intel software

THE PARALLEL UNIVERSE

NeurIPS 億単位の近似最近傍探索コンテストの優勝作品

エンドツーエンドの AI パイプラインの最適化

ハイパーパラメーターのチューニングによる
AI パフォーマンスの向上

Issue
48
2022

目次

編集者からのメッセージ 3

注目記事

NeurIPS 億単位の近似最近傍探索コンテストの優勝作品 6
インテル® Xeon® プロセッサーとインテル® Optane™ テクノロジーで検索パフォーマンスを大幅に向上

エンドツーエンドの AI パイプラインの最適化 15
インテル® Xeon® プロセッサーにおける AI パイプラインの最適化アプローチ

AI アプリケーションの最適化 26
ハイパーパラメーターのチューニングと最適化されたソフトウェアにより AI パフォーマンスを向上

SYCL* 2020 で追加された 5 つの優れた機能 30
SYCL* プログラミング言語が進化

ArrayFire* と oneAPI による 2 次元フーリエ相関アルゴリズムの高速化 39
ヘテロジニアス並列処理について ArrayFire* と oneAPI の抽象化を徹底比較

oneAPI の maxloc リダクション 50
SYCL* と oneDPL でこの一般的な並列パターンを実装

oneDPL による C++ プログラミングの生産性とパフォーマンスの向上 59
ヘテロジニアス・コンピューティングではオープンな標準と移植性が味方になる

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

編集者からのメッセージ

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

SYCL* の重要な問題

以前、私は熱心な Perl プログラマーでした。ある日、同僚が彼のお気に入りの新しい言語である Python* を勧めてきました。それに対し私は、「どうして別のプログラミング言語が必要なんだい？」と言いました。あれから 20 年、現在私は Python* に傾倒しています（ただし、Julia（英語）にも惹かれていますが）。Khronos Group の SYCL* 仕様（英語）は、C++ にヘテロジニアス･データ並列処理をもたらしますが、あのときと同じ「別のプログラミング言語が必要なのか？」という疑問を抱きます。oneAPI 業界イニシアチブ（英語）は、ヘテロジニアス並列処理のダイレクト･プログラミング･アプローチとして SYCL* を採用しており、The Parallel Universe の名誉編集長である James Reinders と Codeplay Software 社の著名なエンジニアである Michael Wong 氏が「SYCL* を使用すべき理由」でこの疑問といくつかの疑問に対して素晴らしい回答を提供しています。

本号では、oneAPI と SYCL* の記事をいくつか掲載していますが、特集記事「**NeurIPS 億単位の近似最近傍探索コンテストの優勝作品**」では、最近実施されたデータサイエンスのコンテストで優勝したソリューションを紹介します。そして、「**エンドツーエンドの AI パイプラインの最適化**」と「**AI アプリケーションの最適化**」の 2 つのデータサイエンスに関する記事が続きます。人工知能（AI）はアナリティクス･パイプラインの華やかな部分ですが、このステップに到達するには大変な努力と多くの計算が必要です。これらの記事は、AI を含むアナリティクス･パイプラインのさまざまな部分を最適化する方法について説明します。

データサイエンスから、再び SYCL* に目を向けます。最近の書籍『Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL』（英語）の共著者である James Brodman と John Pennycook が、「**SYCL* 2020 で追加された 5 つの優れた機能**」を紹介します。

前号の The Parallel Universe で述べたように、私は ArrayFire*（英語）ヘテロジニアス並列ライブラリーを試しています。本号では、ArrayFire 社のソフトウェア･エンジニアである Umar Arshad 氏と私が「**ArrayFire* と oneAPI による 2 次元フーリエ相関アルゴリズムの高速化**」で ArrayFire* と oneAPI の抽象化を徹底比較します。「ArrayFire* と oneAPI、各種ライブラリー、OpenCL* の相互運用性」（The Parallel Universe 47 号）も参考になります。

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。

本号を締めくくるのは、oneAPI データ並列 C++ ライブラリー（oneDPL）に関する 2 つの記事、「**oneAPI の maxloc リダクション**」と「**oneDPL による C++ プログラミングの生産性とパフォーマンスの向上**」です。これらの記事では、プログラマーの生産性とヘテロジニアス並列処理を向上するため、oneDPL で C++ STL 関数を使用する方法について説明します。

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、intel.com/oneapi（英語）を参照してください。

Henry A. Gabb

2022 年 4 月

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

intel. software

将来に通用するコード。

プロプライエタリーの壁を越えて成長。

1 oneAPI

複数の言語をサポートし、ヘテロジニアス計算パフォーマンスを実現する、
単一のオープンなプログラミング・モデルで、コードの幅を広げましょう。

オープンスタンダードに根ざしたインテル® oneAPI は、クロスアーキテクチャーのライブラリー、
コンパイラー、ツールを提供し、コードをより多くのハードウェアに対応させ、
比類ないパフォーマンスを実現します。

インテル® oneAPI の詳細 →

NeurIPS 億単位の近似最近傍探索コンテストの優勝作品

インテル® Xeon® プロセッサーとインテル® Optane™ テクノロジーで検索パフォーマンスを大幅に向上

Mariano Tepper、Cecilia Aguerrebere、および Ted Willke インテル ラボ
Sourabh Dongaonkar および Jawad B Khan インテル ファウンドリー・サービス
Mark Hildebrand カリフォルニア大学デイビス校

近似最近傍探索（ANN）（英語）とも呼ばれる類似性探索は、ウェブスケールのデータベース上で検索、推薦、ランキング操作を必要とする多くの AI アプリケーションのバックボーンであり、精度、スピード、スケール、コスト、そしてサービス品質制約が重要になります。この記事では、インテル® Xeon® プロセッサーとインテル® Optane™ メモリーの能力を活用することで、ANN を進化させるソリューションについて説明します。これらの進化を示すため、我々は NeurIPS'21 億単位の近似最近傍探索コンテスト（英語）に参加し、カスタム・ハードウェア・トラックで優勝しました。我々のソリューションは、CAPEX（資本的支出）を 1/8 ~ 1/19 に削減し、5 年間の OPEX（営業費用）を第 2 位のソリューション（英語）と同等に抑えました。これにより、現代の大規模で高精度かつハイパフォーマンスなシナリオにおける類似性検索の参入障壁を大幅に下げ、普及を促進します。

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。

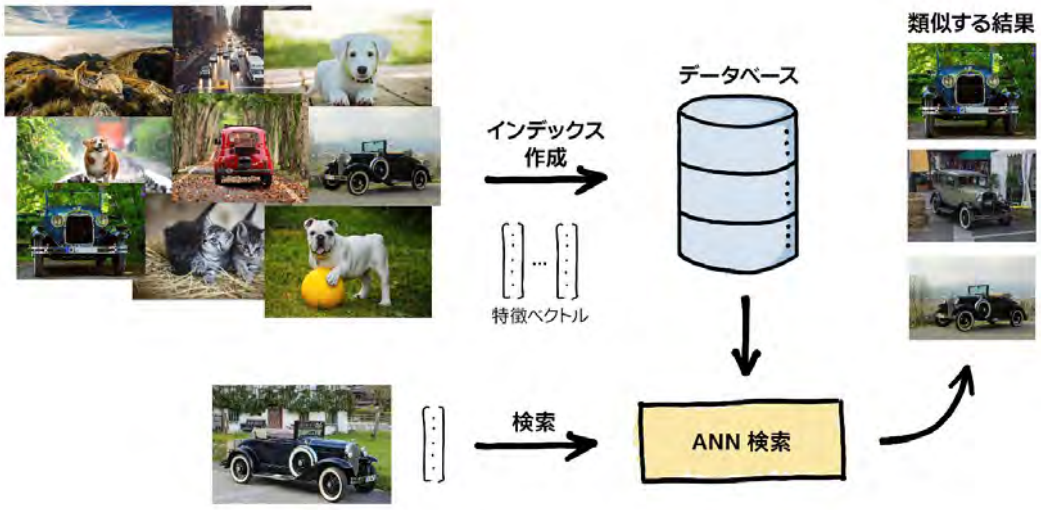


図 1. ANN 検索パイプラインの概略図

近似最近傍探索

類似性検索の目的は、与えられた高次元特徴ベクトルのデータベースと同次元のクエリーベクトルに対し、何らかの類似性関数に基づいて、クエリーと最も類似するデータベース・ベクトルを検索することです（**図 1**）。現代のアプリケーションでは、これらのベクトルはデータ（画像、音声、テキストなど）の内容を表し、類似したベクトルが意味的に関連する項目に対応するように、ディープラーニング・システムを使用して抽出および要約されます。

実際に有用であるには、類似性検索ソリューションはさまざまな価値を提供する必要があります。

- **精度：**検索結果が実用に耐えられる品質でなければなりません（つまり、検索された項目がクエリーに類似している必要があります）。
- **パフォーマンス：**検索は高速で、多くの場合、厳格なサービス品質制約を満たす必要があります。
- **スケーラビリティー：**データベースは、含まれる項目の数や次元数など、ますます大きくなっています。
- **コスト：**本番環境やデータセンターで使用されるため、総所有コスト（TCO）を最小化する必要があります。TCO は通常、資本的支出（CAPEX）と営業費用（OPEX）の組み合わせとして測定されます。

自然なソリューションとして、データベース内の各ベクトルを線形にスキャンし、クエリーと比較し、結果を類似度の降順でランク付けし、最も類似したベクトルを返すことが考えられます。しかし、データの膨大さと豊富さから、このアプローチは不可能であり、計算とメモリーの両方を多用する大規模な類似性検索は、非常に困難な問題です。一般に、次の 2 つのフェーズを含む優れたソリューションを必要とします。

1. インデックス作成時には、データベースの各要素を高次元ベクトルに変換し、検索時にデータベースの一部にのみアクセスするようにインデックスを作成します。
2. 検索時には、与えられたクエリーベクトルに対し、アルゴリズムはインデックスを使用してデータベースをふるいにかけます。その結果は、最終的な用途に応じて、またこれらの意味的に関連した結果に基づいて、さまざまな情報に基づいた行動を取るために使用されます。

NeurIPS'21 億単位の近似最近傍探索コンテスト

2021 年 12 月、NeurIPS カンファレンスの一環として、初の億単位の類似性検索コンテストが開催されました。コンテストの目的は、精選された実際のデータセットで最先端の類似性検索を比較検討し、新しいソリューションの開発を促進することでした。我々は、インテルのハードウェアをフル活用できるカスタム・ハードウェア・トラックに参加しました。インテル® Xeon® プロセッサーとインテル® Optane™ パーシステント・メモリー（PMem）の機能をフル活用したソリューションを開発し、ワンツーアプローチを実現した結果、最終的にコンテストで優勝を勝ち取りました。

データセット間で比較される基本的なメトリックは TCO で、90% の再現率と 10 万クエリー / 秒(QPS)のスループットを持つソリューションの CAPEX + 5 年間の OPEX として定義されました。CAPEX と OPEX（英語）は、コンテスト主催者によって次のように定義されています。

- CAPEX =（すべてのハードウェア・コンポーネントの希望小売価格）×（100,000QPS をサポートするのに必要な最小システム数)
- OPEX =（ベースライン再現率 @10 しきい値以上の最大 QPS）×（キロワット時 / クエリー）×（秒 / 時間）×（時間 / 年）×（5 年間）×（ドル / キロワット時）×（100,000QPS に対応するのに必要な最小システム数)

これらのメトリックは、各ソリューションの電力効率 (OPEX) と生のパフォーマンス (CAPEX) のバランスをとります。

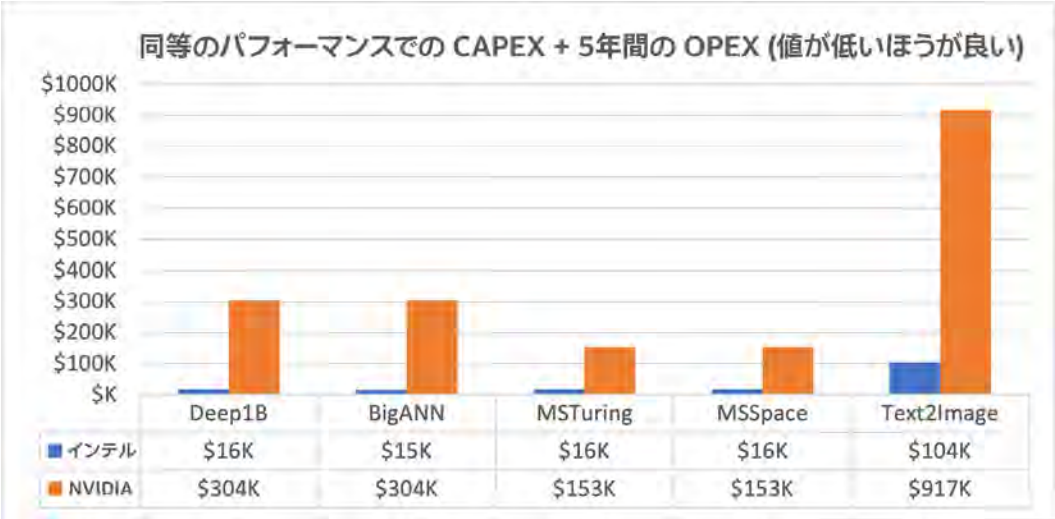


図 2. 優勝したインテルベースのソリューションと第 2 位の NVIDIA* ベースのソリューションの TCO の相違点（5 つの異なるデータセット（X 軸）で最大 20 倍の向上を達成)

我々のソリューションは TCO を画期的に改善し、第 2 位の複数のデータセットで NVIDIA DGX* A100 GPU システムを使用する `cuanns_multigpu` と比較して、8.85 ~ 19.7 倍の効率を実現しています（**図 2**）。この圧倒的な差は、我々と第 2 位の NVIDIA* のハードウェア構成を比較すると明白です（**表 1**）。インテル® Optane™ パーシステント・メモリーを搭載した安価な 1U 2S インテル® Xeon® プロセッサー・ベースのサーバー 1 台で、A100 GPU を 8 個、ANN 検索ワークロード用の 64 コア CPU を 2 個搭載した NVIDIA* のフラッグシップである NVIDIA DGX* A100 サーバーと同等のパフォーマンスを達成することが可能です。

	インテル® Xeon® プロセッサー + インテル® Optane™ メモリー	**NVIDIA DGX* A100**
CPU	デュアル・インテル® Xeon® Gold 6330N プロセッサー 合計 56 コア	デュアル AMD EPYC* 7742 (開発コード名 Rome) 合計 128 コア
システムメモリー	512GB DDR4 2TB インテル® Optane™ パーシステント・メモリー 200 シリーズ	2TB DDR4
GPU	なし	8x NVIDIA* A100 80GB GPU
GPU メモリー	なし	640GB
電力	最大 1.2KW	最大 6.5KW
合計コスト	14,664 ドル（英語）	150,000+ ドル（英語）

表 1. BigANN コンテストのインテルと NVIDIA* のハードウェア構成の比較—2 つの構成は同様のパフォーマンスを達成

インテルベースのソリューションでは同等のパフォーマンスにおける CAPEX が非常に低いことに加え、電力効率も大幅に向上しています。これは、コンテストの全マシンで標準 IPMI インターフェイスにより測定されたクエリー当たりの電力（ジュール単位）でも示されています。インテルベースのソリューションのクエリー当たりの電力は、NVIDIA* ベースのソリューションと比較して、最大 5 倍も優れています（**図 3**）。これは、ANN 検索問題に対するより持続可能なソリューションであるとともに、長期的な OPEX の改善にもつながります。

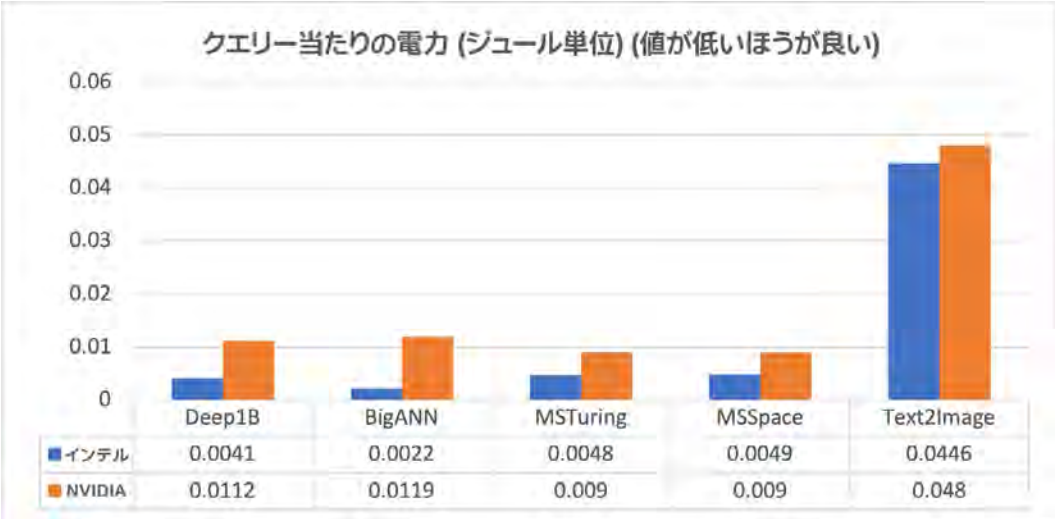


図 3. インテルベースのソリューションのクエリー当たりの電力は NVIDIA* ベースのソリューションと比較して最大 5 倍優れている

このような TCO の大幅な改善は、インテル® Xeon® プロセッサーの利点とインテル® Optane™ パーシステント・メモリーの優れた容量とスループットに加え、ハードウェア・リソースの最適な利用を可能にするアルゴリズムの革新によって実現されました。以下では、このハードウェアとソフトウェアの複合的なアプローチにより、いかにして圧倒的な差をつけてこのコンテストを勝ち抜いたのか、その詳細を紹介します。

アルゴリズムによるアプローチ

ここでは、インテル® Xeon® プロセッサーとインテル® Optane™ パーシステント・メモリーによる ANN 検索アルゴリズム（GraphANN）のパフォーマンスを示します。GraphANN は、インテル® Optane™ パーシステント・メモリー向けに高度に最適化されたグラフベースの Vamana アルゴリズムの拡張版です。データポイントのインデックスを作成するため有向グラフを構築し、グラフをナビゲートして新しいクエリーの最近傍を見つけるため貪欲法による検索を行います。検索中は、2 つの主要なデータ構造（グラフと特徴ベクトル）が使用されます。このソリューションでは、グラフをインテル® Optane™ パーシステント・メモリーに格納し、可能な限り特徴ベクトルを DRAM に保持します。この組み合わせにより、1 ドル当たりのスループットとパフォーマンスが飛躍的に向上します。さらに、インテル® Optane™ パーシステント・メモリーは従来の DRAM よりもはるかに大容量であるため、大規模なデータセットに必要なスケーリングが可能です。最後に、パーシステント・メモリーを使用することで、億単位のデータセットでは非常に時間のかかるメモリーへのインデックスのロードが不要になるという利点もあります。

インテル® Optane™ パーシステント・メモリー（PMem）

インテル® Optane™ パーシステント・メモリーは、SSD やパーシステント・メモリー・アプリケーションに使用できるストレージクラスのメモリーです。歴史的に、メモリーとストレージのパフォーマンスの間には常にギャップがありました。インテル® Optane™ メモリー・テクノロジーはこのギャップを埋めるために設計されています（**図 4**）。高密度でトランジスターを必要としない 3 次元積層可能な設計で、メモリーセルを個別にアドレス指定できます。これらの機能は、手頃な価格で大容量とデータ永続性のサポートというユニークな組み合わせを提供します。特徴的な動作モードを提供する革新的なテクノロジーにより、さまざまなワークロードのニーズに適応できます。例えば、インテル® Optane™ テクノロジーは、ストレージがボトルネックとなっている大規模アプリケーションのログやキャッシュ層のストレージを高速化するために使用されています。

