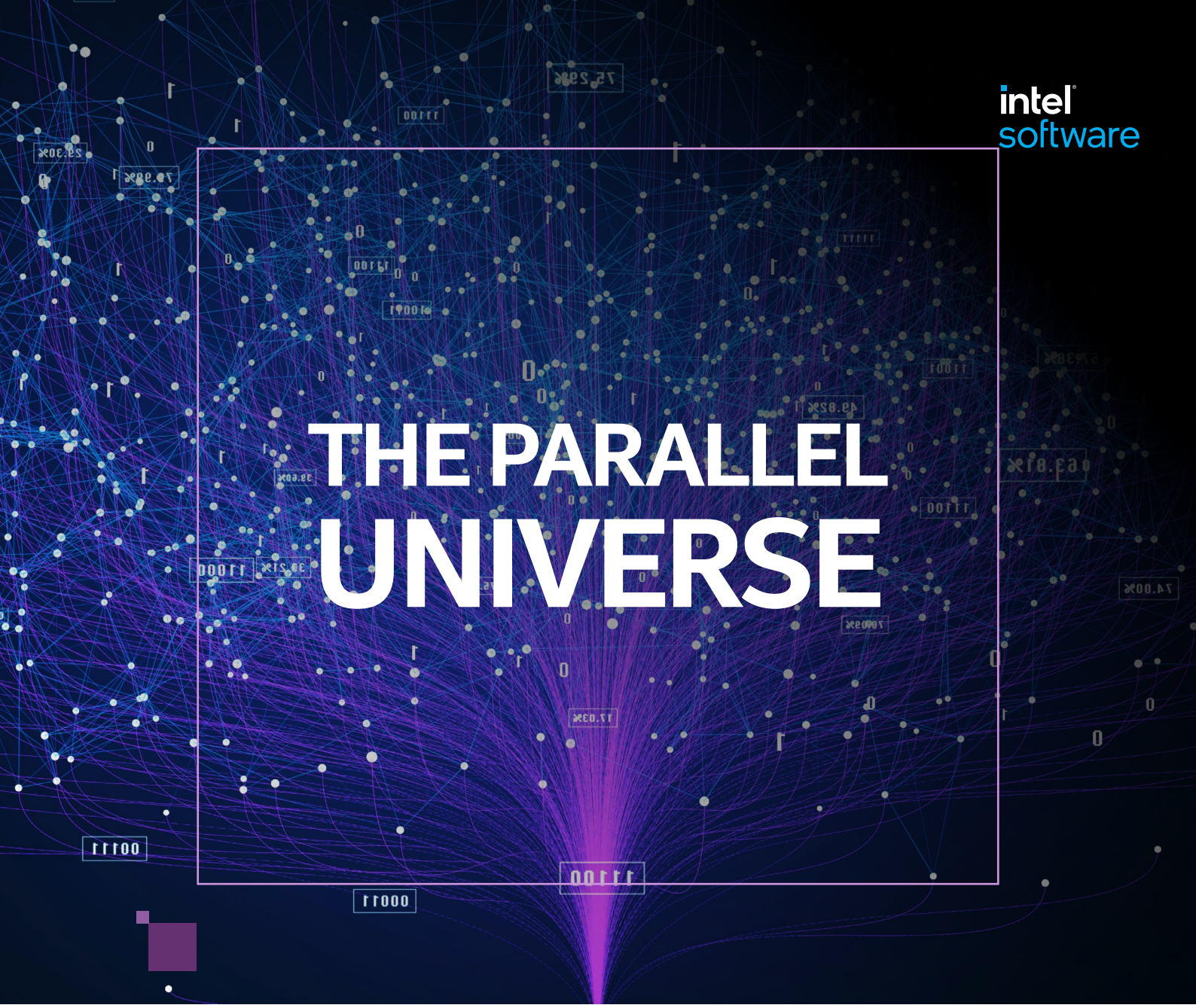


Habana* Gaudi* でディープラーニングのトレーニングを始めよう

コストを考慮したハイパーパラメーター最適化の重要性について

Databricks Runtime ML のスピードアップ

Issue
46
2021

目次

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。

ハイパフォーマンスなデータ・アナリティクス

The Parallel Universe の最近の号では oneAPI、特に DPC++ とインテル® oneMKL などのコンポーネント・ライブラリーを取り上げてきました。本号では、データサイエンスのマシンラーニングとディープラーニング・モデルのトレーニングに注目します。注目記事「**Habana* Gaudi* でディープラーニングのトレーニングを始めよう**」では、Gaudi* HPU（Habana* プロセッサー・ユニット）アーキテクチャーとその使用方法を紹介します。「**Databricks Runtime ML のスピードアップ**」では、クラウド上で人工知能（AI）を利用するためのインテルの最適化について説明します。続く「**ディープラーニングの推奨システム向けの新しいスケールアウト・トレーニング・ソリューション**」では、Facebook と協力してトレーニングのスケーラビリティーを向上した最近の取り組みを紹介します。そして、「**コストを考慮したハイパーパラメーター最適化の重要性について**」では、ハイパーパラメーターのチューニングを向上した Facebook および Amazon との最近の協業について述べます。

本号の後半では、エンドツーエンドのデータ・アナリティクス・パイプラインのもう 1 つの重要な部分である、グラフ・アナリティクスについて取り上げます。インテルはグラフ処理の研究において長い歴史を持ち、効率良くスケーラブルなグラフ・アナリティクスを実現するため、GraphBLAS 仕様（英語）、LDBC Graphalytics ベンチマーク（英語）、包括的なグラフ・アナリティクス分析（英語）、および PIUMA アーキテクチャー（英語）など、グラフ・アナリティクス分野の主要な組織と積極的なコラボレーションを行っています。データサイエンスの分野には、グラフやネットワークの分析に最適な NetworkX* というパッケージがありますが、パフォーマンスはそれほど優れていません。最近、Katana Graph から Python* プログラマー向けのハイパフォーマンスな並列グラフ・アナリティクス・ライブラリーがリリースされました。「**Katana のハイパフォーマンス・グラフ・アナリティクス・ライブラリー**」は、大規模なグラフで計算集約型操作を行う代替手段を提供します。

R プログラミング言語は、データ・サイエンティストや統計学者に人気がありますが、NetworkX* と同様にパフォーマンスはそれほど優れていません。「**インテル® oneAPI マス・カーネル・ライブラリーを使用した R コードの高速化**」では、コードを変更することなく R プログラミング環境をインテル® oneMKL にリンクするだけでパフォーマンスを向上する方法を示します。

本号の締めくくりは、44 号のベクトル化に関する記事「明示的なベクトル化を使用したスキャン操作の最適化」の続きです。「**インテル® AVX-512 命令を使用した Maxloc 操作の最適化**」では、ベクトル組込み関数を使用して、一般的なカーネルの maxloc リダクションを高速化する方法を紹介します。

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、Tech.Decoded（英語）を参照してください。

Henry A. Gabb
2021 年 10 月

Habana* Gaudi* でディープラーニングのトレーニングを始めよう

ディープラーニング向けの新しいハードウェアとソフトウェア・スタック

Greg Serochi Habana Labs アプリケーション・エンジニア兼テクニカル・プログラム・マネージャー

はじめに

ハイパフォーマンスなディープラーニング（DL）のトレーニングに対する需要は、ビデオのイメージやジェスチャー認識、音声認識、自然言語処理、推奨システムなどに基づくアプリケーションやサービスの増加に伴って高まっています。Habana* Gaudi* AI プロセッサーは、トレーニングのスループットと効率を最大限に高めつつ、多くのワークロードやシステムにスケーリングする最適化されたソフトウェアとツールを開発者に提供するように設計されています。Habana* Gaudi* ソフトウェアは、エンドユーザーを考慮して開発されており、ユーザーの独自モデルのニーズに対応した汎用性とプログラミングのしやすさを提供するとともに、ユーザーの既存モデルをシンプルかつシームレスに Gaudi* に移行することができます。Habana 社の Gaudi* HPU（Habana* プロセッサー・ユニット）をぜひ使用してみてください。GPU や CPU でワークロードを実行したことがある方は大勢いるでしょうが、この記

事では、既存の AI ワークロードとモデルを Gaudi* HPU に移行するプロセスを紹介します。Gaudi* でモデルを実行するのに必要なハードウェア・アーキテクチャー、ソフトウェア・スタック、そしてツールについて説明します。

Gaudi* アーキテクチャーと Habana* SynapseAI* ソフトウェア・スイート

基本アーキテクチャーの概要から見ていきましょう。Gaudi* は、DL トレーニング・ワークロードを高速化するためゼロから設計されています。Gaudi* のヘテロジニアス・アーキテクチャーは、完全にプログラム可能なテンソル・プロセシング・ユニット（TPC）のクラスターと、関連する開発ツールやライブラリー、そして構成可能な行列演算エンジンで構成されています。

TPC は、トレーニング・ワークロードに特化した命令セットとハードウェアを備えた VLIW SIMD プロセッサーです。プログラム可能であり、最大限の柔軟性と、次のようなワークロード向けの機能を提供します。

- 高速な GEMM 操作
- テンソル・アドレッシング
- レイテンシーを隠蔽する機能
- 乱数生成
- 特殊関数の高度な実装

TPC は、FP32、BF16、INT32、INT16、INT8、UINT32、UINT16、および UINT8 データ型をネイティブサポートします。

Gaudi* メモリー・アーキテクチャーは、各 TPC にオンダイ SRAM とローカルメモリーを搭載しています。さらに、このチップパッケージには 4 つの HBM デバイスが統合されており、32GB の容量と 1TB/ 秒の帯域幅を実現します。PCIe* インターフェイスはホストとのインターフェイスを提供し、第 3.0 世代と第 4.0 世代の両方のモードをサポートします。

Gaudi* は、RoCE（RDMA over Converged Ethernet）v2 エンジンをオンチップに搭載した最初の DL トレーニング・プロセッサーです。最大 2Tb/ 秒の双方向スループットを持つこのエンジンは、トレーニング・プロセスに必要なプロセッサー間通信において重要な役割を果たします。RoCE がネイティブに統合されているため、同じスケーリング技術をサーバーやラックの内部で使用することができ（スケールアップ）、またラック間でのスケーリングも可能となります（スケールアウト）。これらは、Gaudi* プロセッサー間で直接接続することも、標準的なイーサネット・スイッチを使って接続することもできます。**図 1** にハードウェア・ブロック図を示します。

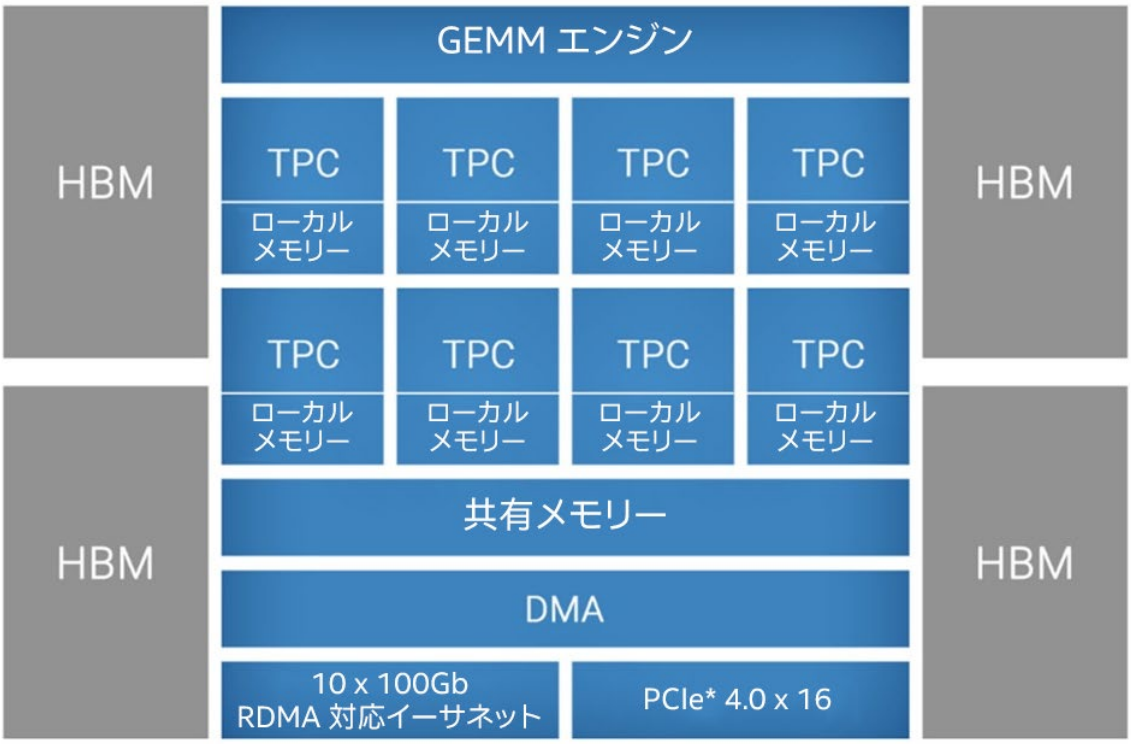


図 1. Gaudi* HPU ブロック図

Habana 社の Gaudi* アクセラレーターでハイパフォーマンスな DL トレーニングを容易にするため設計された SynapseAI* ソフトウェア・スイートは、ニューラル・ネットワークのトポロジーを Gaudi* ハードウェア上に効率良くマッピングできます。ソフトウェア・スタックには、Habana 社のグラフ・コンパイラーとランタイム、TPC カーネル・ライブラリー、ファームウェアとドライバー、カスタムカーネル開発用の TPC SDK や SynapseAI* Profiler などの開発ツールが含まれています。SynapseAI* は、Gaudi* 向けにパフォーマンスが最適化されており、一般的なフレームワークである TensorFlow* や PyTorch* と統合されます。**図 2** に SynapseAI* ソフトウェア・スイートのコンポーネントを示します。SynapseAI* ソフトウェアを作業環境に簡単に統合できるように、Habana* 社では、モデルを実行する環境の作成に必要なすべてのものを含む TensorFlow* と PyTorch* の Docker* イメージを提供しています。これらのライブラリーをモデルに統合する方法を見てみましょう。

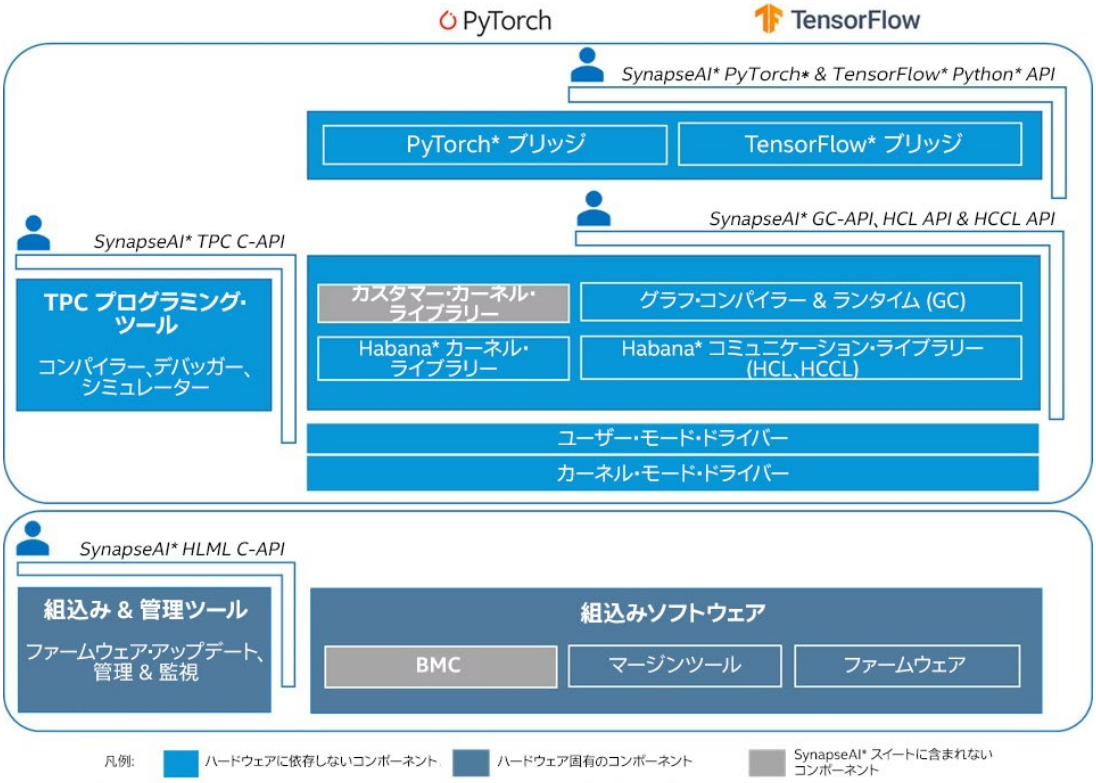


図 2. SynapseAI* ソフトウェア・スイート

環境設定

このセクションでは、環境を設定して、Gaudi* HPU がフレームワークによって認識され、モデル内で操作を実行できるようにするために必要な簡単なステップを追加します。Docker* イメージに加えて、Habana 社では、モデルに適切なコンポーネントを追加するガイドとして利用できるリファレンス・モデルとサンプルを Model-References GitHub* リポジトリー（英語）で公開しています。

