTL -->   0:
__omp_offloading_35_198bc1a__ZN9Benchmark13gridKernelACCERKSt6vector
Target LEVEL0 RTL -->   1:
__omp_offloading_35_198bc1a__ZN9Benchmark15degridKernelACCERKSt6vector

...

Libomptarget --> Launching target execution
__omp_offloading_35_198bc1a__ZN9Benchmark13gridKernelACCERKSt6vector with pointer
0x0000000000b72070 (index=0)
```

図 2a. tConvolveACC の OpenMP* オフロードカーネルのクラスと関数を赤色でハイライト表示

```
Libomptarget --> Entry  0: Base=0x00007f468ce16010, Begin=0x00007f468ce16010, Size=15310080,
Type=0x21, Name=d_iw[:this->wPlane.size()]

Libomptarget --> Creating new map entry with HstPtrBegin=0x00007f468ce16010,
TgtPtrBegin=0xffffd556aaa00000, Size=15310080, Name=d_iw[:this->wPlane.size()]
```

図 2b. tConvolveACC の OpenMP* オフロードカーネルの変数を青色でハイライト表示

```
Libomptarget --> Moving 15310080 bytes (hst:0x00007f468ce16010) -> (tgt:0xffffd556aaa00000)
Target LEVEL0 RTL --> Copied 15310080 bytes (hst:0x00007f468ce16010) -> (tgt:0xffffd556aaa00000)
```

図 2c. tConvolveACC の OpenMP* オフロードカーネルのデータ転送を緑色でハイライト表示

OpenMP* スレッドをターゲットデバイスへマップ

実行時に、OpenMP* スレッド階層がターゲットデバイスへマップされます。**`#pragma omp teams`** 構文はチームのリーグを作成して、各チームの最初のスレッドが領域を実行します。**`#pragma omp distribute`** 構文は、チームの最初のスレッドにワークを分配して、各チームはサブスライス（インテル® GPU）にスケジュールされます。各チーム内では **`parallel for`** 節によりワークがさらに並列化され、チームのスレッドが実行ユニット(EU)スレッドにマップされます。最後に、**`#pragma omp simd`** 構文は、EU ベクトルレーンを使用してベクトル化されたコードを実行します。チーム内のスレッドは、ワークシェア構文の最後で同期されます。**図 3** は、1 スライス、3 サブスライス、サブスライスごとに 8 EU、EU ごとに 7 スレッド、EU ごとに SIMD ベクトル処理ユニットがある第 9 世代インテル® プロセッサー・グラフィックス（Gen9）の図です。Gen9 の対応するユニットへの OpenMP* オフロードプラグマのマップも示しています。

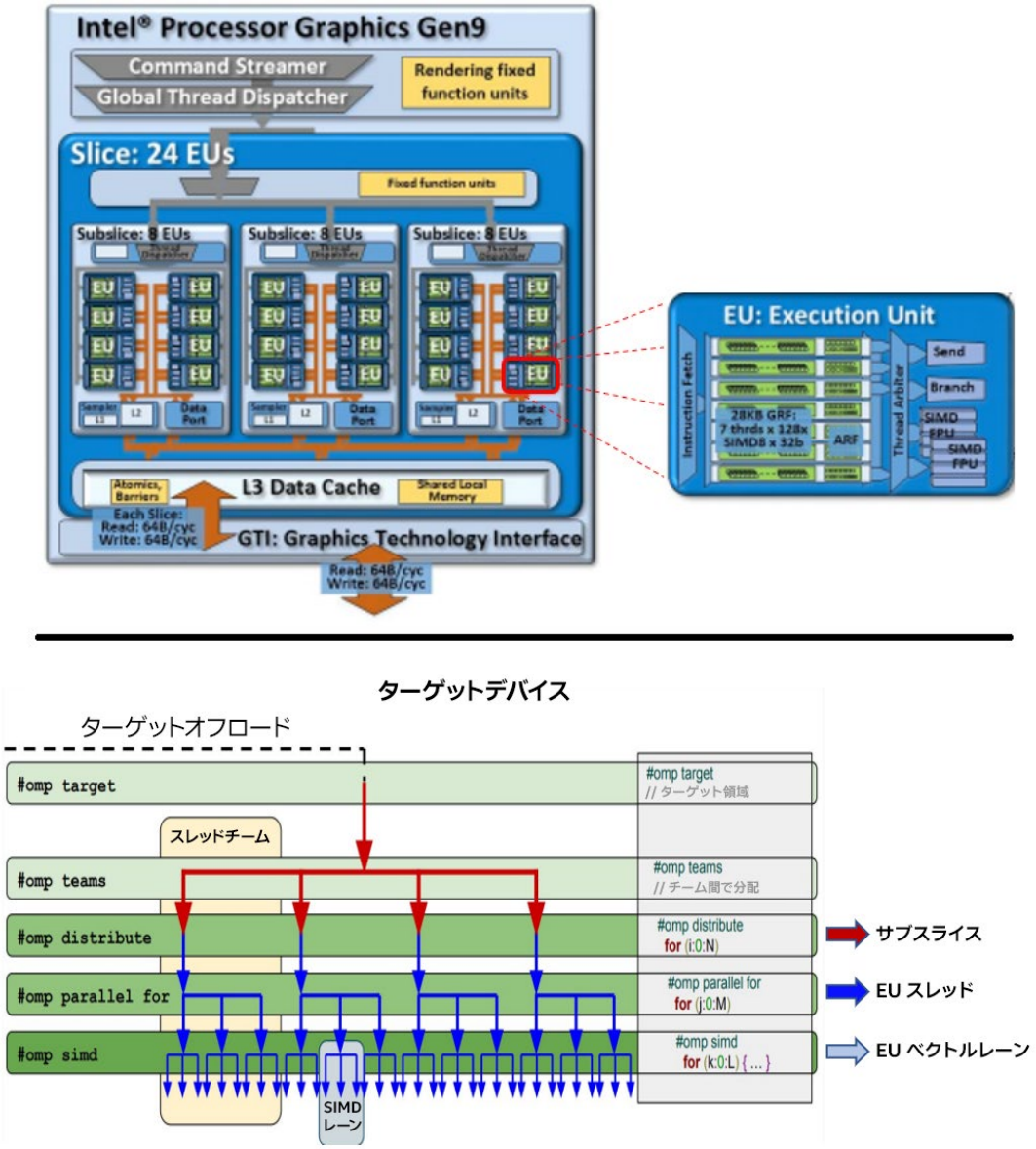

図 3. 第 9 世代インテル® プロセッサー・グラフィックス上のハードウェア機能への OpenMP* オフロードのマップ（「OpenMP* オフロードの確認と検証：ワークフローと 5.0 への道」（英語）より引用）

ターゲットデバイスとの間で効率良くデータ転送を行う OpenMP* プラグマ

アプリケーションをビルドしていくつかのカーネルをターゲットにオフロードできたら、次のステップはデータ転送などの最適化の可能性を調査することです。OpenMP* には、ホストとターゲット間で効率良いデータ転送を実装するプラグマが用意されています。以下に示す **tHogbomCleanACC** の例では、**HogbomClean** 関数に 2 つのオフロードターゲットがあります。単純な OpenMP* オフロードでは、ターゲットの起動時にデータ転送が行われます。**図 4a** のコードスニペットに示すように、これを **g_niters** のループで繰り返す場合、問題はさらに悪化します。

```
HogbomClean(...)
{
...for (unsigned int i = 0; i < g_niters; ++i)
    {
      findPeakOffload(resdata, absPeakVal, absPeakPos, ressize);...
      subtractPSFOffload(psfdata, psfWidth, psfsize, resdata, ...);...
    }...
}

subtractPSFOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}

findPeakOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}
```

図 4a. 2 つの OpenMP* オフロードカーネルの単純な実装では不要なデータ転送が発生

図 4b は、より効率良くデータ転送を行うように最適化された **HogbomClean** 関数の実装です。**#pragma omp target data map** 構文は、ターゲット上でデータが保持されるスコープを定義します。このスコープ内のカーネルオフロードは、(ハンドルを使用して) データを再利用できます。後続のオフロードカーネルへの **map** 呼び出しでは、データ転送は不要です (ただし、明示的に転送が必要とされているものは除く)。

oneAPI
多様なワークロードには多様なアーキテクチャーが必要
インテル® oneAPI ツールキットを使用して、ヘテロジニアス・アプリケーションを素早く正確に開発
ツールキットの詳細 >

```
HogbomClean(...)
{
  ...
  #pragma omp target data map(tofrom: resdata...
  {
    for (unsigned int i = 0; i < g_niters; ++i)
    {
      findPeakOffload(resdata, absPeakVal, absPeakPos, ressize);...
      subtractPSFOffload(psfdata, psfWidth, psfsize, resdata, ...);...
    }...
  }...
}

subtractPSFOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}

findPeakOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}
```

図 4b. 一度コピーしてデータ転送を再利用する効率良い OpenMP* の例

関数バリアントのディスパッチのサポート強化

OpenMP* オフロード仕様は、基本関数の代わりに条件付きで呼び出し可能な関数バリアントをサポートしています。インテル固有の OpenMP* オフロード関数バリアント API の実装は、**`#pragma omp target variant dispatch`** でサポートされます。このプラグマは、関数呼び出しの周囲に条件付きのディスパッチ・コードを発行するようにコンパイラーに指示します。ターゲットデバイスが利用可能な場合、基本関数の代わりに関数バリアントが呼び出されます。**図 5a**、**5b**、および **5c** は、**`target variant dispatch`** の例です。関数バリアントは基本関数と同じ引数に加えて、**`void *`** 型の最後の引数がなければなりません。

```
findPeakOffload(..., void *p)
{
  ...
  #pragma omp target teams distribute parallel for
          reduction(max:threadAbsMaxVal) map(to:data[0:size])
  for (size_t i = 0; i < size; ++i) {
    if ( abs(data[i]) > threadAbsMaxVal)
      threadAbsMaxVal = abs(data[i]);
  }

  #pragma omp target teams distribute parallel for
          map(to:data[0:size]) map(from:tmpPos)
  for (size_t i = 0; i < size; ++i) {
    if (abs(data[i]) == threadAbsMaxVal)
      tmpPos = i;
  }
  maxVal = data[tmpPos];
  maxPos = tmpPos;
}
```

図 5a. ターゲットデバイスで実行する関数バリアント findPeakOffload

```
#pragma omp declare variant(findPeakOffload)
        match(construct={target variant dispatch})
findPeakBase(...)
{
  #pragma omp parallel
  {
    ...
    #pragma omp for schedule(static)
    for (size_t i = 0; i < size; ++i) {
      if (abs(image[i]) > abs(threadAbsMaxVal)) {
        threadAbsMaxVal = image[i];
        threadAbsMaxPos = i;
      }
    }

    #pragma omp critical
    if (abs(threadAbsMaxVal) > abs(maxVal)) {
      maxVal = threadAbsMaxVal;
      maxPos = threadAbsMaxPos;
    }
  }
}
```

図 5b. ホストで実行する基本関数 findPeak