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。

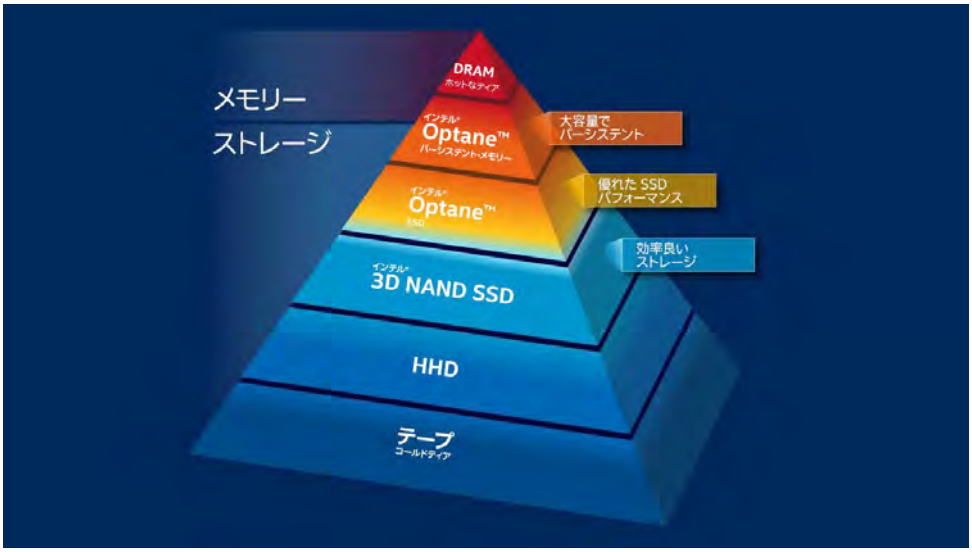


図 4. メモリーとストレージの階層とインテル® Optane™ テクノロジーの位置付け

インテル® Optane™ パーシステント･メモリーと DRAM の類似点は、DIMM 規格パッケージであり、DRAM と同じバス / チャネル上に存在し、DRAM と同様に揮発性データを格納できます。DRAM との相違点は以下の通りです。

- インテル® Optane™ パーシステント･メモリーは従来の DRAM よりもはるかに大きな容量を実現します。モジュールの容量には 128GB、256GB、512GB があり、一般に 16GB から 64GB の範囲である DRAM モジュールよりもはるかに大容量です。
- インテル® Optane™ パーシステント・メモリーは、モジュールへの電源供給がない状態でデータを保存できるパーシステント・モードで動作可能であり、組込みのハードウェア暗号化機能によりデータを安全に維持します。TCO は、GB 当たりのコスト比較において DRAM よりも大幅に改善されており、また DRAM を超える容量への拡張が可能です。

柔軟性を高めるため、インテル® Optane™ パーシステント・メモリーにはメモリーモードとアプリ・ダイレクト・モードの 2 つの動作モードがあります。メモリーモードは、永続性なしでメインメモリー容量を拡張します。インテル® Optane™ パーシステント・メモリーと、そのダイレクト・マップ・キャッシュとして機能する従来の DRAM の組み合わせです。アプリ・ダイレクト・モードでは、インテル® Optane™ パーシステント・メモリーは、DRAM とは別にアドレス指定可能なパーシステント・メモリー・デバイスのように見えます。

インテル® Optane™ パーシステント・メモリー（PMem）を使用した ANN

Vamana アルゴリズムのアクセスパターンを調査した結果、クエリー間のデータ再利用は限定的であることが分かりました。インテル® Optane™ パーシステント・メモリーをメモリーモードで使用しても、キャッシュとしての価値があまりないため、ここではアプリ・ダイレクト・モードを使用しました。

パフォーマンスを最大限に引き出すため、グラフをインテル® Optane™ パーシステント・メモリーに格納し、特徴ベクトルは DRAM に格納するようデータを整理しました。グラフのアクセスパターンは、非常にランダムで、非局所的です。インテル® Optane™ パーシステント・メモリーは 64 バイトのブロックサイズを持つため、パフォーマンスを維持したままグラフ要素にアクセスできます。ちょうど 4 回のアクセスで任意のノードの近傍を取得できるように、各ノードの最大出次数は 127 に制限します（近傍ごとに 4B と近傍数に 4B を使用します）。データを連続して保存することで、この 4 回のアクセスをパイプライン化できます。

Vamana アルゴリズムの最適化バージョンは、Julia（英語）プログラミング言語で記述されています。最適化は、一般的なもの（ソフトウェア・ベース）とインテル® Optane™ テクノロジー固有のものに分けることができます。一般的な最適化は、グラフとデータの表現の最適化、距離計算での VNNI 命令の使用、データベクトルの静的サイズ変更、メモリー・アライメントなどに関連しています。重要な最適化の 1 つは、距離計算ループをベクトルフェッチ・ループから切り離す「プリフェッチ・ホイスティング」です。このアプローチでは、距離計算のステップを開始する前に、x86 組込み命令を使用してメモリーから可能な限り多くのデータをプリフェッチします。これにより、メモリー・レイテンシーがクエリーに与える影響を最小限に抑えられます。もう 1 つの重要な最適化は、DRAM と PMem の間でベクトルを分割することで得られます。距離計算のベクトルのフェッチは検索で最も時間のかかるステップであり、DRAM に可能な限り多くのベクトルを保持することで、PMem トラフィックが軽減され、パフォーマンスが大幅に向上します。

マルチスレッド・アーキテクチャーは、ワーカースレッドに動的に負荷分散されるクエリーの小さなバッチを作成します。各スレッドは、バッチ内で一度に 1 つのクエリーを処理します。さらに、クエリー処理に必要なすべての中間スクラッチ空間のデータ構造は事前に割り当て、各スレッドは専用のプライベート・スクラッチ空間を所有します。これにより、クエリー処理中の動的なメモリー割り当てが不要になり、スレッド間の同期を最小限に抑えられます。

これらの最適化により、従来の最適解（GPU 上で動作する FAISS アルゴリズム）と比較して、ANN 検索パフォーマンスが 10 ~ 100 倍以上向上しました。**図 5** は、5 つの異なるデータセットで最適化したアプローチによって達成された改善を示しています。これらのデータセットには、異なるエンコーディング（Int8、UInt8、Float32）と、異なる距離メトリック（ユークリッドと内積）が含まれます。これらのデータセットにおいて、インテル® Xeon® プロセッサーとインテル® Optane™ メモリー上で GraphANN を実行すると、ベースライン・パフォーマンスが劇的に向上することが分かります。

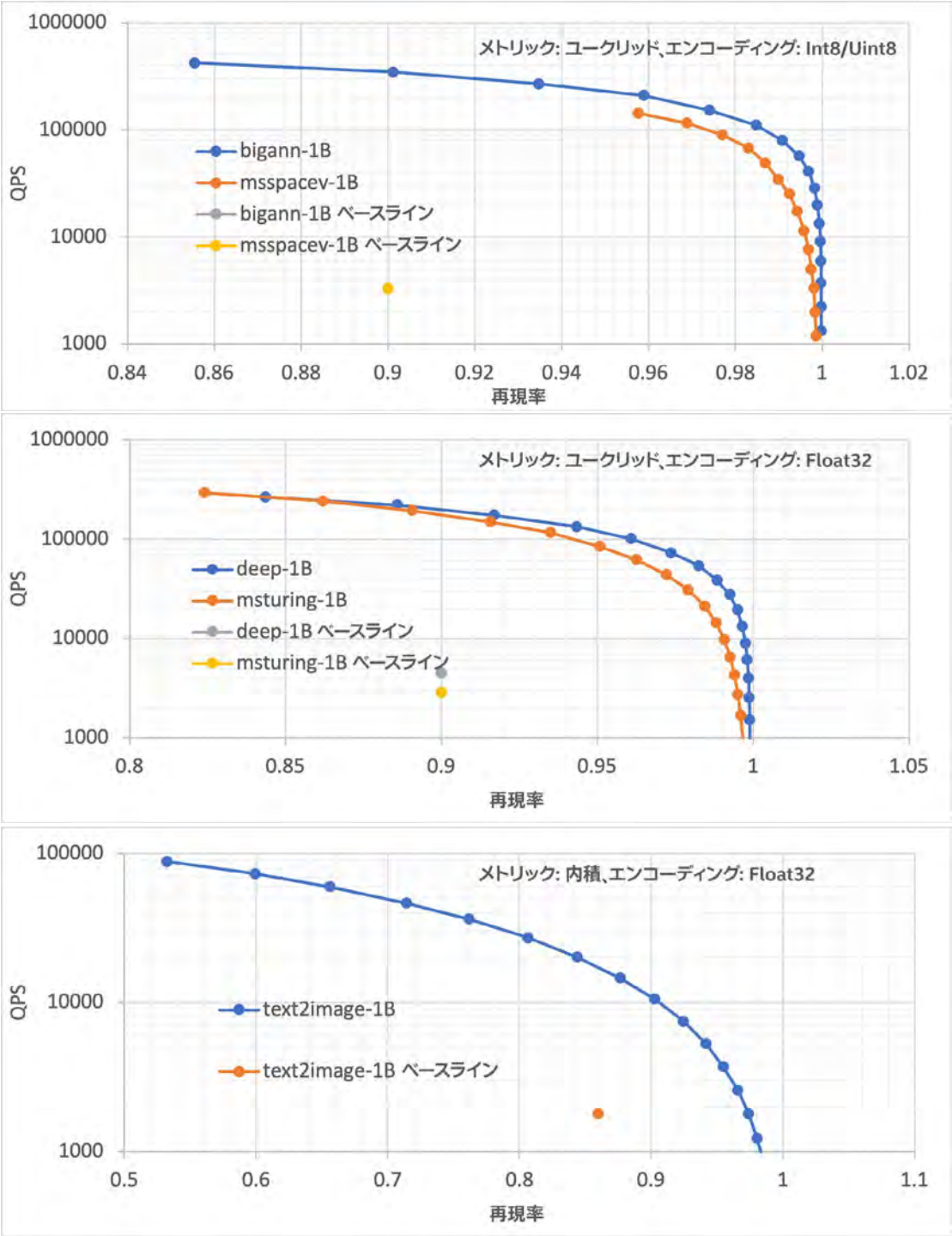


図 5. 5 つの異なるデータセットでの 1 秒当たりのクエリー数（スループット）と再現率—これまで最高のソフトウェア（FAISS）とハードウェア（GPU）に対して、インテル® Optane™ パーシステント・メモリーを使用したソリューションによる改善の程度を示している

まとめ

この記事では、NeurIPS 2021 の億単位の近似最近傍探索コンテストで優勝したアルゴリズム、設計上の選択、および関連ハードウェア設定について説明しました（コンテストのリーダーボードはこちら（英語）を参照してください）。また、関連するコードの変更を伴わないハードウェアのアップグレードから、コードの本格的なカスタム書き換えまで、さまざまな設計シナリオにおいて、インテル® Optane™ パーシステント・メモリーが類似性検索アルゴリズムのパフォーマンスを大幅に向上させることを示しました。

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 >

エンドツーエンドの AI パイプラインの最適化

インテル® Xeon® プロセッサーにおける AI パイプラインの最適化アプローチ

Meena Arunachalam インテル コーポレーション AI & アナリティクス・グループ 主席エンジニア
Vrushabh Sanghavi、Niroop Ammbashankar、および Fahim Mohammad 同シニア・ディープラーニング・ソフトウェア・エンジニア
Yi A Yao、Yi A Zhou、Lifeng A Wang、Zongru Wen、および Ning W Wang 同 AI フレームワーク・エンジニア

エンドツーエンド（E2E）の人工知能（AI）パイプラインは、特定のデータセットとモダリティーで問題を解決する 1 つ以上のマシンラーニング（ML)/ ディープラーニング（DL）モデルと、複数の前処理および後処理ステージで構成されています。DL/ML モデルのトレーニングと推論、最適化されたデータ取り込み、特徴エンジニアリング、メディアコーデック、トークン化などとともに、ML、自然言語処理（NLP)、推薦システム、ビデオ解析、異常検知、顔認識を行うさまざまな最新の AI パイプラインに包括的な最適化アプローチを適用し、E2E パフォーマンスを向上させます。対象パイプラインすべて（ほとんどは推論ベース）にわたる結果から、最適な E2E スループットを得るには、すべてのフェーズを最適化する必要があることが分かります。大容量メモリー、AI アクセラレーション（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）（英語）など)、汎用コードの実行に対応しているインテル® Xeon® プロセッサーは、これらのパイプラインに適しています。

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

最適化の内容は、(1) アプリケーション、フレームワーク、およびライブラリー・ソフトウェア、(2) モデルのハイパーパラメーター、(3) モデルの最適化、(4) システムレベルのチューニング、および (5) ワークロードの分割、に分けることができます。インテル® Neural Compressor（英語）などのツールは、量子化、蒸留、プルーニング、そしてインテル® DL ブーストやインテル® Xeon® プロセッサーに搭載されているほかの AI アクセラレーションを利用できるようにします。これにより、さまざまな E2E パイプラインで 1.8 ~ 81.7 倍のスピードアップを実現します。さらに、1 つまたは非常に限られた数の並列ストリームしかホストできないメモリー制限のあるアクセラレーターと比較して、インテル® Xeon® プロセッサーの高いコア数とメモリー容量により、これらのパイプラインの複数の並列ストリームまたはインスタンスをホストできます。多くの場合、ワークロードを CPU に統合可能であり、TCO と消費電力の面でもメリットがあります。

E2E AI アプリケーション

インテルでは、多くの E2E AI ユースケースとワークロードを紹介しています。それぞれが独自の前処理 / 後処理ステージを含み、ビデオ、画像、表、テキスト、その他のデータタイプに対して、異なる ML/DL アプローチを使用して実装されています（**表 1**）。

ワークロード	アプリケーション名	モデル	前処理 / 後処理ステージ	データセット
ML	米国国勢調査	リッジ回帰	データフレームへのデータのロード、列の削除、行の削除、演算処理、型変換、トレーニング / テスト分割	IPUM の米国国勢調査のデータ
	PLAsTiCC	勾配ブースティング木	データのロード、列の削除、グループ別集計、演算処理、型変換、トレーニング / テスト分割	LSST シミュレーション・データ
	産業用 IoT における予測分析	ランダムフォレスト分類器	データフレームへのデータのロード、不要な列の削除、トレーニング / テスト分割	Bosch* 製造ライン
NLP	文書レベルの感情分析	BERT-Large	データのロード、トークナイザーの初期化、データエンコード、モデルのロード	IMDb
				SST-2
推奨システム	E2E 深層心理進化ネットワーク	DIEN	データの取り込み、ラベル·エンコーディング、履歴シーケンスの取得、ネガティブ・サンプリング、データ分割、モデルのロード	Amazon Books
ビデオ解析	ビデオ・ストリーマー	SSD Resnet-34	ビデオデコード、画像の正規化とリサイズ、境界ボックスとラベル、データのアップロード	モールのビデオ
異常検出	異常検出	ResNet50v1.5	データのロード、画像のリサイズ、画像変換、特徴再構成誤差の評価	MVTec AD
顔認識	顔の検出と認識	SSD MobileNet、Resnet50v1.5	ビデオのロード、フレーム分割、リサイズ、出力の生成	サッカーの祝賀会

表 1. E2E AI アプリケーション

E2E AI アプリケーションは、通常、大きく分けて前処理 / 後処理と AI の 2 つの処理を行います。**図 1** は、E2E の総実行時間に占めるそれぞれの処理の割合を示しています。前処理 / 後処理は 4 ~ 98% で、AI は 2 ~ 96% です。

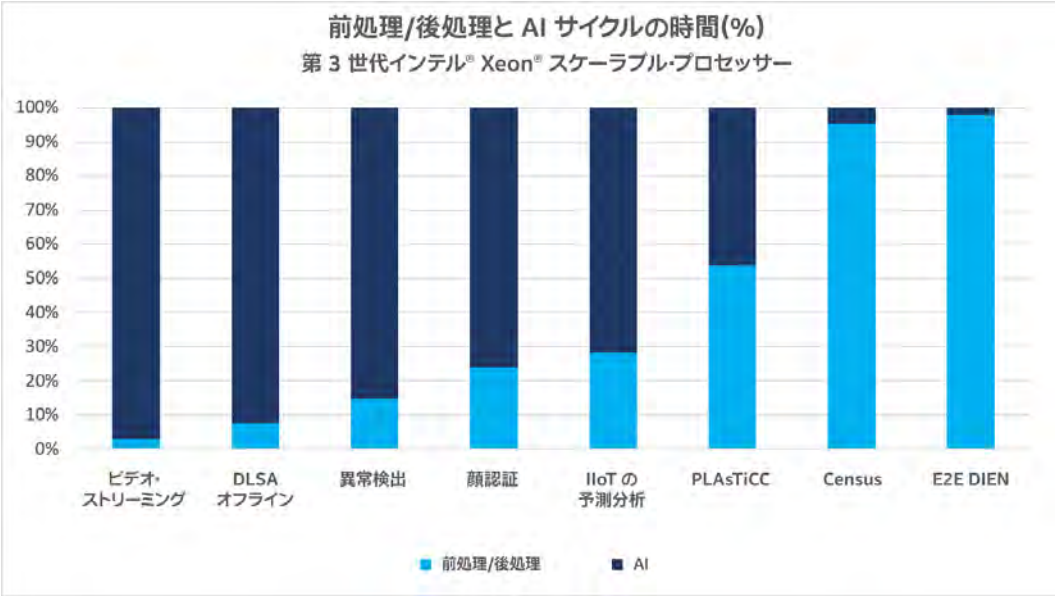


図 1. 前処理 / 後処理と AI の時間（%）

米国国勢調査

Census（米国国勢調査）ワークロードは、1970 年から 2010 年までの米国国勢調査のデータを用いて、リッジ回帰モデルをトレーニングし、個人の教育水準と所得の相関を予測します（**図 2**）。ML に先立ち、データを取り込み、データフレーム操作を行い、モデルのトレーニング用の入力を準備し、特徴セットとそれに続く出力セットを作成します[1]。