最初のステップは、Habana* ドライバー、SynapseAI* ソフトウェア・スタック、およびフレームワークを含む完全なビルド環境を準備することです。ほとんどの場合、この環境を確実に作成する最良の方法は、Habana* 社が提供している(英語)ビルド済みの Docker* イメージを使用することです。この Docker* イメージには、単一ノードとスケールアウトの両方のバイナリーが含まれており、追加のインストール手順は必要ありません。クラウドベースの環境で Gaudi* を使用する場合は、フルドライバーと関連フレームワークを含む、クラウド・サービス・プロバイダーが提供する Habana* Gaudi* イメージを必ず選択してください。フルドライバーと SynapseAI* ソフトウェア・スタックのインストールを別途（オンプレミスのインストールとして）実行する必要がある場合は、Setup and Install GitHub* リポジトリー（英語）にある詳細な手順を参照してください。第 2 ステップでは、Habana* ライブラリーをロードして、Gaudi* HPU デバイスをターゲットにします。次のセクションでは、TensorFlow* と PyTorch* の両方について、この方法を説明します。

TensorFlow* を使用する

Habana* は `tf.load_library` と `tf.load_op_library` を使用して、TensorFlow* フレームワークと SynapseAI* を統合し、ライブラリー・モジュールやカスタムの操作 / カーネルを呼び出します。フレームワークの統合には、3 つの主要コンポーネントが含まれます。

- SynapseAI* ヘルパー
- デバイス
- グラフパス

TensorFlow* フレームワークは、グラフのビルドと実行に必要なオブジェクトのほとんどを制御します。SynapseAI* は、ユーザーがデバイスでグラフを作成、コンパイル、起動できるようにします。グラフ・パス・ライブラリーは、PAMSEN (Pattern Matching, Marking, Segmentation, and Encapsulation — パターン一致、マーク、セグメンテーション、カプセル化) 操作により TensorFlow* グラフを最適化します。TensorFlow* グラフを操作して、Gaudi* のハードウェア・リソースを最大限に活用できるように設計されています。Gaudi* 向けの実装を持つグラフノードのコレクションに対して、PAMSEN はグラフの正当性を維持しながら、できるだけ多くのグラフノードをマージしようとします。グラフのセマンティクスを維持しつつ、1 つのエンティティーに融合できる部分グラフを自動検出することで、PAMSEN はネイティブの TensorFlow* と同等か、それ以上のパフォーマンスを実現します。

モデルを準備するには、Habana* モジュール・ライブラリーをロードする必要があります。`library_loader.py` にある `load_habana_module()` を呼び出します。この関数は、TensorFlow* レベルで Gaudi* HPU を使用するのに必要な Habana* ライブラリーをロードします。

```
import tensorflow as tf
from habana_frameworks.tensorflow import load_habana_module
load_habana_module()
tf.compat.v1.disable_eager_execution()    # tf.session を使用する場合
```

レガシー (TF1.x) モデルを実行するにはいくつかの要件があり、Horovod* をマルチカードやマルチノードのトレーニングに使用する場合は要件が異なります。1 つのカードで TF2.x モデルを実行する場合、最後の文は必要ありません。

ロードすると、Gaudi* HPU は TensorFlow* に登録され、CPU よりも優先されます。つまり、CPU と Gaudi* HPU の両方で利用可能な場合、操作は Gaudi* HPU に割り当てられます。上記の手順で初期モデルの移行が完了したら、`habana_ops` で定義されているように Habana* 操作とカスタム TensorFlow* 操作を統合します。`from habana_frameworks.tensorflow import habana_ops` コマンドでインポートできます。これは、`load_habana_module()` を呼び出した後に使用します。カスタム操作は、標準の TensorFlow* 操作とのパターン一致に使用されます。モデルが Gaudi* HPU に移行されると、ソフトウェア・スタックは CPU と Gaudi* HPU に配置する操作を決定します。最適化パスでは、サポートされていない操作を自動的に CPU に配置します。

これらの手順が完了すると、モデルを Gaudi* HPU インスタンスで実行できるようになります。以下は、MNIST モデルを Gaudi* で有効にする簡単な例です (必要な最小限の変更は太字で示しています)。

```
import tensorflow as tf
from TensorFlow.common.library_loader import load_habana_module
load_habana_module()
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(10),
])
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=128)
model.evaluate(x_test, y_test)
```

PyTorch* を使用する

PyTorch* Habana* ブリッジは、フレームワークと SynapseAI* ソフトウェア・スタック間のインターフェイスとなり、Habana* Gaudi* デバイス上でディープラーニング・モデルの実行を促進します (**図 3**)。Habana 社が提供するインストール・パッケージは、標準の PyTorch* リリースに変更を加えたもので、Habana 社が提供する Docker* イメージに含まれています。このインストール・パッケージのカスタマイズされたフレームワークを使用して、PyTorch* と Habana* ブリッジを統合する必要があります。ここでは、PyTorch* Habana* プラグイン・ライブラリーと `import habana_frameworks.torch.core` モジュールをロードして Habana* ブリッジと統合し、PyTorch* のディープラーニング・モデルのトレーニング・スクリプトを変更します。

Habana* PyTorch* は、Gaudi* HPU 上での PyTorch* モデルの実行で 2 つのモードをサポートしています。

- Eager モード – 標準の PyTorch* の Eager モードスクリプトで定義されているように操作ごとに実行します。
- Lazy モード – Eager モードと同様に、スクリプトから 1 つずつ提供される操作で構成されるグラフを遅延実行します。Eager モードを Gaudi* のパフォーマンスで体験できます。

一般に、初期モデルの開発では、機能を確立するため Eager モードで実行し、その後、最高のパフォーマンスを得るため Lazy モードで実行することが推奨されます。モードは、ランタイムフラグで選択できます (以下の**ステップ 5** を参照)。

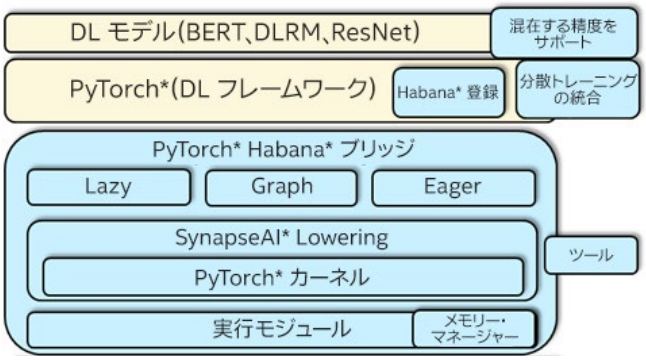


図 3. PyTorch* Habana* フルスタック・アーキテクチャー

Habana* 上でモデルを実行するには、以下のコードを追加する必要があります。以下のステップは、Eager モードと Lazy モードの実行に適用されます。

1. Habana* PyTorch* プラグイン・ライブラリー `libhabana_pytorch_plugin.so` をロードします。

```
import torch
from habana_frameworks.torch.utils.library_loader import load_habana_module
load_habana_module()
```

2. Gaudi* HPU デバイスをターゲットに指定します。

```
device = torch.device("hpu")
```

3. モデルをデバイスに移動します。

```
model.to(device)
```

4. Habana* Torch ライブラリーをインポートします。:

```
import habana_frameworks.torch.core as htcore
```

5. 以下のように環境変数を設定して Lazy 実行モードを有効にします。コードを Eager モードで実行する場合は、この環境変数を設定しないでください。

```
os.environ["PT_HPU_LAZY_MODE"] = "1"
```

6. Lazy モードでは、Habana* デバイスからホストにデータが読み取られると実行がトリガーされます。例えば、トポロジーを実行していて、`loss.item()` でホストがデバイスからロス値を取得すると実行がトリガーされます。コードに `mark_step()` を追加して実行をトリガーすることもできます。

```
htcore.mark_step()
```

トレーニング・スクリプトの以下の場所に `mark_step()` を配置する必要があります。

- トレーニングの反復を明示的にするため、`optimizer.step()` の直後に配置します。
- Habana* カスタム・オプティマイザーを使用する場合は `loss.backward` と `optimizer.step()` の間に配置します。

Gaudi* HPU で実行するためモデルが移行されると、ソフトウェア・スタックは CPU と HPU に配置する操作を決定します。この決定は、操作が HPU をバックエンドとして PyTorch* に登録されているかどうかに基づいて行われます。HPU をバックエンドとして PyTorch* に登録されていない場合、操作の実行は自動的にホスト CPU にフォールバックします。

Habana 社は、Habana* デバイス向けにカスタマイズされたいくつかの複雑な PyTorch* 操作の独自の実装を提供しています。モデルでこれらの複雑な操作をカスタム Habana* バージョンに置き換えると、パフォーマンスが向上します。

以下は、Habana* デバイスで現在サポートされているカスタム・オプティマイザーのリストです。

- FusedAdagrad – torch.optim.Adagrad（英語）を参照
- FusedAdamW – torch.optim.AdamW（英語）を参照
- FusedLamb – LAMB optimizer paper（英語）を参照
- FusedSGD – torch.optim.SGD（英語）を参照

まとめ

この記事では、既存のディープラーニング・モデルを Gaudi* に移行するプロセスを紹介し、モデルを実行できるようにする基本的な手順を示しました。製品の最新情報は、Habana.ai（英語）を参照してください。メールニュースにサインアップすると、選択したトピックの今後の活動に関する通知を受け取ることができます。

関連資料

- 開発者サイト（英語）は開始点として最適です。モデル、ドキュメント、その他のリソースの情報が得られます。
- Habana* ドキュメント（英語）には、**移行ガイド**をはじめとする詳細なドキュメントがあります。
- Habana* Vault（英語）では、最新のドライバーや Docker* イメージをダウンロードできます。
- ユーザーフォーラム（英語）は、関連するすべてのトピックについて議論するフォーラムです。

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）>

Databricks Runtime ML のスピードアップ

インテルにより最適化されたクラウド上の人工知能

Haifeng Chen インテル コーポレーション シニア・ソフトウェア・アーキテクト
Qing Yao 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
Xiangxiang Shen 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

1 インテルにより最適化されたマシンラーニング・ライブラリー

1.1 scikit-learn*

scikit-learn* は、Python* プログラミング言語用の人気の高いオープンソースのマシンラーニング（ML）ライブラリーです。サポート・ベクトル・マシン、ランダムフォレスト、勾配ブースティング、K 平均法、DBSCAN など、さまざまな分類、回帰、クラスタリングのアルゴリズムを搭載し、Python* の数値および科学ライブラリーである NumPy* や SciPy* と相互運用できるように設計されています。

インテル® oneAPI AI アナリティクス・ツールキット（英語）に含まれるインテル® Extension for scikit-learn* は、ML パフォーマンスを向上します。これにより、データ・サイエンティストはより多くの時間をモデルに費やすことができます。インテルはまた、インテル® ディストリビューションの Python* により Python* 自体のパフォーマンスを

最適化するとともに、scikit-learn* と一緒に使用される XGBoost、NumPy*、SciPy* などの主要なデータ・サイエンス・ライブラリーを最適化しています。これらの拡張のインストールと使用については、こちらの記事（英語）を参照してください。

1.2 TensorFlow*

TensorFlow* もまた、エンドツーエンドの ML およびディープラーニング（DL）アプリケーションを開発するオープンソースのフレームワークとして人気があります。TensorFlow* は、ツール、ライブラリー、コミュニティー・リソースからなる包括的で柔軟なエコシステムを備えており、研究者は簡単にアプリケーションを構築して展開できます。

インテル® プロセッサーのパフォーマンスを最大限に活用するため、TensorFlow* 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）プリミティブを使用して最適化されています。最適化の詳細やパフォーマンス・データについては、「最新のインテル® アーキテクチャーでの TensorFlow* の最適化」（英語）を参照してください。

2 Databricks Runtime ML

Databricks は、データ・エンジニアリング、ML、コラボレーティブ・データ・サイエンスのための統合データ・アナリティクス・プラットフォームです。データ集約型アプリケーションを開発する包括的な環境を提供します。Databricks Runtime for Machine Learning (Databricks Runtime ML)（英語）は、実験の追跡、モデルのトレーニング、特徴の開発と管理、特徴とモデルの提供を行うマネージドサービスを含む、統合されたエンドツーエンドの環境です。TensorFlow*、PyTorch*、Keras*、XGBoost などの代表的な ML/DL ライブラリーに加えて、Horovod* などの分散学習に必要なライブラリーも含まれています。

Databricks は、Amazon Web Services*、Microsoft* Azure*、Google Cloud Platform* サービスと統合されています。これらのクラウド・サービス・プロバイダーは、プロダクション用インフラストラクチャーの管理やプロダクション用ワークロードの実行に大きな利便性をもたらします。クラウドサービスは無料ではありませんが、最適化されたライブラリーを利用することで、運用コストを軽減できる可能性があります。この記事では、Azure* Databricks を使用して、ソリューションとパフォーマンスを示します。

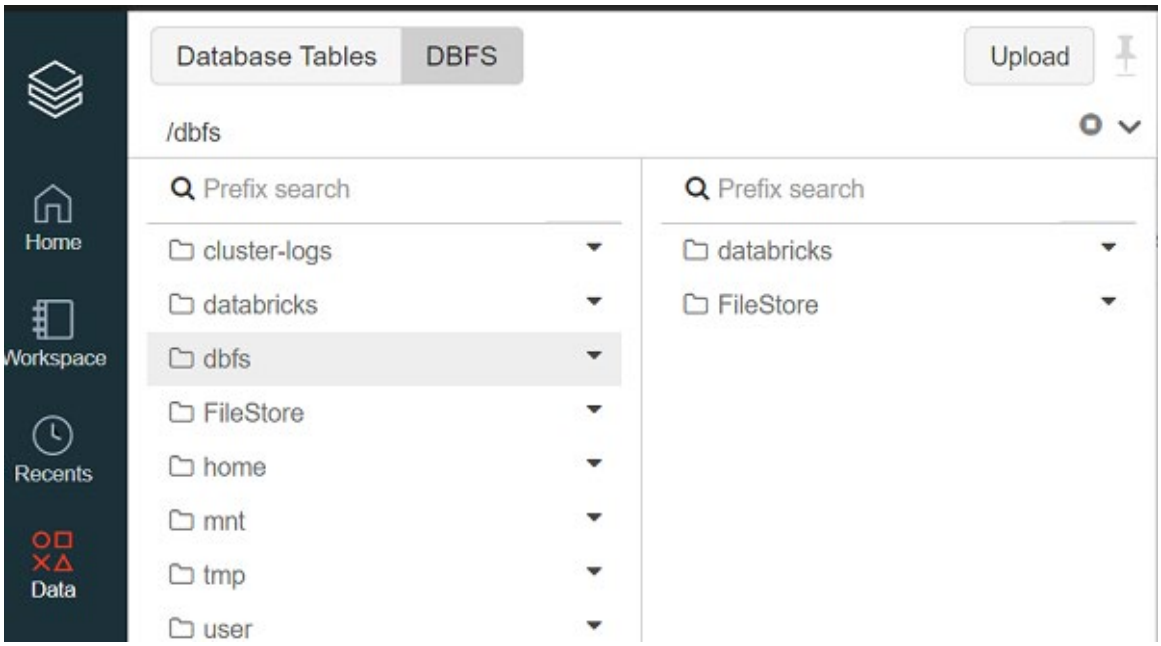
3 Azure* Databricks 上のインテルにより最適化された ML ライブラリー