```
HogbomClean(...)
{
  ...
  #pragma omp target data map(to: psfdata[0:psfsize])
                          map(tofrom: resdata[0:ressize])
  {
    for (unsigned int i = 0; i < g_niters; ++i)
    {
      #pragma omp target
      findPeakBase(resdata, absPeakVal, absPeakPos, ressize);
      ...
    }
  }
}
```

図 5c. 呼び出しはホストバージョンでなければならない—ターゲットデバイスが利用可能な場合は オフロードターゲット関数が実行され、そうでない場合はホストバージョンが実行される

まとめ

OpenMP* 標準のプラットフォームやベンダーに依存しないデバイス・オフロード・サポートにより、ユーザーは同じコードベースで複数のヘテロジニアス・アーキテクチャーに容易に対応できます。そのため、ユーザーやハードウェア / ソフトウェア・ベンダーの間で、OpenMP* のヘテロジニアスな並列処理の採用が進むと予想されます。

関連情報

1. **https://techdecoded.intel.io/resources/using-openmp-accelerator-offload-for-programming-heterogeneous-architectures/**（英語）

インテル® oneAPI ツールキットを使用した分散型 AI トレーニングの最適化

インクリメンタルなチューニングによりパフォーマンスを大幅に向上

Abhay Rawat 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
Amarpal S Kapoor 博士 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

ディープラーニング（DL）ワークロードは急速に増加しています。DL ベースのアルゴリズムは、膨大な量のデータを処理してパターンを見つけ、画像分類、物体検出、時系列予測などを行います。データの可用性が高まるにつれ、DL モデルの複雑さも増しています。**ResNet**（英語）や **VGG**（英語）などのモデルでは、数百万個のパラメーターがあり、数十億回の浮動小数点演算を行います。**GPT-3**（英語）や **BERT**（英語）などの最近のモデルでは、数十億から 1 兆個のパラメーターがあります。そのため、DL モデルのトレーニングは計算量が多く、時間のかかる処理です。トレーニング時間を短縮するには、シングルコアやマルチコアのパフォーマンスを最適化するだけでなく、複数のノードへのスケールアウト（分散トレーニング）も検討する必要があります。これには、モデルを分割する方法（モデル並列処理）、データを分割する方法（データ並列処理）、または両方を組み合わせた方法（ハイブリッド並列処理）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。

この記事では、クラスターの計算ノードでの DL ベースのアルゴリズムのチューニングとスケーリングに注目します。半教師あり敵対的生成ネットワーク（S-GAN）を使用して画像を分類し、OpenMP*、TensorFlow*、およびインテル® MPI ライブラリーでさまざまなチューニング・オプションを使用して最適化します。最適化により、単一ノードでは 2 倍のスピードアップを達成し、8 ノードのクラスターでは全体で 16 倍のスピードアップを達成します。インテル® MPI ライブラリーの画像スループット（1 秒あたりの画像数）は、Open MPI ライブラリーよりも、単一ノードでは最大 27%、8 ノードのクラスターでは最大 18% 向上します。パフォーマンスの向上の多くは、コードを変更することなく達成されており、これらの最適化がほかのアプリケーションにも効果的に適用できることを示しています。

半教師あり敵対的生成ネットワーク（S-GAN）

教師あり学習には大量のラベル付きデータが必要です。ラベル付けとアノテーションは専門家が手動で行う必要があるため、手間とコストがかかります。半教師あり学習とは、ラベル付きのデータとラベルなしのデータの両方を使ってモデルを学習する手法です。通常、ラベル付けされたデータポイントの数は、ラベルなしのデータポイントよりもかなり少ないです。半教師あり学習は、データのパターンや傾向を利用して分類します。

S-GAN は、生成モデルを用いてデータポイントを生成することで、膨大な量の学習データを用意します。敵対的生成ネットワーク（GAN）は、大規模なラベルなしデータセットで、画像識別器を利用して画像生成モデルをトレーニングするアーキテクチャーです。GAN は、生成モデルと識別モデルで構成されます。2 つのモデルはゼロサムゲームで一緒にトレーニングされます。生成器は、データセットに存在するデータに似たデータを生成します。識別器は、生成されたデータの中から実際のデータを識別します。S-GAN は、分類タスクに教師あり識別器（分類器）を追加することで GAN アーキテクチャーを拡張します（**図 1**）。これにより、初見のデータに対しても一般化できる分類器になります。

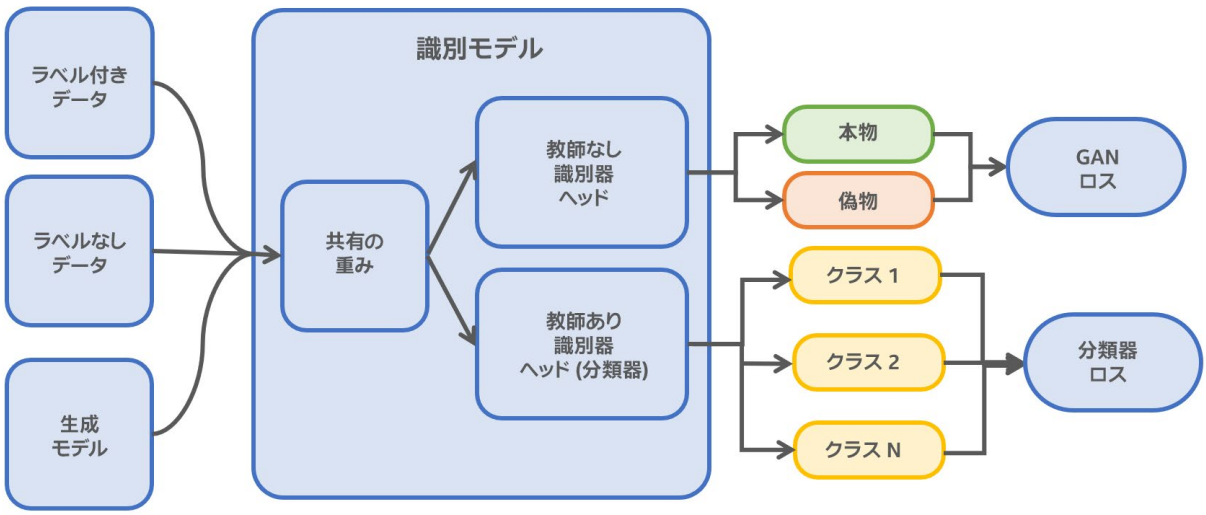

図 1. S-GAN アーキテクチャー

ソフトウェアとハードウェア

実験では、3 つの**インテル® oneAPI ツールキット**（バージョン 2021.1）を使用しました。

1. インテル® oneAPI ベース・ツールキット
2. インテル® oneAPI HPC ツールキット
3. インテル® oneAPI AI アナリティクス・ツールキット

Python* の基本数値パッケージを高速化するため、インテル® oneAPI マス・カーネル・ライブラリーとインテル® oneAPI データ・アナリティクス・ライブラリーを利用する**インテル® ディストリビューション for Python***（バージョン 3.6）を使用しました。S-GAN モデルは、**インテル® Optimization for TensorFlow***（英語）（バージョン 1.15）を使用して実装しました。分散トレーニングには Horovod*（バージョン 0.20.2）を使用しました。Horovod* はノード間通信に MPI を使用しているため、インテル® MPI ライブラリーと Open MPI（バージョン 4.0.5）のパフォーマンスを比較しました。パフォーマンスの解析には、**インテル® VTune™ プロファイラー**を使用しました。すべてのテストは、100Gbps で動作する**インテル® Omni-Path ファブリック**（英語）で接続された**インテル® Xeon® Platinum 8260L** プロセッサー・ベースの **Endeavor**（英語）クラスターで実行しました（**図 2**）。

```
プロセッサー名          : インテル® Xeon® Platinum 8260L プロセッサー
パッケージ (ソケット)   : 2
コア                    : 48
プロセッサー (CPU)      : 96
パッケージごとのコア    : 24
コアごとのスレッド      : 2
```

図 2. ノード構成（$ cpuinfo -g）

チューニング手法

アプリケーションを最適化する前にベースライン・パフォーマンスを測定します。プロファイル・ツールは、スレッド、ベクトル化、I/O、マルチノード通信など、最適化の候補を特定するのに役立ちます。ここでは、インテル® VTune ™ プロファイラーの**アプリケーション・パフォーマンス・スナップショット（APS）**（英語）を使用します。APS プロファイルは、アプリケーションがスポーンしたスレッドが多すぎることを示しました。OpenMP* スレッドもあれば、TensorFlow* から呼び出される Eigen ライブラリーのスレッドもあります。OpenMP* スレッドと Eigen スレッドの数が、ノードあたりの論理コア数を超えています。通常、これはパフォーマンスの低下を招きます。

最適なスレッド数の選択

最初に、最適なスレッド数を特定します。単一の計算ノードで OMP_NUM_THREADS 環境変数を使用して、時間を短縮するため低解像度（256 x 256 ピクセル）の画像でエポック数を 2 に制限して、さまざまな OpenMP* スレッド数をテストした結果、50 スレッドで 19.67 画像 / 秒という最高のパフォーマンスを達成できることが分かりました（**図 3**）。

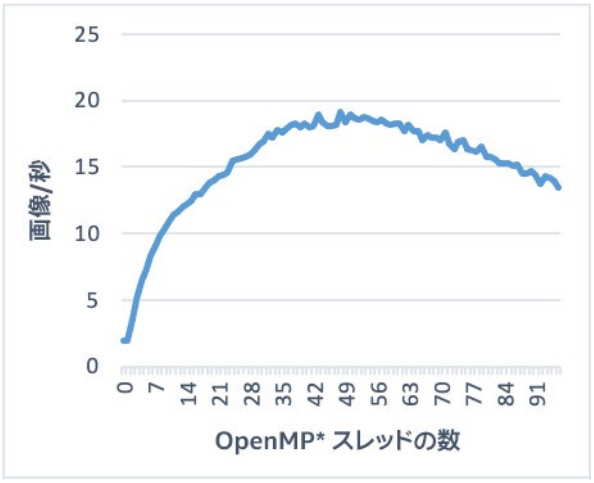

図 3. 最適な OpenMP* スレッド数の特定

次に、TensorFlow* のスレッド API を使用してスレッド数を制御します。**`inter_op_parallelism_threads`** 値は、独立した非ブロッキング操作によって使用されるスレッド数を指定し、**`intra_op_parallelism_threads`** 値は行列乗算やリダクションなどの個々の操作に使用されるスレッド数を指定します。**図 4** は、この 2 つのパラメーターにさまざまな値を設定した場合のパフォーマンスです。濃い緑色のブロックはパフォーマンスが良好であることを示し、濃い赤色のブロックはパフォーマンスが悪いことを示します。白色のブロックは実行に失敗したことを示します。

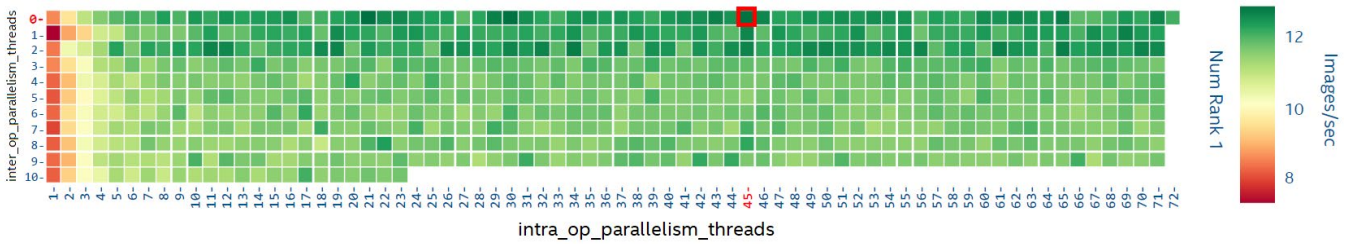

図 4. 最適な inter/intra_op_parallelism_threads 値の特定

ノードあたりの MPI ランクが 1 つの場合、この 2 つのパラメーターの最適な値はそれぞれ 0 と 45 で、12.34 画像 / 秒を達成します。これは、**`OMP_NUM_THREADS`** を使用した場合の 19.67 画像 / 秒（**図 3**）よりも低いため、ここでは TensorFlow* スレッド API を使用する代わりに、この環境変数でスレッド数を制御します。

最適なノードあたりの MPI ランク数の選択

アプリケーションは、MPI/OpenMP* ハイブリッド並列処理を使用しているため、最適な OpenMP* スレッド数とノードあたりの MPI ランク数の組み合わせを特定することが重要です。そこで、異なるノードあたりの MPI ランク数（$2^1 \le$ **`ppn`** $\le 2^4$、**`ppn`** はノードあたりのランク数) で**図 3** のテストを繰り返し実施したところ (1 ≤ **`OMP_NUM_THREADS`** ≤ 96)（**図 5**)、ノードあたり 8 MPI ランクと 9 OpenMP* スレッドで最高のパフォーマンス（26.3 画像 / 秒）が得られました。

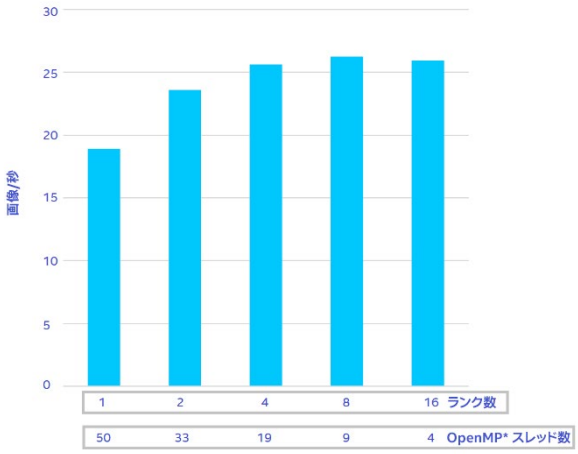

図 5. 最適な OpenMP* スレッド数とノードあたりの MPI ランク数の組み合わせの特定

メモリーリークの解決

これまでの実験では、低解像度の画像を使用しました。高解像度の画像（1360 x 1360 ピクセル）では、各ワーカープロセスのメモリー消費量が大幅に増加するため、ノードあたり 8 MPI ランクを実行するとメモリー不足になります。1 ランクのメモリー消費量は 129GB になり、各ノードには約 200GB の DRAM しかないため、ノードごとに 1 つのランクしか実行できません。これは、NUMA の問題から、デュアルソケット・マシンでは最適ではありません。アプリケーションを Python* 向けの**メモリー・プロファイラー**（英語）ユーティリティーで実行したところ、メモリーリークが見つかりました。モデルのフォワードパスで約 56GB のメモリーが解放されていませんでした（**図 6** の上）。これは、TensorFlow* の Keras **`Model.predict`** メソッドの既知の問題です（TensorFlow* **問題番号：33009**（英語）、Keras **問題番号：13118**（英語））。TensorFlow* コミュニティーの推奨に従って、**`Model.predict`** を **`Model.predict_on_batch`** に置き換えました。これにより、プロセスごとのメモリー消費量が減りました（**図 6** の下）。メモリーリークを解決しても、高解像度の画像ではノードあたり 2 ランクに制限されました。メモリー消費をさらに最適化できる可能性がありますが、ここでは行いません。