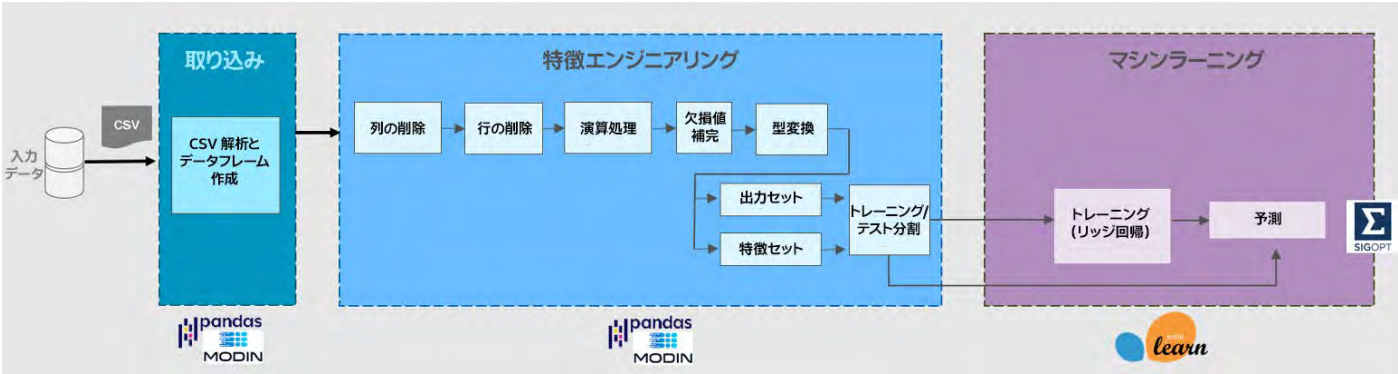


図 2. 米国国勢調査アプリケーションのパイプライン

PLAsTiCC

PLAsTiCC は、シミュレーションされた天文時系列データを使用して、明るさの異なる天体を分類するオープン・データ・チャレンジです（**図 3**）。パイプラインは、データを読み込み、データフレームの操作、変換、処理を行い、XGBoost ライブラリーのヒストグラム・ツリー法を使用して分類器のトレーニングとモデルの推論を行います。

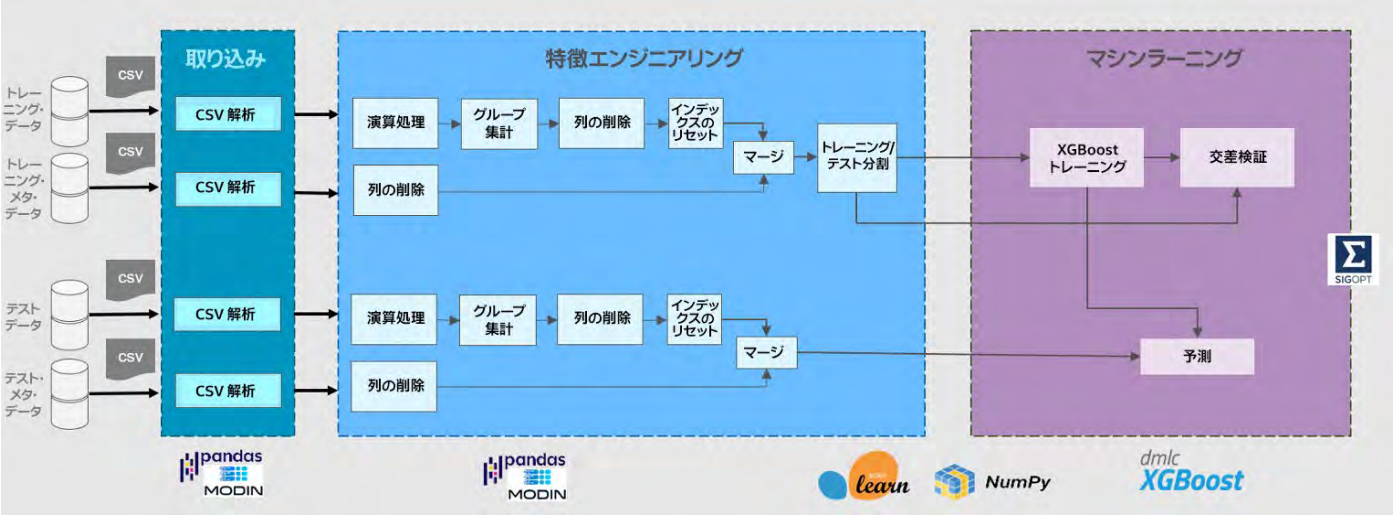


図 3. PLAsTiCC アプリケーションのパイプライン

産業用 IoT における予測分析

これは産業用 IoT における E2E 教師なし学習のユースケースで、製造時の内部不良を予測することで、生産ラインの品質とパフォーマンスの維持に貢献します（**図 4**）。ワークフローは、CSV ファイルから計測値を読み込み、必要な特徴量のみを含むようにクリーニングします。このステップでは、高度に最適化された[インテル® ディストリビューションの Modin（英語）](#)[2] が使用されます。モデルの生成には、[scikit-learn* 向けインテル® エクステンション（英語）](#)[3] が使用されます。

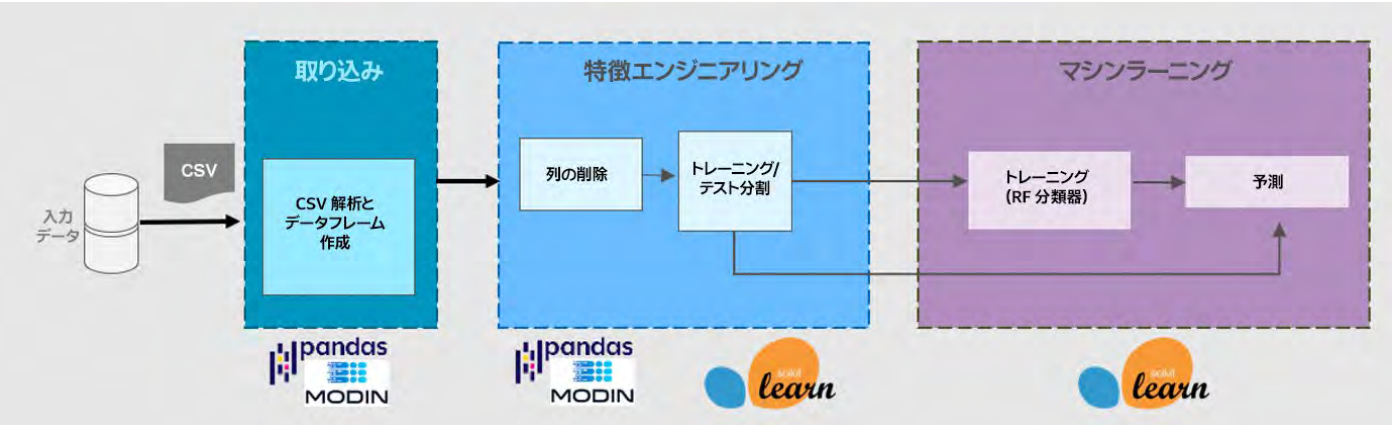


図 4. 産業用 IoT における予測分析のパイプライン

文書レベルの感情分析（DLSA）

図 5 に示す DLSA ワークフローは、Hugging Face 変換 API を使用して構築された、文書レベルの感情分析を実行するリファレンス NLP パイプラインです。これは、大規模な英語テキストコーパスで事前トレーニング済みの BERT-LARGE（大文字小文字を区別しない）のような言語モデルを使用します。

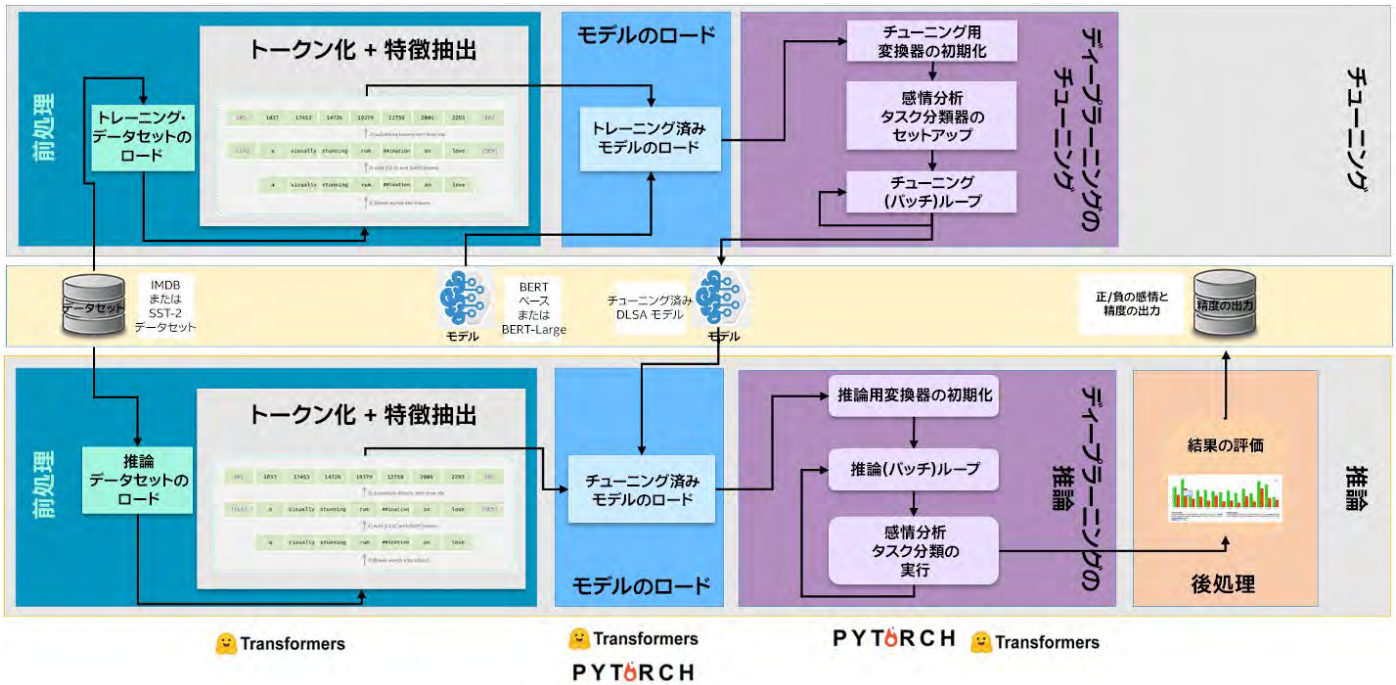


図 5. 文書レベルの感情分析のパイプライン

E2E 深層心理進化ネットワーク（DIEN）推奨システム

図 6 に示す DIEN ワークフローは、ユーザーのクリック確率を大規模に推定する推薦推論パイプラインです。

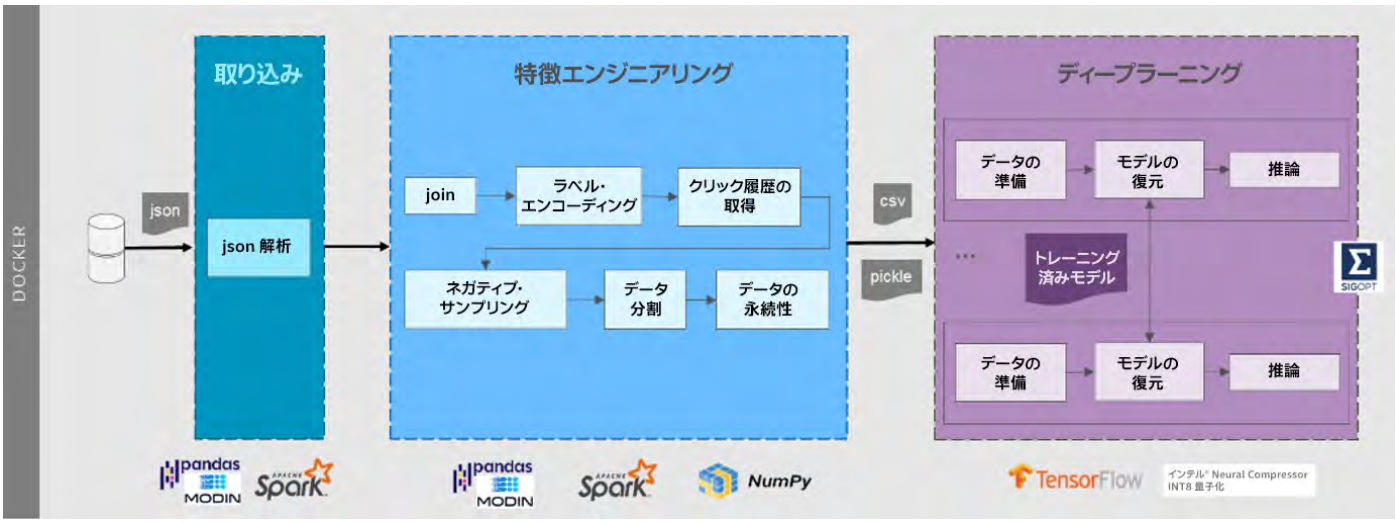


図 6. E2E DIEN 推奨システムのパイプライン

ビデオ・ストリーマー

ビデオ・ストリーマー・パイプライン（**図 7**）は、リアルタイム・ビデオ解析を模倣して設計されています。リアルタイム・データは、シングルショットのオブジェクト検出を実行する推論エンドポイントに提供されます。推論中に作成されたメタデータは、データベースにアップロードされ、キュレーションされます。パイプラインは GStreamer、TensorFlow*、OpenCV* をベースに構築されています。入力ビデオは、GStreamer によってフレーム単位で画像にデコードされます。その後、GStreamer のバッファーは NumPy* 配列に変換されます。TensorFlow* は画像の正規化とリサイズを行い、その後、事前トレーニング済みの SSD-ResNet34 モデルでオブジェクト検出を行います。最後に、境界ボックス座標とクラスラベルの結果がデータベースにアップロードされます。

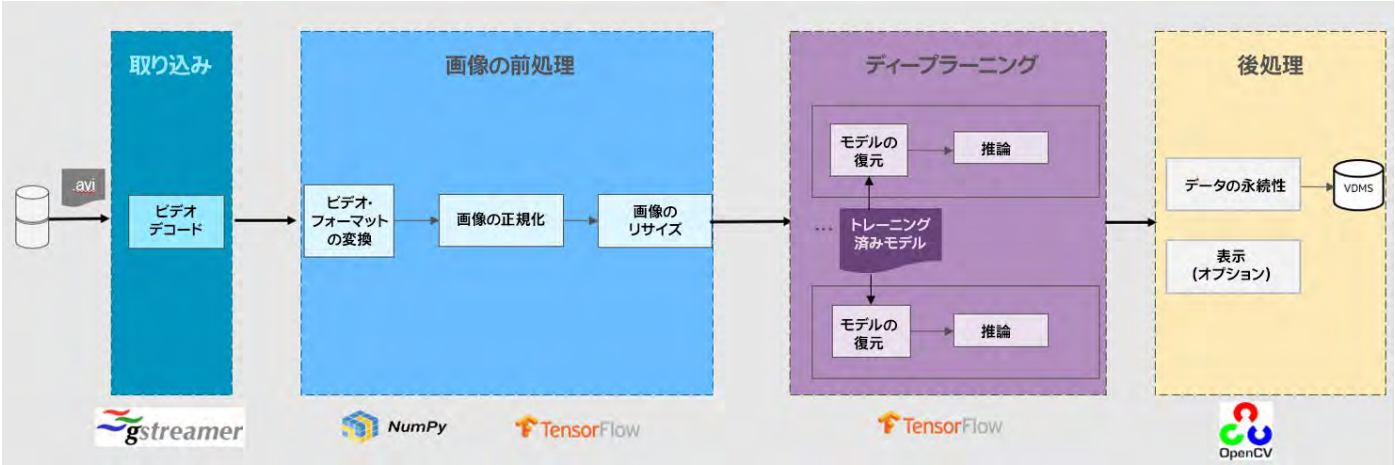


図 7. ビデオ・ストリーマー・アプリケーションのパイプライン

異常検出

異常検出の目的は、工業生産ラインで製造される部品の画像を、ディープ・ニューラル・ネットワークと確率モデリングを使用して解析し、まれに発生する不良品を見つけることです（**図 8**）。分布から外れるものを見つけるため、正常なデータのいくつかの最終層の特徴マップから教師なし方式で正規性のモデルを学習し、モデルからの逸脱に対し異常としてフラグを立てます。分布を学習する前に、PCA を使用して特徴空間の次元を下げ、分布のパラメーターを推定する際に行列の特異点やランク不足が発生しないようにします。

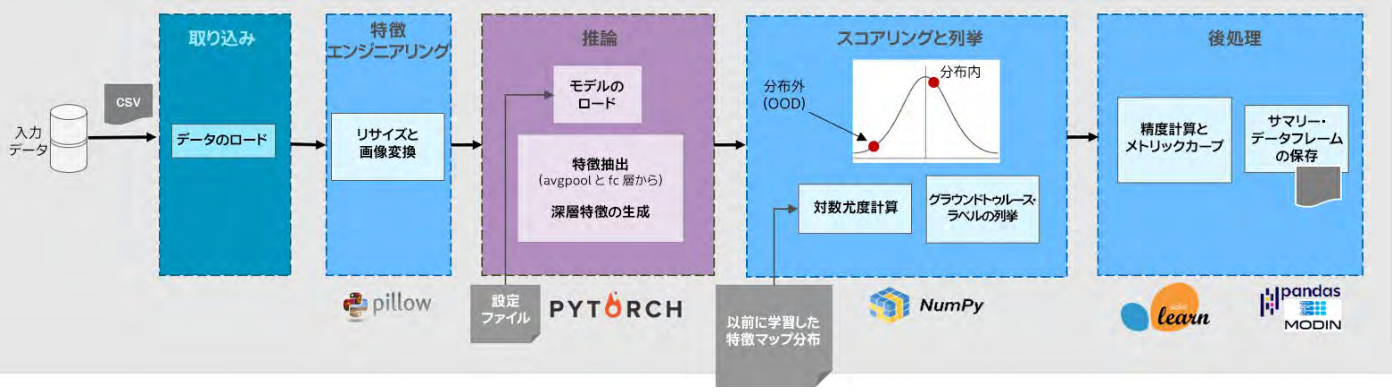


図 8. 異常検出のパイプライン

顔認識

この E2E パイプラインは、すぐに使える 2 つの事前トレーニング済みモデルをつないで、リアルタイムの顔認識を実現します（**図 9**）。カメラからの入力は、圧縮または非圧縮ビデオとして、フレーム分割とリサイズが行われます。各フレームは検出モデル（SSD MobileNet）に送られ、物体検出が行わます。そして、NMS 境界ボックスを認識モデル（ResNet50v1.5）に渡して、顔を認識します。顔認識された出力フレームは、表示するか、データベースに保存できます。

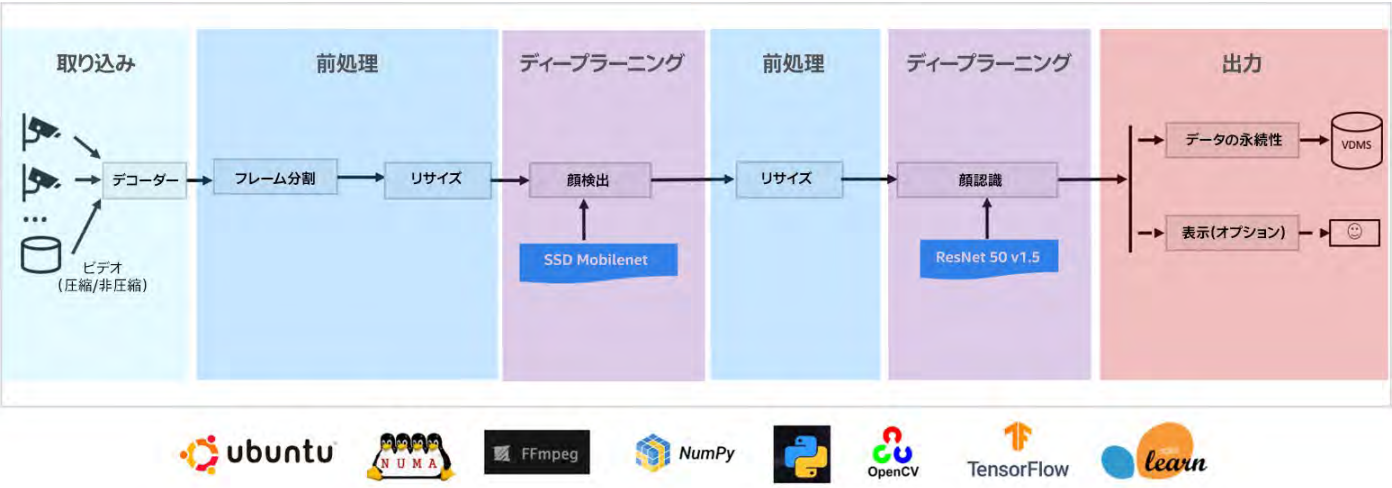


図 9. 顔認識のパイプライン

「効率良い AI」の実現 : E2E 最適化アプローチ

E2E のパフォーマンス効率に優れた AI には、AI ソフトウェアの高速化、システムレベルのチューニング、ハイパーパラメーターとランタイム・パラメーターの最適化、およびワークフローのスケーリングからなる首尾一貫した最適化アプローチが必要です。ユーザーの生産性とワークロードのパフォーマンス効率を向上させるには、すべてのフェーズ（データの取り込み、データの前処理、特徴エンジニアリング、モデル構築）に総合的に対処する必要があります（**図 10**）。