Databricks Runtime ML には、scikit-learn* と TensorFlow* のストックバージョンが含まれています。パフォーマンスを向上するため、これらをインテルにより最適化されたバージョンに置き換えます。Databricks は、カスタマイズを容易にするため初期化スクリプト（英語）を提供しています。このスクリプトは、各クラスターノードの起動時に実行されます。インテルにより最適化されたバージョンの scikit-learn* と TensorFlow* を組み込むため、静的にパッチを適用したバージョンをインストールするかどうかに応じて、2 つの初期化スクリプトを開発しました。

1. init_intel_optimized_ml.sh（英語）は、静的にパッチを適用した scikit-learn* と TensorFlow* をランタイム環境にインストールします。
2. init_intel_optimized_ml_ex.sh（英語）は、インテル® Extension for scikit-learn* と TensorFlow* をランタイム環境にインストールします。

以下の手順に従って、クラスターを作成できます。最初に、初期化スクリプトを DBFS にコピーします。

1. `init_intel_optimized_ml.sh` または `init_intel_optimized_ml_ex.sh` をローカルフォルダーにダウンロードします。
2. 左のサイドバーにある **[Data]** アイコンをクリックします。
3. 画面上部にある **[DBFS]** ボタンをクリックしてから、**[Upload]** ボタンをクリックします。
4. **[Upload Data to DBFS]** ダイアログでターゲット・ディレクトリー（例：`FileStore`）を選択します。
5. 以前のステップでローカルフォルダーにダウンロードしたローカルファイルを参照して、**[Files]** ボックスにアップロードします。

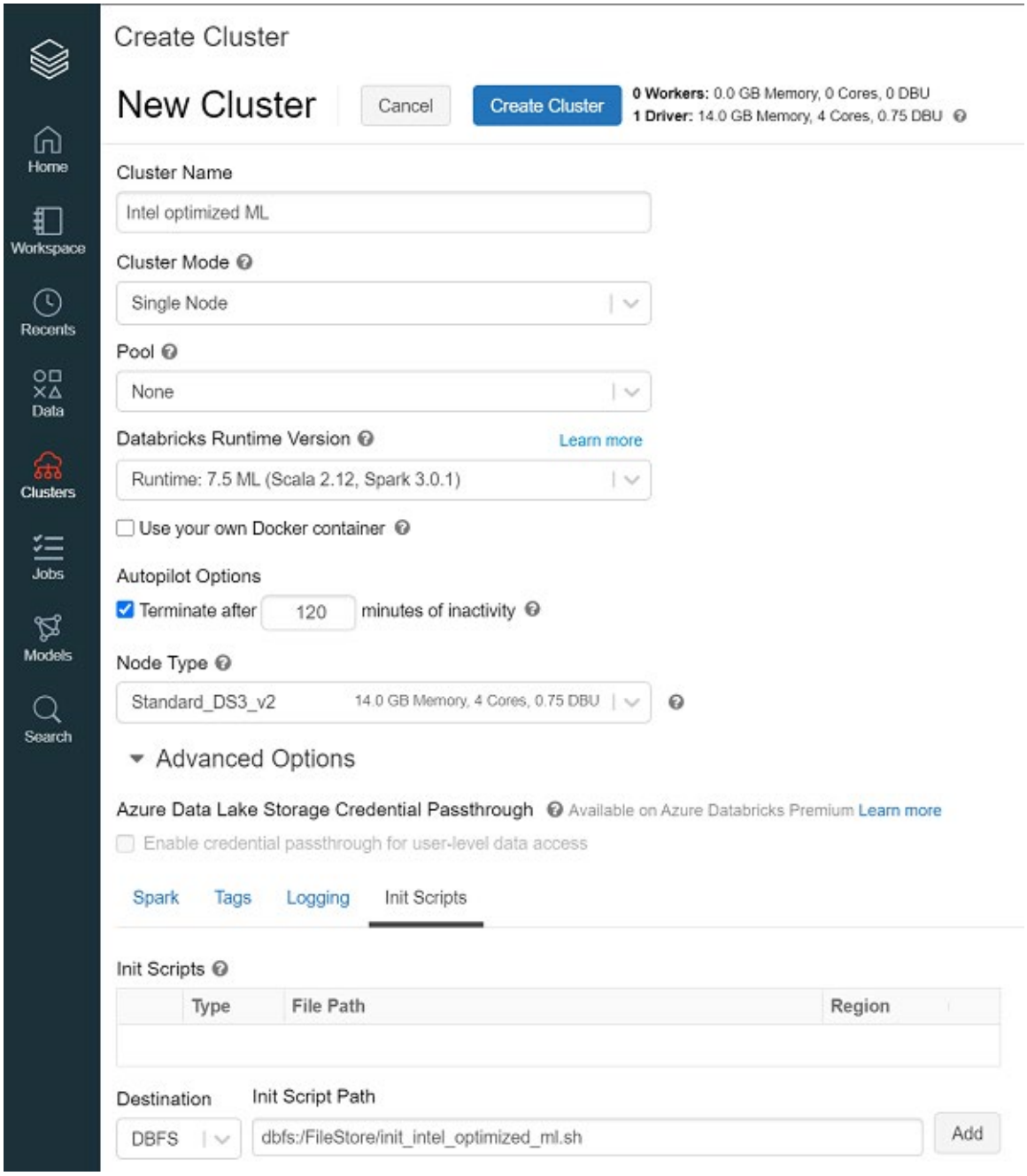


アップロードした初期化スクリプトを使用して、Databricks クラスターを起動します。

1. クラスターの設定ページで **[Advanced Options]** トグルをクリックします。
2. 右下の **[Init Scripts]** タブをクリックします。
3. **[Destination]** ドロップダウン・メニューから **[DBFS]** デスティネーション・タイプを選択します。
4. 以前のステップでアップロードした初期化スクリプトのパスを指定します。

   `dbfs:/FileStore/init_intel_optimized_ml.sh` または
   `dbfs:/FileStore/init_intel_optimized_ml_ex.sh`

5. **[Add]** をクリックします。



詳細は、「Databricks 向けにインテルにより最適化された ML」（英語）を参照してください。

4 パフォーマンス測定

4.1 scikit-learn* トレーニングと推論パフォーマンス

以下のベンチマークでは Databricks Runtime バージョン 7.6 ML を使用します。scikit-learn_bench（英語）により、インテルによる最適化を使用した場合としない場合の一般的な scikit-learn* アルゴリズムのパフォーマンスを比較します。便宜上、Databricks Cloud 上で `scikit-learn_bench` を実行する benchmark_sklearn.ipynb（英語）ノートブックが提供されています。

単一ノードの Databricks クラスターを、ストック・ライブラリーとインテルにより最適化されたバージョンで作成し、トレーニングと推論パフォーマンスを比較します。どちらのクラスターも Standard_F16s_v2 インスタンス・タイプを使用しています。

どちらのクラスターでもベンチマーク・ノートブックを実行しました。トレーニングと推論の正確なパフォーマンス・データが得られるように、アルゴリズムごとに複数の設定（英語）を使用しました。**表 1** に各アルゴリズムの 1 つの設定のパフォーマンス・データを示します。

アルゴリズム	入力設定	トレーニング時間（秒）		推論時間（秒）	
		ストック scikit-learn* (ベースライン)	インテル® Extension for scikit-learn*	ストック scikit-learn* (ベースライン)	インテル® Extension for scikit-learn*
`kmeans`	`config1`	517.13	24.41	6.54	0.42
`ridge_regression`	`config1`	1.22	0.11	0.05	0.04
`linear_regression`	`config1`	3.1	0.11	0.05	0.04
`logistic_regression`	`config3`	87.5	5.94	0.5	0.08
`svm`	`config2`	271.58	12.24	86.76	0.55
`kd_tree_knn_classification`	`config4`	0.84	0.11	1584.3	25.67

表 1. トレーニングと推論パフォーマンスの比較（時間はすべて秒単位）

各アルゴリズムにおいて、インテル® Extension for scikit-learn* は、トレーニングと推論パフォーマンスを大幅に向上し、`svm` や `brute_knn` などの一部のアルゴリズムでは、桁違いの高速化を達成しました（**図 1**）。

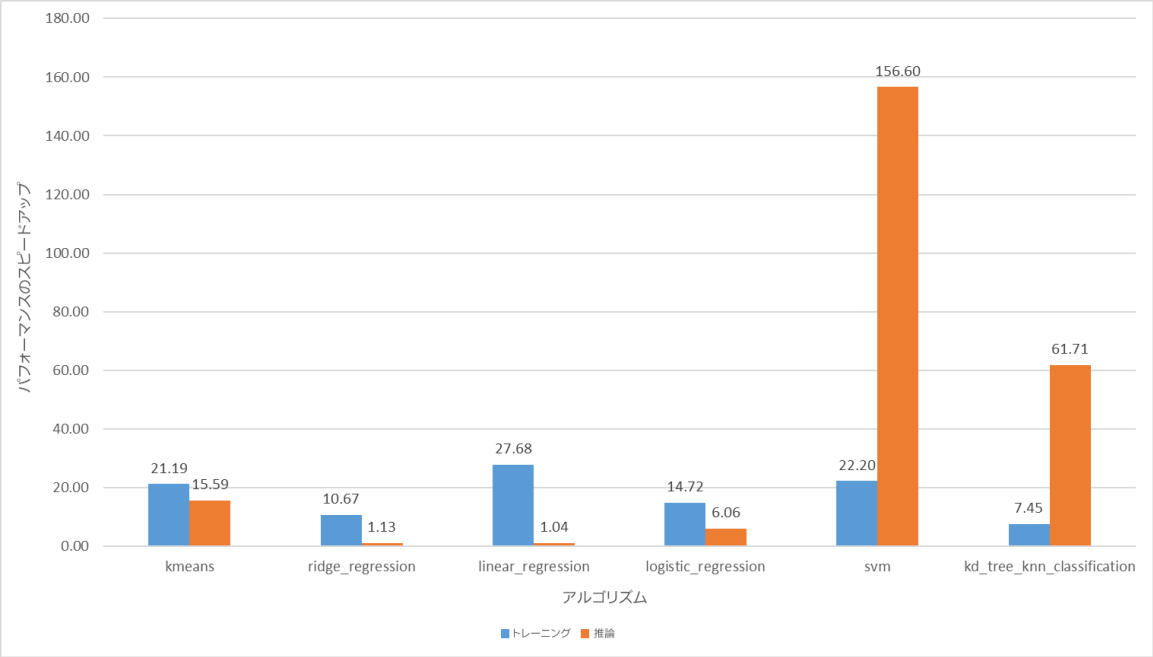


図 1. ストックバージョンに対するインテル® Extension for scikit-learn* のトレーニングと推論スピードアップ

4.2 TensorFlow* のトレーニングと推論パフォーマンス

BERT（Bidirectional Encoder Representations from Transformers）（英語）は、言語表現を事前にトレーニングする新しい手法で、さまざまな自然言語処理タスクで最先端の結果を得ることができます。Model Zoo（英語）には、インテル® Xeon® スケーラブル・プロセッサー向けにインテルが最適化した、多くの一般的なオープンソースのマシンラーニング・モデルのトレーニング済みモデル、サンプルスクリプト、ベスト・プラクティス、ステップごとのチュートリアルへのリンクが含まれています。

ここでは、Model Zoo（英語）を使用して、SQuADv1.1 データセットで BERT-Large モデル（英語）を実行し、ストックバージョンとインテル® Optimization for TensorFlow* のパフォーマンスを比較します。ここでも、Databricks Cloud でベンチマークを実行するノートブック (benchmark_tensorflow_bertlarge.ipynb（英語)) が提供されています。詳細は、「パフォーマンス比較ベンチマークの実行」（英語）を参照してください。

TensorFlow* のパフォーマンスを評価するため、Standard_F32s_v2、Standard_F64s_v2、および Standard_F72s_v2 インスタンス・タイプの Databricks Cloud Single Node を使用しました。インスタンス・タイプごとに、ストックバージョンの TensorFlow* とインテル® Optimization for TensorFlow* のトレーニングと推論パフォーマンスを比較しました。後者は、Standard_F32s_v2、Standard_F64s_v2、および Standard_F72s_v2 インスタンスの Databricks Runtime ML で、それぞれ 2.09 倍、1.95 倍、および 2.18 倍の推論パフォーマンスを達成しました（**図 2**)。トレーニングでは、インテル® Optimization for TensorFlow* は、Standard_F32s_v2、Standard_F64s_v2、および Standard_F72s_v2 インスタンスで、それぞれ 1.76 倍、1.70 倍、および 1.77 倍のパフォーマンスを達成しました（**図 3**)。

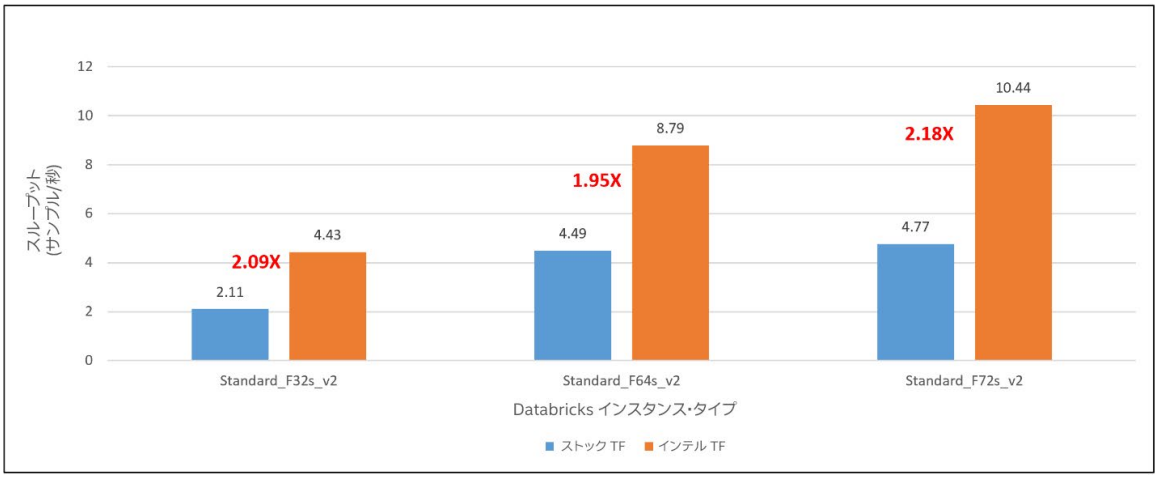


図 2. ストックバージョンに対するインテル® Optimization for TensorFlow* の推論スピードアップ

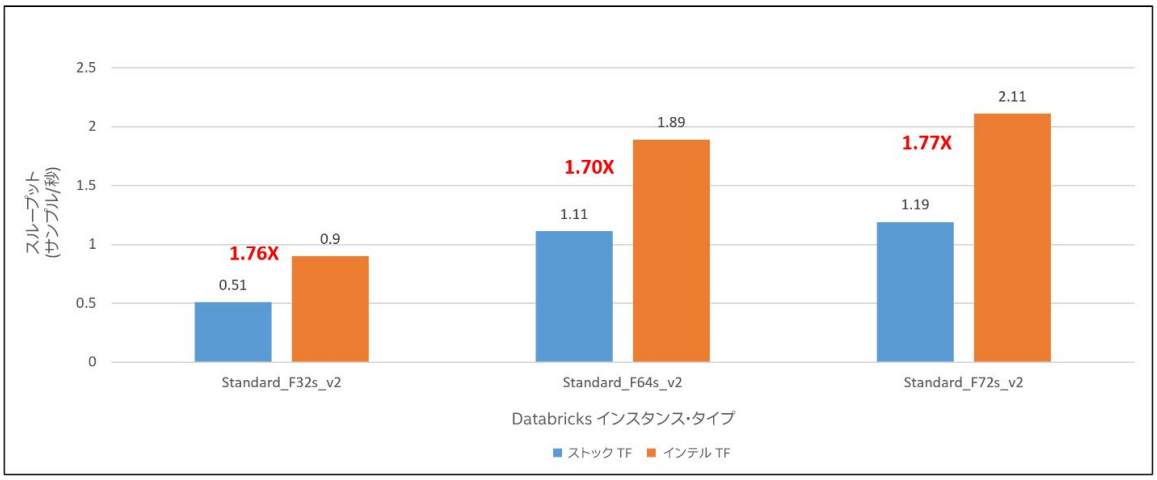


図 3. ストックバージョンに対するインテル® Optimization for TensorFlow* のトレーニング・スピードアップ

5 まとめ

インテルにより最適化されたバージョンの scikit-learn* と TensorFlow* は、インテルの XPU でトレーニングと推論パフォーマンスを大幅に向上しました。この記事では、Databricks Runtime ML に含まれるストックバージョンの scikit-learn* と TensorFlow* を最適化されたバージョンに置き換えることで、パフォーマンスを向上し、コストを軽減できることを示しました。