```
784 129167.6 MiB  -1089.7 MiB          60               if (i+1) % (bat_per_epo * 1) == 0:
785 129109.5 MiB    -68.0 MiB           2                   acc_train, acc_valid, f1_train, f1_valid = summarize_performance(
786 129131.5 MiB  56285.0 MiB           2                       i, g_model, c_model, latent_dim, dataset,validation, outdir)
787 129131.5 MiB      0.0 MiB           2                   acc_train_array.append(acc_train)

784  72271.0 MiB  -1052.4 MiB          30               if (i+1) % (bat_per_epo * 1) == 0:
785  72210.4 MiB    -60.6 MiB           1                   acc_train, acc_valid, f1_train, f1_valid = summarize_performance(
```

図 6. メモリー・プロファイラーによって検出されたメモリーリーク

その他の OpenMP* およびインテル® MPI ライブラリー環境変数の設定

ここでは、3 つの環境変数（**KMP_BLOCKTIME**、**KMP_AFFINITY**、および **I_MPI_PIN_DOMAIN**）の最適な設定を調査します。**パフォーマンスの推奨事項**（英語）に従って、**表 1** に示す 6 つの実験を行いました。セット 5 の環境変数の設定は残りの 5 セットでも使用しました。セット 3 で最高のトレーニング・パフォーマンスが得られました（**図 7**）。

セット	環境変数の設定
1	KMP_BLOCKTIME=1 KMP_AFFINITY=granularity=fine,verbose,compact,1,0 I_MPI_PIN_DOMAIN=auto:compact
2	KMP_BLOCKTIME=0 KMP_AFFINITY=granularity=fine,verbose,compact,1,0 I_MPI_PIN_DOMAIN=auto:compact
3	KMP_BLOCKTIME=1 I_MPI_PIN_DOMAIN=auto:compact
4	KMP_BLOCKTIME=0 I_MPI_PIN_DOMAIN=auto:compact
5（共通）	I_MPI_DEBUG=100 HOROVOD_FUSION_THRESHOLD=33554432
6	KMP_BLOCKTIME=200 I_MPI_PIN_DOMAIN=auto:compact

表 1. その他の OpenMP* およびインテル® MPI ライブラリー環境変数の設定に関する実験

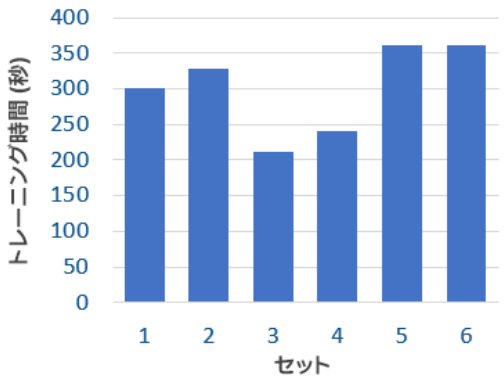

図 7. 表 1 の 6 つの実験によるパフォーマンスの比較

MPI ライブラリーの選択

次に、以前のチューニング実験から最適な設定を収集し、高解像度の画像（1360 x 1360 ピクセル）を使用して、ノードあたり 2 MPI ランクと 33 OpenMP* スレッドで合計 100 エポックを実行して、インテル® MPI ライブラリーと Open MPI のパフォーマンスを比較しました。以下のインテル ® MPI ライブラリーのコマンドラインを使用しました。

```
$ OMP_NUM_THREADS=33 KMP_BLOCKTIME=1 I_MPI_PIN_DOMAIN=auto:compact \
   mpirun -n 2/4/8/16 -ppn 2 -f $HOSTFILE \
   python SGAN_pob.py --image_size 1360 --num_epochs 100 --save_checkpoints False
```

Open MPI では、ほぼ同等のランタイム設定を使用しました。