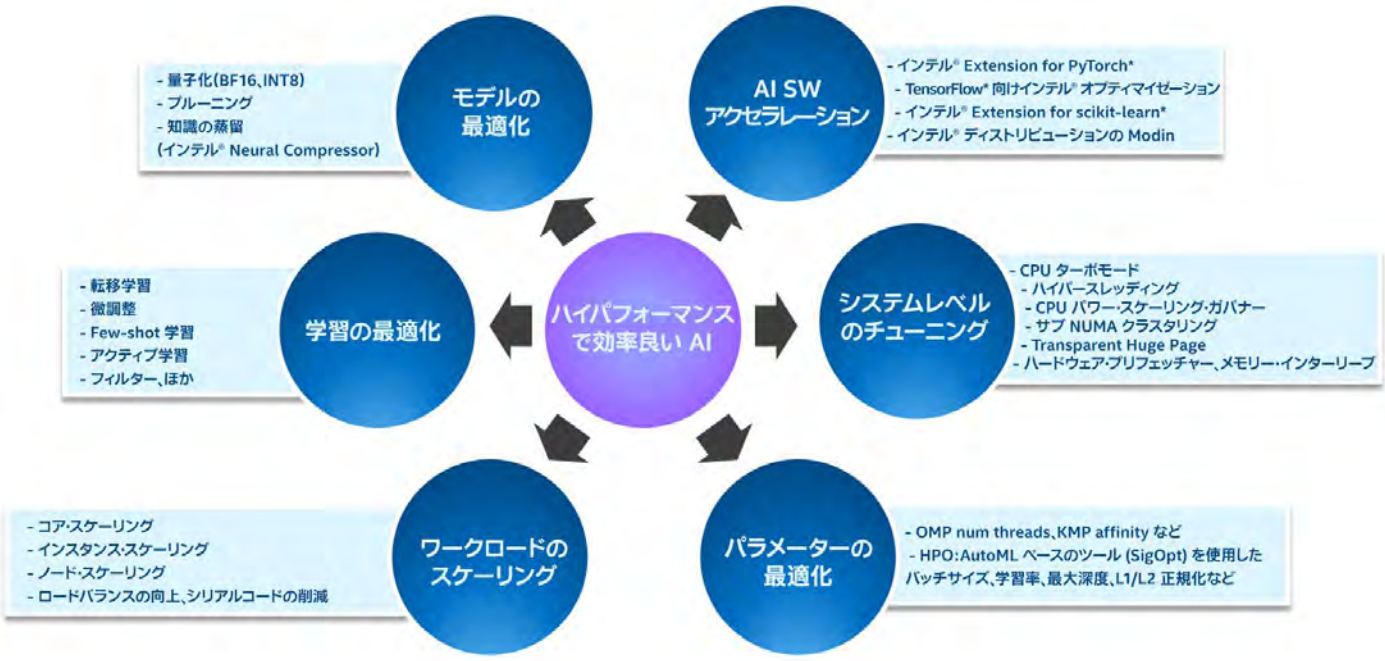


図 10. 効率良い AI

AI ソフトウェアの高速化

インテル® ディストリビューションの Modin は、pandas* API と互換性のあるハイパフォーマンスなマルチスレッドの並列データ・フレーム・ライブラリーです。軽量で堅牢なデータフレームと CSV の操作を行い、pandas* とは異なり、コア数に応じて効率良くスケーリングし、数行のコードを変更するだけで大幅なスピードアップを実現します。異なるフェーズのデータフレーム操作が 1.12 ~ 30 倍高速化されます。

scikit-learn* 向けインテル® エクステンションは、古典的な ML における一般的な推定器、変換器、クラスタリング・アルゴリズムを高速化します。米国国勢調査ワークロードにおけるリッジ回帰のトレーニングと推論は、DGEMM ベースのメモリー依存アルゴリズムであり、scikit-learn* 向けインテル® エクステンションのベクトル化、キャッシュフレンドリーなブロッキング、マルチスレッドを利用して複数の CPU コアを効率良く使用します。

インテルにより最適化された XGBoost と CatBoost ライブラリーは、効率良い並列ツリー・ブースティングを提供します。XGBoost カーネルは、インテル® プロセッサー上でのキャッシュ効率、リモート・メモリー・レイテンシー、メモリー・アクセス・パターン向けに最適化されています。

PyTorch* 向けインテル® エクステンション（英語）[4] は、インテル® プロセッサー上で PyTorch* のパフォーマンスを向上します。PyTorch* 向けインテル® エクステンションを使用すると、異常検出パイプラインと DLSA パイプラインでインテル® DL ブーストを活用できます。TensorFlow* 向けインテル® オプティマイゼーション[5] は、インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー（インテル® oneDNN）を使用しており、畳み込み、正規化、活性化、内積、その他のプリミティブがインテル® AVX-512 命令でベクトル化されています。DIEN、顔認識、およびビデオ・ストリーマー・アプリケーションは、TensorFlow* 向けインテル® オプティマイゼーションを使用して、ベクトル化と最適化されたグラフ操作（操作の融合やバッチ正規化など）により、インテル® プロセッサー上でスケーラブルなパフォーマンスを実現します。

モデルの最適化

量子化により、高精度データ（32 ビット浮動小数点、FP32）から低精度（8 ビット整数、INT8）への変換が容易になり、畳み込みや行列乗算などの重要な操作において、精度をほとんど落とさずに大幅な高速化が可能になります。インテル® Neural Compressor は、DL モデルの低精度レシピを自動的に最適化し、期待される精度条件で最適なパフォーマンスとメモリー使用量を達成するように調整します。DLSA とビデオ・ストリーマー・アプリケーションでは、INT8 量子化だけで最大 4 倍のスピードアップを達成しました（**表 2**）。

パラメーターの最適化

SigOpt モデル開発プラットフォームでは、しきい値の精度やレイテンシー・レベルでの最大スループットなどの目的に合わせたチューニングを行いながら、あらゆるパイプラインの実行追跡、トレーニングの可視化、ハイパーパラメーター最適化のスケーリングを容易に行うことができます。SigOpt の多目的の最適化により、AI パイプラインの構成に関する情報を簡単に収集して、最適なパフォーマンスの要約と解析を表示できます。PLAsTiCC では、「精度」と「タイミング」のメトリックを最適化し、目的を達成するため、モデルのハイパーパラメーター（XGBoost の並列スレッド数、木の本数、学習率、最大深度、L1/L2 正規化など）を計算します[6]。DLSA では、推論インスタンスの数とバッチサイズをチューニングすることで、高い E2E スループットを実現します。

TensorFlow* のランタイムオプションもパフォーマンスに大きな影響を与えます。行列乗算やリダクションなどの操作では、`intra_op_parallelism_threads` を利用可能な物理コアの数と同じに設定して、タスクをスレッドプール内にスケジュールするように並列性を制御し、一方で `inter_op_parallelism_threads` をソケット数と同じに設定して TensorFlow* グラフ内で独立している処理を同時実行することが推奨されます。また、データレイアウト、OpenMP*、NUMA の制御も可能で、さらにパフォーマンスをチューニングできます[5]。

ワークロードのスケーリング

マルチインスタンス実行により、1 台のインテル® Xeon® スケーラブル・プロセッサー・ベースのサーバーでアプリケーションの並列ストリームを実行できます。この利点は、製造パイプラインのさまざまな段階で不良を検出するため複数のカメラを配置する異常検出において実証されます。ResNet50 モデルで標準 30FPS を超える処理を行う 10 個のストリームは、単一の第 3 世代インテル® Xeon® スケーラブル・プロセッサーで処理できます。同様に、E2E DIEN はソケットあたり 1 コア / インスタンス、40 推論インスタンスで動作し、DLSA と DL パイプラインはソケットあたり 4 ~ 8 コア / インスタンス、10 ~ 5 推論ストリームで動作します。これは大容量メモリーを持つ CPU ならではの利点です。

BIOS でシステムレベルのチューニングを行うことで、効率を向上できます。最高のパフォーマンスを得るため、ハイパースレッディング、CPU パワー・スケーリング・ガバナー、NUMA 最適化、ハードウェア・プリフェッチャーなどを制御するチューニング設定を利用できます。

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

	インテル® ディストリビューションの Modin	scikit-learn* 向けインテル® エクステンション	XGBoost	PyTorch* 向けインテル® エクステンション	TensorFlow* 向けインテル® オプティマイゼーション	INT8 量子化
米国国勢調査	6x	59x	-	-	-	-
PLAsTiCC	30x	8x	1x	-	-	-
産業用 IoT における予測分析	4.8x	113x	-	-	-	-
文書レベルの感情分析	-	-	-	4.15x	-	3.90x
E2E 深層心理進化ネットワーク	23.2x	-	-	-	9.82x	-
ビデオ・ストリーマー	-	-	-	-	1.36x	3.64x
異常検出	1.12x	3.4x	-	1.8x	-	-
顔認識	-	-	-	-	1.7x	-

表 2. E2E AI アプリケーションのソフトウェア最適化と量子化によるパフォーマンスの向上

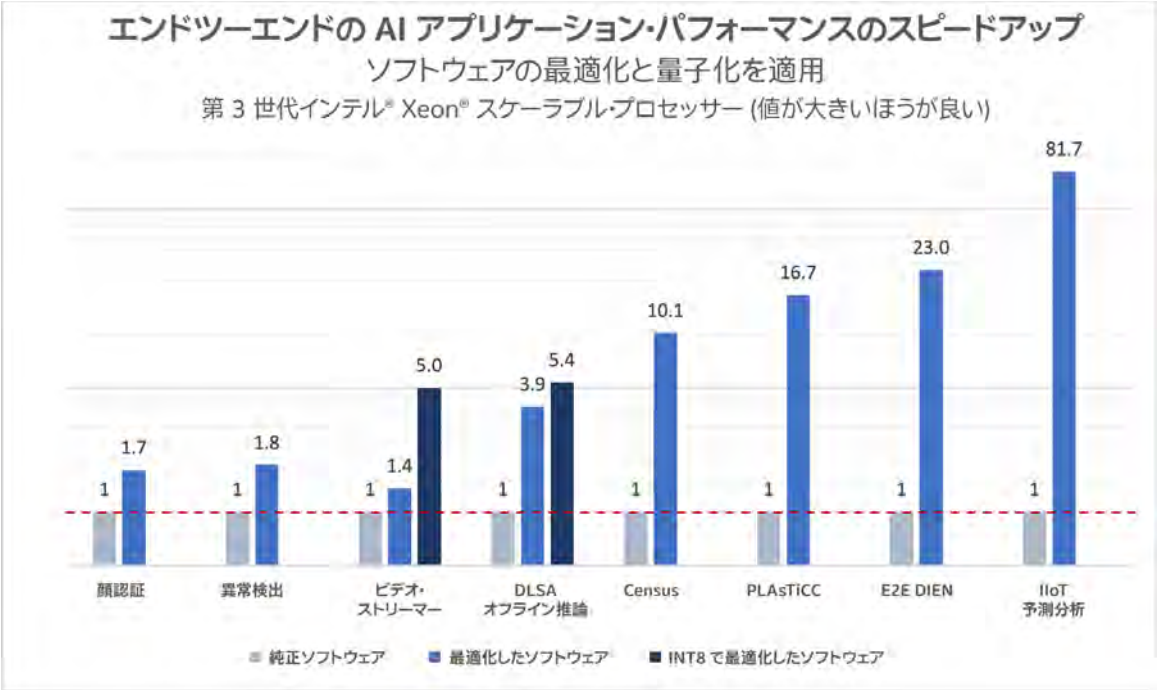


図 11. E2E AI アプリケーションのパフォーマンスの向上

システム構成：シングルノード、デュアルソケットの第 3 世代インテル® Xeon® Platinum 8380 プロセッサー、ソケット当たり 40 コアでパフォーマンスを測定（DIEN と DLSA を除く）。DIEN と DLSA は、第 3 世代インテル® Xeon® Gold 6348 プロセッサー、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・テクノロジー無効、BIOS:SE5C620.86B.01.01.0003.2104260124、カーネル:5.13.0-28-generic、OS：Ubuntu* 21.10、メモリー 512GB（16 スロット /32GB DIMMs/3200MHz）、OS ドライブ：インテル® SSD 480GB。

異常検出：Python* 3.7.11、torch 1.11.0、torchvision 0.11.3、PyTorch* 1.10、NumPy* 1.22.1、pandas* 1.3.5、scikit-learn-intelex 2021.4.0。
顔認識：Python* 3.7.9、TensorFlow* 2.8.0、NumPy* 1.22.2、opencv-python 4.5.3.56、ffmpy 0.3.0。
ビデオ・ストリーマー：Python* 3.8.12、TensorFlow* 2.8.0、opencv-python 4.5.2.54、pillow 8.3.1、gstreamer1.0、vdms 0.0.16。
DLSA オフライン推論：Python* 3.7.11、PyTorch* 1.10、Hugging Face Transformer 4.6.1。
E2E DIEN：Python* 3.8.10、Modin 0.12.0、TensorFlow* 2.8.0、NumPy* 1.22.2。
米国国勢調査：Python* 3.9.7、Modin 0.12.0、scikit-learn-intelex 2021.4.0。
PLAsTiCC：Python* 3.9.7、Modin 0.12.0、scikit-learn-intelex 2021.4.0、XGBoost 1.5.0。
産業用 IoT における予測分析：Python* 3.9.7、Modin 0.12.0、scikit-learn-intelex 2021.4.0。

結論として、ソフトウェア、システム、ハードウェア、モデル構築、ハイパーパラメーターにわたる最適化アプローチを組み合わせた結果、インテル® Xeon® プロセッサーにおける E2E パフォーマンスが 1.8 ~ 81.7 倍向上しました。

参考資料（英語）

1. インテル® oneAPI AI アナリティクス・ツールキット用の米国国勢調査ワークロード・サンプル：https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/End-to-end-Workloads/Census
2. インテル® ディストリビューションの Modin：https://www.intel.com/content/www/us/en/developer/tools/oneapi/distribution-of-modin.html
3. scikit-learn* 向けインテル® エクステンション導入ガイド：https://www.intel.com/content/www/us/en/developer/articles/guide/intel-extension-for-scikit-learn-getting-started.html
4. PyTorch* 向けインテル® エクステンション：https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-intel-optimization-of-pytorch.html
5. CPU 上で TensorFlow* パフォーマンスを最大化する：https://www.intel.com/content/www/us/en/developer/articles/technical/maximize-tensorflow-performance-on-cpu-considerations-and-recommendations-for-inference.html
6. AI アプリケーションの最適化：ハイパーパラメーターのチューニングと最適化されたソフトウェアにより AI パフォーマンスを向上：https://medium.com/intel-analytics-software/optimizing-artificial-intelligence-applications-1bc22b5d707b

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

AI アプリケーションの最適化

ハイパーパラメーターのチューニングと最適化されたソフトウェアにより AI パフォーマンスを向上

Vrushabh Sanghavi インテル コーポレーション シニア・ディープラーニング・ソフトウェア・エンジニア

データ・サイエンティストは、AI アプリケーションのパフォーマンスを向上させる方法を常に探しています。純正のパッケージではなく、最適化されたマシンラーニング・ソフトウェアを使用することは、これを実現する簡単な方法です。SigOpt（英語）のような AutoML ベースのプラットフォームを使用してモデルのハイパーパラメーターをチューニングするのもその 1 つです。ここでは、Kaggle の PLAsTiCC Classification Challenge（英語）を使用して、パフォーマンスの可能性について説明します。

PLAsTiCC は、等級の異なる天体を分類するオープン・データ・チャレンジです。チリ北部に設置される大型可視光赤外線望遠鏡による観測に備え、シミュレーションされた天文時系列データを使用します。このチャレンジでは、各天体が 14 クラスの天文フィルターのいずれかに属する確率を、小さなトレーニング・セット（140 万行）から非常に大きなテストセット（1 億 8,900 万行）へとスケーリングして決定します。

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

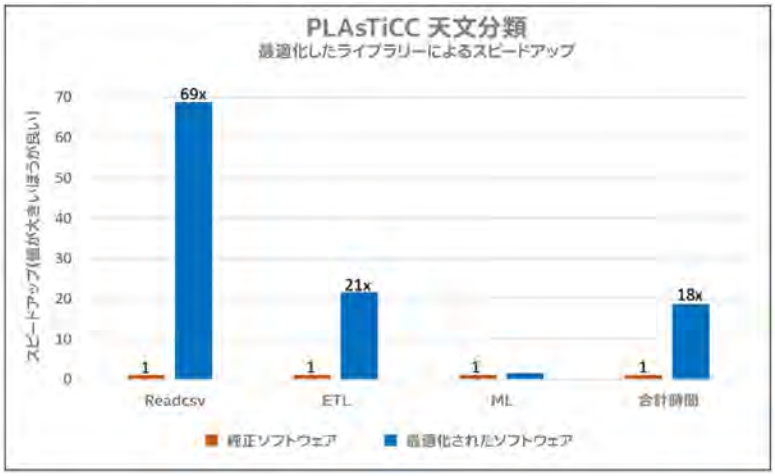
このコードは 3 つのフェーズに分けられます。

1. **Readcsv:**CSV 形式のトレーニング･データとテストデータ、およびそれに対応するメタデータを pandas* のデータフレームに読み込みます。
2. **ETL :** トレーニング･アルゴリズムに入力するデータフレームを操作、変換、処理します。
3. **ML :** XGBoost ライブラリーのヒストグラム･ツリー法を使用して分類モデルをトレーニングします。モデルは交差検証され、トレーニング済みモデルは膨大なテストセット内のオブジェクトの分類に使用されます。

下図は、これらの各フェーズで使用される純正および最適化されたソフトウェアと、ハイパーパラメーターのチューニングに使用する SigOpt を示したものです。

インテル® ディストリビューションの Modin（英語）は、Readcsv と ETL のパフォーマンスを向上させるために使用されます。この並列分散データフレーム･ライブラリーは、pandas* API を使用します。1 行のコードを変更するだけで、データフレーム操作のパフォーマンスを大幅に向上できます。PLAsTiCC ML のパフォーマンスを向上させるため、インテル® アーキテクチャー向けに最適化された XGBoost パッケージがメインブランチにアップストリームされています。これは、XGBoost の最新バージョン（英語）をインストールすると利用できます（「Distributed XGBoost with Modin on Ray（Ray 上での Modin を使用した分散 XGBoost）」（英語）を参照してください）。