ディープラーニングの推奨システム向けの新しいスケールアウト・トレーニング・ソリューション

MLPerf* ベンチマークで優れた並列スケーリングを達成

Liangang Zhang 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
Guokai Ma 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
Roger Feng インテル コーポレーション ディープラーニング・ソフトウェア・エンジニア
Fan Zhao インテル コーポレーション ディープラーニング・ソフトウェア・エンジニア
Ke Ding インテル コーポレーション 主席 AI エンジニア

DLRM（Deep Learning Recommendation Model）は、Facebook が導入したディープラーニング・ベースの推奨モデルです。最先端のモデルであり、MLPerf* トレーニング・ベンチマークの一部となっています。DLRM は、計算、メモリー、I/O 依存の処理のバランスを取る必要があるため、シングルソケットおよびマルチソケットの分散型トレーニングにおいて独自の課題を抱えています。この課題を解決するため、データとモデルの並列化、新しいハイブリッドの Split-SGD + LAMB オプティマイザー、より大きなグローバル・バッチ・サイズでモデルを収束させる効率良いハイパーパラメーター・チューニング、スケールアップおよびスケールアウトをサポートする新しいデータローダーを用いて、インテル® Xeon® プロセッサー・ベースのクラスター上で DLRM トレーニングの効率的なスケールアウト・ソリューションを実装しました。MLPerf* v1.0 のトレーニング結果では、64 個のインテル® Xeon® 8376H プロセッサーを使用して、DLRM を 15 分でトレーニングできることが示されています。これは、MLPerf* v0.7（16 個のインテル® Xeon® 8380 プロセッサーにしかスケーリングできなかった）に比べて 3 倍のパフォーマンス向上です。この記事では、このパフォーマンス向上を実現するために行った最適化について説明します。

垂直分割埋め込みテーブルベースのハイブリッド並列処理

DLRM MLPerf* トレーニングのスケーラビリティーを向上させるためハイブリッド並列ソリューションを使用し、さらにスケーラビリティー向上のため垂直分割埋め込みテーブルを使用します（**図 1**）。

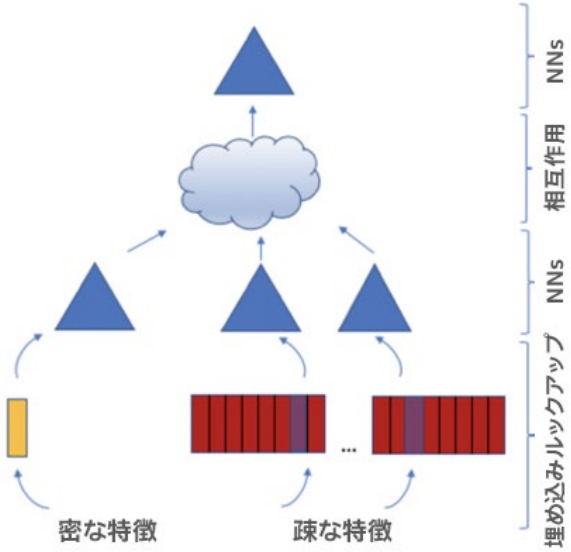


図 1. DLRM モデルの構造。密な特徴は下位 MLP(多層パーセプトロン)層の入力であり、疎な特徴は埋め込みテーブルの入力です。下位 MLP と埋め込みからの出力は、上位 MLP 層の入力となります。

MLPerf* の DLRM トレーニングには 26 個の埋め込みテーブルがあります。テーブルのエントリー番号は、40M、40M、40M、40M、40790948、3067956、590152、405282、39060、20265、17295、12973、11938、7424、7122、2209、1543、976、155、108、63、36、14、10、4、および 3 です。各エントリーのセマンティクスをモデル化するため、128 個の Float32/Bloat16 数値が使用されています。埋め込みテーブルを処理する簡単な方法は、疎な all-reduce によるデータ並列処理です。モデルのウェイトは、モデル・インスタンス間で複製する必要があります。また、データ並列処理ではすべての埋め込みテーブルを複製する必要があるため、26 個の埋め込みテーブルを使用すると、1 つのモデル・インスタンスに 100GB 以上のメモリーが必要になります。

通信のオーバーヘッドと各デバイスの必要メモリーを軽減するため、ハイブリッド並列分散トレーニング・ソリューションを使用しています（**図 2** と **3**）。埋め込みテーブルは、密な勾配を使用する小さなテーブルと、疎な勾配を使用する大きな埋め込みテーブルに分けられます。MLPerf* DLRM の場合、埋め込みテーブルはエントリー番号が 2048 未満の場合は小さなテーブルとして、それ以上は大きなテーブルとして扱われます。ここでは、10 個の小さな埋め込みテーブルと 16 個の大きな埋め込みテーブルがあります。モデルを並列処理する関係からモデル・インスタンスは、大きな埋め込みテーブルの一部のローカルコピーを保持します。例えば、8 つのソケットを使用し、1 つのソケットに 1 つのインスタンスを使用した場合、すべてのインスタンスが 2 つの大きな埋め込みテーブルを保持します。16 ソケットの場合、各インスタンスは 1 つの大きな埋め込みテーブルしか保持しません。

各モデル・インスタンスは、ローカルバッチのインデックスで埋め込みテーブルをルックアップする代わりに、グローバルバッチのインデックスでローカルの埋め込みテーブルをルックアップします。ルックアップ操作の後、モデル・インスタンスは、自分のローカルバッチのルックアップ・エントリーだけでなく、ほかのインスタンスのバッチのルックアップ・エントリーも保持することになります。ランク間の埋め込み情報の交換には、all-to-all の集合通信が使用されます。データ並列処理を行うため、下位 MLP、上位 MLP、10 個の小さな埋め込みテーブルをすべてのモデル・インスタンスで複製し、ランク間の勾配を平均化するため all-reduce 集合通信が使用されます。

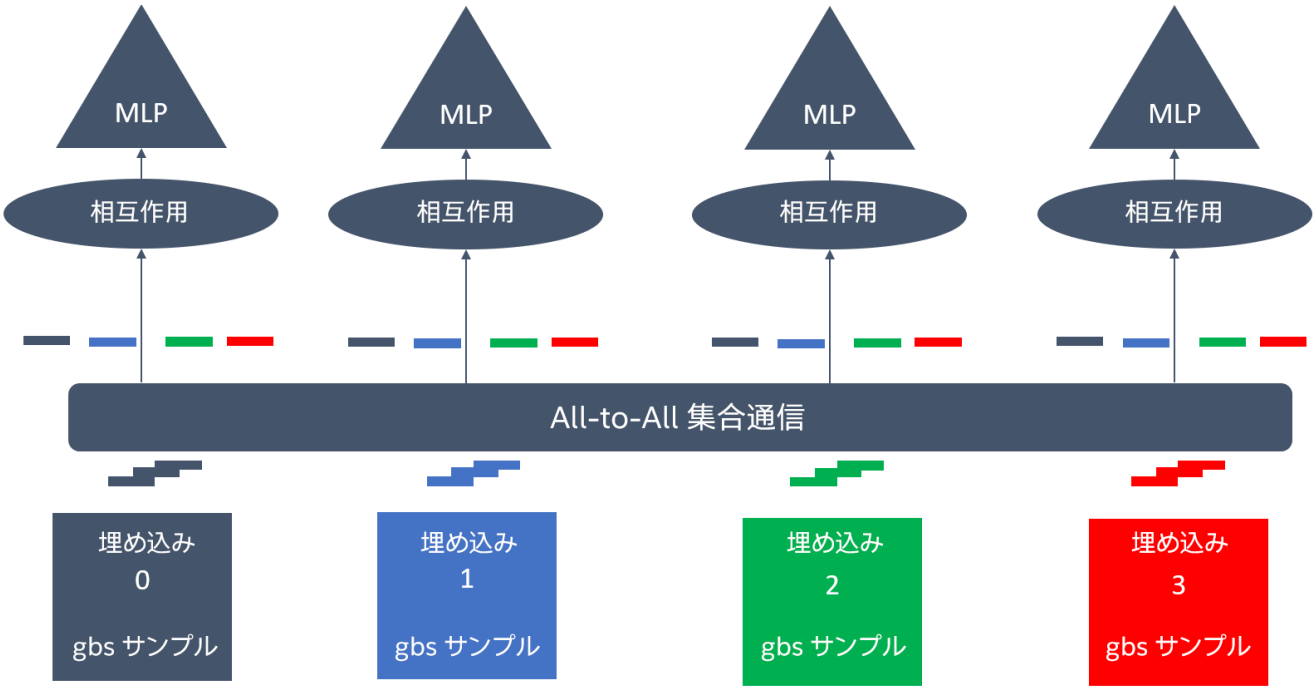


図 2. 4 つの埋め込みテーブルを持つ 4 つのモデル・インスタンス間のハイブリッド並列処理を示しています。色付きのブロックは異なる埋め込みテーブルを示します（gbs: グローバル・バッチ・サイズ）。同じ埋め込みテーブルからのルックアップ・エントリーが異なるインスタンスに分散されています。

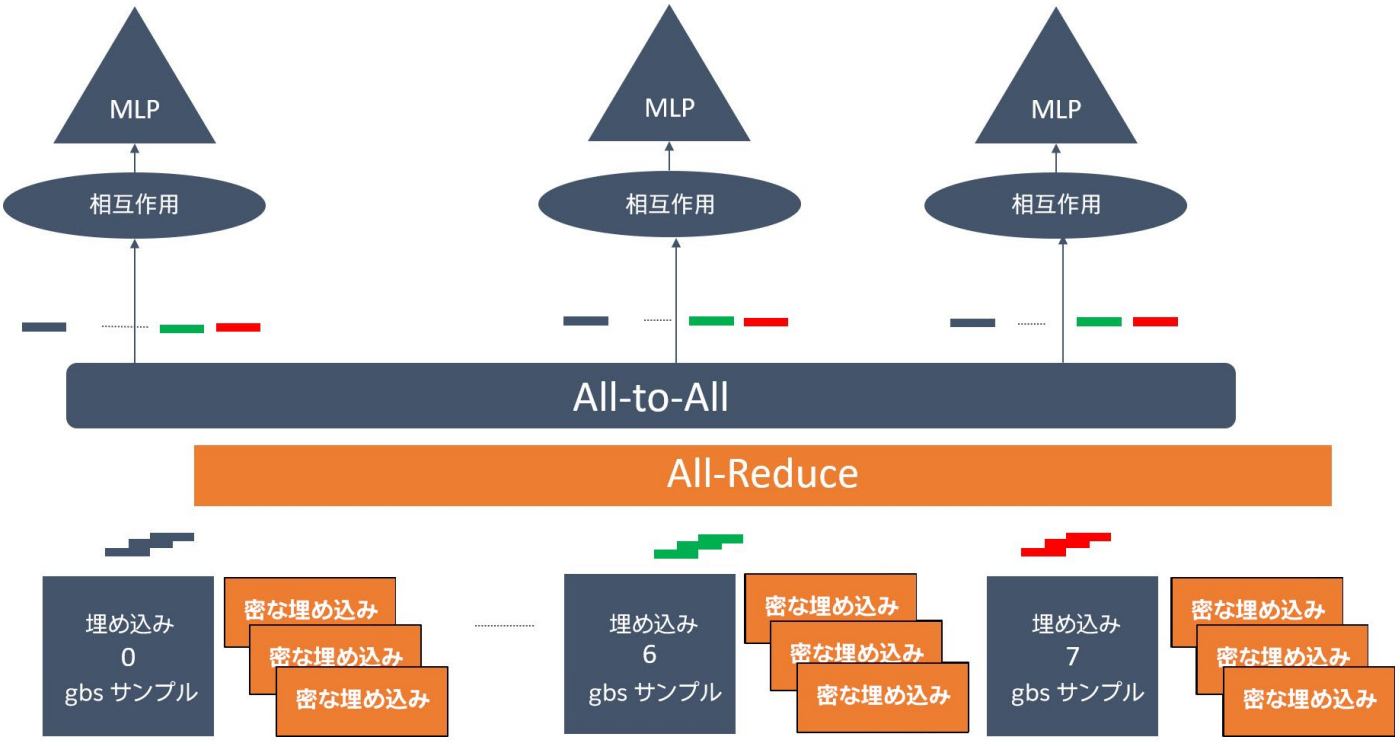


図 3. 小さな埋め込みテーブルの勾配を密な勾配に変換します。データ並列処理の勾配の同期には all-reduce を、モデル並列処理の埋め込み情報の交換には all-to-all を使用しています。

この手法には、インスタンス数は大きな埋め込みテーブル数を超えることができないという制限があります。MLPerf* DLRM では、16 個の大きな埋め込みテーブルがあるため、17 以上のインスタンスにはスケーリングできません。スケーリングを向上するには、垂直分割埋め込みベースのモデル並列化を使用します（**図 4**）。この手法では、大きな埋め込みテーブルを元のテーブルと同じエントリー番号の複数の埋め込みテーブルに垂直分割します。各テーブルは、元のテーブルの列のサブセットになります。そして、各モデル・インスタンスは分割されたテーブルの 1 つを保持し、all-to-all 通信を行います。`p`（ランク数）が `N`（モデルの埋め込みテーブルの数）で割り切れ、グループ番号が `g=p/N` であるとします。各埋め込みテーブルを `g` 個のテーブルに分割します。垂直分割後の埋め込みテーブルは `g*N=p` 個になります。各モデル・インスタンスに 1 つの埋め込みテーブルを配置して、グローバルバッチで各テーブルを検索し、all-to-all でインスタンス間の検索エントリーを転置します。その後、同じ元の埋め込みテーブルに属するエントリーを連結し、データ並列アプローチと同様に上層を通過させます。

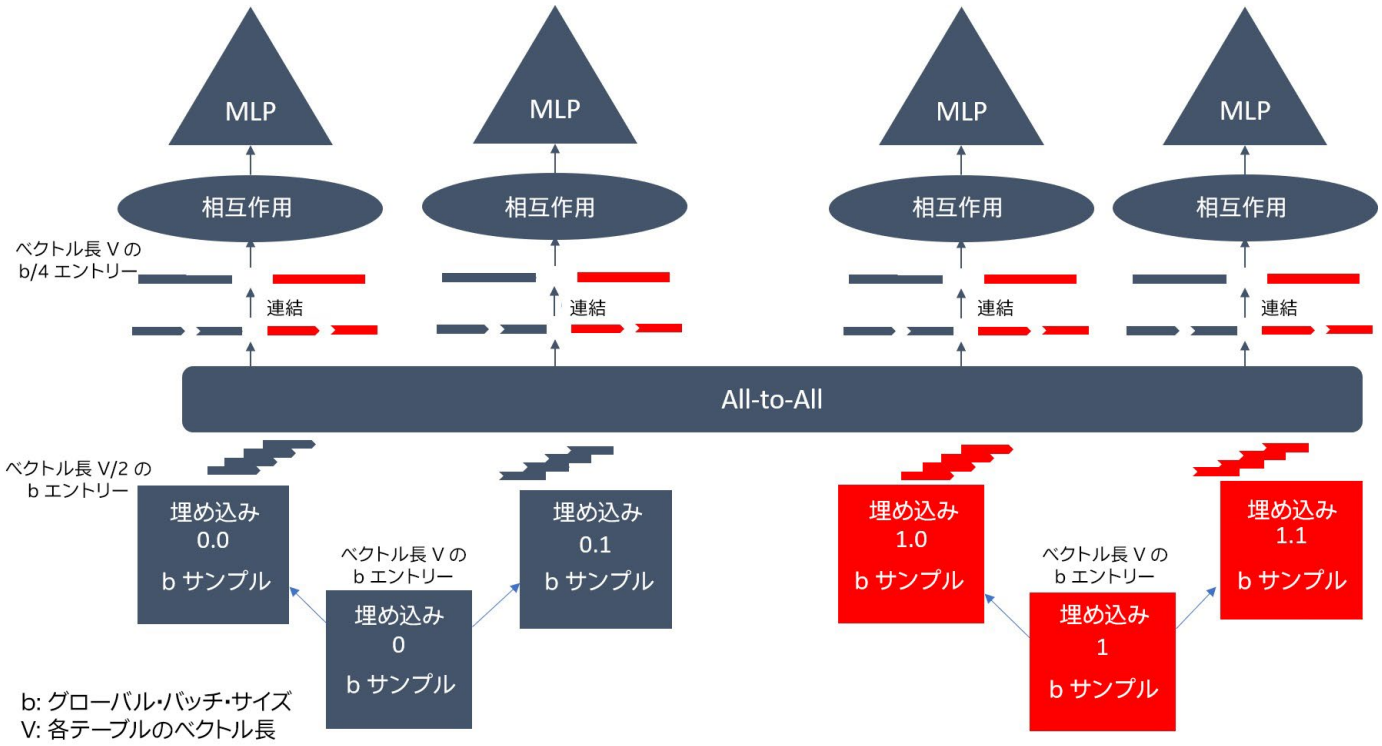


**図 4. 2 つの埋め込みテーブルを持つモデルを 4 つのインスタンスでトレーニングした場合。
各テーブルは縦に 2 つのテーブルに分割されます（V : 埋め込みテーブルの各行のベクトル長）。
ルックアップ・エントリーはすべてのインスタンス間で転置され、同じテーブル（同じ色）に属するエントリーは連結されます。
これにより、従来は不可能だった、4 つのランクで 2 つの埋め込みテーブルを持つモデルのトレーニングを可能にします。**