```
$ OMP_NUM_THREADS=33 KMP_BLOCKTIME=1 \
   mpirun -n 2/4/8/16 -hostfile $HOSTFILE -bind-to socket -map-by ppr:1:socket \
   --mca btl ^openib --mca pml ob1 --report-bindings -x LD_LIBRARY_PATH \
   python SGAN_pob.py --image_size 1360 --num_epochs 100 --save_checkpoints False
```

インテル® MPI ライブラリーは、単一ノードとマルチノードの両方で Open MPI のパフォーマンスを上回りました（**図 8**）。

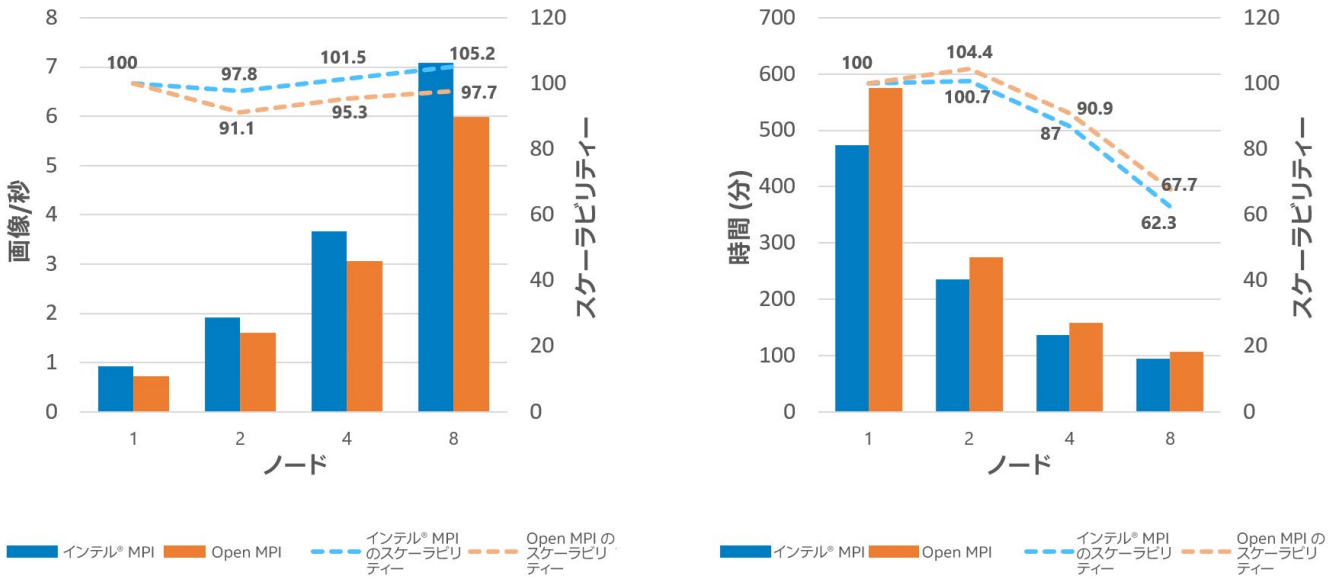

図 8. インテル® MPI ライブラリーと Open MPI のパフォーマンスの比較

画像 / 秒に基づく並列スケーラビリティーは線形に見えますが、時間に基づくスケーリングは劣線形であり、アプリケーションが非効率であることを示しています。さらに詳しく分析したところ、評価の頻度は合計ノード数に応じて正しくスケーリングしないことが分かりました。ノード数の増加に伴い、評価の頻度も増加していました。また、これらの評価は MPI のマスターランクが行うため、マスターランクが評価を完了するまで、ほかのランクはすべてストールしていました。この問題を解決するため、ノード数にかかわらず、15 回のトレーニング反復ごとに評価ステップを実行するようにしました（**図 9** と **10**）。この変更によるモデル精度への影響はありませんでした。

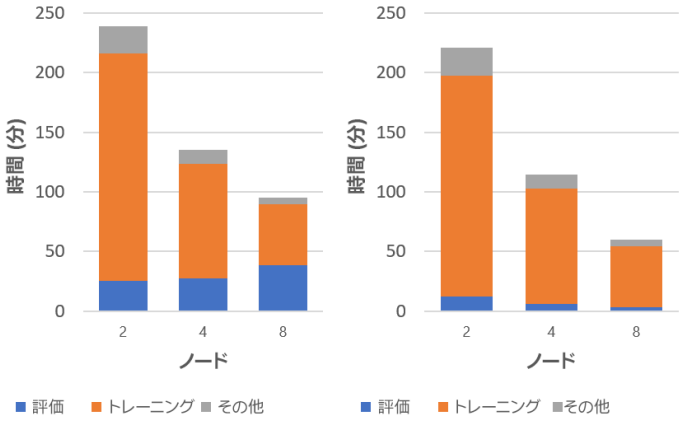

図 9. 最適化の前（左）と後（右）のトレーニングと評価の合計時間に占める割合

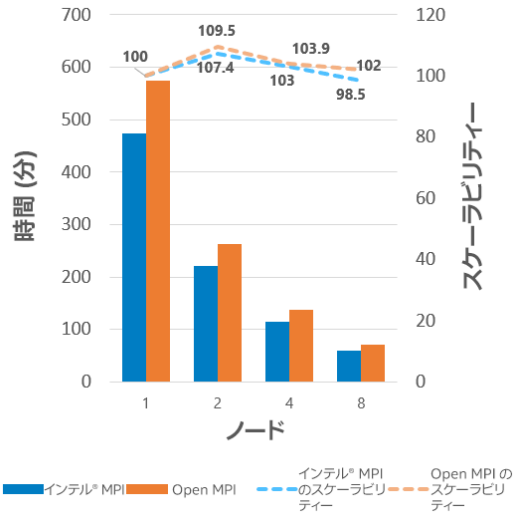

図 10. 線形の時間スケーリングのマルチノード・パフォーマンス

まとめ

この記事では、S-GAN のトレーニング・パフォーマンスを向上するインクリメンタルな最適化アプローチを紹介しました。ランタイムとソースコードベースの最適化の両方を行い、メモリーの問題、スケーリングの問題、単一ノードの非効率なパフォーマンスを解決しました。マルチノード・クラスター上で S-GAN モデルの分散型トレーニングを実装するため、Horovod* を使用しました。そして、2 つの MPI ライブラリー（インテル® MPI ライブラリーと Open MPI）を比較しました。

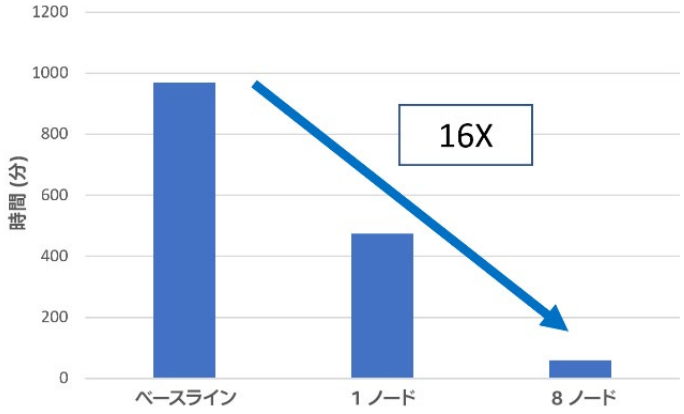

図 11. インテル® MPI ライブラリーで達成された最終的なスピードアップ

図 11 は最適化によるパフォーマンスの向上を要約したものです。単一ノードの最適化は 2 倍のスピードアップを達成し、マルチノードの最適化はさらに 8 倍のスピードアップを達成して、モデル精度を低下させることなく全体的な問題解決までの時間を 1/16 に短縮しました。

ディープラーニングにおける CPU の役割と可能性

Sparsh Mittal インド工科大学ルーキー校助教授

[この記事は **HPCwire**（英語）に掲載された記事を許可を得て掲載したものです。]

この寄稿記事で、Sparsh Mittal 博士は、多様化するプロセッサー空間でのディープラーニング・ワークロードにおける中央処理装置（CPU）の役割を示し、CPU が優れたパフォーマンスを発揮するユースケースを確認して、ディープラーニング・アプリケーションにより促進されたアーキテクチャーの改善と方向性に注目します。この記事では、IEEE Transactions on Neural Networks and Learning Systems で今年 4 月に発表された新しい（Sparsh Mittal 博士などにより執筆された）**調査研究論文**（英語）を紹介します。

ディープラーニング (DL) アプリケーションには、独自のアーキテクチャー特性と効率要件があるため、コンピューティング・システムの選択は、ユーザーが最終的に利用できる DL 処理のサイズに大きな影響を及ぼします。アクセラレーターは、一般に汎用コンピューティング・システム (CPU) よりも高いスループットを提供しますが、CPU が推奨されるまたは優れているメトリックや使用シナリオもあります。インド工科大学ハイデラバード校の Poonam Rajput 博士およびインテル コーポレーションの Sreenivas Subramoney 氏と共著した最近の調査論文 (「**CPU 上のディープラーニングの調査 : 機会と最適化**」) (英語) では、DL における CPU の長所と、さらなる最適化の可能性を説明しています。

CPU には長所があり、アクセラレーターは万能薬ではない

スパース・ディープ・ニューラル・ネットワーク (DNN) は、メモリーアクセスが不規則で、キャッシュタイリングやベクトル化などの最適化を活用できないため、超並列プロセッサーでは非効率である傾向があります。また、再帰型ニューラル・ネットワーク (RNN) は、ステップ間の依存関係により**並列化が困難** (英語) です。同様に、InceptionNet バリアントなどの DNN では、フィルター形状が 1×1、3×3、1×3、3×1 などであり、不規則なメモリーアクセスと層の間での可変量の並列処理が発生します。CPU は、高度なメモリー管理手法を備えており、このような並列処理が制限されるアプリケーションに適しています。例えば、ライス大学の研究者は、Amazon-670K や Delicious-200K などのスパース・データセットを利用した全結合ネットワークで、DL トレーニング問題を検索問題としてモデル化できることを**示しました** (英語)。列の乗算演算はハッシュテーブルに置き換えることができます。これらの CPU 上の手法は、GPU 上の TensorFlow* ベースの実装よりも優れたパフォーマンスを提供します。

バッチサイズが大きな畳み込みニューラル・ネットワーク (CNN) は、2D でも 3D でも大量のメモリーが必要です。クラウドとデータセンターのシナリオでは、CPU 管理ホストのメモリー容量はアクセラレーターよりもはるかに大きいため、CPU でメモリーを大量に消費する操作を実行することは、魅力的であるだけでなく、多くの場合は必須です。TPU などのアクセラレーターは大きなバッチサイズで高いスループットを提供しますが、リアルタイムの推論が必要なアプリケーションでは、大きなバッチサイズの使用は推奨されません。小さなバッチサイズでは、CPU は一般に容認できるレイテンシーを提供します。CPU 上の DL アプリケーションのさらなるチューニングには、ハードウェアを考慮したプルーニング、ベクトル化、キャッシュタイリング、近似計算などの手法を適用できます。調査論文では、これらの多くの手法を要約しています。

幅広く対応可能 : 小さなウェアラブル・デバイスから大規模データセンターまで

IoT デバイスやウェアラブル・デバイスは、電力とサイズの要件が厳しいため、過度な特殊化を排除しています。例えば、スマートウォッチ・チップは、音声 / オーディオ / 画像 / ビデオ処理に個別のアクセラレーターを持つことはできません。Android* を実行しているスマートフォンでは、モバイル GPU や DSP のプログラミング・サポートはまだ完全に成熟していません。実際、一般的なモバイル SoC では、モバイル CPU の理論上のピーク・パフォーマンスはモバイル GPU のピーク・パフォーマンスと同じです。また、ソーシャル・ネットワークなどのウェブサービスをサポートするデータセンターでは、時間の経過とともにコンピューティングの需要が大きく変動します。CPU は、DL タスクと非 DL タスクの両方で高い可用性と効率性を実現するため、この需要の変動に対応できます。最後に、セキュリティー認証が必要な防衛や医療などの非常に厳格な環境では、CPU が唯一のプラットフォームとして選択されることがあります。

明白な利点：経済性と使いやすさ

アクセラレーターは、長い設計サイクルと巨額の投資が必要であり、既存のエコシステムに統合するには、コストのかかる開発作業が必要になります。対照的に、CPU のハードウェア / ソフトウェア・スタックはすでに広く確立され、よく理解されているため、幅広いアプリケーションで十分なスピードアップを達成できます。大規模企業にはカスタム・アクセラレーターを構築および保守するリソースがありますが、中小規模の企業には CPU（または GPU）が引き続き最適なプラットフォームです。

将来の見通し：思っているよりも明るい

今後は、ピーク・パフォーマンスを高めるだけでは不十分です。強化学習や敵対的生成ネットワークなど、幅広い DL アプリケーションのパフォーマンスを向上するには、画期的な改良が必要です。最近の CPU は、低精度コンピューティング・ハードウェアをサポートし始めています。インメモリー・コンピューティングが成熟すると、CPU の大きなキャッシュは大規模な計算ユニットとなる可能性があります。RISC-V などのオープンソース ISA の開発は、アクセラレーターの移植性と独自性の壁を打ち破るでしょう。

注目するメトリックは多種多様であり、最先端の DL モデルにも同じことが言えます。将来的には、「汎用プロセッサー対アクセラレーター」で議論するのではなく、両方の長所を組み合わせた CPU アクセラレーターのヘテロジニアス・コンピューティング・アプローチが主流になると考えています。

oneAPI

ヘテロジニアス・アプリケーションの開発を高速化

次のハードウェア・プラットフォーム向けにコードを書き直すのではなく、イノベーションに取り組む

インテル® oneAPI ツールキット >

SigOpt と Ray Tune を使用した MiniNAS ニューラル・アーキテクチャー検索

SigOpt を使用してモデルのアーキテクチャーを系統的に検索

Ellick Chan インテル® AI Academy、University Relations & Research 責任者
Barrett Williams SigOpt 製品マーケティング・リード

ニューラル・アーキテクチャー検索（NAS）（英語）は、構造の反復と並べ替えにより、画像分類など、指定されたタスク向けに最適化されたニューラル・ネットワーク（NN）を見つける最新の手法です。ネットワークの深さ、畳み込みフィルターの数、プーリング、エポック数、学習率などのネットワーク・パラメーターは、ネットワークの精度、推論スループット、指定されたデータセットのレイテンシーに大きな影響を与えます。

これらのパラメーターの検索空間は大きいため、NAS トレーニングの計算時間は長くなります。この記事では、**SigOpt**（英語）が提供するスマートな検索アルゴリズムと、**Ray Tune**（英語）が提供するクラスター計算能力を組み合わせて、このプロセスを高速化する方法を説明します。簡単な例を使用して、開発者がこの手法を各自のワークフローに適用できるようにします。

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。

NAS の主要概念を説明するため、**図 1(a)**のオリジナル・ネットワークについて考えます。このリファレンス・ネットワークは、1 つの入力層と、それに続くブロック 1 のコピーで構成されます。ブロック 1 は、32 個の 3×3 の畳み込みフィルターで構成された畳み込みプーリングモチーフに基づいていて、オプションでプーリング操作が続きます。このパターンに、同様に 32 個の 3 × 3 畳み込みフィルターで構成されたブロック 2 のコピーが続きます。そして、これらの畳み込みブロックはベクトルにフラット化され、全結合層で処理されて、最終的な分類のためのソフトマックス関数がネットワークの終端となります。

NAS は、データ・サイエンティストがリファレンス・アーキテクチャーのさまざまな並べ替えをテストするのに役立ちます。**図 1 (b)** は、ブロック 2 を繰り返してネットワークの有効な深さを増やす、「深度スケーリング」と呼ばれるオプションを示しています。適切に測定するため、このオプションは 1024 ニューロンの別の全結合層を追加します。このチュートリアルでは、2 つの全結合層は同じサイズですが、実際のアプリケーションではサイズが異なる場合があります。

図 1 (c) は、ネットワークの深さは一定のままで、パラメーターの値が変化する、「幅スケーリング」を示しています。ここでは、L1(層 1)の畳み込みの数を 32 から 16 に減らし、L2 の畳み込みの数を 32 から 64 に増やします。また、全結合層を 1024 から 2048 に広げます。NAS はすべてのパラメーターを一度に検索する必要はないことに注意してください。一度に 1 つのパラメーターを最適化して、ほかのパラメーターを修正できます。オプティマイザーが SigOpt のようにインテリジェントな場合、複数のパラメーターを一度に更新して、最適なネットワーク・アーキテクチャーを迅速に見つけることができるため、より効率的です。

図 1 (d) は、3 × 3 フィルターのカーネルサイズをスケーリングするという試みを示します。ブロック 1 のフィルターを 5 × 5 フィルターに、ブロック 2 のフィルターを 7×7 フィルターに置換します。このスケーリングは、データの特性と入力画像の解像度に応じて、特定のモデルとデータセットのパフォーマンスを向上するのに役立ちます。

ここまでは、簡単な例でも、カスタマイズおよび調査する NN パラメーターに多くの組み合わせがあることは明らかです。この記事の残りの部分では、SigOpt と Ray Tune を使用して、**CIFAR-10 データセット**（英語）で画像の分類に使用する単純な NN の空間を微調整する方法を示します。

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。

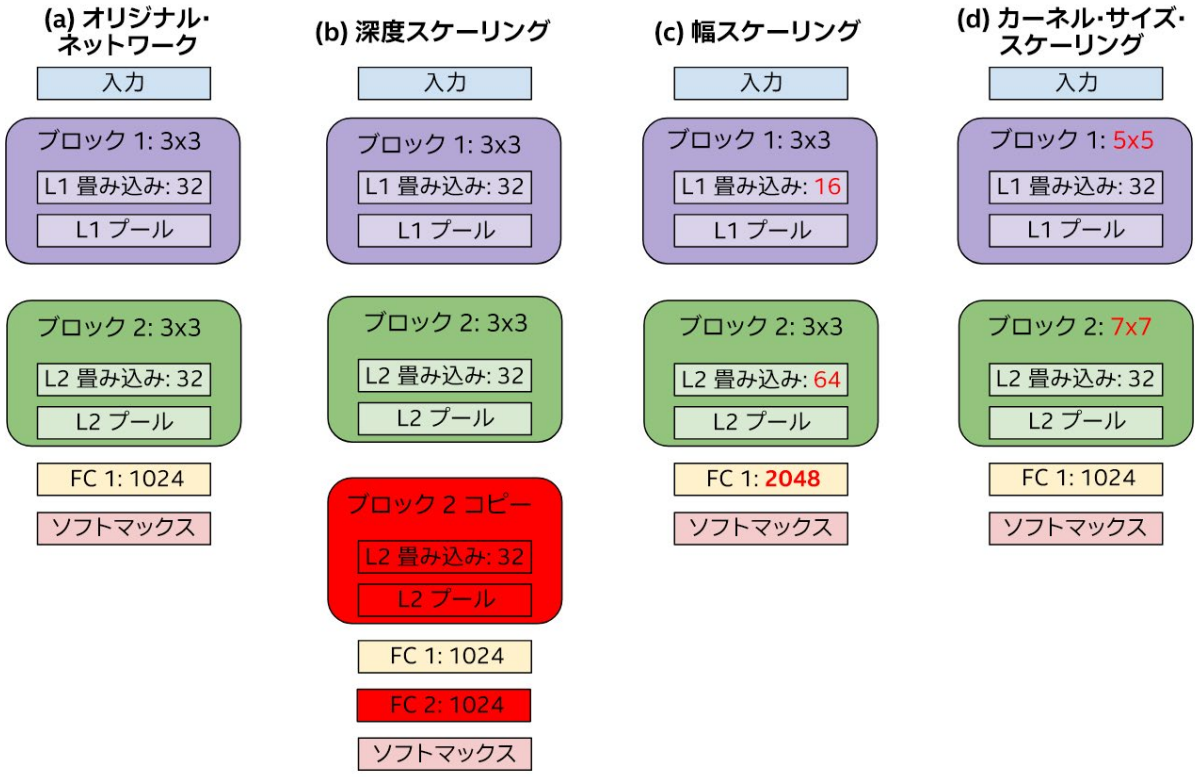

図 1. このチュートリアルで使用したネットワークのバリエーション

全体的なワークフロー

1. NN トレーニング・タスクの定義：データセットとモデルのテンプレート（CIFAR-10 や畳み込みニューラル・ネットワーク [CNN] など）を選択し、チューニングするパラメーター（層の数やフィルターの数など）を定義します。
2. Ray Tune を適用して、モデルのパラメーターの予備セットを検索します。
3. 検索アルゴリズムを SigOpt に適応させて、適切なパラメーターを効率良く取得します。

モデルのパラメーター化

この記事では、NN トレーニング・タスクを 1 つ以上の畳み込みブロックがある畳み込みネットワークとして定義します。CIFAR-10 データセットと **TensorFlow*** （英語）の **Keras* API** （英語）を使用します。

```
# Install prerequisites
!pip install -qqq ray[tune] pandas

# Download dataset
from tensorflow.keras.datasets import cifar10
d = cifar10.load_data()

# Ray Tune calls this function many times in parallel with different hyperparameters
def train(config):
    import os, numpy as np
    import psutil
    os.environ['OMP_NUM_THREADS'] = str(int(psutil.cpu_count()/2))
    from tensorflow import keras
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

    # Load the dataset
    (X_train, Y_train), (X_test, Y_test) = keras.datasets.cifar10.load_data()

    # Create the model
    model = Sequential()

    # Build first convolutional block motif
    model.add(Conv2D(config['nconv0'], kernel_size=(3, 3),
                     activation='relu', input_shape=(32, 32, 3)))
    for i in range(config['nblocks1']):    # Repeat this block nblocks1 times
        if len(model.layers) > config['layers']: break    # Limit depth to "layers"
        # Add nconv1 3x3 conv kernels with relu activation
        model.add(Conv2D(config['nconv1'], kernel_size=(3, 3), activation='relu'))
        # Add pooling
        if config["pooling"] == "True": model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))    # Add dropout

    for i in range(config['nblocks2']):
        if len(model.layers) > config['layers']: break
        model.add(Conv2D(config['nconv2'], kernel_size=(3, 3), activation='relu'))
        if config["pooling"] == "True": model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

    model.add(Flatten())    # Flatten output tensor to a vector
    for i in range(config['nfcll']):    # number of fully connected last layers
        if len(model.layers) > config['layers']: break
        model.add(Dense(1024, activation='relu'))