以下の棒グラフは、PLAsTiCC の各フェーズにおいて、最適化されたソフトウェア･スタック（青色で表示）を使用した場合、純正ソフトウェア（オレンジ色で表示）よりも高速化されることを示しています。最適化されたソフトウェアを使用することで、エンドツーエンドで 18 倍という大幅なスピードアップを達成しています。インテル® ディストリビューションの Modin は、pandas* とは異なり、軽量かつ堅牢なデータフレームと Readcsv の操作を実行し、コア数に応じて効率良くスケーリングします。XGBoost カーネルは、インテル® アーキテクチャー上でキャッシュ効率、リモート･メモリー･レイテンシー、メモリー･アクセス･パターンを最適化し、高いプロセッサー周波数、キャッシュサイズ、キャッシュ帯域幅を最適に使用します。



マシンラーニング・モデルのハイパーパラメーターを調整することで、エンドツーエンドのワークロードのパフォーマンスをさらに向上できます。SigOpt は、これを簡単に実現するモデル開発用プラットフォームです。トレーニング実験を追跡し、トレーニングを可視化するツールを提供し、あらゆるタイプのモデルに対してハイパーパラメーターの最適化をスケーリングします。

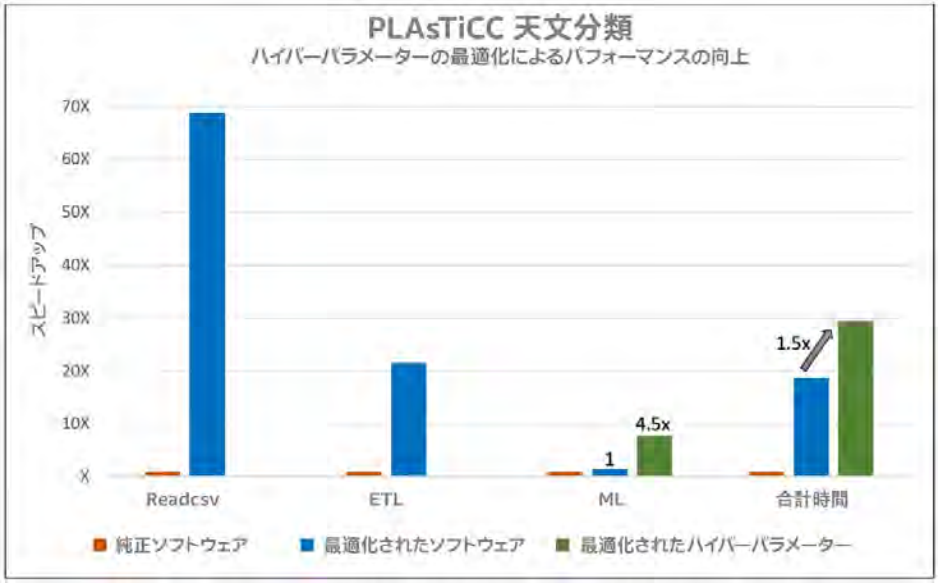
SigOpt はモデルの最適なパラメーター値を見つけ、最適化ループの中で定義されているメトリック全体を最適化します。PLAsTiCC では、**精度**と**タイミング**のメトリックを最適化し、目的を達成するため、モデルのハイパーパラメーター（XGBoost の並列スレッド数、木の本数、学習率、最大深度、L1/L2 正規化など）を計算します。目的関数の大域的な最大値または最小値を求めるには、最小限の観測を実行する必要があり、実験回数を実験中のパラメーター数の 10 ~ 20 倍に設定すると、ほとんどの場合、収束が見られます。

次の表は、デフォルトのモデル・パラメーターと、SigOpt AutoML の実験によって計算されたチューニング・パラメーターを示しています。

パラメーター	デフォルト	チューニング済み
nthread	56	48
subsample	0.92	0.54
colsample_bytree	0.47	0.75
nround	43	18
max_depth	1	2
max_bin	38	60
min_child_weight	9.80	0.80
learning_rate (eta)	0.93	0.76
gamma	0.90	0.65
(reg_) alpha	0.38	0.39
(reg_) lambda	0.75	0.54
scale_pos_weight	7	6

手動でこれらの順列をすべてチューニングして実行することはほとんど不可能ですが、SigOpt は数時間でそれを行うことができます。モデルの対数損失と検証損失は増加していないため、これらの改善はモデルの精度に影響を与えることなく達成されています。

以下のグラフでは、SigOpt のハイパーパラメーター・チューニングにより ML パフォーマンスがさらに 5.4 倍向上し、全体のパフォーマンスが 1.5 倍向上したことを示すため、前のグラフに今回の結果を追加しています。



典型的なエンドツーエンド・パイプラインで行ったこれらの手順は、インテルにより最適化されたさまざまなソフトウェア・パッケージ、ライブラリー、および最適化ツールを使用することで、AI ワークロードのパフォーマンスを大幅に向上できることを示しています（詳細は、「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」を参照してください）。

ハードウェアとソフトウェアの構成

ハードウェア：2x インテル® Xeon® Platinum 8280L プロセッサー（28 コア）、OS : Ubuntu* 20.04.1 LTS Mitigated、384GB RAM（384GB RAM: 12 x 32GB 2933MHz）、カーネル : 5.4.0–65-generic、マイクロコード: 0x4003003、CPU ガバナー:performance。**ソフトウェア:**scikit-learn* 0.24.1、pandas* 1.2.2、XGBoost 1.3.3、Python* 3.9.7、scikit-learn-intelex 2021.2、modin 0.8.3、omniscidbe v5.4.1。

SYCL* 2020 で追加された 5 つの優れた機能

SYCL* プログラミング言語が進化

James Brodman インテル コーポレーション 主席エンジニア
John Pennycook インテル コーポレーション ソフトウェア・イネーブリング & 最適化アーキテクト

SYCL* 2020 は、アクセラレーターの活用を考えている C++ プログラマーにとって、エキサイティングなアップデートです。私たち 2 人は、SYCL* 仕様、SYCL* に関する書籍(英語)、SYCL* を LLVM に実装する DPC++ オープンソース・プロジェクト (英語) に貢献する機会に恵まれました。ここでは、SYCL* 2020 (英語) で追加された新機能の中で、私たちが特に気に入っているものを紹介します。これは、Khronos を代表してではなく、インテルのエンジニアである私たちの意見として提供します。

SYCL*

SYCL* は、C++ にヘテロジニアス・プログラミングのサポートをもたらす Khronos の標準規格です。2020 年末に SYCL* 2020 仕様が確定し、それ以降、コンパイラーのサポートが拡大しています（実装については、Khronos のウェブサイト（英語）を参照してください)。

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* については、「C++ のヘテロジニアスな将来の考察」や、sycl.tech（英語）に掲載されている多数のリソースなど、多くの場所で取り上げられています。簡単に言えば、SYCL* は、「C++ でヘテロジニアス・プログラミングを可能にし、ベンダーやアーキテクチャーを超えた移植性を実現するにはどうすればよいか？」という重要な課題に取り組んでいます。

コミュニティーからの強力なフィードバックにより、SYCL* 2020 は、さまざまなマルチベンダーとマルチアーキテクチャーに対応したエキサイティングな新機能を備えています。この記事では、これらの新機能とその目的について説明します。

5 つの優れた機能

SYCL* 2020 の重要な目標は、SYCL* を ISO C++ と整合させることであり、これには 2 つの利点があります。1 つは、C++ プログラマーにとって SYCL* が自然であることを保証します。もう 1 つは、SYCL* が、ほかの C++ ライブラリーや ISO C++ 自体にも影響を与える、ヘテロジニアス・プログラミングに対するマルチベンダー、マルチアーキテクチャー・ソリューションとして機能できるようになります。

SYCL* 2020 の構文変更の多くは、ベース言語を C++11 から C++17 へ更新したことに伴うもので、開発者はクラス・テンプレート引数推定（CTAD）（英語）や推定ガイドなどの機能を利用できるようになりました。しかし、多くの新機能も追加されています。この記事では、SYCL* 2020 の新機能の中から 5 つを取り上げ、それらが重要な理由を説明します。

1. **バックエンド**は、OpenCL* 以外の言語 / フレームワークで構築された SYCL* 実装への扉を開き、SYCL* がより多様なハードウェアをターゲットにすることを可能にします。
2. **統合共有メモリー（USM）**は、ポインターベースのアクセスモデルで、SYCL* 1.2.1 のバッファー / アクセサーモデルの代替となるものです。
3. **リダクション**は、一般的なプログラミング・パターンであり、SYCL* 2020 は「組込み」ライブラリーによってこれを高速化します。
4. **グループ・ライブラリー**は、協調的なワークアイテムの抽象化を提供し、（ベンダーに関係なく）ハードウェア機能と整合性を取ることで、アプリケーションのパフォーマンスとプログラマーの生産性を向上させます。
5. **アトミック参照**は C++20 の `std::atomic_ref` と整合性があり、C++ メモリーモデルをヘテロジニアス・デバイスに拡張します。

これらの機能は、オープンで、マルチベンダー、かつマルチアーキテクチャーの SYCL* エコシステムを確立し、C++ プログラマーが現在および将来のヘテロジニアス・コンピューティングの可能性を十分に引き出すのに役立ちます。

1. バックエンド

バックエンドの導入により、SYCL* 2020 は OpenCL* 以外の言語 / フレームワークで構築された実装への扉を開きます。その結果、名前空間は `cl::sycl::` から `sycl::` に短縮され、SYCL* ヘッダーファイルは `<CL/sycl.hpp>` から `<sycl/sycl.hpp>` になりました。

これは表面的な変更ではなく、SYCL* にとって重大な意味を持ちます。引き続き OpenCL* 上に実装できますが (そして多くの実装がそうしていますが)、汎用バックエンドのサポートにより、SYCL* はより多様なヘテロジニアス API とハードウェアをターゲットにできるプログラミング・モデルに変わりました。SYCL* は、C++ アプリケーションとベンダー固有のライブラリーとの間の「接着剤」として機能するようになり、開発者はより簡単に、しかもコードを変更することなく、さまざまなプラットフォームをターゲットにできます。

SYCL* 2020 は真にオープンで、クロスアーキテクチャー、かつクロスベンダーを実現

オープンソースの DPC++ コンパイラー・プロジェクト（英語）は、LLVM（clang）で SYCL* 2020 を実装し、この柔軟性を活かして NVIDIA*、AMD*、およびインテルの GPU をサポートします。SYCL* 2020 は真にオープンで、クロスアーキテクチャー、かつクロスベンダーを実現します（**図 1**）。

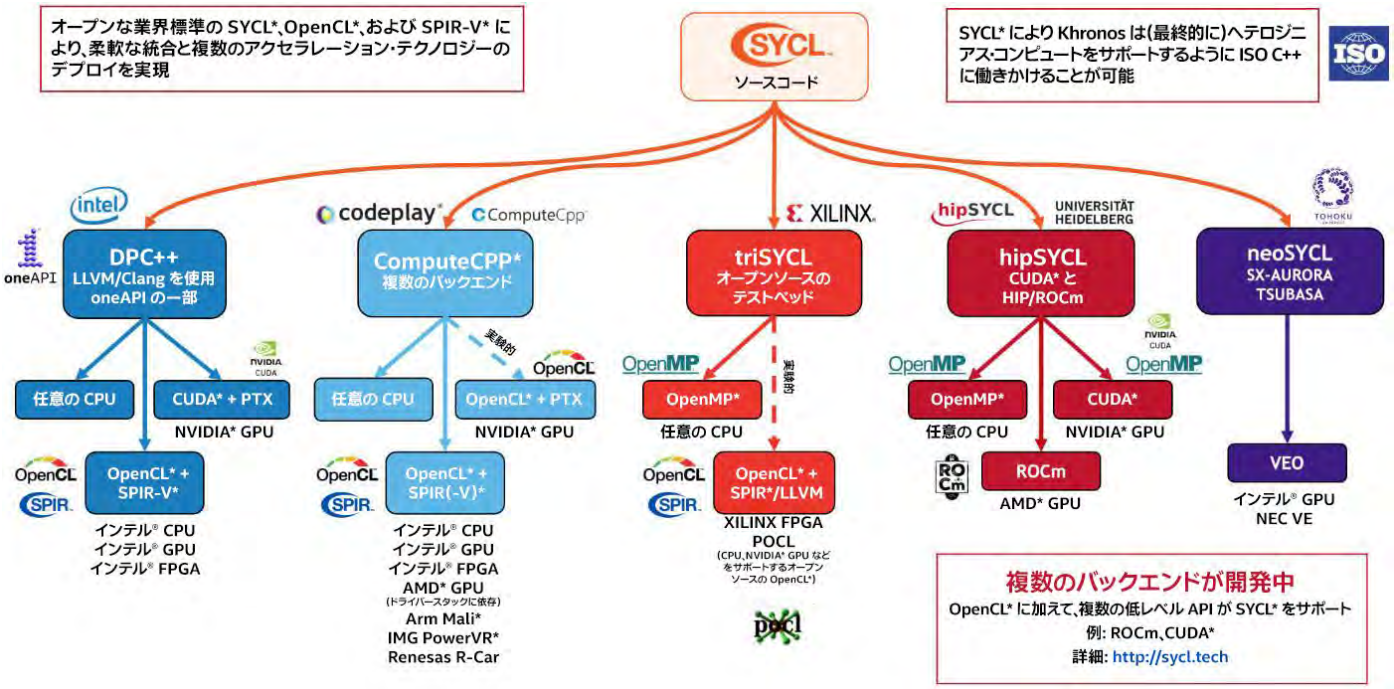


図 1. https://www.khronos.org/sycl/**（英語）の複数のバックエンドをターゲットとする SYCL* 実装**

2. 統合共有メモリー

デバイスによっては、ホスト（CPU）と統一されたメモリービューをサポートできます。SYCL* 2020 では、これを**統合共有メモリー（USM）**と呼び、SYCL* 1.2.1 のバッファー / アクセサーモデルに代わるポインターベースのアクセスモデルを可能にします。

USM を使ったプログラミングには、2 つの重要な利点があります。第一に、USM はホストとデバイスで単一の統合アドレス空間を提供します。USM 割り当てへのポインターはデバイス間で一貫しており、カーネルに直接引数として渡すことができます。これにより、既存のポインタベースの C++ および CUDA* コードの SYCL* への移行が大幅に

簡素化されます。第二に、USM はデバイス間で自動的に移行する共有割り当てを可能にし、プログラマーの生産性を向上させ、C++ コンテナー（`std::vector` など）や C++ アルゴリズム（**図 2** に示すように インテル® oneDPL（英語）を介して）との互換性を提供します。

```
sycl::usm_allocator<int, sycl::usm::alloc::shared> alloc(q.get_context(),
                                                         q.get_device());
std::vector<int, decltype(alloc)> vec(n, alloc);

auto policy = oneapi::dpl::execution::make_device_policy(q);
std::fill(policy, vec.begin(), vec.end(), 0);
```

図 2. 書籍で紹介している例**（英語）の C++ コンテナーとアルゴリズムでの USM の使用**

3 種類の USM 割り当ては、プログラマーに必要なデータ移動の制御を提供します。デバイス割り当てでは、プログラマーがアプリケーションのデータ移動を完全に制御できます。ホスト割り当ては、データの使用頻度が低く、移動のコストが見合わない場合、またはデータのサイズがデバイスのメモリーに収まらない場合に有効です。共有割り当ては、使用される場所に自動的に移行され、パフォーマンスと生産性の両方の利点をもたらします。

3. リダクション

SYCL* 2020 のリダクションに対するアプローチは、P0075（英語）の提案や、Kokkos（英語）および RAJA（英語）ライブラリーで実装されている機能など、ほかの C++ リダクションを参考にしています。

`reducer` クラスと `reduction` 関数を使用することで、SYCL* カーネルのリダクション・セマンティクスの変数表現が大幅に簡素化されます。また、コンパイル時にリダクション･アルゴリズムを指定するように実装できるため、多くのベンダーのさまざまなデバイスでハイパフォーマンスが得られます。

SYCL* 2020 のリダクションがもたらす改善の実例として、ブリストル大学で開発された人気の高い BabelStream（英語）ベンチマークを挙げることができます。BabelStream には、カーネル内のすべてのワークアイテムの浮動小数点総和を計算するシンプルなドット積カーネルが含まれています。SYCL* 1.2.1 バージョン（英語）は 43 行で、特定のアルゴリズム（ワークグループのローカルメモリーのツリー・リダクション）により記述されます。また､デバイスに最適な work-group サイズは､ユーザーが選択する必要があります(**図 3**)。SYCL* 2020 バージョン（英語）は、わずか 20 行と短いだけでなく、アルゴリズムや work-group サイズの選択を実装に委ねることで、パフォーマンスの移植性を高めています（**図 4**)。

```
template <class T>
T SYCLStream<T>::dot()
{
  queue->submit([&](handler &cgh)
  {
    auto ka   = d_a->template get_access<access::mode::read>(cgh);
    auto kb   = d_b->template get_access<access::mode::read>(cgh);
    auto ksum = d_sum->template get_access<access::mode::write>(cgh);

    auto wg_sum = accessor<T, 1, access::mode::read_write,
access::target::local>(range<1>(dot_wgsize), cgh);

    size_t N = array_size;
    cgh.parallel_for<dot_kernel>(nd_range<1>(dot_num_groups*dot_wgsize, dot_wgsize),
[=](nd_item<1> item)
    {
      size_t i = item.get_global_id(0);
      size_t li = item.get_local_id(0);
      size_t global_size = item.get_global_range()[0];

      wg_sum[li] = 0.0;
      for (; i < N; i += global_size)
        wg_sum[li] += ka[i] * kb[i];

      size_t local_size = item.get_local_range()[0];
      for (int offset = local_size / 2; offset > 0; offset /= 2)
      {
        item.barrier(cl::sycl::access::fence_space::local_space);
        if (li < offset)
          wg_sum[li] += wg_sum[li + offset];
      }

      if (li == 0)
        ksum[item.get_group(0)] = wg_sum[0];
    });
  });

  T sum = 0.0;
  auto h_sum = d_sum->template get_access<access::mode::read>();
  for (int i = 0; i < dot_num_groups; i++)
  {
    sum += h_sum[i];
  }

  return sum;
}
```

図 3. BabelStream のドット積カーネルの SYCL* 1.2.1 バージョン

```
template <class T>
T SYCLStream<T>::dot()
{
  queue->submit([&](sycl::handler &cgh)
  {
    sycl::accessor ka {d_a, cgh, sycl::read_only};
    sycl::accessor kb {d_b, cgh, sycl::read_only};

    cgh.parallel_for(sycl::range<1>{array_size},
      sycl::reduction(d_sum, cgh, std::plus<T>(), sycl::property::reduction::initiali
ze_to_identity{}),
      [=](sycl::id<1> idx, auto& sum)
      {
        sum += ka[idx] * kb[idx];
      });
  });

  sycl::host_accessor result {d_sum, sycl::read_only};
  return result[0];
}
```

図 4. BabelStream のドット積カーネルの SYCL* 2020 バージョン

4. グループ・ライブラリー

SYCL* 2020 は、SYCL* 1.2.1 の work-group 抽象化機能を拡張し、新しい sub-group 抽象化機能とグループベース・アルゴリズムのライブラリーを提供します。