この垂直分割埋め込みベースのハイブリッド並列アプローチには、以下の利点があります。

1. 疎な all-reduce データ並列処理と比較して、小さなテーブルを密なものとして扱い、大きな疎なテーブルを垂直に分割することで、埋め込みテーブルを保持するモデルの通信オーバーヘッドを軽減します。これにより、TTT（トレーニング時間）が短縮され、複数のモデル・インスタンスを効率良くスケーリングできます。
2. DLRM トレーニングは、大きな埋め込みテーブルを使用することから、メモリー依存のワークロードであると言えます。垂直分割埋め込みでは、大きな埋め込みテーブルの列のサブセットのローカルコピーだけを扱うため、必要なメモリーを減らすことができます。例えば、26 個の埋め込みテーブルは、単一ノードでトレーニングした場合、100GB 以上のメモリーを必要とします。前述のソリューションにより 64 ランクにスケーリングすると、すべてのランクは 1 つの大きなテーブルのサブセットのみを扱い、特徴サイズは単一ノードでトレーニングする場合の 128 ではなく 32 となります。つまり、10 個の小さな埋め込みテーブルと、元の大きな埋め込みテーブルの列のサブセットを保持する大きな埋め込みテーブルに必要なメモリーは、約 6GB だけです。そのため、垂直分割埋め込みによる最適化は、大きな埋め込みテーブルを持つトレーニング・ワークロードに対する一般的な解決策でもあります。

Split-LAMB オプティマイザーを使用した大きなバッチサイズの DLRM BFloat16 トレーニング

さらに優れたスケーリング効率を得るため、LAMB と呼ばれる層別適応型大規模バッチ最適化手法を使用して、大きなバッチサイズのトレーニングを可能にします。また、Split-LAMB + SGD を使用して、インテル® DL ブーストの BFloat16 命令を活用します。これにより、64 ソケット以上へのスケーリングが可能となり、DLRM トレーニングの TTT を短縮できます。DLRM では、32K グローバル・バッチ・サイズが一般的です。より多くのランクにスケーリングすると、ローカル・バッチ・サイズが極端に小さくなり、ローカル・ワークロードがプロセッサーを飽和させることができなくなります。この場合、通信と計算をオーバーラップさせることができません。そのため、より多くのランクにスケーリングする際には、より大きなグローバル・バッチ・サイズを使用する必要があります。

SGD（確率的勾配降下法）は、DLRM のリファレンス・コードにおけるデフォルトのオプティマイザーです。これは、64K グローバル・バッチ・サイズでは 0.75 エポックで収束しますが、より大きなバッチサイズ（256K）では収束に失敗します。ここで紹介するソリューションでは、LAMB（Adam ベースのオプティマイザー）を使用して、256K グローバル・バッチ・サイズのトレーニングを可能にし、DLRM のトレーニングにおいて 0.8 エポックで収束します。LAMB は、すべてのウェイトについて 1 次および 2 次のモーメントを格納します。標準の SGD オプティマイザーと比較して、LAMB は 3 倍のメモリー・フットプリントを必要とします。DLRM は、大きな埋め込みテーブルを使用し、その勾配が疎であることから、メモリー依存です。メモリー・フットプリントを軽減するため、LAMB オプティマイザーは計算のデータ並列部分でのみ使用し、疎な埋め込みテーブルでは SGD オプティマイザーを使用します。

DLRM のトレーニングを高速化するため、インテル® DL ブーストの BFloat16 命令を使用します。マスターウェイトは、BFloat16 でトレーニング精度を維持するためによく使用されます。マスターウェイトとは、ウェイトを更新するためにオプティマイザーに保存されている Float32 ウェイトのコピーで、フォワードパスやバックワード・パスにはマスターウェイトから変換された BFloat16 ウェイトも必要になります。これは、Float32 トレーニングに比べて約 1.5 倍のメモリー・フットプリントを必要とし、DLRM のメモリー依存を高めます。ここでは、Split オプティマイザーを使用して、BFloat16 トレーニングのメモリー・フットプリントを軽減します（**図 5**）。**SCOPE I** のすべての入力パラメーターは、前方および後方のトレーニング段階で BFloat16（対応する Float32 パラメーターを切り捨てたもの）で処理され、その後 BFloat16 演算子（`InnerProduct`、`EmbeddingBag`）に投入されインテル® DL ブーストの BFloat16 を活用します。パラメーターの更新段階（SGD オプティマイザーのスコープ）に入ると、**SCOPE I** の BFloat16 データと **SCOPE II** の下半分のデータ（BFloat16 も存在する）をフル精度の Float32 パラメーターにパックし、Float32 で通常の計算を行います。更新後、Float32 データを **SCOPE I** と **SCOPE II** の別々の BFloat16 表現に分割して戻します。そのため、Split-SGD では、ウェイトごとに追加のメモリー・オーバーヘッドは発生しません。Split-LAMB では、同じ方法でウェイトのパックとアンパックを行い、モメンタムを Float32 で保持します。

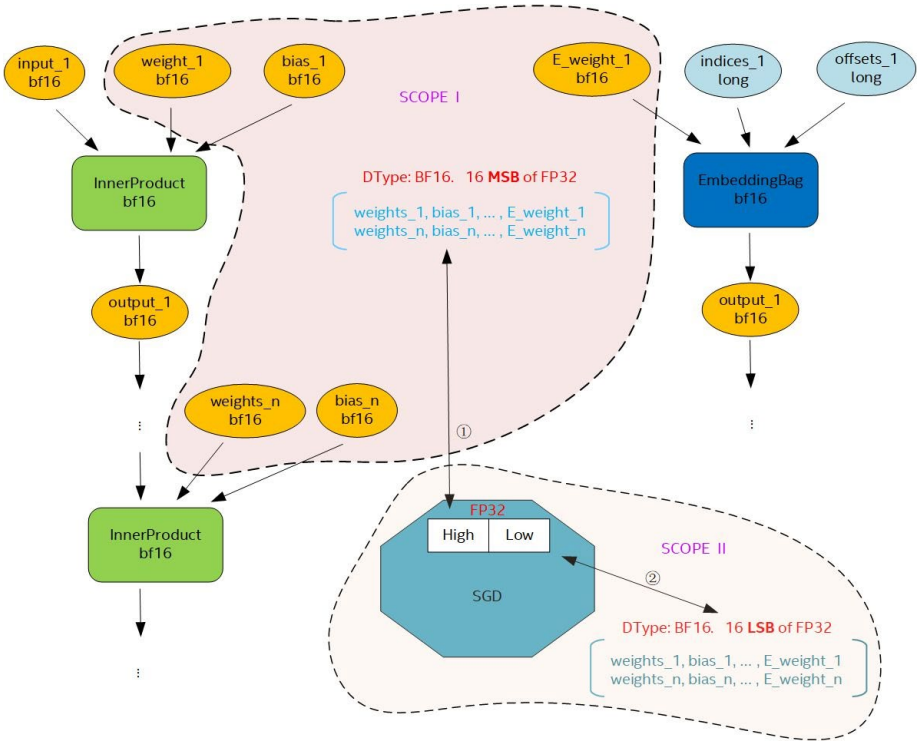


図 5. Split-SGD の模式図

SigOpt によるハイパーパラメーター最適化

革新的なデータとモデルの並列化、BFloat16 最適化、および新しいハイブリッドの Split-SGD + LAMB オプティマイザーの組み合わせにより、メトリックに対するパフォーマンスを最大化するにはハイパーパラメーターを最適化することが重要です。ハイパーパラメーター最適化は、グリッド検索やランダム検索などの従来の手法では多くの時間とリソースを必要とします。ここでは、最適なハイパーパラメーターを見つけるため、よりサンプル効率の高い検索方法を採用しています。

ここでは、あらゆるタイプのモデル（ディープラーニング、マシンラーニング、ハイパフォーマンス・コンピューティング、シミュレーションなど）向けに、実行の追跡とスケーラブルなハイパーパラメーター最適化を組み合わせた最先端の実験プラットフォームである SigOpt を使用します。2020 年 10 月にインテルに買収された（英語）SigOpt は、スケジューラーで任意のハイパーパラメーター最適化（HPO）手法（ランダム探索、グリッド探索、ベイズ最適化など）を利用できるだけでなく、さまざまなベイズ最適化アルゴリズムとグローバル最適化アルゴリズムの最良の属性を組み合わせた独自のオプティマイザーであり、ここでの目的には最適であると考えました。

DLRM のトレーニングは数回の反復で収束し、256K グローバル・バッチ・サイズでは AUC（曲線下面積）のしきい値である 0.8025 に達しました（**図 6**）。これは、32K グローバル・バッチ・サイズで達成された 0.75 AUC を上回ります。SigOpt は、しきい値を満たすハイパーパラメーター・セットを素早く見つけ、しきい値を超えて向上し続けていることが分かります。SigOpt のウェブ・ダッシュボードでは、さまざまな可視化、チャート、プロット、比較、テーブルをすぐに確認できます。SigOpt のパラメーター重要度分析には、実験の重要なパラメーターが表示されます（**図 7**）。

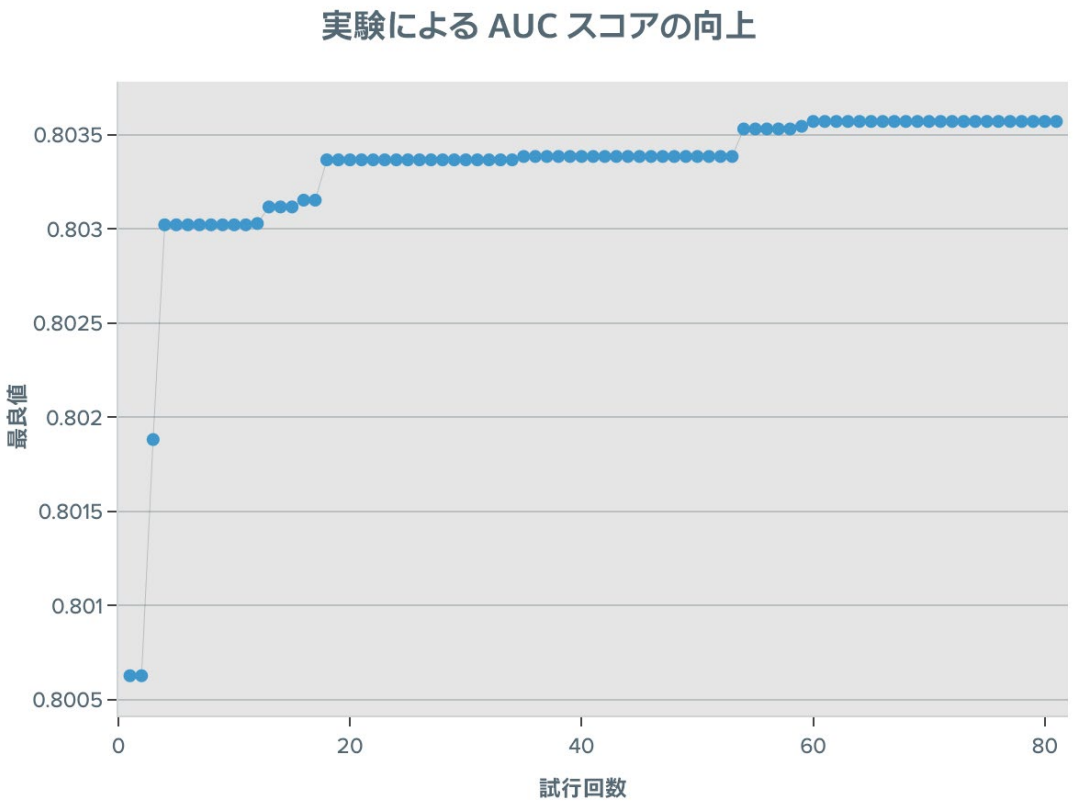


図 6. DLRM 実験による AUC スコアの向上

Parameter Importance

Name	Importance
learning-rate	
lr-num-warmup-steps	
lamblr	
lr-num-decay-steps	
lr-decay-start-step	

図 7. DLRM におけるパラメーターの重要性

新しいモデル並列データローダー

これまで、より多くのランクにスケーリングし、メモリー・フットプリントを軽減する垂直分割埋め込みテーブルについて説明してきました。LAMB の大きなグローバル・バッチ・サイズも、うまく利用することで、より優れたスケーリング効率が得られます。大きな埋め込みテーブルでは、グローバル・バッチ・サイズのエントリーをモデル並列でルックアップする必要がありますが、そのためには、グローバル・バッチ・サイズ入力をディスクから読み込む必要があります。これは、マルチソケット・システムでは I/O がボトルネックになる可能性があります。このオーバーヘッドを軽減するため、新しいモデル並列データローダーを使用します。このローダーは、グローバル・バッチ・サイズ入力の一部であるローカル・バッチ・サイズ入力のみを読み込み、all-to-all 通信を使用してグローバル・バッチ・サイズ入力を取得します。

MLPerf* DLRM モデルのトレーニングには、テラバイトのデータセットを使用します。データは行優先で、メモリー内で連続しています。各サンプルには 40 の要素（1 つのラベル、13 の数値特徴、26 のカテゴリー特徴）があり、各要素は 4 バイトを使用します。数値特徴は下位 MLP の入力となり（データ並列処理）、カテゴリー特徴は埋め込みの入力となります（小さなテーブルはデータ並列化、大きなテーブルはモデル並列化されます）。単一インスタンスの場合は、反復ごとにローカル・バッチ・サイズ（LBS）のサンプルを読み込む必要があります。データ並列のスケールアウト・ソリューションのみを使用すると、すべてのインスタンスは反復ごとに LBS サンプルを読み込みますが、ハイブリッド並列アプローチを使用すると、すべてのインスタンスは大きな埋め込みテーブルの一部のローカルコピーのみを扱います。そのため、現在のインスタンスの大きな埋め込みテーブルと、すべてのインスタンスの入力データに対して、グローバル・バッチ・サイズ（GBS）の埋め込みインデックスが必要となります。

LBS 入力では、サンプルごとに 26 個のカテゴリー特徴があり、現在のインスタンスは自分の大きなテーブルの LBS カテゴリー特徴だけでなく、ほかのモデル・インスタンスの LBS カテゴリー特徴も読み取ります。そこで、すべてのインスタンスが反復ごとに LBS サンプルのみを読み込み、all-to-all 通信を使用して大きな埋め込みテーブルの GBS カテゴリー特徴を取得するようにします（**図 8**）。ここで、すべてのインスタンスの入力は、第 1 インスタンスの大きな埋め込みテーブルに対する LBS カテゴリー特徴を意味し、すべてのランク 0 は GBS カテゴリー特徴に統合されます。ランク 1 は、第 2 インスタンスの大きな埋め込みテーブルの LBS カテゴリー特徴を意味し、以降のランクも同様になります。埋め込みテーブルをルックアップする前に、all-to-all 通信を使用してすべてのインスタンスの GBS カテゴリー特徴を収集します。`N` 個のインスタンスを使用する場合、標準のデータローダーと比べて `(N-1)/N` の I/O 帯域幅を軽減できます。