    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))    # Apply softmax classification
    print("Layers: %d max layers: %d" % (len(model.layers), config['layers']))

    # Compile and setup training
    from tensorflow.keras.utils import to_categorical
    model.compile(loss='categorical_crossentropy',
      # Adam uses an adaptive learning rate that we do not explicitly tune this
      optimizer=keras.optimizers.Adam(lr=0.0001, decay=1e-6), metrics=['accuracy'])

    # Train the model
    model.fit(X_train / 255.0, to_categorical(Y_train),
      batch_size=128, shuffle=True, verbose=0,
      epochs=config["epochs"])

    # Evaluate the model
    scores = model.evaluate(X_test / 255.0, to_categorical(Y_test))
    tune.report(Accuracy=scores[1], Loss=scores[0])
```

モデルをパラメーター化するため、以下の項目を定義します。

- **Epochs**：モデルをトレーニングするエポックの数
- **Layers**：目的のモデルの層の最大数（後続の層はプルーニングされます）
- **Nconv0**：入力層の 3 × 3 畳み込みフィルターの数
- **Nfcll**：全結合された最後の層の数、それぞれ 1024 ニューロン
- **Pooling**：畳み込みブロック 1 および 2 のプーリングを有効 / 無効にするグローバル設定
- **Nblocks1**：畳み込みブロック 1 のコピーの数
- **Nconv1**：畳み込みブロック 1 および 2 の 3×3 畳み込みフィルターの数
- **Nblocks2**：畳み込みブロック 2 のコピーの数
- **Nconv2**：ブロック 2 の 3 × 3 畳み込みフィルターの数

パート 2（後日公開）でのクラスターの展開で一貫性を保つため、コマンドラインから Ray を起動します。単一ノードで実行している場合、次のコマンドは必要ありません。

```
!ray stop
!sleep 3
!nohup ray start --head --num-cpus 1
!sleep 3
import ray
ray.init('localhost:6379')
```

ジョブ・スケジューラー（Portable Batch System など）を使用するクラスター（インテル® DevCloud など）で実行している場合、次のコマンドにより、複数のノードでワーカープロセスを開始します。

```
!which qsub && echo ray start --address `hostname`:6379 --block --num-cpus 1 | qsub
!which qsub && echo ray start --address `hostname`:6379 --block --num-cpus 1 | qsub
!which qsub && echo ray start --address `hostname`:6379 --block --num-cpus 1 | qsub
!which qsub && sleep 20
```

最後に、パラメーターを設定します。

```
from ray import tune
config = {
        "epochs":   tune.randint(20, 30),
        "layers":   tune.randint(1, 20),      # maximum number of layers
        "nconv0":   tune.randint(16, 64),     # input layer
        "nfcll":    tune.randint(0, 2),       # fully connected last layer
        "nblocks1": tune.randint(1, 3),       # conv block 1
        "nconv1":   tune.randint(16, 64),
        "nblocks2": tune.randint(1, 3),       # conv block 2
        "nconv2":   tune.randint(16, 64),
        "pooling":  tune.choice(['True', 'False'])
    }
```

Ray Tune の適用

Ray Tune は、スケーリングされた実験と SigOpt によるハイパーパラメーターの最適化を容易にするため、複数のワーカーノードが検索空間を並列検索できるようにする Python* ライブラリーです。定義したパラメーター空間のナイーブグリッド検索では、約 12 億の構成が検索されます[1]。この記事では、ランダム検索を使用してこのプロセスを高速化する方法を示した後、SigOpt を使用してスマートなガイド付き検索を行い、2 つのアプローチのパフォーマンスと出力を比較します。

Ray Tune で最も重要な入力は、最適化する関数（train）とパラメーターの検索空間（config）です。この 2 つの入力はどちらも定義済みです。対応するコードを以下に示します。その他のオプションは、ガイド付き検索の検索アルゴリズムとスケジューラーを選択することです。

```
# Common options for Ray Tune
tune_opts = {
    # Number of sample points to try, increase for better results
    'num_samples': 5,
    # Some net configs are invalid (pooling too many times creates negative dim)
    'raise_on_failed_trial': False
}

import subprocess, psutil

# Enable accelerator, if present
try:
  if subprocess.run('nvidia-smi').returncode == 0:
    tune_opts['resources_per_trial'] = {'cpu': 1, 'gpu': 1}
except FileNotFoundError: pass

analysis = tune.run(
    train_wrapper,
    config=config,
    verbose=1,
    **tune_opts)

# To see the full optimization results, inspect the results dataframe
analysis.results_df

# Visualize Ray Tune results
d = analysis.results_df
d.plot.scatter('timestamp', 'Accuracy')
```

SigOpt の統合

SigOpt を使用するには、**サインアップ・フォーム**（英語）を使用して無料のアカウントを作成してください。アカウントを作成すると、Google Colab* Notebook または**インテル® DevCloud Jupyter* Notebook**（英語）で使用できる API キーにアクセスできるようになります。

[1] 10*20*(64-16)*3*3*(64-16)*3*(64-16)*2 から算出した値。

```
# Fill in your SigOpt key here
SIGOPT_TOKEN   = "YOUR_SIGOPT_API_KEY_HERE"
SIGOPT_PROJECT = "raytune-simplenas"
!pip install -qqq sigopt

# Convert Ray Tune parameter space to SigOpt format
def convert_space_to_sigopt(config):
    c = []
    for k, v in config.items():
        print(k,v)
        if isinstance(v, ray.tune.sample.Float):
            c.append({'name': k, 'type': 'double',
                      'bounds': {'min': v.lower, 'max': v.upper}})
        elif isinstance(v, ray.tune.sample.Integer):
            c.append({'name': k, 'type': 'int',
                      'bounds': {'min': int(v.lower), 'max': int(v.upper)}})
        elif isinstance(v, ray.tune.sample.Categorical):
            vals = [{'enum_index': i+1, 'name': str(z),
                     'object': 'categorical_value'}
                    for i, z in enumerate(v.categories)]
            c.append({'name': k, 'type': 'categorical', 'categorical_values': vals})
        else:
            print('Unknown type:', k, type(v))
            raise ValueError
    print('config:', c)
    return c

import ray, os
from ray.tune.suggest.sigopt import SigOptSearch
from ray.tune.schedulers import FIFOScheduler
sigopt_connection = ray.tune.suggest.sigopt.Connection(client_token=SIGOPT_TOKEN)
algo = SigOptSearch(
    convert_space_to_sigopt(config),
    name=SIGOPT_PROJECT,
    connection=sigopt_connection,
    project=SIGOPT_PROJECT,
    max_concurrent=3,
    metric="Accuracy",
    mode="max")

analysis_sigopt = tune.run(
    train_wrapper,
    search_alg=algo,
    verbose=1,
    **tune_opts)