`sub_group` クラスは、カーネル内で「一緒に」実行される協調的なワークアイテムのセットを表し、異なるベンダーのハードウェア機能に移植可能な抽象化を提供します。DPC++ コンパイラーでは、sub-group は常に、インテル® アーキテクチャーの SIMD ベクトル化、NVIDIA* アーキテクチャーの「warps」、AMD* アーキテクチャーの「wavefronts」といった重要なハードウェア概念にマッピングされ、SYCL* アプリケーションの低レベルのパフォーマンス・チューニングを可能にします。

ISO C++ との整合性のもう 1 つの例として、SYCL* 2020 では C++17 のグループベース・アルゴリズムの `all_of`、`any_of`、`none_of`、`reduce`、`exclusive_scan`、および `inclusive_scan` が追加されています。各アルゴリズムは異なるスコープでサポートされており、SYCL* 実装では、work-group や sub-group の並列処理を使用して、これらの関数の高度にチューニングされた協調的なバージョンを提供できます。

SYCL* 2020 のグループ・ライブラリーは、より多くのグループタイプと、より幅広いグループベース・アルゴリズムの基盤となります。

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。

5. アトミック参照

C++20 では、アトミック参照（`std::atomic_ref`）で型をラップする機能が導入され、アトミック操作が大きく進化しました。SYCL* 2020 では、この設計を採用して拡張し（`sycl::atomic_ref` として）、アドレス空間とメモリースコープをサポートして、多様なヘテロジニアス・コンピューティングに完全に対応するアトミック参照の実装を実現しました。

SYCL* 2020 では、ISO C++ を拡張することについて慎重に検討し、パフォーマンスを犠牲にすることなく移植可能なプログラミングを可能にするには、メモリースコープの概念が必須であると考えました。ヘテロジニアス・システムには、複雑なメモリー階層があります（**図 5**）。

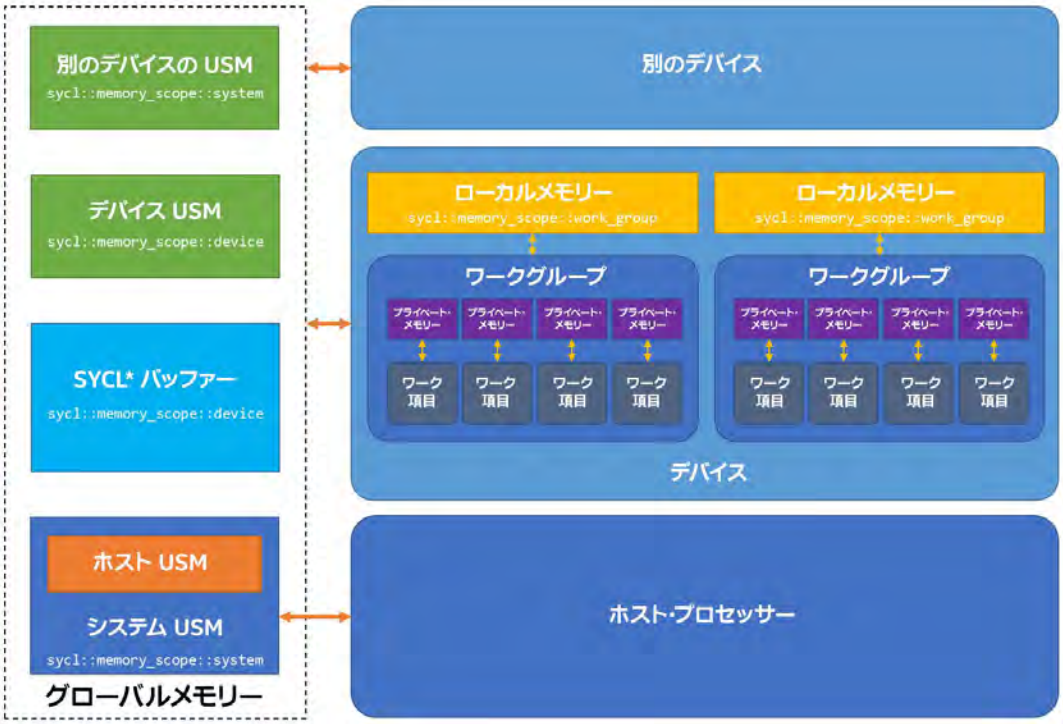


図 5. メモリースコープを使用することで、アトミック参照で一貫性が必要なメモリーを指定し、メモリーの更新を「見る」ことができるワークアイテムやデバイスをきめ細かく制御可能

メモリーモデルとアトミック操作は複雑なため、SYCL* ではできるだけ多くのデバイスをサポートできるように、すべてのデバイスで C++ メモリーモデルを完全にサポートすることを要求していません。むしろ、SYCL* はさまざまなデバイス機能を提供しています。これは、あらゆるベンダーのデバイスにオープンであることのもう 1 つの素晴らしい例です。

SYCL* 2020 の先 : ベンダー拡張

SYCL* 2020 でより多くのバックエンドとハードウェアがサポートされたことで、ベンダーによる拡張機能の開発が促進されています。これらの拡張機能は、それを必要とするデバイスに実用的なソリューションを提供し、将来の SYCL* 標準の方向性を示すイノベーションを可能にします。標準化プロセスにおいて拡張は重要な役割を果たします。この記事で取り上げたいくつかの機能は、DPC++ コンパイラー・プロジェクトで検討された拡張機能に由来するものです。

ここでは、SYCL* 2020 のベンダー拡張機能として DPC++ コンパイラー・プロジェクトでサポートされている 2 つの新機能を簡単に説明します。

カーネルスコープでのグループローカル・メモリー

SYCL* 1.2.1 は、ローカルアクセサーを介してグループローカル・メモリーをサポートしています。グループローカル・メモリーは、カーネルの外側で宣言され、カーネルの引数として取得される必要があります。OpenCL* や CUDA* などの言語のプログラマーにとって、これは不自然に感じられるかもしれません。そこで、グループローカル・メモリーをカーネル関数内で宣言できるようにする拡張機能（英語）が設計されました。この変更により、カーネルはより自己完結的になり、（コンパイル時にローカルメモリーのサイズが分かっている場合）コンパイラーの最適化にも影響を与えることができます。

FPGA 固有の拡張機能

DPC++ コンパイラー・プロジェクトでインテル® FPGA を使用できるようになりました。この拡張機能またはそれに近いものは、すべてのベンダーの FPGA に移植可能であると考えています。FPGA は重要なアクセラレーターであり、この先駆的な取り組みが、ほかのベンダーの拡張プロジェクトとともに、将来の SYCL* 標準に反映されることを期待しています。

FPGA セレクターの追加により、FPGA ハードウェアまたは FPGA エミュレーション・デバイスを簡単に利用できるようになりました。これは、FPGA をターゲットとするソフトウェア開発者にとって重要な考慮事項である、迅速なプロトタイプ作成を可能にします。FPGA の LSU 制御は、FPGA のロード / ストア操作をチューニングし、特定の方法でグローバル・メモリー・アクセスの実装を構成するように明示的に要求できます。FPGA メモリーチャネル経由で FPGA 設計をチューニングできるように、外部メモリーバンク（DDR チャネルなど）を持つデータの配置制御が追加されています。FPGA レジスターにより、ハイパフォーマンスな FPGA パイプラインを実現する主要なチューニング制御が有効になります。

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。

まとめ

ヘテロジニアスが普及し、ハードウェアの選択肢はますます多様化し、より優れたパフォーマンスとワット当たりのパフォーマンスを追求するものが多くなっています。この傾向は、SYCL* のようなオープンで、マルチベンダー、かつマルチアーキテクチャーのプログラミング・モデルの必要性をさらに高めます。

この記事では、SYCL* 2020 の移植性とパフォーマンスの移植性を可能にする 5 つの新機能を紹介しました。SYCL* 2020 により、C++ プログラマーはヘテロジニアス・コンピューティングの可能性を十分に活用できるようになります。

詳細は、sycl.tech（英語）を参照してください。オンライン・チュートリアル、SYCL* 書籍（無料の PDF（英語））へのリンク、現在の SYCL* 標準仕様（英語）へのリンクがあります。

[訳者注 : SYCL* 2020 リビジョン 4 の日本語参考訳が iSUS で公開されています。]

ArrayFire* と oneAPI による 2 次元フーリエ相関アルゴリズムの高速化

ヘテロジニアス並列処理について ArrayFire* と oneAPI の抽象化を徹底比較

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兼 The Parallel Universe 編集長
Umar Arshad ArrayFire ソフトウェア・エンジニア

「oneAPI によるフーリエ相関アルゴリズムの実装」（The Parallel Universe 44 号）では、SYCL* とインテル® oneMKL 関数の組み合わせを使用して 2 つの信号の 1 次元相互相関を計算する方法を紹介しました。この記事では、2 次元相互相関を調べて、2 つの類似した画像の最適な重なりを見つけます（**図 1**）。2 次元のユースケースの説明に加えて、1 つのコードでどこでも実行できるヘテロジニアス並列処理である oneAPI と ArrayFire* の 2 つのアプローチを比較して、以前のワークを拡張します。[**編集者注：**「ArrayFire* と oneAPI、各種ライブラリー、OpenCL* の相互運用性」（The Parallel Universe 47 号）も参考になります。]

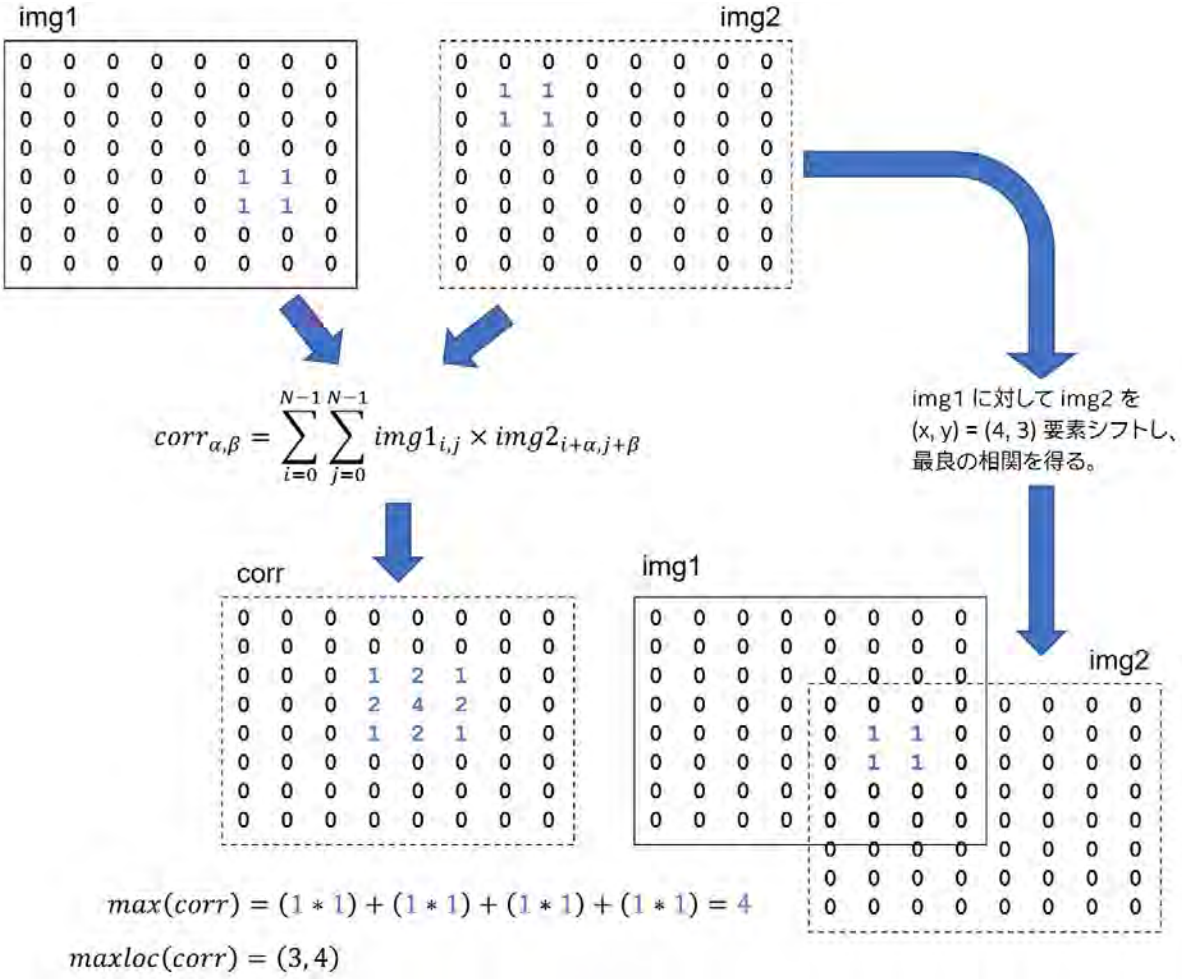


図 1.（2 進数の正方行列として表された）2 つの画像の最適な配置を見つける。
ここで、(α , β) は img1 に対する img2 の変位。
相関を計算するとき、2 番目の画像は循環シフトされることに注意。

図 1 の総当たり法の総和は非効率で、大規模な問題では実行できない可能性があるため、以前の 1 次元の例と同様に、2 次元相互相関でもフーリエ相関アルゴリズムを利用して計算の複雑さを大幅に軽減します（**図 2**）。両方のアプローチを使用して 2 次元フーリエ相関を実装し、コードを並べて表示して違いと相対的な強みを示します。

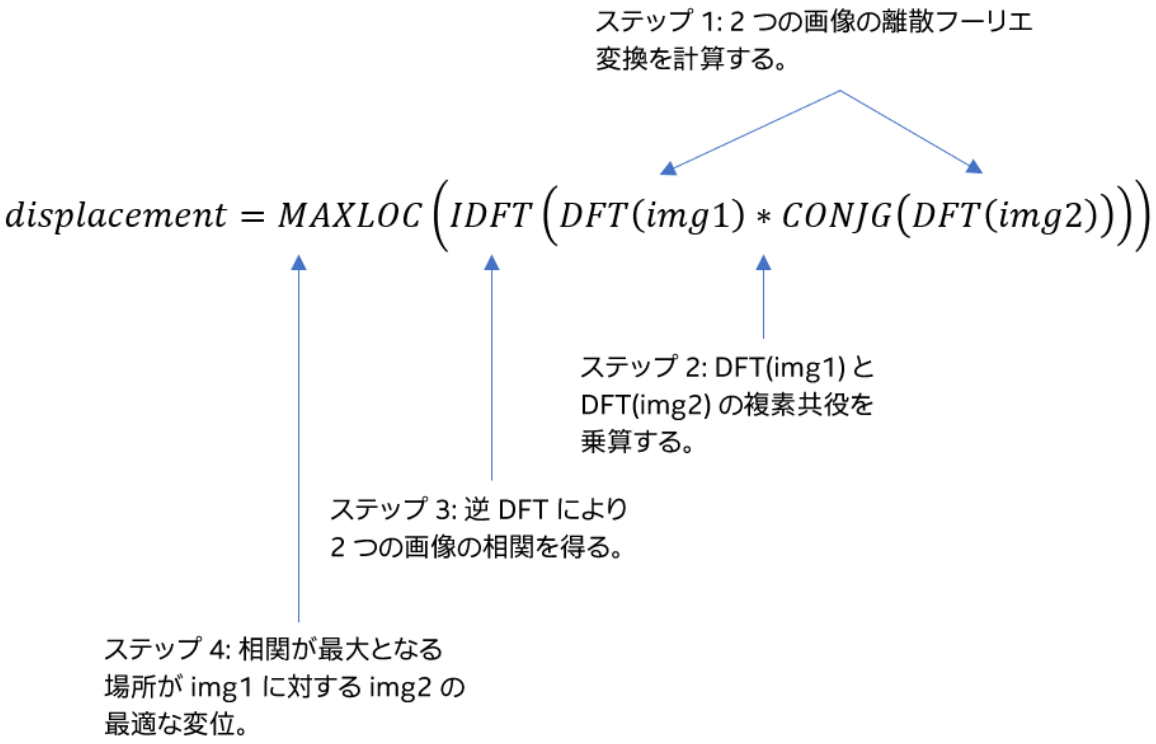


図 2. 4 ステップのフーリエ相関アルゴリズム。DFT は離散フーリエ変換、IDFT は逆 DFT、CONJG は複素共役、MAXLOC は最大相関スコアの場所。

2 次元の oneAPI 実装では、以前の 1 次元実装との違いと改善点も示します。まず、oneMKL の多次元フーリエ変換では、実数領域と複素数領域のデータレイアウトを考慮する必要があります。以降の例で示します。次に、1 次元実装では、SYCL* で記述された `MAXLOC` リダクション演算子を使用しました。2 次元実装では、このコードを oneAPI DPC++ ライブラリー (oneDPL (英語)) の標準関数に変更します。この変更により、コードが明確で簡潔になります。

フーリエ相関アルゴリズムの実装

理想的には、画像がデバイスのメモリーにロードされた後に、すべての計算がデバイスで実行されるべきです。ホストで必要なのは、最終的な変位(2 次元相関の 2 つのスカラー)のみです。その他のホストとデバイス間のデータ転送は不要であり、特に画像が大きい場合、パフォーマンスが低下します。

ArrayFire* によるアクセラレーター・オフロード

図 2 の各ステップは実際には 1 つの文にマップされるため（**図 3**）、ArrayFire* でフーリエ相関アルゴリズムを実装するのは簡単です。C++ 、MATLAB（英語）、Fortran、NumPy* の配列表記を知っているプログラマーなら、コメントがなくても、このコードが理解できるでしょう。このコードには明示的なループはなく、明示的なホストとデバイス間のデータ転送やデバイスのオフロードもありません。コーディングの抽象化は暗黙的にデータ並列であり、これらの詳細は ArrayFire* ランタイム・ライブラリーによって処理されます。

```
#include <iostream>
#include <arrayfire.h>

int main(int argc, char **argv)
{
    // デバイスを選択して ArrayFire* 情報を表示
    af::setDevice(0);
    af::info();

    // 2 次元画像配列を割り当てて初期化（注： ArrayFire* は列優先）
    unsigned int n_rows = 8, n_cols = 8;

    auto img1 = af::constant(0.0, n_rows, n_cols, af::dtype::f32);
    auto img2 = af::constant(0.0, n_rows, n_cols, af::dtype::f32);
    auto corr = af::constant(0.0, n_rows, n_cols, af::dtype::f32);

    img1(af::seq(4, 5), af::seq(5, 6)) = 1.0f;  // 最初の画像の右下に要素を設定
    img2(af::seq(1, 2), af::seq(1, 2)) = 1.0f;  // 2 番目の画像の左上に要素を設定

    // ステップ 1: DFT(img1) と DFT(img2) を計算
    img1 = af::fftR2C<2>(img1, 0.0);
    img2 = af::fftR2C<2>(img2, 0.0);

    // ステップ 2: DFT(img1) * CONJG(DFT(img2)) を計算
    corr = img1 * af::conjg(img2);

    // ステップ 3: 逆 DFT を実行
    corr = af::fftC2R<2>(corr, 0.0);

    // ステップ 4: img1 に対する img2 の最適な変位を見つける
    af::array max_score, shift;
    af::max(max_score, shift, af::flat(corr));

    auto max_corr = max_score.scalar<float>();
    auto s = shift.scalar<unsigned>();
    int x_shift = s / n_cols;
    int y_shift = s % n_rows;

    std::cout << std::endl << "Shift the second image (x, y) = (" << x_shift << ", " << y_shift
              << ") elements relative to the first image to get a maximum," << std::endl
              << "normalized correlation score of " << max_corr
              << ". Treat the images as circularly shifted versions of each other." << std::endl;
}
```