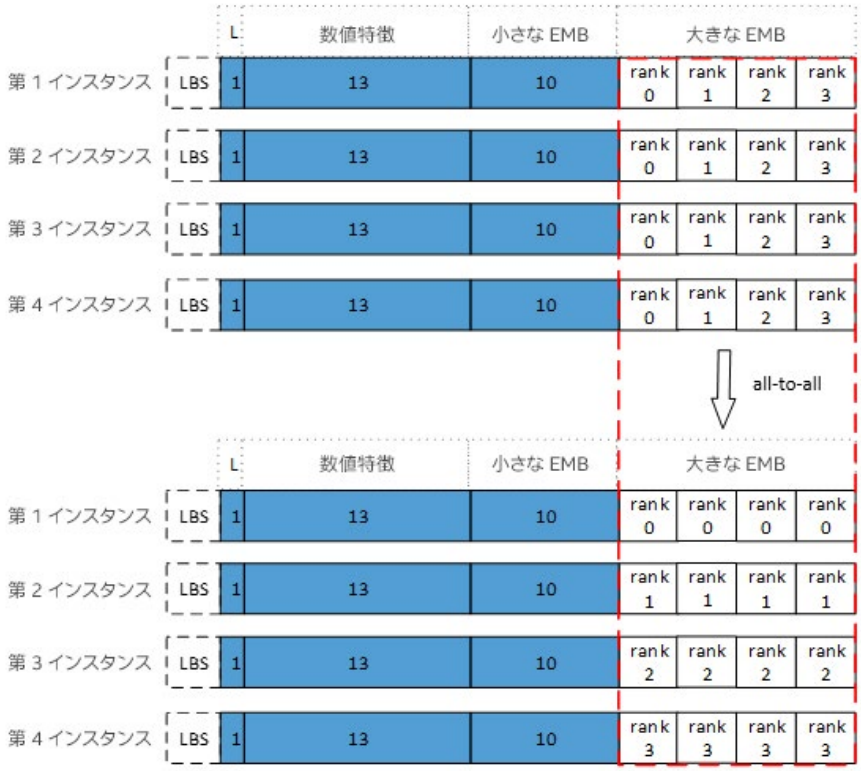


図 8. 新しいモデル並列データローダー。すべてのインスタンスは、まず LBS 入力を読み込み、次に all-to-all を使用して現在のインスタンスの大きな埋め込みテーブルの GBS カテゴリー特徴を取得します。

結果

DLRM のスケールアウト・ソリューションは、PyTorch*、インテル® Extension for PyTorch* (IPEX)、およびインテル® oneAPI コレクティブ・コミュニケーション・ライブラリー (インテル® oneCCL) を使用して実装されています。MLPerf* ベンチマークはクローズド部門とオープン部門の両方に提出されました (**図 9**)。クローズド部門の結果 (英語) (2021 年 6 月 30 日実施) では、インテル® Xeon® Platinum 8380H プロセッサーで 4 ソケットのみを使用した場合、32K GBS の収束におよそ 2 時間かかりました。一方、LAMB を使用して 256K GBS と垂直分割埋め込みテーブルを有効にし、インテル® Xeon® Platinum 8376H プロセッサーで 64 ソケットにスケーリングした場合は、わずか 15 分で収束しました (オープン部門の結果 (英語)、2021 年 6 月 30 日実施)。

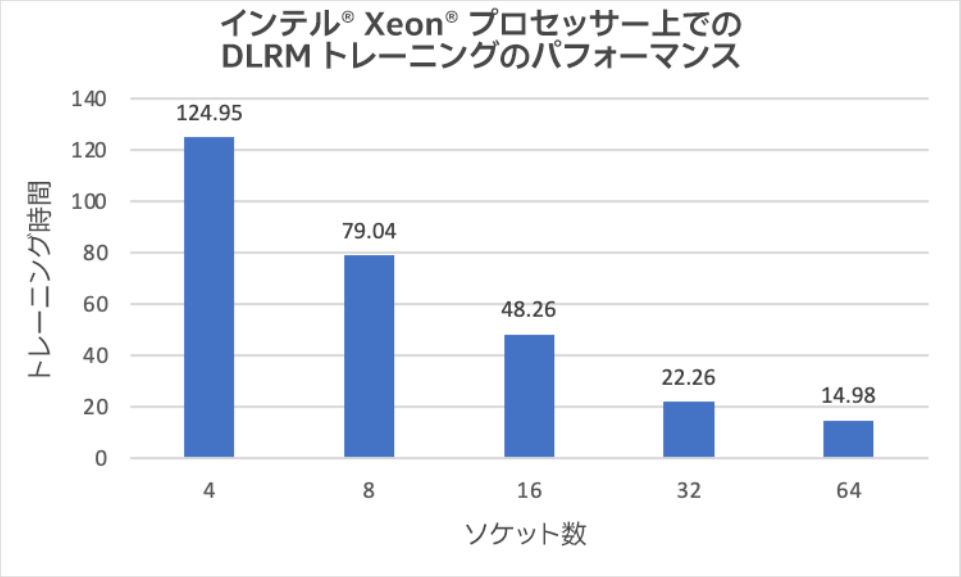


図 9. 異なるソケット数の DLRM トレーニング・パフォーマンス。
1 つのモデル・インスタンス（ランク）/ ソケットを使用しています。4、8、16 ソケットの結果は MLPerf* のクローズド部門に提出され、32 ソケットと 64 ソケットの結果はオープン部門に提出されました。

まとめ

ここでは、DLRM MLPerf* のトレーニング向けに、計算、メモリー、メモリー容量、I/O に依存するワークのバランスを取るという課題を解決するため、完全なスケールアウト・トレーニング・ソリューションを提供しました。最初に、ハイブリッド並列処理を用いて、通信コストとメモリー消費量を軽減しました。垂直分割埋め込みテーブルは、より多くのランクにスケーリングできるだけでなく、大きな埋め込みテーブルを持つほかのワークロードもトレーニングできる素晴らしいソリューションです。次に、LAMB オプティマイザーにより、大きなバッチサイズのトレーニングを可能にし、スケーリング効率を向上しました。Split オプティマイザーは、インテル® DL ブーストの BFloat16 命令を利用したトレーニングにも圧倒的な効果を発揮します。最後に、新しいモデル並列データローダーにより、I/O 帯域幅の要件を軽減しました。このソリューションでは、DLRM MLPerf* モデルのトレーニングを 15 分（より多くのソケットを使用する場合はそれ以下）で行うことができます。この記事で使用されているほとんどの手法は、ほかの分散トレーニング・アプローチにも一般化できます。

コストを考慮したハイパーパラメーター最適化の重要性について

実際に重要なコストメトリックを把握する

Eric Hans Lee SigOpt リサーチエンジニア
Michael McCourt SigOpt エンジニアリング責任者

この記事では、Conference on Uncertainty in Artificial Intelligence（英語）で発表した弊社のコスト制約のあるベイズ最適化への非短絡的アプローチ（英語）に関する研究を紹介します。この研究は、Facebook や Amazon のサイエンティストと共同で行われました。この要約では、この研究の動機となった要因について説明します。

通常のハイパーパラメーター最適化：反復による進捗状況の測定

ほとんどの実用的なハイパーパラメーター最適化パッケージは、一定の反復回数を実行して最適なハイパーパラメーターを決定しようとします。例えば、HyperOpt*、Optuna*、SKOpt、SigOpt を 100 回ずつ反復する例について考えてみます。

HyperOpt* :	Optuna* :

`hyperopt.fmin(`
`  objective,`
`  space = searc_space,`
`  max_evals = num_iterations)`
	`study = optuna.create_study()`
	`study.optimize(`
	`  objective,`
	`  n_trials = num_iterations)`

SKOpt :	**SigOpt :**

`skopt.gp_minimize(`
`  func=objective,`
`  n_calls = num_iterations)`
	`sigopt_conn = Connection()`
	`for _ in range(num_iterations):`
	`  suggestion =`
	`  conn.experiments().`
	`  suggestions().`
	`  create()`
	
	`  value = objective(`
	`  suggestion.assignments)`
	`  conn.experiments().`
	`  observations().`
	`  create(`
	`  suggestion = suggestion.id,`
	`  value = value)`

ほとんどのオプティマイザーのインターフェイスは基本的に同じで、最大化 / 最小化したいもの（最適化の目的）と実行する反復回数を入力します。多くの開発者はこのインターフェイスを当たり前のように使っていますが、これは本当に目的を最適化する最良の方法なのでしょうか？ 累積トレーニング時間などが重要である場合に反復回数を求めることは、パフォーマンスを重視していないと言えます。

課題：ハイパーパラメーターの評価コストのばらつき

最適化の進捗状況を反復回数で測定することは、各評価にかかる時間が同じであれば合理的ですが、ハイパーパラメーター最適化（HPO）では、ハイパーパラメーターにより設定のトレーニング時間が大きく異なる場合があります。前述の研究では、最も一般的に使用されている 5 つのマシンラーニング・モデルでこれを確認しています。

- K 近傍法（KNN）
- 多層パーセプトロン（MLP）
- サポート・ベクトル・マシン（SVM）
- 決定木（DT）
- ランダムフォレスト（RF）

これらのモデルは、データ・サイエンティストが使用する一般的なモデルの大部分を構成しています。ここでは、ディープラーニング・モデルは省略していますが、これらのモデルについても結果は同じです。

5 つのモデルについて、一般的なベンチマークである OpenML* w2a（英語）データセットで、標準の探索空間からランダムに選択した 5,000 個のハイパーパラメーター設定を使用してトレーニングしました。そして、各モデルのトレーニング時間の分布を図にしました（**図 1**）。各モデルのトレーニング時間は大きく異なり、1 桁以上の差があることも珍しくないことが分かります。これは、各モデルにおいて、いくつかのハイパーパラメーターが、モデルのパフォーマンスだけでなく、トレーニング時間にも大きく影響するためです（ニューラル・ネットワークの層のサイズや、フォレストのツリーの数など）。実際に、ほとんどすべての実用アプリケーションにおいて、評価コストは探索空間の異なる領域で大きく変化することが分かっています。

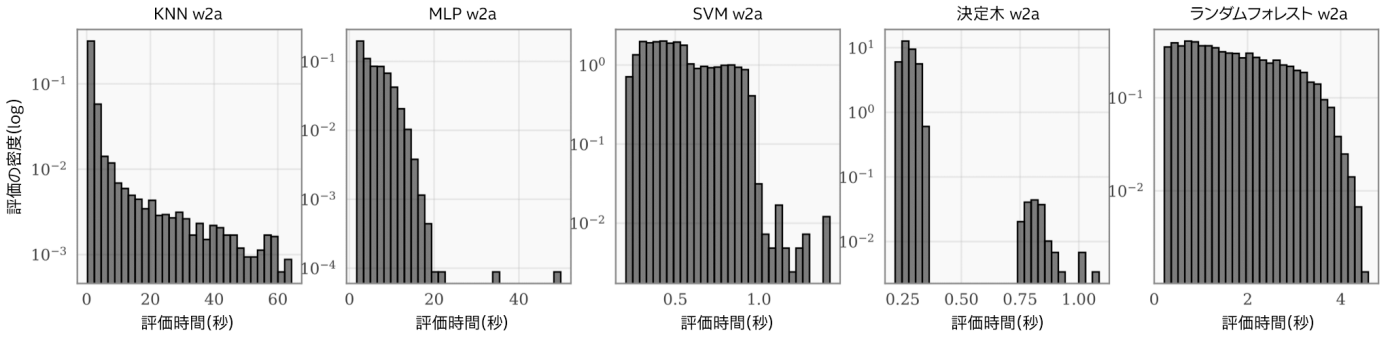


図 1. KNN、MLP、SVM、DT、RF の各ハイパーパラメーター最適化問題について、ランダムに選択した 5,000 点のランタイム分布（ログスケール）。各ヒストグラムの X 軸はランタイム、Y 軸は密度（全評価数に対する割合）を示します。

そのため、これらのハイパーパラメーターのチューニングにかかる累積トレーニング時間は、反復回数に正比例しません。実際、図 1 のヒストグラムから、あるオプティマイザーが 1 つのハイパーパラメーター設定を評価し、別のオプティマイザーが 100 個のハイパーパラメーター設定を評価した場合に、両者の時間が同じになる可能性は十分にあります。

課題：多様なコストへの対応

SigOpt では、目的関数の多様なコストを考慮し、次にどこを評価するかを判断するオプティマイザーを開発しました。これは、コストを考慮した最適化ルーチンとして知られています。この最適化ルーチンは、反復回数ではなく、時間などのコストメトリックを考慮して最適化を判断します。例えば、最適化ルーチンに 100 回の反復で最適なハイパーパラメーターを計算するように指示する代わりに、100 分のトレーニング時間で最適なハイパーパラメーターを計算するように指示することができます。この最適化ルーチンは、次のように呼び出します。

```
optimizer.minimize(func=objective, num_minutes=100)
```

ハイパーパラメーター最適化を制約するものは、費用や時間などユーザーにより異なるため、ハイパーパラメーター最適化コミュニティーではこの重要な研究課題に積極的に取り組んでいます。

コストを考慮するメリット

SigOpt では、「コストを考慮したベイズ最適化(CA-BO)」アルゴリズムを開発しました。簡単に説明するため、当社の CA-BO 手法と Preferred Networks 社の優れたオープンソース・ツールである Optuna* (英語) を比較する例について考えてみましょう。どちらのオプティマイザーも、100 回の最適化反復で XGBoost モデルの 2 項分類精度を最大化します。反復回数が増えるにつれて、Optuna* が CA-BO を徐々に上回ることが分かります (**図 2**)。パフォーマンスの差は明らかです。CA-BO が少なくとも Optuna* と同等のパフォーマンスを発揮することを期待したのですが、何が起こっているのでしょうか？

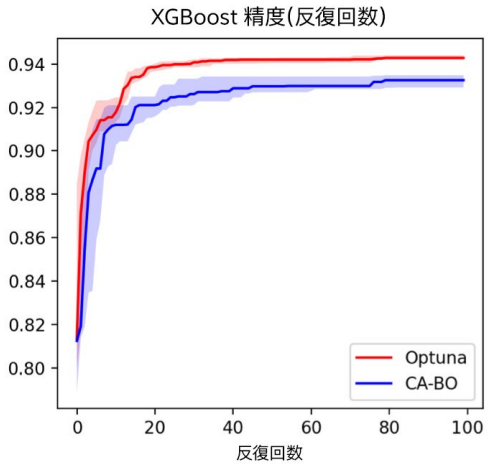


図 2. 最適化の反復回数を固定した場合の CA-BO と Optuna* の比較

この差は、CA-BO がコストを考慮しているためです。評価コストが変化することを認識し、それに応じた判断をしています。X 軸を反復回数ではなく、累積トレーニング時間に置き換えて、評価コストの変化を考慮した場合の最適化パフォーマンスを見てみましょう (**図 3**)。CA-BO のほうが、はるかに良いパフォーマンスであることが分かります。Optuna* は最終的に CA-BO を上回りますが、それは CA-BO が目的の時間内で実行し終わった後のことです (目的の時間内では、CA-BO が Optuna* を常に上回っています)。

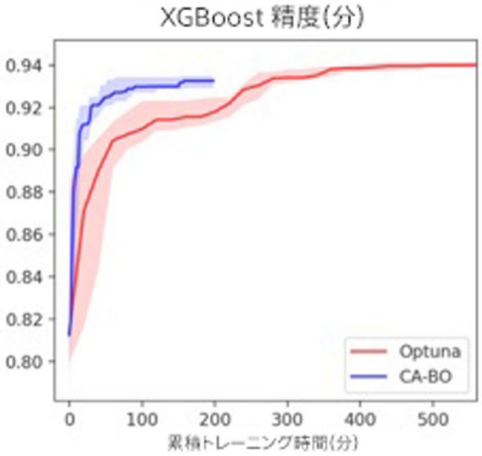


図 3. トレーニング時間に基づく CA-BO と Optuna* の比較

対照比較では、コストを考慮した最適化ルーチンとコストを考慮しない最適化ルーチンの差が顕著に表れています（**図 4**）。右のコストを考慮したアプローチの図では、CA-BO が 200 分以内に優れたハイパーパラメーターを見つけていることが分かります。左のコストを考慮しないアプローチの図では、トレーニングに 500 分以上かかる精度の低いハイパーパラメーターを 100 回の反復で見つけていることが分かります。