analysis_sigopt.results_df

# Visualize Ray Tune results
a = analysis.results_df[['timestamp', 'Accuracy']]
s = analysis_sigopt.results_df[['timestamp', 'Accuracy']]
a['timestamp'] -= a['timestamp'].min()
s['timestamp'] -= s['timestamp'].min()
ax = a.plot.scatter('timestamp', 'Accuracy', c='b',
                    label='Random Search- Acc %0.3f%%' % a['Accuracy'].max())
s.plot.scatter('timestamp', 'Accuracy', c='r', ax=ax,
               label='SigOpt- Acc %0.3f%%' % s['Accuracy'].max(),
               title='Accuracy vs time')
ax.legend(loc='lower right')
```

SigOpt の結果の解釈

上記の例では、検証を素早く完了するため、限定数のサンプルポイントを使用しました。多くのデータポイントをサンプリングした結果を**図 2** に示します。SigOpt の有向検索のほうがランダム検索よりも効率が良いことが分かります。

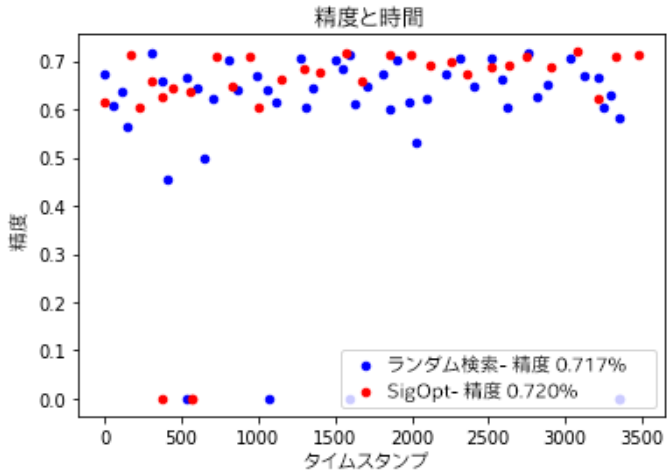

図 2. SigOpt の有向検索とランダム検索

SigOpt は、データ・サイエンティストが NAS に最も重要なパラメーターを理解するのに役立ちます。エポックの数がネットワークの精度に最も大きな影響を及ぼし、プーリング層、畳み込みフィルターの数が続きます (**図 3**)。SigOpt は、この空間をインテリジェントに検索して、最適な値を効率良く見つけます。

データ・サイエンティストがさまざまなパラメーターの影響を理解できるように、SigOpt は、サンプリングしたポイントに関する相対的なパラメーターの重要性を視覚化します。ポイントは (ランダムではなく) オプティマイザーによりインテリジェントに選択されるため、この例は少し偏ったサンプルであることに注意してください。

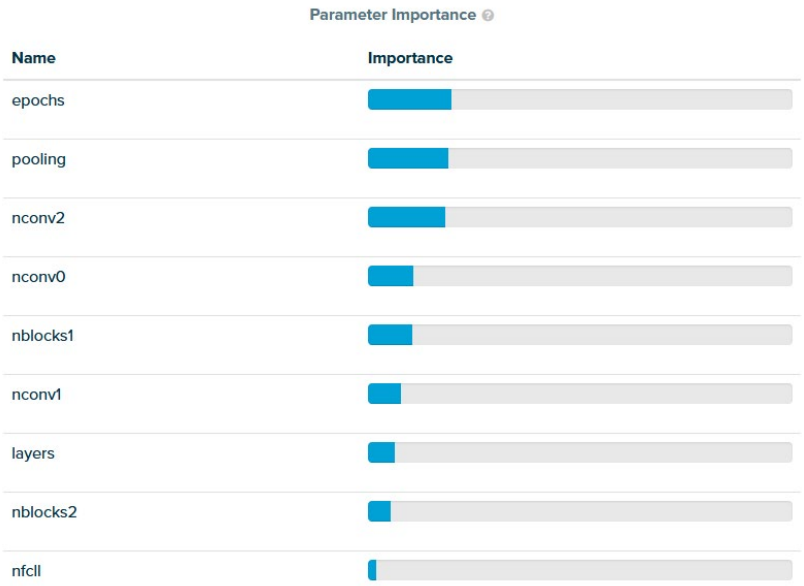

図 3. 最も重要なパラメーターを判断する

パラメーターの相対的な重要性を考慮して、畳み込みフィルターのパラメーター **nconv0** と **nconv1** の関係を調べたところ、この特定の問題では、**nconv0** に約 50 個のフィルターが、**nconv1** に少量のフィルターが使用されると良いと分かります（**図 4**）。このグラフは、変数の任意のペアを視覚化できます。

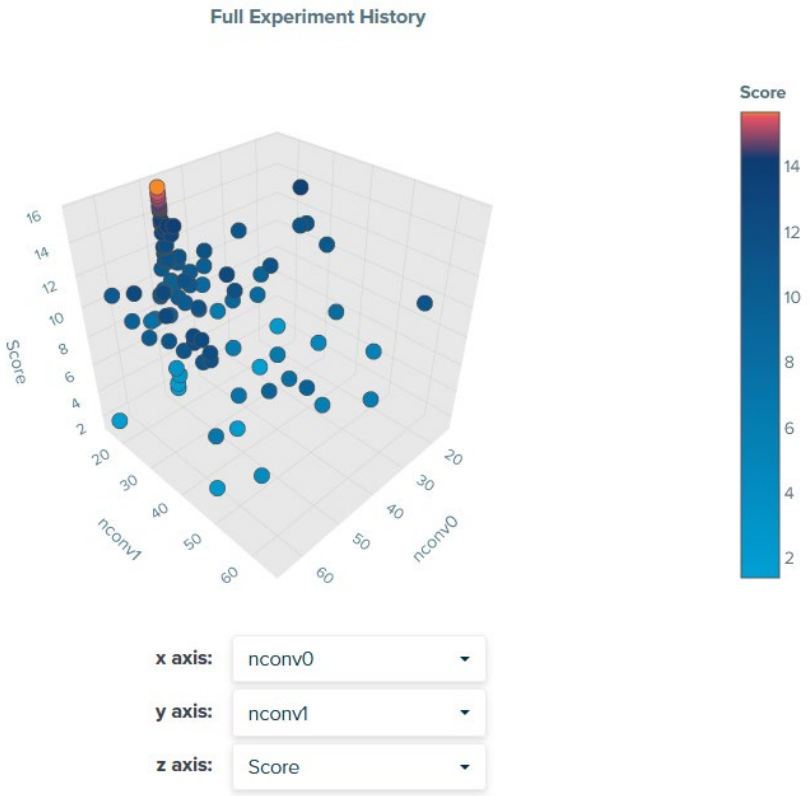

図 4. パラメーター間の関係を視覚化

平行座標グラフは、パラメーター検索の軌跡を示します（**図 5**）。最高のスコアは、多数のエポック、プーリング、および層のパラメーターのさまざまな組み合わせで取得されます。このグラフは、この特定の問題が何を優先するかを示しています。データセットまたは目的が変更されると、優先されるパラメーターが変更されることがあります。

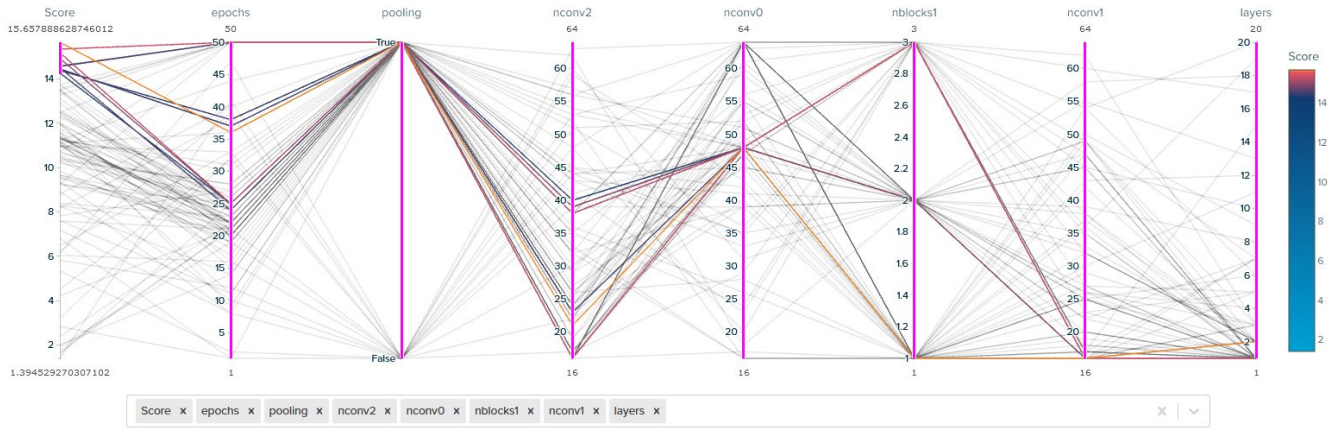

図 5. パラメーター検索の軌跡

パラメーター間の関係を理解することは、データ・サイエンティストが問題のパラメーター値を適切に最適化し、トレードオフを適切に管理するのに役立ちます。**SigOpt の無料アクセスにサインアップ**（英語）して、最適化、追跡、体系化を開始してください。

エンドツーエンドの AI パイプラインのパフォーマンス最適化

インテル® プロセッサー向けに最適化されたフレームワークとライブラリー

Meena Arunachalam インテル コーポレーション 主席エンジニア

最新の人工知能（AI）およびマシンラーニング（ML）アプリケーションは、未処理のデータを詳細な情報に変換するさまざまなタスクを実行します。データ・サイエンティストは、AI および ML アプリケーション向けの複数のフェーズのエンドツーエンド・パイプラインを作成します（**図 1**）。フェーズには、データの取り込み、データのクリーニング、データの調査、特徴量エンジニアリングが含まれ、プロトタイピング、モデルの構築が続き、最後にモデルの展開が行われます。通常、フェーズは何度も繰り返され、パイプライン全体をクラスター全体にスケーリングしたり、クラウドに展開したりする必要があります。

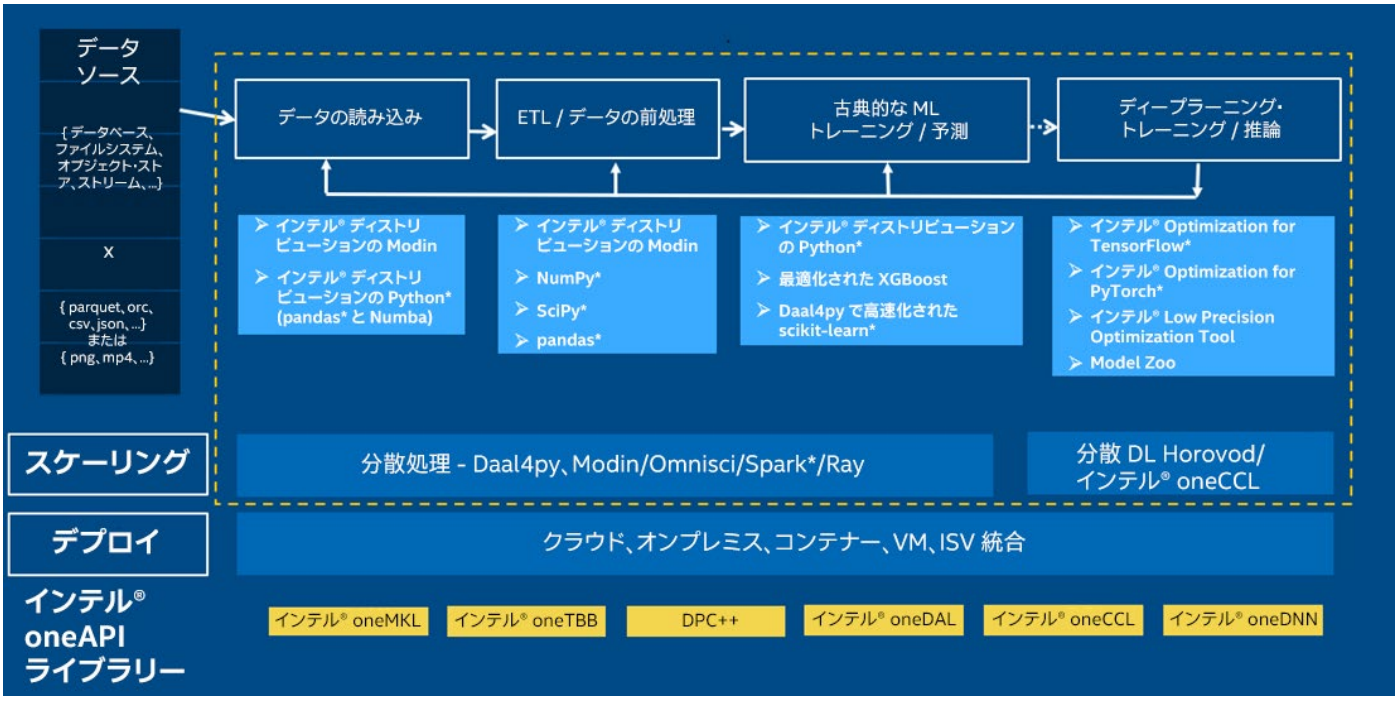

図 1. エンドツーエンドのアナリティクス・パイプライン

インテル® oneAPI AI アナリティクス・ツールキット（英語）は、これらのパイプラインのフェーズを高速化するハイパフォーマンスな API と Python* パッケージを提供します（**図 2**）。

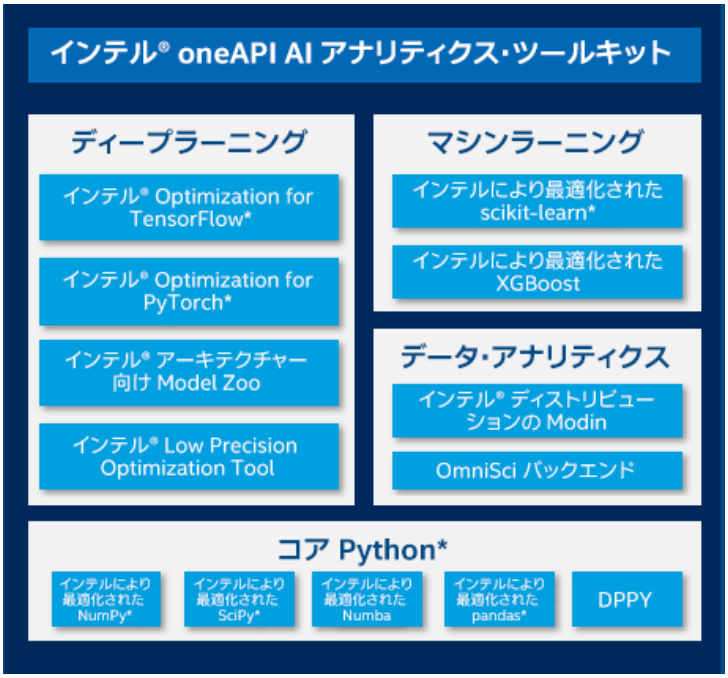

図 2. インテル® oneAPI AI アナリティクス・ツールキット

インテル® ディストリビューションの Modin（英語）は、OmniSci* DB エンジンを使用して、コードを 1 行変更するだけで、スケーラブルな pandas* API を提供します。Modin は、pandas* のデータフレーム処理のパフォーマンスとスケーラビリティーを大幅に向上します。

古典的な ML トレーニングと推論向けに、インテル® oneAPI AI アナリティクス・ツールキットには、一般的な推定器（ロジスティック回帰、特異値分解、主成分分析など）、トランスフォーマー、クラスタリング・アルゴリズム（K 平均法、DBSCAN など）を高速化する**インテル® Extension for Scikit-learn**（英語）が含まれています。

勾配ブースティング向けに、インテルでは XGBoost ライブラリーと CatBoost ライブラリーも最適化しました。これらのライブラリーは、多くのデータサイエンスの問題を迅速かつ正確に解決するために使用される効率の良い並列木ブースティングを提供します。

インテル® oneAPI AI アナリティクス・ツールキットがデータ・サイエンティストの AI パイプラインの高速化を支援する 2 つの例を見てみましょう。

1. **Census :** このワークロードは、米国国勢調査のデータ（1970 年から 2010 年、**IPUMS**（英語）により公開）を使用して、教育レベルを予測するリッジ回帰モデルをトレーニングします。
2. **PLAsTiCC Astronomical Classification :** このワークロードは、夜空の物体を分類することを目的とした **Kaggle***（英語）のオープン・データ・チャレンジです。シミュレートされた天文時系列データを使用して物体を分類します。

両方のワークロードには、3 つの大きなフェーズがあります。

1. **取り込み**は、数値データをデータフレームにロードします。
2. **前処理と変換**は、さまざまな ETL 操作を実行して、列の削除、欠損値の処理、型変換、算術演算、集約など、モデリングのためにデータをクリーンアップして準備します。
3. **データモデリング**は、個別のトレーニング・セットとテストセット、モデルの構築とトレーニング、モデルの検証、推論を作成します。

図 3 は、両方のワークロードについてフェーズごとの内訳を示しています。エンドツーエンドのパイプライン全体を高速化するには、各フェーズの最適化が重要であることが分かります。

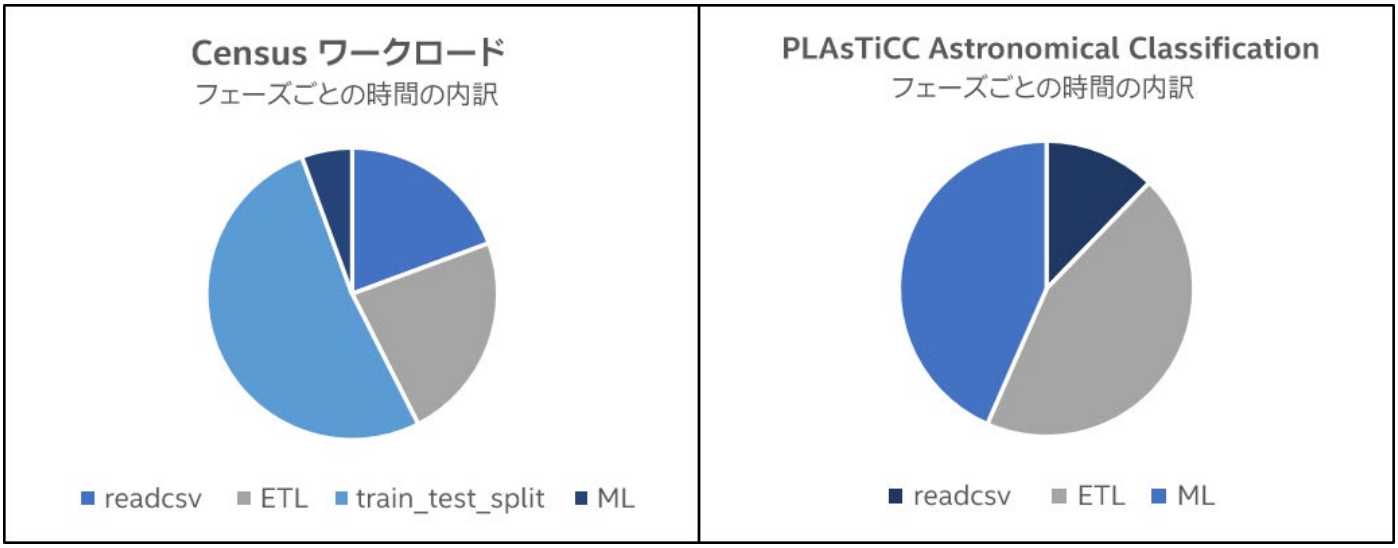

図 3. 各ワークロードのフェーズごとの内訳

図 4 と **5** は、ストック・ソフトウェア（オレンジで表示）とインテルにより最適化されたソフトウェア・スタック（青で表示）を使用した場合の、各フェーズの相対的なパフォーマンスとスピードアップを示しています。第 2 世代インテル® Xeon® スケーラブル・プロセッサーでは、インテルにより最適化されたソフトウェア・スタックを使用した場合、ストック・ソフトウェア・スタックと比較して、Census で 10 倍、PLAsTiCC で 18 倍のスピードアップが達成されています。

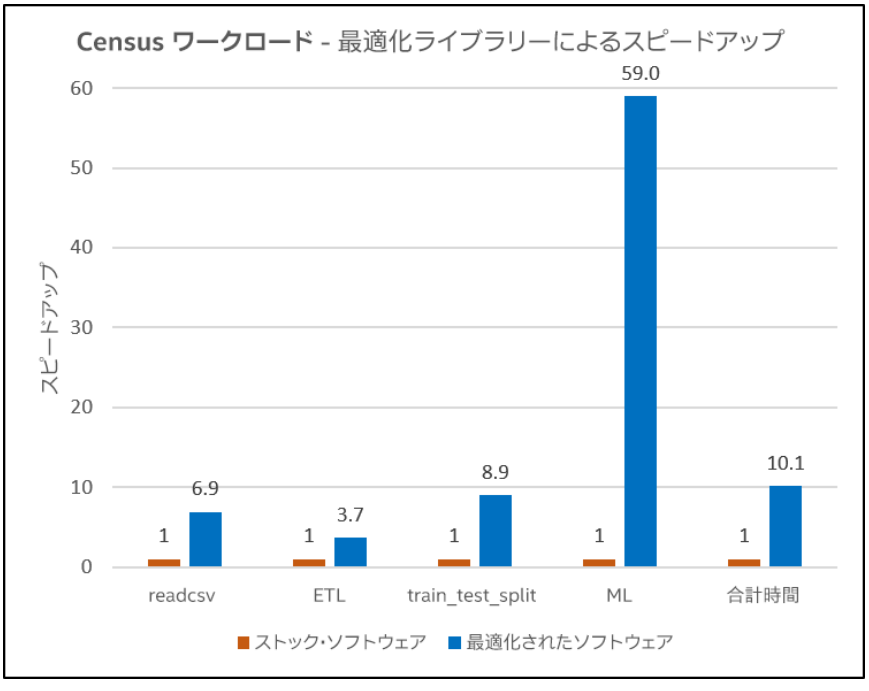

図 4. Census ワークロードのエンドツーエンド・パフォーマンス、各フェーズのスピードアップを表示

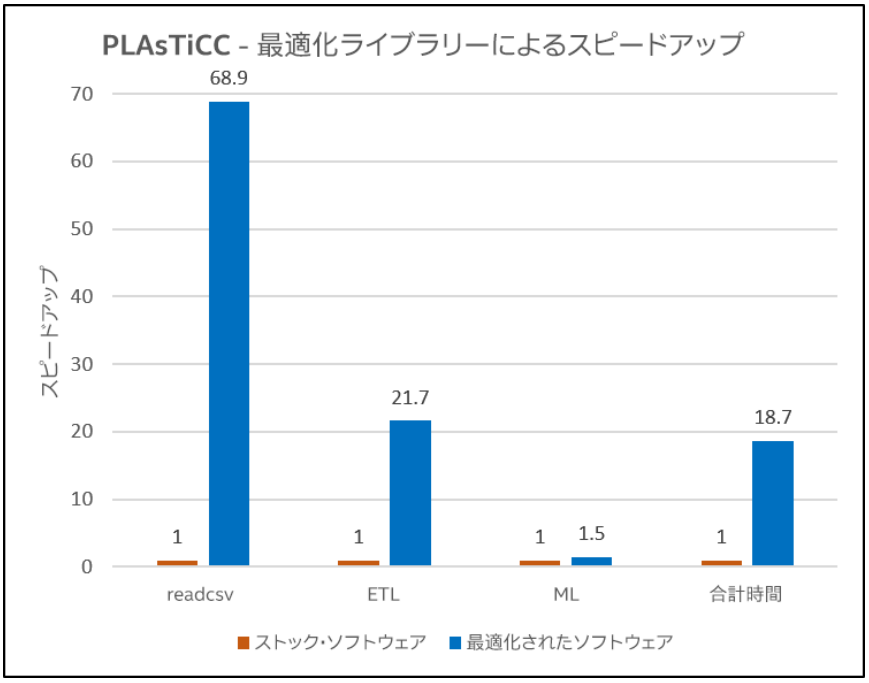

図 5. PLAsTiCC ワークロードのエンドツーエンド・パフォーマンス、各フェーズのスピードアップを表示

Census では、pandas* の代わりにインテル® ディストリビューションの Modin を使用すると、readcsv は 7 倍、ETL 操作は 4 倍にスピードアップしました。トレーニングと予測は、ストックパッケージの代わりにインテルにより最適化された scikit-learn* を使用すると、**`train_test_split`** 関数は 9 倍、トレーニングと予測は 59 倍にスピードアップしました。PLAsTiCC では、インテル® ディストリビューションの Modin を使用すると、readcsv は 69 倍に、ETL 操作は 21 倍にスピードアップしました。これらの高速化は、並列化、ベクトル化、コア・スケーリング、メモリーレイアウトの改善、キャッシュの再利用、キャッシュフレンドリーなブロッキング、効率の良いメモリー帯域幅の使用、プロセッサー命令セットの効果的な使用など、インテル® oneAPI AI アナリティクス・ツールキットのさまざまな最適化により達成されます。

図 6 と **7** は、第 2 世代および第 3 世代インテル® Xeon® スケーラブル・プロセッサー、第 2 世代および第 3 世代 AMD EPYC* プロセッサー、NVIDIA* Tesla* V100 プロセッサーおよび A100 プロセッサーにおける 2 つのワークロードのエンドツーエンドのパフォーマンスを示しています。インテルと AMD の CPU では同じ最適化されたソフトウェア・スタックを使用し、NVIDIA の GPU では RAPIDS スタックを使用しました。ハードウェアとソフトウェアの構成は、記事の最後に記載されています。

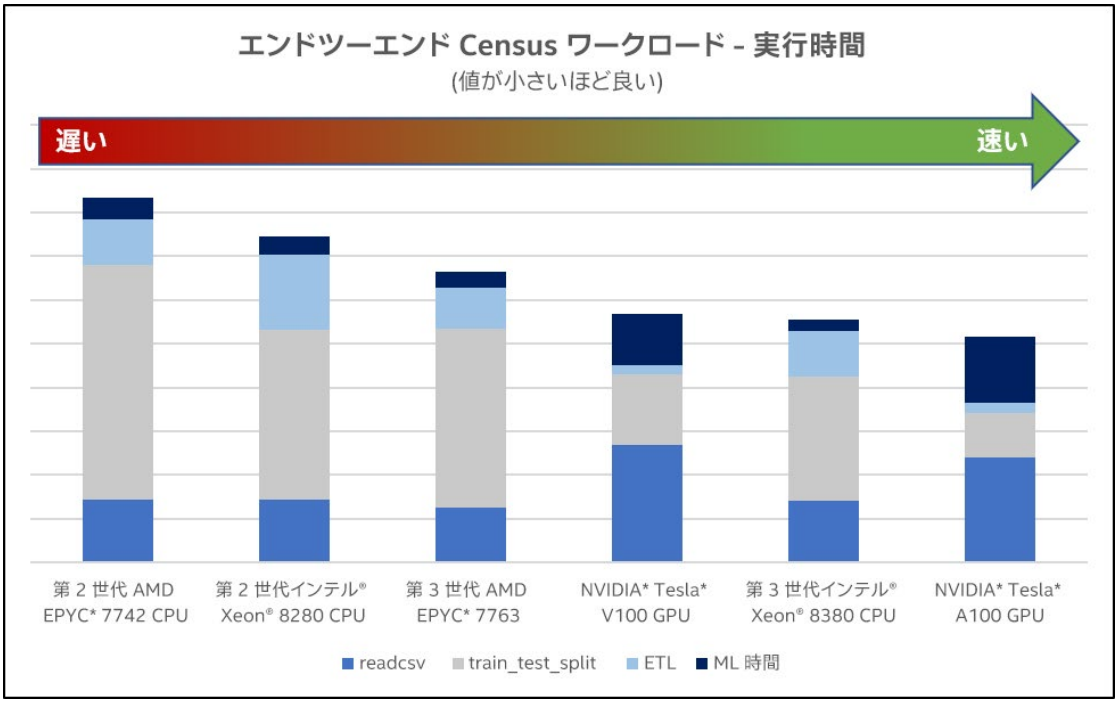

図 6. Census パイプラインのすべてのフェーズで競争力の高いパフォーマンスを発揮

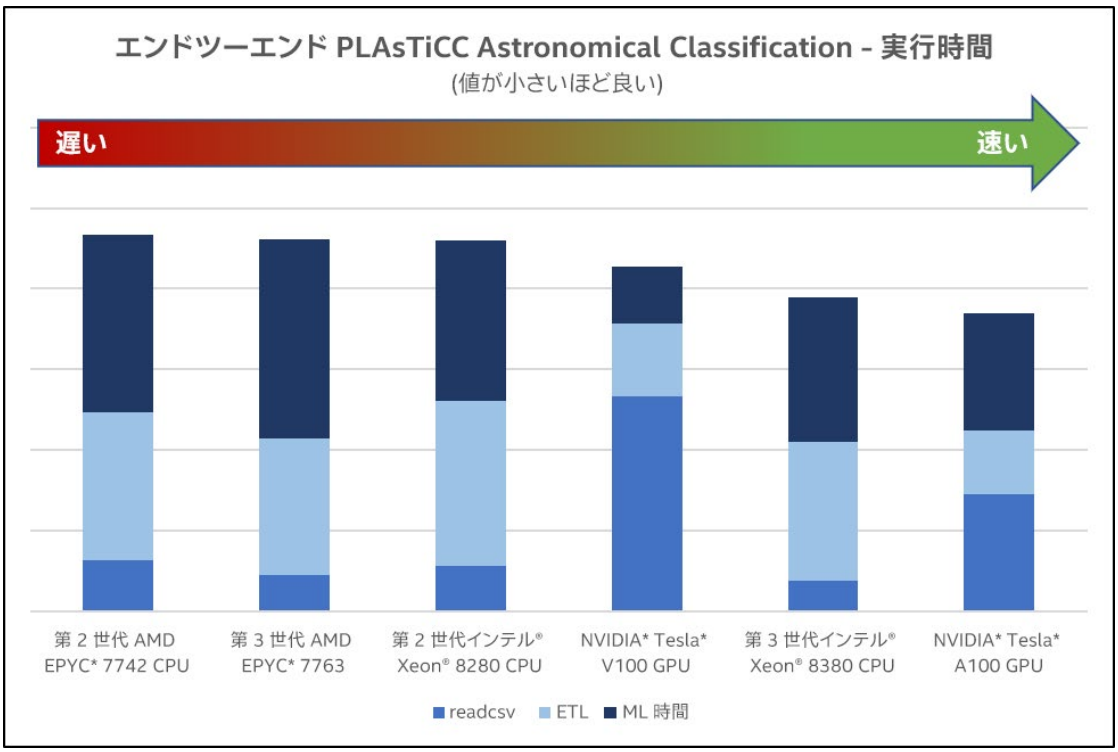

図 7. PLAsTiCC パイプラインのすべてのフェーズで競争力の高いパフォーマンスを発揮

この記事では、ストック・データ・アナリティクス・ソフトウェアを最適化されたソフトウェア・パッケージに単純にドロップイン置換することで、インテル® Xeon® プロセッサー上のパフォーマンスが大幅に向上する（10 倍から 18 倍スピードアップする）ことを示しました。CPU と GPU はパイプラインのさまざまなフェーズで優れていて、インテル® Xeon® Platinum 8380 プロセッサー（第 3 世代インテル® Xeon® スケーラブル・プロセッサー）は NVIDIA* V100 を上回り、NVIDIA* A100 に匹敵しています。第 3 世代インテル® Xeon® スケーラブル・プロセッサーは、低コストで電力効率も優れています。これらの結果は、データ・アナリティクスにおいて一般性が重要であるという概念を補強するものです。

Modin、XGboost、scikit-learn* 拡張、その他のインテル® アーキテクチャー向けに最適化されたソフトウェアは、インテルのウェブサイト、YUM、APT、Anaconda* などの一般的なチャネルから入手できます。好みのディストリビューション・パッケージを**選択してダウンロード**（英語）し、『**入門ガイド**』（英語）に従ってインストールしてください。