図 3. ArrayFire* を使用して実装したフーリエ相関アルゴリズム

ほとんどの ArrayFire* 関数は非同期であるため、呼び出し元は関数がリターンするのを待たずに続行します。関数呼び出しは、内部のインオーダー・デバイス・キューに追加されます。しかし、明示的な同期を使用して、キューが空になるまで待機することもできます。次に例を示します。

```
// ステップ 1: DFT(img1) と DFT(img2) を計算
img1 = af::fftR2C<2>(img1, 0.0);
img2 = af::fftR2C<2>(img2, 0.0);
af::sync();
```

`sync` 文は、2 つの順変換が完了するまで呼び出し元を待機させます。

ArrayFire* は、一部の計算の遅延評価も行います。この場合、命令の順序は保存されますが、結果が必要になるまでデバイスキューにワークが送信されません。その後にカーネルが生成され、キューに追加されます。共役乗算文は、遅延評価の例です。

```
// ステップ 2: DFT(img1) * CONJG(DFT(img2)) を計算
corr = img1 * af::conjg(img2);
corr.eval();
af::sync();
```

`eval` 文は、カーネルを強制的に作成してキューに追加します。`sync` 文は、共役乗算操作が完了するまで呼び出し元を待機させます。

oneAPI によるアクセラレーター・オフロード

oneAPI を使用した 1 次元フーリエ相関の実装は、以前の記事およびウェビナー（英語）で紹介しました。その原理は 2 次元の場合も同じであるため、このセクションでは、oneAPI 実装の各ステップを説明するのではなく、oneAPI のコードと ArrayFire* のコードを比較して、ヘテロジニアス並列処理に対する 2 つのアプローチの違いを説明します。

デバイスの画像の初期化

ホスト CPU とさまざまなアクセラレーター・デバイス間のデータ移動は、ヘテロジニアス並列プログラミングにおける重要な考慮事項です。アルゴリズムの一部のステップをホストで実行し、ほかのステップをデバイスで実行すると、ホストとデバイス間のデータ転送により、アクセラレーター・オフロードのパフォーマンスの利点が制限されます。幸いなことに、フーリエ相関アルゴリズムの各ステップはデバイスで実行できます。デバイスメモリーに転送した画像を、ホストに戻す必要はありません。

デバイスのデータを初期化する ArrayFire* コードと SYCL* コードを**図 4** に示します。簡単にするため、**図 1** と同じ人工の画像を使用しています。オフロードデバイスを設定した後、ArrayFire* コードは便利な関数と配列構文を使用してデータを初期化します（**図 4**、左）。同様に、oneAPI コードはデフォルトデバイスの SYCL* キューを初期化し、統合共有メモリーにインプレースの実数から複素数の変換に十分な空間を割り当てて、データレイアウトを定義します（**図 4**、右）[多次元 DFT のデータレイアウトの説明は本題から外れるためここでは取り上げません。

FFTW のドキュメントに優れた説明（英語）が記載されています。])。SYCL* コードは、SYCL* キューにワークを送信して（つまり、`parallel_for` と `single_task` カーネルにより)、デバイスで初期化を実行します。SYCL* キューは非同期であるため、これらのカーネルは、ワークが完了するまで待機してから続行するように明示的に指示されています。

```
af::setDevice(0);

auto img1 = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);
auto img2 = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);
auto corr = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);

img1(af::seq(4, 5), af::seq(5, 6)) = 1.0f;
img2(af::seq(1, 2), af::seq(1, 2)) = 1.0f;
```

```
// SYCL* キューを初期化
sycl::queue Q(sycl::default_selector{});

// 2 次元画像と相関配列を割り当て
auto img1 = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);
auto img2 = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);
auto corr = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);

// 行優先アドレス指定の一般化されたストライドを設定
int r_stride = 1;
int c_stride = (n_cols / 2 + 1) * 2;
int c_stride_h = (n_cols / 2 + 1);

// 人工データで入力画像を初期化
// デバイスで初期化を行う
Q.parallel_for<>(sycl::range<2>{n_rows, n_cols},
                 [=](sycl::id<2> idx)
{
   unsigned int r = idx[0];
   unsigned int c = idx[1];
   img1[r * c_stride + c * r_stride] = 0.0;
   img2[r * c_stride + c * r_stride] = 0.0;
   corr[r * c_stride + c * r_stride] = 0.0;
}).wait();

Q.single_task<>([=]()
{
   // 最初の画像の右下に要素を設定
   img1[4 * c_stride + 5 * r_stride] = 1.0;
   img1[4 * c_stride + 6 * r_stride] = 1.0;
   img1[5 * c_stride + 5 * r_stride] = 1.0;
   img1[5 * c_stride + 6 * r_stride] = 1.0;

   // 2 番目の画像の左上に要素を設定
   img2[1 * c_stride + 1 * r_stride] = 1.0;
   img2[1 * c_stride + 2 * r_stride] = 1.0;
   img2[2 * c_stride + 1 * r_stride] = 1.0;
   img2[2 * c_stride + 2 * r_stride] = 1.0;
}).wait();
```

図 4. ArrayFire*（左）と SYCL*（右）を使用したデバイスのデータの初期化。

ホストとデバイス間の不要な転送を防ぐため、ホストはデバイスメモリーのデータを変更しないようにします。画像の要素は `single_task` カーネルを使用して設定されています。このカーネルがないと、要素がホストで設定されるため、oneAPI ランタイムはホストとデバイスのデータの整合性を取る必要があります。データが大きい場合（医療用画像処理アプリケーションのボリューム画像など）、この処理によりパフォーマンスが低下します。

ArrayFire* コードは、よりコンパクトで直感的です。oneAPI コードは、データが割り当てられる場所、計算が実行される場所とタイミングが、より明確です。また、人気の高いオープンソースの高速フーリエ変換パッケージである FFTW（英語）との整合性も高くなっています。

ステップ 1：順変換を実行

ArrayFire* コードはシンプルで直感的です（**図 5**、左）。関数呼び出しは、入力データにより定義された精度で、2 次元の実数から複素数(R2C)の非正規化 FFT を指定します。oneMKL DFT 記述子のアプローチ(**図 5**、右)は、それほどコンパクトではありませんが、以前のインテル® MKL DFTI ユーザーと FFTW ユーザーには馴染み深いものです。oneMKL コードは、必要なサイズと次元の、単精度の実数から複素数の変換の記述子を初期化し、この記述子を SYCL* キューにコミットした後、順変換を計算します。`compute_forward` 関数は、後で同期に使用する SYCL* イベントを返します。

```
img1 = af::fftR2C<2>(img1, 0.0);
img2 = af::fftR2C<2>(img2, 0.0);
```

```
// FFT 記述子を初期化
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                             oneapi::mkl::dft::domain::REAL>
                                       forward_plan({n_rows, n_cols});

// 実数領域のデータレイアウト
std::int64_t real_layout[4] = {0, c_stride, 1};

// 共役偶数領域のデータレイアウト
std::int64_t complex_layout[4] = {0, c_stride_h, 1};

forward_plan.set_value(oneapi::mkl::dft::config_param::INPUT_STRIDES,
                       real_layout);
forward_plan.set_value(oneapi::mkl::dft::config_param::OUTPUT_STRIDES,
                       complex_layout);
forward_plan.commit(Q);

auto evt1 = oneapi::mkl::dft::compute_forward(forward_plan, img1);
auto evt2 = oneapi::mkl::dft::compute_forward(forward_plan, img2);
```

図 5. ArrayFire*（左）と oneMKL（右）を使用したデバイスでの順変換（実数から複素数）の実行。

ステップ 2：複素共役乗算

ArrayFire* と oneMKL は、異なるアプローチを使用して共役乗算操作を実行します。ArrayFire* は、単純な配列表記を使用します（**図 6**、左）。プログラマーは、デバイスやデータのレイアウトを指定する必要はありません。oneMKL は、便利な `mulbyconj` 関数を提供します（**図 6**、右）。`mulbyconj` 関数はより複雑ですが、プログラマーは、データレイアウト、計算を実行する場所とタイミング（つまり、SYCL* キューとイベント）を明示的に制御できます。

`corr = img1 * af::conjg(img2);`	`oneapi::mkl::vm::mulbyconj(Q, n_rows * c_stride_h,`
	`reinterpret_cast<std::complex<float>*>(img1),`
	`reinterpret_cast<std::complex<float>*>(img2),`
	`reinterpret_cast<std::complex<float>*>(corr),`
	`{evt1, evt2}).wait();`

図 6. ArrayFire*（左）と oneMKL（右）を使用した複素共役乗算。

ステップ 3：逆変換を実行

ステップ 1 とステップ 3 のコードは似ていますが、1 つの複素数から実数の変換のみ実行される点が異なります。ArrayFire* コード（**図 7**、左）は、`fftR2C` 関数ではなく `fftC2R` 関数を呼び出しています。oneMKL コード（**図 7**、右）は、複素数から実数のデータレイアウトを指定する新しい DFT 記述子を初期化します。

`corr = af::fftC2R<2>(corr, 0.0);`	`oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,`
	`oneapi::mkl::dft::domain::REAL>`
	`backward_plan({n_rows, n_`
	`cols});`
	
	`// 共役偶数領域のデータレイアウト`
	`backward_plan.set_value(oneapi::mkl::dft::config_param::INPUT_STRIDES,`
	`complex_layout);`
	
	`// 実数領域のデータレイアウト`
	`backward_plan.set_value(oneapi::mkl::dft::config_param::OUTPUT_`
	`STRIDES,`
	`real_layout);`
	`backward_plan.commit(Q);`
	
	`auto bwd = oneapi::mkl::dft::compute_backward(backward_plan, corr);`
	`bwd.wait();`

図 7. ArrayFire*（左）と oneMKL（右）を使用したデバイスでの逆変換（複素数から実数）の実行。

ステップ 4 : MAXLOC リダクション

以前の実験で使用した SYCL* `MAXLOC` リダクション演算子を使用すると、コードは複雑になります (fcorr_1d_usm.cpp（英語）を参照)。oneDPL は、C++ プログラマーに馴染み深い 2 つの関数、max_element（英語）および distance（英語）と同じ機能を提供します。**図 8**（右）の `MAXLOC` リダクションは、これらの関数の oneDPL 実装を使用しています。SYCL* キューは、`max_element` に計算を実行する場所を指示します。SYCL* カーネルとは異なり、oneDPL アルゴリズムは同期的であるため、明示的な `wait` 文はありません。

ArrayFire* コード（**図 8**、左）は以前のステップほど単純ではありません。`af::flat` 関数で 2 次元 `corr` 配列を平坦化した後、`af::max` 関数で最大相関スコアと平坦化された配列内での場所を見つけます。この場所は、ArrayFire* が列優先（**図 3** を参照）であることを考慮して、2 つの画像の最適な配置を提供する x および y シフトに変換されます。oneDPL コード（**図 8**、右）も同様の操作を実行しますが、oneMKL のデータレイアウトを考慮する必要があります。

```
af::array max_score, shift;
af::max(max_score, shift, af::flat(corr));

auto max_corr = max_score.
scalar<float>();
auto s = shift.scalar<unsigned>();
int x_shift = s / n_cols;
int y_shift = s % n_rows;
```

```
auto policy = oneapi::dpl::execution::make_device_policy(Q);
auto maxloc = oneapi::dpl::max_element(policy,
                                       corr,
                                       corr + (n_rows * n_cols * 2 + 2));

auto s = oneapi::dpl::distance(corr, maxloc);
float max_corr = corr[s];
int x_shift = s % (n_cols + 2);
int y_shift = s / (n_rows + 2);
```

図 8. ArrayFire*（左）と oneDPL（右）を使用した MAXLOC リダクションの実行

まとめ

oneAPI と ArrayFire* のアプローチはどちらも、1 つのコードでどこでも実行できるヘテロジニアス並列処理を実現できます。すべての 2 次元フーリエ相関計算が oneAPI または ArrayFire* のライブラリーで行われるため、パフォーマンスについては説明しません。The Parallel Universe では、アプリケーション開発者とコンパイラー / ライブラリー開発者の間の関心の分離（英語）が繰り返しテーマになっています。後者のグループは、主にパフォーマンスと計算効率に関心があります。前者のグループは、主に問題を可能な限り生産的に解決することに関心があります。このグループに属している場合は、優れたパフォーマンスを発揮する高レベルのプログラミング抽象化を好むでしょう。ライブラリーが優れたパフォーマンスを発揮することは当然のことです。

ArrayFire* は高レベルの抽象化を提供するため、ArrayFire* のフーリエ相関の実装は簡潔です。その配列表記は、Fortran、MATLAB、Python* NumPy* プログラマーには馴染み深いものです。ArrayFire* には Python* API も用意されています。遅延評価は計算を開始するときのランタイム制御を意味しますが、上記のように、プログラマーは必要に応じて制御を行うことができます。

oneAPI の実装は、プログラマーがホストとデバイス間のデータ転送、計算を実行する場所とタイミングを細かく制御できるため、より複雑になります。oneMKL の DFT 記述子とデータレイアウトは、以前のインテル® MKL DFTI ユーザーと FFTW ユーザーには馴染み深いものです。実際に、インテル® oneAPI マス・カーネル・ライブラリーは FFTW インターフェイスをサポートしています。最後に、oneDPL の関数は C++ プログラマーには受け入れやすいものです。

最終的には、プロジェクトの要件と個人的な好みで oneAPI と ArrayFire* のどちらかを選択することになるでしょう。

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

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

oneAPI の maxloc リダクション

SYCL* と oneDPL でこの一般的な並列パターンを実装

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兼 The Parallel Universe 編集長
Alexey Kukanov インテル コーポレーション 主席ミドルウェア・エンジニア
John Pennycook インテル コーポレーション ソフトウェア・イネーブリング & 最適化アーキテクト

maxloc 操作とは

最大値の場所（`maxloc`）を見つけることは、配列で実行される一般的な検索操作です。多くのプログラミング言語とライブラリーが、組込みの `maxloc` 関数を提供しています（NumPy* argmax（英語）、Fortran maxloc（英語）、BLAS amax（英語）、C++ max_element（英語）関数など）。最近の記事、「インテル® AVX-512 命令を使用した Maxloc 操作の最適化」（The Parallel Universe 46 号）では、`maxloc` 検索をベクトル化してパフォーマンスを向上する方法を説明しました。相互相関を含む、多くのアルゴリズムで重要な操作であることは明白です（**図 1**）。

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。

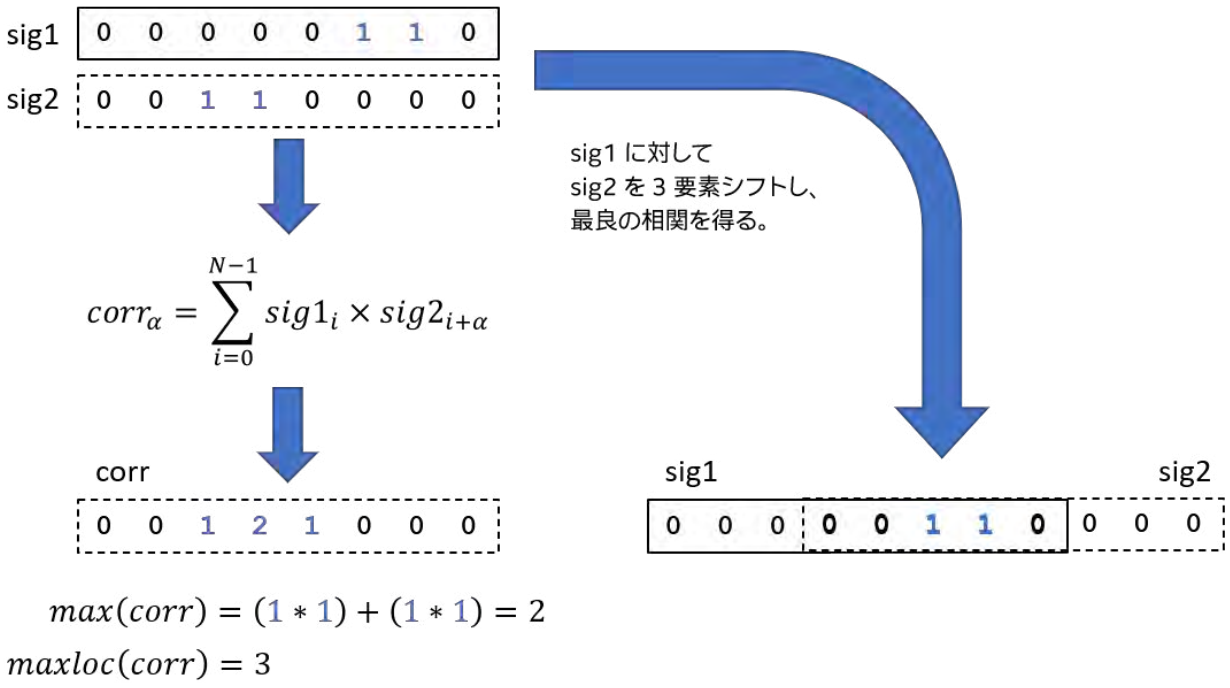


図 1.（2 進数の配列として表された）2 つの離散信号の重なりが最大になる変位を見つける。ここで、α は sig1 に対してシフトされる sig2 の要素の数。相関を計算するとき、2 番目の信号は循環シフトされることに注意。

以前の記事、「oneAPI によるフーリエ相関アルゴリズムの実装」（The Parallel Universe 44 号）では、SYCL* とインテル® oneMKL 関数の組み合わせを使用して 2 つの信号の 1 次元相互相関を計算する方法を紹介しましたが、最後の `maxloc` ステップは省略されていました（**図 2**）。ステップ 1 から 3 はアクセラレーター・デバイスにオフロードされていましたが、ステップ 4 はホスト CPU で計算されていました。つまり、最終的な相関配列をホストに転送する必要があることを意味します。理想的には、信号がデバイスのメモリーにロードされた後に、すべての計算がデバイスで実行されるべきです。ホストで必要なのは、最終的な変位（1 次元相関の 1 つのスカラー）のみです。その他のホストとデバイス間のデータ転送は不要であり、パフォーマンスの低下につながります。そのため、この記事の主題である、デバイス上で `maxloc` を実行するさまざまな方法を実験しました。