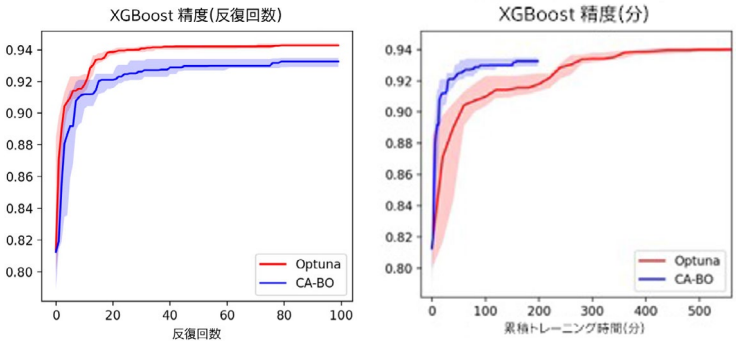


図 4. コストを考慮したオプティマイザーとコストを考慮しないオプティマイザーの対照比較

左の図は HPO の現状を示しています。右の図は、モデルをプロダクション環境に移行する際に重要なこと、つまり、考慮すべきメトリックに基づいてインテリジェントな決定を行う手法を示しています。

まとめ

CA-BO は、急速に進化している HPO アルゴリズムです。CA-BO の詳細は、SigOpt のアプローチの技術的な詳細を含む論文（英語）を参照してください。ほかの最適化手法やアプリケーションについては、SigOpt の Research ページ（英語）を参照してください。このような手法は、SigOpt Intelligent Experimentation Platform（英語）の柱の 1 つである SigOpt オプティマイザーに組み込まれています。今すぐサインアップ（英語）して、オプティマイザーとその他のプラットフォームを無料でお試しいただけます。

Katana のハイパフォーマンス・グラフ・アナリティクス・ライブラリー

Python* プログラマーに新しいグラフ・アナリティクスの選択肢を提供

Gurbinder Gill Katana Graph, Inc. シニア・ソフトウェア・エンジニア

Gartner 社（英語）によると、グラフ処理は 2021 年のデータ・アナリティクスのトレンドのトップ 10 に入っています。グラフ処理は、新たな応用分野であると同時に、リンクされたデータセット（社会、通信、金融ネットワーク、ウェブ・トラフィック、生化学的経路など）を扱うデータ・サイエンティストにとって必要なツールでもあります。実際に使われるグラフは大きくなる傾向にあります。例えば、ソーシャル・ネットワークでは、ノードとエッジの数が数十億にもなるため、ハイパフォーマンスな並列コンピューティングが不可欠です。

このため、Katana Graph はインテルと共同で、次の機能を備えたハイパフォーマンスで使いやすいグラフ・アナリティクス Python* ライブラリーを設計しました。

a. 重要なグラフ・アナリティクス・アルゴリズムを高度に最適化した並列実装
b. 基礎となる C++ グラフエンジン上にカスタム並列アルゴリズムを記述するための高レベルの Python* インターフェイス
c. pandas*、scikit-learn*、Apache Arrow*、インテルの AI ソフトウェア・スタック（英語）のツールやライブラリーとの相互運用性
d. さまざまなフォーマットからの抽出、変換、読み込み（ETL）の包括的なサポート
e. Metagraph*（英語）プラグイン

この記事では、ライブラリーの内容、ライブラリーの入手方法、使用例、パフォーマンス・ベンチマークについて説明します。

ライブラリーのグラフ・アナリティクス・アルゴリズム

グラフ処理のパイプラインで使用される主要なアルゴリズムは、Katana ライブラリーにあらかじめパッケージされています。現在、利用可能なアルゴリズムは以下の通りです。

- **幅優先検索：**ソースノードを起点とした幅優先探索から構築された指向性ツリーを返します。
- **単一始点最短経路：**ソースノードから始まるすべてのノードへの最短経路を計算します。
- **連結成分：**グラフの成分（ノードのグループ）のうち、内部ではつながっているが、ほかの成分とはつながっていないものを探します。
- **ページランク：**受け取ったリンクの構造に基づいて、グラフ内のノードのランクを計算します。
- **媒介中心性：**各ノードを通過する最短パスの数に基づいて、グラフ内のノードの中心性を計算します。
- **三角形カウント：**グラフ内の三角形の数をカウントします。
- **Louvain コミュニティー検出：**Louvain ヒューリスティックスを使用して、モジュール性を最大化するグラフのコミュニティーを計算します。
- **部分グラフ抽出：**グラフの誘導部分グラフを抽出します。
- **Jaccard 類似度：**与えられたノードとグラフ内のほかのすべてのノードとの Jaccard 係数を計算します。
- **ラベル伝搬法によるコミュニティー検出：**ラベル伝搬アルゴリズムを使用して、グラフ内のコミュニティーを計算します。
- **局所クラスタリング係数関数：**グラフ内のノードがどの程度集まっているかを測定します。
- **K トラス：**少なくとも 3 つの頂点を含み、すべての辺が少なくとも K - 2 個の三角形に入射するグラフの最大誘導部分グラフを見つけます。
- **K コア：**次数 K 以上のノードを含む最大の部分グラフを見つけます。

新たなアルゴリズムが次々に追加されていますが、ユーザーが独自のアルゴリズムを追加することも容易です。

Katana Graph ライブラリーの入手方法

Katana Graph のアナリティクス・ライブラリーはオープンソースであり、3 条項 BSD ライセンス（英語）の下で自由に利用できます。GitHub*（英語）または Anaconda.org からインストールできます。

```
$ conda install -c katanagraph/label/dev -c conda-forge katana-python
```

Katana Graph ライブラリーの使用法

Katana の Python* ライブラリーは、隣接行列、pandas* DataFrame、NumPy* 配列、エッジリスト、GraphML*、NetworkX* など、さまざまなフォーマットからの ETL をサポートしています。いくつかの例を以下に示します。

```
import numpy as np
import pandas
from katana.local import Graph
from katana.local.import_data import (
     from_adjacency_matrix,
     from_edge_list_arrays,
     from_edge_list_dataframe,
     from_edge_list_matrix,
     from_graphml)
```

隣接行列からの入力

```
katana_graph = from_adjacency_matrix(
                    np.array([[0, 1, 0], [0, 0, 2], [3, 0, 0]]))
```

エッジリストからの入力

```
katana_graph = from_edge_list_arrays(
                    np.array([0, 1, 10]), np.array([1, 2, 0]),
                    prop = np.array([1, 2, 3]))
```

pandas* DataFrame からの入力

```
katana_graph = from_edge_list_dataframe(
                    pandas.DataFrame(dict(source=[0, 1, 10],
                                          destination=[1, 2, 0],
                                     prop = [1, 2, 3])))
```

GraphML* からの入力

```
katana_graph = from_graphml(input_file)
```

グラフ・アナリティクス・アルゴリズムの実行

以下の例は、入力グラフの媒介中心性を計算します。

```
import katana.local
from katana.example_utils import get_input
from katana.property_graph import PropertyGraph
from katana.analytics import betweenness_centrality,
                             BetweennessCentralityPlan,
                             BetweennessCentralityStatistics
katana.local.initialize()

property_name = "betweenness_centrality"
betweenness_centrality(katana_graph, property_name, 16,
                       BetweennessCentralityPlan.outer())
stats = BetweennessCentralityStatistics(g, property_name)

print("Min Centrality:", stats.min_centrality)
print("Max Centrality:", stats.max_centrality)
print("Average Centrality:", stats.average_centrality)
```

Katana の Python* ライブラリーは、pandas*、scikit-learn*、Apache Arrow* と相互運用性があります。

これまでに挙げたパッケージ化されたルーチンに加えて、データ・サイエンティストは、Katana Graph の最適化された C++ エンジン[1] とその並列データ構造や並列ループ構造を利用する簡単な Python* インターフェイスを使って、独自のグラフ・アルゴリズムを記述することもできます。Katana Graph のライブラリーには、すでに幅優先検索の実装が含まれていますが、以下の例では API を使用してこのようなアルゴリズムをいかに簡単に実装できるかを示しています。

```
def bfs(graph: Graph, source):
    """
    ソースからすべてのノードへの BFS を計算

    レベルごとに一括同期するアルゴリズム

    :param graph: 入力グラフ
    :param source: トラバースするソースノード
    :return: ノード ID でインデックス付けされた距離の配列
    """
    next_level_number = 0

    # Katana のコンカレント・データ構造を使用した現在のレベルと
    # 次のレベルのワークリスト
    curr_level_worklist = InsertBag[np.uint32]()
    next_level_worklist = InsertBag[np.uint32]()

    # 距離配列を作成して初期化
    # ソースは 0、それ以外はすべて INFINITY
```

```
    distance = np.empty((len(graph),), dtype=np.uint32)
    distance[:] = INFINITY

    distance[source] = 0

    # ソースノードのみで処理を開始
    next_level_worklist.push(source)

    # ワークリストが空になるまで実行
    while not next_level_worklist.empty():
        # 現在と次のワークリストをスワップ
        curr_level_worklist, next_level_worklist = next_level_worklist,
                                                    curr_level_worklist

        # 次のレベルのためにワークリストをクリア
        next_level_worklist.clear()
        next_level_number += 1

        # 現在のワークリストの各要素に bfs_operator を適用して並列処理
        do_all(
            curr_level_worklist,
            # この呼び出しは bfs_operator の初期引数をバインド
            bfs_operator(graph, next_level_worklist,
                                next_level_number, distance)
        )

    return distance

# この関数は、Katana の演算子としてマークされるネイティブコードにコンパイルされ、
# Katana do_all で使用できるようになる
@do_all_operator()
def bfs_operator(graph: Graph, next_level_worklist,
                               next_level_number, distance, node_id):
    """
    ワークリストの各ノードで演算子が呼び出される

    最初の 4 つの引数は上記の bfs によって提供される
    node_id はワークリストから受け取り、do_all でこの関数に渡される

    :param next_level_worklist: 次のノードを追加するワークリスト
    :param next_level_number: 見つけたノードに割り当てるレベル
    :param distance: データが格納される距離配列
    :param node_id: 処理するノード
    :return:
    """
    # ノードのアウトエッジを反復
    for edge_id in graph.edges(node_id):
        # エッジのデスティネーションを取得
        dst = graph.get_edge_dest(edge_id)
```

```
# デスティネーションにまだ到達していない場合は、
# そのレベルを設定し、ワークリストに追加することで、
# そのアウトエッジを次のレベルで処理できるようにする
if distance[dst] == INFINITY:
    distance[dst] = next_level_number
    next_level_worklist.push(dst)
# 競合状態だが安全。
# 演算子への複数の呼び出しが同じデスティネーションを追加すると、
# 同じレベルが設定される。ノードは次のレベルで複数回処理されるため、
# より多くのワークが作成されるが、アトミック操作を回避するため、
# 低次数グラフではまだ利点が得られる。
```

Metagraph* サポート

Katana Graph の Python* アナリティクス・ライブラリーは、Metagraph* (英語) プラグインを介して利用できます。Metagraph* は、Python* でグラフ・アナリティクスを行う一貫したエントリーポイントを提供します。標準の API でグラフ・ワークフローを記述し、それを Metagraph* にプラグインした互換性のあるグラフ・ライブラリーにディスパッチできます。これにより、オープンソースのグラフ・コミュニティーは、Katana Graph のハイパフォーマンスなアプリケーションを直接利用できるようになります。Metagraph* プラグインは、Anaconda* パッケージで提供されており、以下のようにインストールして起動できます。

```
$ conda create -n metagraph-test -c conda-forge \
                                 -c katanagraph/label/dev \
                                 -c metagraph metagraph-katana
```

```
import metagraph as mg
bfs = mg.algos.traversal.bfs_iter(katana_graph, <start node>)
```

Katana Graph ライブラリーはどれくらい高速か？

Katana ライブラリーは、ほかのグラフ・アナリティクス・フレームワークと比較して幅広いベンチマークが行われており、GAP ベンチマーク・スイート (英語)[2] では常に同等以上のパフォーマンスを示しています。**表 1** は、多様なドメインのさまざまなグラフについて、GAP のリファレンス実装と比較した Katana Graph のパフォーマンスを示しています。

	アルゴリズム	リアルグラフ			合成グラフ	
		ウェブ	Twitter	Road	Kron	Urand
GAP と比較した Katana Graph のスピードアップ	BFS	1.6	0.7	1.0	1.3	0.6
	SSSP	1.1	1.0	0.8	1.3	0.6
	CC	1.4	1.1	1.5	1.1	0.6
	ページランク	1.0	1.3	1.5	0.9	0.8
	BC	1.3	0.6	0.7	0.8	1.2
	TC	1.6	1.4	1.0	1.3	1.3

表 1. GAP ベンチマーク・スイートを使用した Katana Graph パフォーマンスの測定結果。出展 : Azad ほか（2020）[2]。システム構成 : デュアルソケット インテル® Xeon® Platinum 8153 プロセッサー（2GHz、64 論理コア）、384GB DDR4 メモリー。性能やベンチマーク結果に関する詳細は、http://www.intel.com/benchmarks/（英語）を参照してください。

Katana Graph ライブラリーは、インテル® Optane™ DC パーシステント・メモリーのようなバイト・アドレス・メモリー・テクノロジー上で、Clueweb12[3] や WDC12[4]（それぞれ 420 億エッジと 1,280 億エッジで、一般に公開されているグラフの中では最大級）などの非常に大きなグラフに対しても良好なパフォーマンスを示すことが分かっています[5、6]（**図 1**）。

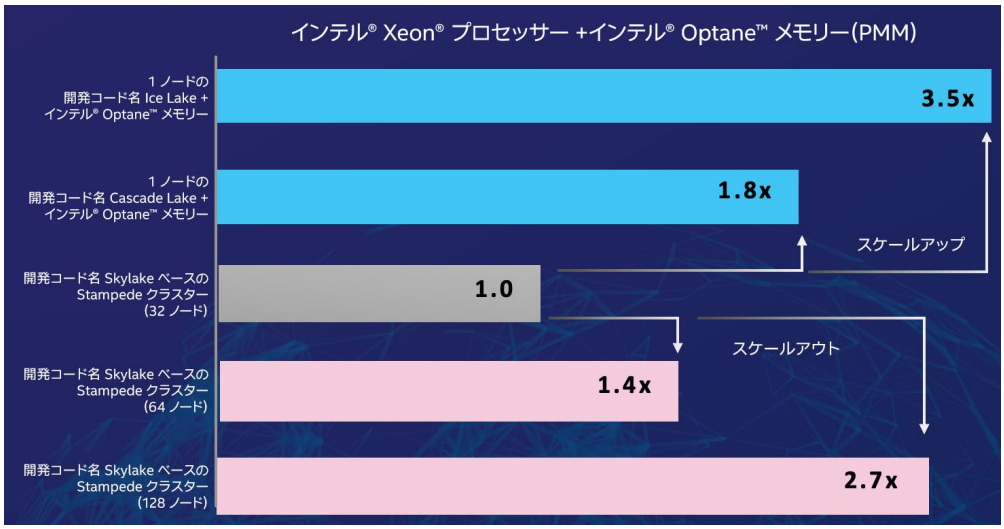


図 1. 大規模グラフにおける Katana Graph の BFS パフォーマンス。インテル® Optane™ メモリーベースのシングルノードとマルチノードのクラスターを比較しています。開発コード名 Skylake ベースの各 TACC Stampede クラスターノードには、2 基のインテル® Xeon® Platinum 8160 プロセッサー 2.10GHz と 192GB DDR4 メモリーが搭載されています。開発コード名 Cascade Lake ベースのサーバーには、第 2 世代インテル® Xeon® スケーラブル・プロセッサー 2.20GHz、6TB インテル® Optane™ パーシステント・メモリー、384GB DDR4 DRAM が搭載されています。開発コード名 Ice Lake ベースのサーバーには、2 基のインテル® Xeon® Platinum 8352Y プロセッサー 2.20GHz、8TB インテル® Optane™ パーシステント・メモリー、1TB DDR4 DRAM が搭載されています。このグラフは、参考文献 [5] と [6] のデータを使用してコンパイルされました。性能やベンチマーク結果に関する詳細は、http://www.intel.com/benchmarks/（英語）を参照してください。

詳細情報

Katana Graph ライブラリーが、グラフ・アナリティクス向けの多機能でハイパフォーマンスな選択肢であることをご理解いただけたでしょうか。GitHub* サイト（英語）では、ライブラリーの詳細を確認したり、質問や機能要求を投稿することができます。

参考文献

[1] Nguyen, D., Lenharth, A., and Pingali, K. (2013). A lightweight infrastructure for graph analytics（英語）. Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13).