ハードウェアとソフトウェアの構成

インテル® Xeon® Platinum 8380 プロセッサー（第 3 世代インテル® Xeon® スケーラブル・プロセッサー）: デュアルソケット・サーバー、ソケットあたり 40 コア、ベース周波数 2.30GHz、インテル® ターボ・ブースト・テクノロジー有効、インテル® ハイパースレッディング・テクノロジー有効。OS : Ubuntu* 20.04.1 LTS、512GB RAM（16x 32GB 3200MHz）、カーネル : 5.4.0–64-generic、マイクロコード : 0x8d055260、BIOS : SE5C620.86B.OR.64.2021.10.3.02.0417、CPU ガバナー : performance。

インテル® Xeon® Platinum 8280L プロセッサー（第 2 世代インテル® Xeon® スケーラブル・プロセッサー）: デュアルソケット・サーバー、ソケットあたり 28 コア、ベース周波数 2.70GHz、インテル® ターボ・ブースト・テクノロジー有効、インテル® ハイパースレッディング・テクノロジー有効。OS : Ubuntu* 20.04.1 LTS、384GB RAM（12x 32GB 2933MHz）、カーネル : 5.4.0–65-generic、マイクロコード : 0x4003003、BIOS : SE5C620.86B.OR.64.2020.51.2.04.0651、CPU ガバナー : performance。

AMD EPYC* 7763 プロセッサー（第 3 世代 AMD EPYC* プロセッサー）: デュアルソケット・サーバー、ソケットあたり 64 コア、ベース周波数 1.50GHz、同時マルチスレッディング有効。OS : Red Hat* Enterprise Linux* 8.3（Ootpa）、1024GB RAM（16x 64GB 3200MHz）、カーネル : 4.18.0–240.el8.x86_64、マイクロコード : 0xa001119、BIOS : Gigabyte* バージョン M03、CPU ガバナー : performance。

AMD EPYC* 7742 プロセッサー（第 2 世代 AMD EPYC* プロセッサー）: デュアルソケット・サーバー、ソケットあたり 64 コア、ベース周波数 1.50GHz、同時マルチスレッディング有効。OS : Ubuntu* 20.04.1 LTS、512GB RAM（16x 32GB 3200MHz）、カーネル : 5.4.0–62-generic、マイクロコード : 0x8301038、BIOS : AMIBIOS* 2.1c、CPU ガバナー : performance。

NVIDIA* Tesla* A100 GPU : DGX-A100 の一部、デュアルソケット、第 2 世代 AMD EPYC* 7742 ホスト CPU。OS : Ubuntu* 18.04.5 LTS、512GB RAM（16x 32GB 3200MHz）、カーネル : 5.4.0–42-generic、マイクロコード : 0x8301034、BIOS : リビジョン 0.23、CPU ガバナー : performance。

NVIDIA* Tesla* V100 GPU : 32GB GPU、デュアルソケット第 2 世代インテル® Xeon® Platinum 8268 ホスト CPU。OS : CentOS* 7.8.2003、384GB RAM（12x 32GB 2933MHz）、カーネル : 5.4.69、マイクロコード : 0x5003003、BIOS : SE5C620.86B.OR.64.2020.51.2.04.0651、CPU ガバナー : performance。

CPU SW:scikit-learn* 0.24.1（daal4py 2021.2 により高速化）、modin 0.8.3、omniscidbe v5.4.1、pandas* 1.2.2、XGBoost 1.3.3、Python* 3.9.7。

GPU SW : NVIDIA* RAPIDS 0.17、CUDA* Toolkit 11.0.221、Python* 3.7.9。

CatBoost のパフォーマンスを最大 4 倍に最適化

マシンラーニングのトレーニングのパフォーマンスを向上するコツ

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

データサイエンスの勾配ブースティング

実際のアプリケーションで精度と効率を実現する、よく知られた勾配ブースティング・フレームワークは多くあります。これらは、さまざまな種類のマシンラーニングの問題に対処する多目的ツールと見なされています。

最新の **Kaggle* 2020 サーベイ**（英語）によれば、データ・サイエンティストの 61.4% が定期的に勾配ブースティング（XGBoost、CatBoost、LightGBM など）を使用しており、これらのフレームワークはさまざまな種類のニューラル・ネットワークよりも広く使用されています。そのため、勾配ブースティングの計算コストを削減することは非常に重要です。

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

この記事では、CatBoost 勾配ブースティング・ライブラリーについて説明します。CatBoost は、ほかのライブラリーよりもカテゴリー別に特徴を効率良く処理し、さまざまなポリシーを提供します。例えば、SymmetricTree ポリシーは、トレーニング済みモデルの分散を減らし、トレーニング時間を大幅に改善します。CatBoost v0.25 では、以前のリリースと比較してトレーニングを最大 4 倍に高速化する最適化が導入されました（**図 1**）。

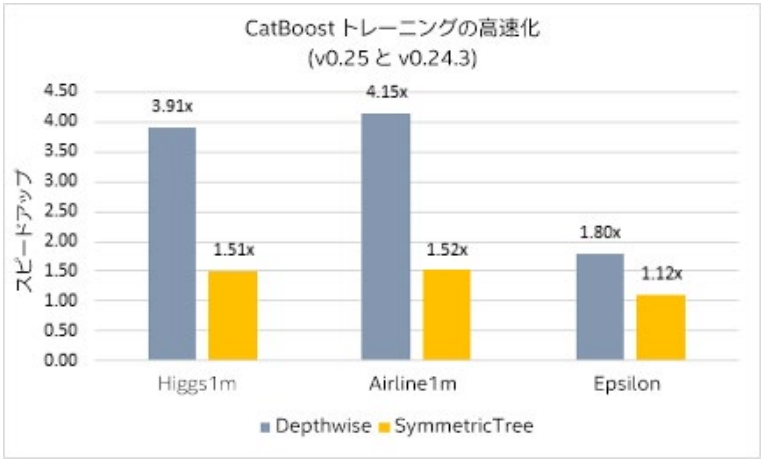

図 1. Higgs1m および Airline1m データセット [1] と Epsilon データセット [2] での CatBoost v0.25 の v0.24.3 に対する相対的なスピードアップ

これらの最新の CatBoost の最適化を詳しく見てみましょう。

最適化の詳細

勾配ブースティングのトレーニング段階は非常に複雑です。不規則なメモリー・アクセス・パターンを軽減し、依存関係のあるループを並列化し、分岐の予測ミスを排除する、特定の最適化が必要な多くの計算カーネルがあります。メモリー依存のワークロード向けのいくつかの最適化（例えば、最小の整数型 **`int8`** を使用）は、CatBoost の以前のバージョンですでに実装されていましたが、拡張の可能性はまだ残されていました（ヒストグラム計算で引き算を使用、スレッド・オーバーヘッドの削減、効率の良いデフォルトのスレッド層バックエンドの導入など）。これらの最新の最適化を見てみましょう。

引き算のトリック

Depthwise ポリシーでは、各トレーニングの反復でレベルごとにツリーが構築されます。この一般的なアプローチは、同じツリーレベルのノードの特徴列とノード構築を効率良く並列化できます。トレーニングの前に、CatBoost は特徴列のビンへの量子化を行います。ビンの数は、**`max_bin`** パラメーターで制御します。トレーニング中に、CatBoost は各決定木ノードのヒストグラムを計算します。ヒストグラムの計算は、各ビンの勾配とヘッセ行列の合計を見つけることから構成されます。

ヒストグラムの計算はトレーニング段階で最も計算量の多い部分ですが、ヒストグラムの加法性から、ヒストグラムの計算プロセスを最適化するソリューションを考えます。親ノードのヒストグラムは、その子ノードのヒストグラムの合計と等しいため、同じレベルの各ノードの正確なヒストグラムを計算する必要はありません。代わりに、親ノードの最小の子のヒストグラムを計算した後、以前に保存した親のヒストグラムから最小の子のヒストグラムを引くことで、最大の子のヒストグラムを計算できます。

このアルゴリズムの最適化は、CatBoost GitHub* リポジトリーのメインブランチに追加されました。Depthwise ポリシーを使用すると、トレーニング時間は大幅に短縮されます（**表 1**）。

CatBoost パフォーマンス (Depthwise) スピードアップ	Higgs1m	Airline1m	Epsilon
	2.37	3.14	1.53

表 1. v0.24.3 から v0.25 に移行して Depthwise ポリシーを使用した場合の CatBoost トレーニングのスピードアップ

スレッド層の改良

前のセクションで説明したパフォーマンスの向上は、CatBoost メンテナーによるスレッド修正がなければ不可能でした。インテル® VTune™ プロファイラーを使用したところ、大量のスレッド・オーバーヘッドが検出されました。オーバーヘッドの多くは各タスクのサイズを増やすことで解消されましたが、根本的な問題は解決されませんでした。

カスタム CatBoost スレッド層バックエンドの代替としてインテル® oneAPI スレッディング・ビルディング・ブロック (インテル® oneTBB) ライブラリーを提供することにより、スレッド層を改善しました。長い実績を誇り、広くサポートされ、高度に最適化されたインテル® oneTBB ライブラリーを CatBoost に統合することにより、効率の良いタスク・スケジューリングと適切にネストされたスレッドが提供され、スレッドのオーバーヘッドの問題が解決しました（**表 2**）。

CatBoost パフォーマンス (SymmetricTree)	Higgs1m (秒)	Airline1m (秒)	Epsilon (秒)
インテル® oneTBB 統合前	76.8	64.8	27.7
インテル® oneTBB 統合後	50.9	42.8	24.8
スピードアップ	1.51	1.51	1.12

表 2. v0.24.3 から v0.25 に移行して SymmetricTree ポリシーを使用した場合の CatBoost トレーニングのパフォーマンスの向上（秒単位）

SymmetricTree ポリシーを使用する場合、インテル® oneTBB の統合により、以前のスレッド層に比べて最大 1.5 倍パフォーマンスが向上しました。CatBoost のメンテナーは、**最大 2 倍のスピードアップを報告**（英語）しています。引き算のトリックと組み合わせることで、インテル® oneTBB スレッド層は Depthwise ポリシーのトレーニング・パフォーマンスを改善し（Higgs1m で 1.65 倍、Airline1m で 1.32 倍、Epsilon で 1.17 倍のスピードアップ）、合計で Higgs1m で 3.9 倍、Airline1m で 4.1 倍、Epsilon で 1.8 倍のスピードアップが達成されました。

CatBoost v0.25 以降、インテル® oneTBB はデフォルトのスレッド層です。

まとめ

CatBoost を使用してマシンラーニングのモデルをトレーニングする場合は、必ず最新バージョンを使用してください。v0.25 の最適化により最大で 4 倍のスピードアップが得られますが、CatBoost のパフォーマンス向上のためにできることはまだあります。コア・スケーラビリティーのさらなる改善、メモリー帯域幅の使用率の向上、ベクトル命令の使用などが、将来のリリースで追加されると考えています。

ハードウェアとソフトウェアの構成

インテル® Xeon® Platinum 8280L プロセッサー（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、ソケットごとに 28 コア、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。合計メモリー 384GB（12 スロット / 16GB/2933MHz）。CatBoost 0.24.3（最適化前）、0.25（最適化後）。NumPy* 1.15.1。scikit-learn* 0.24.1。インテル® oneTBB 2021.2。

トレーニングのパラメーター

Higgs1m [1] および Airline1m [1]。