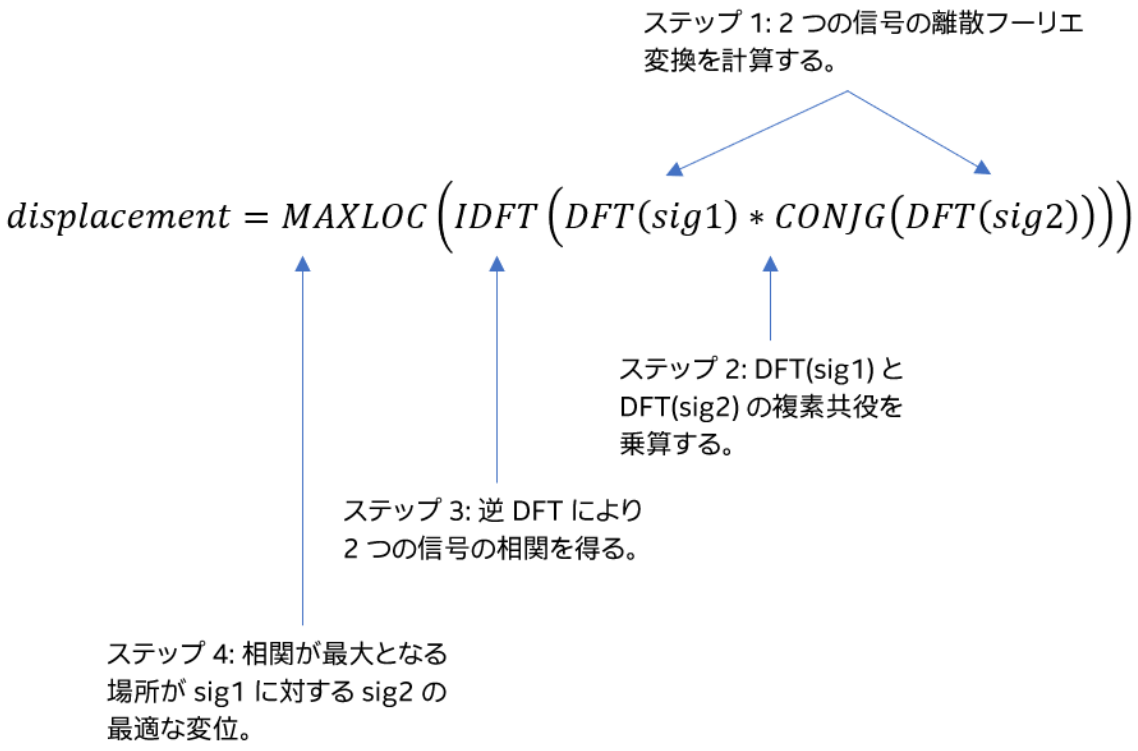


図 2. maxloc リダクションはフーリエ相関アルゴリズムの最後のステップ。DFT は離散フーリエ変換、IDFT は逆 DFT、CONJG は複素共役、MAXLOC は最大相関スコアの場所。

SYCL* のリダクション演算子

リダクションは、複数の値を 1 つの値に減らす一般的な並列パターンです。例えば、総和リダクションは、配列内の値を加算して 1 つの総和を得ます。配列内の最小値や最大値、それらの値の場所を見つけることも、リダクション操作です。SYCL* は、並列カーネルで使用できる組込みのリダクション演算子を提供しています(**図 3**)。

```
#include <CL/sycl.hpp>
#include <iostream>

int main() {

    sycl::queue Q;
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    int sum;
    std::vector<int> data{1, 1, 1, 1, 1, 1, 1, 1};

    sycl::buffer<int> sum_buf(&sum, 1);
    sycl::buffer<int> data_buf(data);

    Q.submit([&](sycl::handler& h)
    {
        sycl::accessor buf_acc{data_buf, h, read_only};

        h.parallel_for(sycl::range<1>{8},
                       sycl::reduction(sum_buf, h, std::plus<>()),
                       [=](sycl::id<1> idx, auto& sum)
        {
            sum += buf_acc[idx];
        });
    });
    sycl::host_accessor result{sum_buf, read_only};
    std::cout << "Sum equals " << result[0] << std::endl;

    return 0;
}
```

図 3. SYCL* の総和リダクションの例。

SYCL* のリダクション・クラスを使用して `maxloc` のようなほかの演算子を実装することもできます（**図 4**）。このコードは、『Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL（データ並列 C++: C++ と SYCL* を使用したヘテロジニアス・システムのプログラミングで DPC++ を使いこなす）』（英語）の第 14 章「Common Reduction Patterns（一般的なリダクション・パターン）」（英語）（334 ~ 339 ページ）の `minloc` の例から引用したものです。SYCL* キューは非同期であるため、`wait` 文を使用して続行する前に計算が完了することを保証しています。

```
#include <iostream>
#include <CL/sycl.hpp>

template <typename T, typename I>
using maxloc = sycl::maximum<std::pair<T, I>>;

constexpr size_t L = 1;

int main(int argc, char **argv)
{
   sycl::queue Q;
   const size_t n = 7;
   float* data = sycl::malloc_shared<float>(n, Q);
   data[0] = 1; data[1] = 1; data[2] = 1; data[3] = 2; data[4] = 1; data[5] = 1; data[6] = 1;

   std::pair<float, int>* max_res = sycl::malloc_shared<std::pair<float, int>>(1, Q);
   std::pair<float, int> max_identity = {
       std::numeric_limits<float>::min(), std::numeric_limits<int>::min()
   };
   *max_res = max_identity;
   auto red_max = sycl::reduction(max_res, max_identity, maxloc<float, int>());

   Q.parallel_for(sycl::nd_range<1>{n, L}, red_max, [=](sycl::nd_item<1> item, auto& max_res) {
       int i = item.get_global_id(0);
       std::pair<float, int> partial = {data[i], i};
       max_res.combine(partial);
   }).wait();

   std::cout << "Maximum value = " << max_res->first << " at element " << max_res->second <<
std::endl;

   sycl::free(data, Q.get_context());
   sycl::free(max_res, Q.get_context());

   return 0;
}
```

図 4. SYCL* のリダクション演算子として maxloc を実装。

これは SYCL* のリダクション･クラスの簡単な例ですが、アーキテクチャーをチューニングする方法はさまざまです。以前の記事、「データ並列 C++ のリダクション操作」(The Parallel Universe 44 号) および「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」(The Parallel Universe 45 号) では、さまざまな実装とパフォーマンス特性の例を示しました。組込みのリダクション演算子を使用すれば、最適化の責任を負うのはコンパイラーとライブラリーの開発者です。アプリケーション開発者は、組込みの実装がターゲット CPU やアクセラレーターで優れたパフォーマンスを提供することを期待できます。

oneDPL で同じリダクションを行う

oneDPL は、SYCL* のリダクション・クラスを使用しないで関数を呼び出したいプログラマーに別の方法を提供します。Data Parallel C++（英語）（339 ページ）では次のように説明されています。

> 「C++ 標準テンプレート・ライブラリー（STL）には、並列パターンに対応する複数のアルゴリズムが含まれています…STL のアルゴリズムは通常、イテレーターのペアで指定されたシーケンスに適用されます。C++17 以降は、それらをシーケンシャルに実行するか並列で実行するかを示す実行ポリシー引数をサポートします。oneDPL は、この実行ポリシー引数を活用して、内部の DPC++ で記述されたカーネルを活用する並列プログラミングに生産性の高いアプローチを提供します。STL アルゴリズムの機能のみを使用してアプリケーションを表現できる場合、oneDPL を使用すると、DPC++ カーネルコードを 1 行も記述することなく、システムでアクセラレーターを使用できます。」

この説明には多くのポイントが含まれていますが、2 つのポイントに注目します。

1. **C++ STL :** 関数は C++ プログラマーには馴染み深いものです。
2. **高い生産性 :** コーディングとパフォーマンスの責任は、STL の開発者が負います。アプリケーション開発者は、低レベルの SYCL* カーネルを記述することなくアクセラレーターにアクセスできます。

カーネルベースの `maxloc` コード（**図 4**）と oneDPL 実装（**図 5**）を比較すると、利点が明らかになります。後者は、`max_element` 関数を使用して `maxloc` リダクションを実行します。前の説明で述べたように、SYCL* カーネルは「内部」で活用されます。oneDPL のデフォルトの実行ポリシーは、アクセラレーターが利用可能な場合は、計算をアクセラレーターにオフロードします。利用できない場合は、計算をホスト CPU で実行します。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};

   auto policy = oneapi::dpl::execution::dpcpp_default;
   auto maxloc = oneapi::dpl::max_element(policy, data.cbegin(), data.cend());

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data.cbegin(), maxloc) <<
std::endl;

   return 0;
}
```

図 5. oneDPL の max_element と distance 関数、および暗黙的なホストとデバイス間のデータ転送を使用した基本的な maxloc リダクション。

図 5 ではホストとデバイス間のデータ転送が暗黙的に処理されていることに注意してください。計算がアクセラレーターにオフロードされている場合、oneDPL ランタイムは一時バッファーによりデータを自動的にラップします。大規模な oneAPI プログラムで、データがすでに SYCL* バッファーに含まれている場合、oneDPL 関数はバッファーを使用して反復処理を行うことができます（**図 6**）。バッファリングされたデータがデバイスに暗黙的に転送されるか、（SYCL* バッファー経由で）明示的に転送されるかに関係なく、バッファーが変更されない限りホストに転送する必要がないため、不要なオーバーヘッドを回避できます。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};
   sycl::buffer<int> data_buf(data);

   auto policy = oneapi::dpl::execution::dpcpp_default;
   auto maxloc = oneapi::dpl::max_element(policy, oneapi::dpl::begin(data_buf), oneapi::dpl::end(data_
buf));

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element "
         << oneapi::dpl::distance(oneapi::dpl::begin(data_buf), maxloc) << std::endl;

   return 0;
}
```

図 6. oneDPL と明示的なホストとデバイス間のデータ転送のために SYCL* バッファリングを使用した maxloc リダクション。

oneDPL 関数は、統合共有メモリー（USM）のポインターを使用してホストとデバイス間のデータ転送を処理することもできます（**図 7**）。この例では、SYCL* の `malloc_shared` 関数と SYCL* キューを使用して、適切な USM に空間を割り当てています。同じキューが oneDPL 実行ポリシーの設定に使用されます。oneDPL アルゴリズムは同期的であるため、**図 5 ~ 7** には明示的な `wait` 文はありません。一方、SYCL* カーネルは非同期です。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   sycl::queue Q(sycl::default_selector{});
   auto policy = oneapi::dpl::execution::make_device_policy(Q);

   const size_t n = 7;
   auto data = sycl::malloc_shared <int>(n, Q);

   data[0] = 1; data[1] = 1; data[2] = 1; data[3] = 2; data[4] = 1; data[5] = 1; data[6] = 1;

   auto maxloc = oneapi::dpl::max_element(policy, data, data + n);

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data, maxloc) << std::endl;

   sycl::free(data, Q);
   return 0;
}
```

図 7. oneDPL とホストとデバイス間のデータ転送に USM を使用した maxloc リダクション。

SYCL* リダクションと oneDPL のどちらを使用すべきか

多くのプログラミングの質問と同様に、決定的な答えはありません。どちらのアプローチにも使用する正当な理由があります。どちらを使用するかは要件に依存します。oneDPL が要件を満たすアルゴリズムを提供している場合、標準関数を呼び出すほうが SYCL* カーネルを作成するよりも簡単です。例えば、デバイスメモリーにすでに大きな配列が含まれていて、リダクションからのスカラー出力のみホストで必要な場合は、oneDPL 関数を呼び出すのが最適でしょう。しかし、関数呼び出しは同期的であるため、関数がリターンするまでプログラムがブロックされることに注意する必要があります。SYCL* リダクションは複雑ですが、非同期で柔軟性が高く、より多くのチューニングの機会を提供します。例えば、リダクションを実行しているデータを変換する場合や、複数のリダクションを同時に実行する場合は、SYCL* カーネルを作成することを推奨します。

The Parallel Universe では、アプリケーション開発者とコンパイラー / ライブラリー開発者の間の関心の分離（英語）が繰り返しテーマになっています。後者のグループは、主にパフォーマンスと計算効率に関心があります。前者のグループは、主に問題を可能な限り生産的に解決することに関心があります。このグループに属している場合は、優れたパフォーマンスを発揮する高レベルのプログラミング抽象化を好むでしょう。**図 4** の SYCL* `maxloc` リダクション演算子は複雑で低レベルであり、最高のパフォーマンスを達成するには、アーキテクチャー固有のチューニング（ワークグループのサイズなど）が必要になります。**図 5 ~ 7** の oneDPL の例は単純で、C++ STL ユーザーに馴染み深く、ホストとデバイス間のデータ転送という点で汎用的です。さらに重要なのは、主にパフォーマンスに関心がある oneDPL 製品の開発者がチューニングを行うことです。その結果、oneDPL 関数は、アーキテクチャーに関係なく、優れたパフォーマンスを提供します。これは、ヘテロジニアス・コンピューティングのための oneAPI ソフトウェア抽象化の約束です。

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。

oneDPL による C++ プログラミングの生産性とパフォーマンスの向上

ヘテロジニアス・コンピューティングではオープンな標準と移植性が味方になる

Pablo Reble インテル コーポレーション ソフトウェア・エンジニア

プログラマーが新しいマルチコア・プロセッサーで直面する課題について、Herb Sutter 氏が「The Free Lunch Is Over（フリーランチは終わった）」（英語）と述べた 2005 年以来、我々は長い道のりを歩んできました。今日のコンピューティング・アーキテクチャーとアクセラレーターのポートフォリオは非常に豊富で、絶えず成長しています。半導体の根本的な制限と、強力で電力効率の良いコンピューティングに対する要望により、開発が推進されています。コンピューティングの世界は、ますますヘテロジニアスになり、プログラマーの課題を生み出しています。

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

C++ は、現在でも最も人気の高い 5 つのプログラミング言語の 1 つです（TIOBE（英語）のプログラミング言語ランキングでは、2022 年 1 月時点で第 4 位です）。メモリー管理の完全な制御や汎用プログラミングのサポートなどの属性により、ヘテロジニアス・プログラミングの課題に取り組むための優れた言語として評価されています。プログラミング言語を選択する際の一般的な懸念事項は、開発者の生産性とコード保守のコストです。幸いなことに、以前の調査では、並列ビルディング・ブロックと C++ アルゴリズムを組み合わせると生産性の向上が期待できることが示されています。例えば、特定のアーキテクチャーに共通の機能とパターン（リダクションなど）を最適化した組込み実装は、パフォーマンスと開発者の生産性の両方を向上させます。[1, 2]

業界最高レベルの oneDPL（英語）の実装は、オープンソースの LLVM プロジェクトに貢献しました。そのおかげで、マルチスレッドの世界における開発者の労力は大幅に削減されます。[1, 6]

進化したクラシック STL アルゴリズム

古いものと新しいものでコードを高速化

C++ 言語は進化しており、その標準テンプレート・ライブラリー（STL）も進化しています。例えば、5 年前に実行ポリシーがアルゴリズム・ライブラリーに追加されたことにより、既存の C++ コードでも最新のプロセッサーのユビキタスな並列処理の恩恵を受けることができます。oneDPL は、異なるベンダーが移植可能な方法で従来のアルゴリズムの高速化バージョンを実装できるようにする、進化した C++ STL と考えることができます。

oneDPL は、SYCL* を使用して C++ アルゴリズム・ライブラリーを実装します。

> 「SYCL*（「シクル」と読みます）は、ロイヤルティー・フリーのクロスプラットフォーム抽象化レイヤーであり、同じソースファイルに含まれるアプリケーションのホストコードとカーネルコードと標準の ISO C++ を使用して、ヘテロジニアス・プロセッサーに対応したコードを記述できます。」[3]

SYCL* には、ダイレクト・アクセラレーター・プログラミングの学習曲線があります。C++ はプログラマーにメモリー管理の完全な制御を提供しますが、ホストメモリーとデバイスメモリーの分離という概念はありません。oneDPL は、ホストとデバイス間でデータを共有するための移植可能な方法として、SYCL* のメモリー抽象化に依存しています。oneDPL のアルゴリズム関数はすぐに使えて、C++ プログラマーには馴染み深く、さまざまなアクセラレーター向けに最適化されています。これにより、学習曲線が平坦になり、コードのパフォーマンスと開発者の生産性が向上します。

[1] 「oneDPL ライブラリーは SYCL* の上に構築されているため、ネイティブ SYCL* コードよりも優れていることを目にするのは非常に興味深いことです。」Deakin ほか、2021（40 ページ）[2]

oneDPL の能力を説明する簡単な例を次に示します。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};

   auto policy = oneapi::dpl::execution::dpcpp_default;
   auto maxloc = oneapi::dpl::max_element(policy, data.cbegin(), data.cend());

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data.cbegin(), maxloc) <<
std::endl;

   return 0;
}
```

この例は、一般的な `maxloc` リダクション（つまり、最大値を含むデータセット内の要素の特定）を実行ポリシーで指定したアクセラレーターにオフロードします。インクルードされたヘッダーは ISO C++ に準拠しています。`max_element` のブロッキング動作も同様です。データの移動は、この例では暗黙的に処理されます。つまり、計算がアクセラレーターにオフロードされた場合、ランタイムは SYCL* バッファーによりデータをラップして、ホストとデバイス間のデータ転送を自動的に処理します。プログラマーがホストとデバイス間のデータ転送を明示的に制御できるモードも存在します。

SYCL* での並列アルゴリズムの実装に加えて、oneDPL は、カスタム・イテレーターなど、デバイス・プログラミングに必須の拡張機能をサポートしています。これらの拡張機能は、異なるプラットフォーム間の相互運用性を確保するために、oneDPL 仕様（英語）に追加されました。[4]

次のステップ
将来を占う

現在開発中の、ISO C++ に完全に追加されていない、いくつかの強力で試験的な oneDPL の機能と、それらにアクセスする方法を見てみましょう。

- C++20 では、C++ STL アルゴリズムを使用する際の表現力を大幅に向上できる `Ranges` が追加されています。`Ranges` は、`Views` で複雑なデータ・アクセス・パターンをサポートすることにより、アルゴリズムの有用性を強化します。これはすべて、より少ないコード行で行われます。現在、ISO C++ の `Ranges` アルゴリズムは実行ポリシー（つまり、アクセラレーター）をサポートしていません。oneDPL は、選択したアルゴリズムの Ranges を有効にし、カスタム SYCL* ビューなどの拡張機能を提供して、デバイス・プログラミングを有効にします。[7]

- 従来の C++ アルゴリズムは、関数呼び出しのブロッキング動作を含め、明確に定義されています。しかし、計算をアクセラレーターにオフロードするときに、ホスト・プロセッサーをブロックすることが常に適切であるとは限りません。ホストとデバイスの実行とデータ転送のインターリーブを可能にするため、一連の非同期アルゴリズムが oneDPL に追加されました。それらの機能は C++ アルゴリズムに似ていますが、ブロッキング動作はありません。非ブロッキング動作を制御するため、結果の代わりに C++ の `future` 形式のオブジェクトを返します。[8]

ほかにもあります。自動デバイス選択[4] のような機能も将来のリリースで計画されています。最新情報は、GitHub*（英語）をフォローしてください。

まとめ

oneDPL は、CPU、GPU、FPGA、その他のアクセラレーターに、高いパフォーマンスと生産性を備えた C++ ビルディング・ブロックを提供します。オープンスタンダード・ベースで、その仕様により、異なるプラットフォーム間での相互運用性が保証されています。インテルのリファレンス実装は、許容範囲でライセンスされたオープンソース・プロジェクトです。[5]

参考資料（英語）

1. 並列リサーチカーネル
2. リダクション抽象化機能の解析
3. SYCL* プログラミング言語
4. oneDPL 仕様
5. oneAPI DPC++ ライブラリー
6. インテルの Parallel STL と C++17 並列アルゴリズムを使用してパフォーマンスを向上する方法
7. oneDPL 範囲ベース API のアルゴリズム
8. oneDPL 非同期 API のアルゴリズム

oneAPI と oneDPL を使用したプログラミングの詳細（英語）

- oneDPL を使用してクロスプラットフォーム・プログラミングの労力を減らしハイパフォーマンスな並列コードを実現する
- インテル® oneAPI ベース・トレーニング・モジュール

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/2206/PDF/XL/SPI/ND