```
{iterations=1000, learning_rate=0.1, grow_policy='Depthwise/SymmetricTree', score_
function='L2' , border_count=256, scale_pos_weight=2, l2_leaf_reg=1,logging_
level='Silent', depth=8}
```

Epsilon [2]。

```
{max_depth: 8, learning_rate: 0.1, reg_lambda: 0, iterations=500, grow_
policy='Depthwise/SymmetricTree', logging_level='Silent'}
```

1.Airline1m および Higgs1m データセットは、GitHub* の **dmlc/xgboost-bench**（英語）リポジトリーにあります。
2.Epsilom データセットは、GitHub* の **NVIDIA/gbm-bench**（英語）リポジトリーにあります。

intel software

THE PARALLEL UNIVERSE

インテルのテクノロジーを使用するには、対応したハードウェア、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。詳細については、OEM 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。

実際の費用と結果は異なる場合があります。

インテルは、サードパーティーのデータについて管理や監査を行っていません。ほかの情報も参考にして、正確かどうかを評価してください。

最適化に関する注意事項 : インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。注意事項の改訂 #20110804。
https://software.intel.com/en-us/articles/optimization-notice#ja-jp

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。構成の詳細は、補足資料を参照してください。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。

性能の測定結果はシステム構成の日付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。詳細は、システム構成を参照してください。絶対的なセキュリティーを提供できる製品またはコンポーネントはありません。

本資料は、(明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず) いかなる知的財産権のライセンスも許諾するものではありません。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、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/2108/PDF/XL/SPI/ND