[2] Azad, A. et al. (2020). Evaluation of graph analytics frameworks using the GAP Benchmark Suite（英語）, IEEE International Symposium on Workload Characterization (IISWC).

[3] The ClueWeb12 Dataset（英語）

[4] Web Data Commons – Hyperlink Graphs（英語）

[5] Gill, G., Dathathri, R., Hoang, L., Peri, R., and Pingali, K. (2020). Single machine graph analytics on massive datasets using Intel Optane DC Persistent Memory（英語）. Proceedings of the VLDB Endowment, 13(8), 1304–1318.

[6] Dathathri, R. et al. (2019). Gluon-Async: A bulk-asynchronous system for distributed and heterogeneous graph analytics（英語）. Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT).

インテル® oneAPI マス・カーネル・ライブラリーを使用した R コードの高速化

コードを変更せずに R のパフォーマンスを向上

Khang Nguyen 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

R 統計計算パッケージは、IT、金融、電子商取引、ヘルスケア、製造業など、さまざまな分野で利用されています。この記事では、インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) (英語) を使用して R のパフォーマンスを向上する方法を紹介しますが、その前に R とは何かを探ってみましょう。

R は、統計学とデータ・アナリティクスのためのオープンソースのプログラミング環境です。また、統計学に特化したドメイン固有のプログラミング言語でもあります (詳細は、www.r-project.org (英語) を参照)。多くの R ユーザーは、インテル® oneMKL のようなハイパフォーマンスな数学ライブラリーにリンクすることで、簡単に計算パフォーマンスを大幅に向上できることを知りません。

インテル® oneMKL には、科学、工学、金融などのアプリケーションで使用される一般的な数学演算に対して、高度に最適化、スレッド化、およびベクトル化された関数が含まれています (**図 1**)。密 / スパース線形代数 (BLAS、

LAPACK、PARDISO)、FFT、ベクトル演算、サマリー統計、スプラインなどを提供します。コードを分岐させることなく、プロセッサー向けに最適化されたコードを自動的に実行します。また、シングルコアのベクトル化やキャッシュの利用についても最適化されています。さらに、マルチコア CPU や GPU で並列処理を自動的に使用し、一部の計算をシングルシステムからクラスターへとスケールアップします。

図 1. インテル® oneMKL がサポートする数学ドメイン

インテル® oneMKL は、インテル® oneAPI ベース・ツールキットの一部です。R にリンクするには、インテル® oneAPI HPC ツールキットが必要です。どちらも無料でダウンロードできます。

- インテル® oneAPI ベース・ツールキット（英語）を入手する
- インテル® oneAPI HPC ツールキット（英語）を入手する

R とインテル® oneMKL をリンクすることで、後述するように、開発者は R のコードを変更することなく、大幅なパフォーマンスの向上を実現できます。インテル® oneMKL は、R アプリケーションの 1 つ下のレイヤーです。R エンジンと連携し、適切なインテル® oneMKL 関数を使用してパフォーマンスを向上します。インテル® oneMKL の機能は、インテル® アドバンスト・ベクトル・エクステンション 512（インテル® AVX-512)、インテル® アドバンスト・ベクトル・エクステンション 2（インテル® AVX2)、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）といった、インテル® プロセッサーのハードウェア機能を自動的に利用します。インテル® oneMKL は、開発者がハードウェアを気にすることなく、アプリケーションに集中できるように設計されています。例えば、インテル® AVX をサポートするシステム上で作成されたインテル® oneMKL アプリケーションは、後続の拡張機能をサポートするシステムに移行した場合、インテル® AVX-512 の利点も得られます。

R とインテル® oneMKL のリンクは簡単です（「インテル® MKL の BLAS と LAPACK を R に素早くリンクする」（英語）を参照）。以下の説明は、Linux* の場合です。R をインテル® oneMKL にリンクすると、適切な R 関数が最適化されたインテル® oneMKL 関数にリダイレクトされます。ただし、R と連携するには、特定のインテル® oneMKL 環境変数を設定することが重要です。

```
$ export MKL_INTERFACE_LAYER=GNU,LP64
$ export MKL_THREADING_LAYER=GNU
```

これらの環境変数は、インテル® MKL のインターフェイスとスレッド層を GNU* と LP64 に設定します。R がインテル® oneMKL にリンクされていることを確認するには、R のコマンドプロンプトから `sessionInfo()` を実行します（**図 2** と **3**）。

```
> sessionInfo()
R version 4.1.1 (2021-08-10)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.3 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
[1] compiler_4.1.1
```

図 2. R をインテル® oneMKL にリンクしない場合の sessionInfo() の出力

```
> sessionInfo()
R version 4.1.1 (2021-08-10)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.3 LTS

Matrix products: default
BLAS/LAPACK: /opt/intel/oneapi/mkl/2021.3.0/lib/intel64/libmkl_rt.so.1

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
[1] compiler_4.1.1
>
```

図 3. R をインテル® oneMKL にリンクした場合の sessionInfo() の出力

インテル® oneMKL にリンクするだけでも十分ですが、さらにできることがあります。

- R はシングルスレッドですが、インテル® oneMKL はシングルスレッドまたはマルチスレッド・モードで実行でき、デフォルトではマルチスレッド（環境変数 `MKL_DYNAMIC=TRUE`）を使用します。スレッドの生成を正当化するのに十分な作業がある場合は、通常これが最適です。言い換えると、インテル® oneMKL は大規模なデータセットに最適です。
- システムをオーバーサブスクライブ（利用可能なプロセッサーよりも多くのスレッドを使用すること）しないことが重要です。一般的には、スレッドの数をシステムのコア数と同じに設定します。
- ただし、利用可能なリソースを飽和させるだけの作業がない場合、この方法では常に最高のパフォーマンスが得られるわけではありません。このような場合は、手動でスレッド数を設定することで、パフォーマンスが向上します（例えば、`MKL_DYNAMIC=FALSE` 環境変数や `MKL_NUM_THREADS=4` 環境変数を設定します）。
- BIOS でハイパースレッディングを無効にします。
- `MKL_VERBOSE=1` 環境変数を設定して、呼び出されているインテル® oneMKL 関数と、使用されているスレッド数を確認します。

R ベンチマーク v2.5（英語）を使用して、R をインテル® oneMKL にリンクした場合としない場合のパフォーマンスを測定します。R をインテル® oneMKL にリンクしない場合、ベンチマークは 27.9 秒かかります。

```
Total time for all 15 tests_________________________ (sec):  27.9263333333334
Overall mean (sum of I, II and III trimmed means/3)_ (sec):  0.648128318174555
                        --- End of test ---
```

R をインテル® oneMKL にリンクした場合、ベンチマークは 2.8 秒かかります。

```
Total time for all 15 tests_________________________ (sec):  2.847
Overall mean (sum of I, II and III trimmed means/3)_ (sec):  0.16466777393087
                        --- End of test ---
```

つまり、R をインテル® oneMKL にリンクするだけで 9.8 倍のスピードアップが得られます。R コードの変更は必要ありません。

インテル® AVX-512 命令を使用した Maxloc 操作の最適化

一般的な Maxloc リダクション操作のベクトル化ガイド

Vamsi Sripathi インテル コーポレーション シニア HPC アプリケーション・エンジニア

対象要素（最小値、最大値、またはそれらの絶対値）のインデックスを特定する操作は、多くのアプリケーションで使用されています。この記事では、コンパイラーのインテル® アドバンスト・ベクトル・エクステンション 512（インテル® AVX-512）組込み関数を使用して、この操作（以降、「Maxloc」）を高速化し、インテル® Xeon® スケーラブル・プロセッサーでパフォーマンスを測定します。インテル® AVX-512 は、SIMD（Single Instruction, Multiple Data）実行を容易にする幅広い命令セットを提供します。すべてのインテル® Xeon® スケーラブル・プロセッサーでサポートされており、512 ビットのベクトルレジスターを使用してよりワイドな実行ユニットで動作させることで、最大の効率を実現します。インテル® AVX-512 を慎重に適用することで、対象要素のインデックスを特定する命令や比較の数を最小限に抑え、大幅な高速化を実現します。

Maxloc は、配列上で実行される一般的な検索操作です（例えば、NumPy* の argmax 関数（英語）、TensorFlow* の GlobalMaxPool1D 関数（英語）、インテル® oneMKL の amax 関数（英語）などがあります）。

ここでは、インテル® C/C++ コンパイラー・クラシック(英語)のインテル® AVX-512 SIMD API(ベクトル組込み命令)を使用します(以前の記事「明示的なベクトル化を使用したスキャン操作の最適化」も参照してください)。以下では、インテル® AVX-512 ベクトル組込み命令を使用したインテル® AVX-512 Maxloc の実装について説明します。

ベースライン実装

コードリスト 1 は、シングルパスのスカラーバージョンの Maxloc ベースライン実装です。このコードから分かるように、n 個の要素を含むベクトルは n 回の比較操作を必要とします。入力ベクトルには一度だけアクセスし、最大値に対応するインデックス位置を格納します。**表 2** は、ベースライン実装で生成される命令です。詳しくは、パフォーマンス評価のセクションで説明します。

```
1 #include "float.h"
2
3 int ref_max (int *p_n, float *p_x)
4 {
5   int n = *p_n;
6   float max_val = FLT_MIN;
7   int idx;
8
9   for (int i=0; i<n; i++) {
10    if (p_x[i] > max_val) {
11      max_val = p_x[i];
12      idx = i;
13    }
14  }
15  return idx;
16 }
```

コードリスト 1. Maxloc のベースライン実装

インテル® AVX-512 SIMD 実装

表 1 は、今回使用したインテル® AVX-512 SIMD 命令の一覧です。**図 1** は、`vmaxpd` 命令、`vcmpps` 命令、`vblendmps` 命令がサンプルのレジスターステートに対して行う操作を視覚的に表したものです。

操作名	命令	インテル® AVX-512 ベクトル組込み関数 API	説明
SIMD ロード	vmovups	__m512d _mm512_loadu_ps (void *mem_address);	512 ビット (16 個のパックド単精度(32 ビット) 浮動小数点要素)をメモリーから dst にロードします。
SIMD 最大	vmaxps	__m512d _mm512_max_ps (__m512d a, __m512d b);	a と b のパックド単精度(32 ビット)浮動小数点要素を比較して、パックド最大値を dst に格納します。
SIMD 比較	vcmpps	__mmask16 _mm512_cmp_ps_mask (__m512 a, __m512 b, const int imm8);	imm で指定された比較演算子に基づいて、a と b のパックド単精度(32 ビット) 浮動小数点要素を比較して、結果をマスクベクトル k に格納します。
SIMD ブレンド	vblendmps	__m512 _mm512_mask_blend_ps (__mmask16 k, __m512 a, __m512 b);	制御マスク k を使用して、a と b のパックド単精度(32 ビット) 浮動小数点要素をブレンドして、結果を dst に格納します。
SIMD ブロードキャスト	vpbroadcastd	__m512i _mm512_mask_set1_epi32 (__m512i src, __mmask16 k, int a);	書き込みマスク k を使用して、32 ビット整数 a を dst のすべての要素にブロードキャストします(対応するマスクビットが設定されていない場合、要素は src からコピーされます)。
SIMD 抽出	vextractf32x8	__m256 _mm512_extractf32x8_ps (__m512 a, int imm8);	imm8 を使用して、a から選択した 256 ビット(8 個の単精度 (32 ビット) 浮動小数点要素)を抽出し、結果を dst に格納します。

表 1. Maxloc で使用されたインテル® AVX-512 SIMD 命令

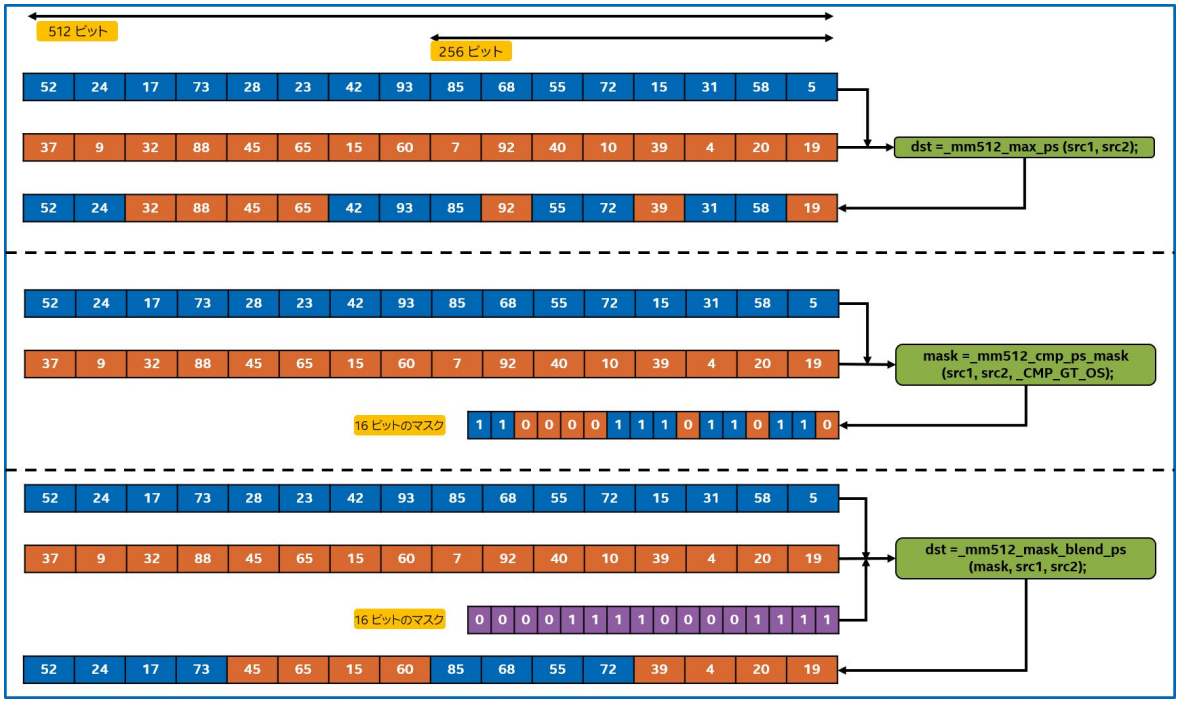


図 1. インテル® AVX-512 の vmaxps、vcmpps、および vblendmps で実行される操作を視覚的に表した図

Maxloc では、ループをアンロールして一連の `vmaxpd` 命令を適用し、512 ビットのインテル® AVX-512 レジスターに格納される 16 個の FP32（32 ビット浮動小数点）最大値のシーケンスを取得します。さらに、入力データを処理する際に、最大値のインデックスを保持します。これは、追加の比較操作（現在の反復の最大値と前の反復の最大値の比較）を行い、その結果を使って最新の最大値とそれに対応するアンロールブロック ID をブレンドすることで実現しています。アルゴリズムの主要なステップは以下の通りです。

1. **メインブロック：**
   a. 各反復では、64 個の入力要素を 4 つのインテル® AVX-512 ベクトルレジスターにロードし、ペア単位の比較を行って 4 組ずつの最大値を特定し、オフセット 16 で最大値を表す現在の `max` レジスターを形成します。
   b. 前に入力された最大値を比較し、16 ビットのマスクを形成します（各ビットはインテル® AVX-512 レジスターの FP32 要素を表します）。現在の反復で見つかった 4 組ずつの最大値が、前回の 4 組ずつの最大値よりも大きい場合、マスクビットが設定されます。
   c. このマスクは、現在の最大値と前回の最大値の両方からの値をブレンドして、今後の反復処理で使用する新しい 4 組ずつの最大値をレジスター内に形成するために使用されます。
   d. インデックスを追跡するため、同じマスクを再利用して、現在のループ・インデックス（展開されたブロック ID）を対応する最大値が配置されているブロックを表すインデックス・レジスターに設定します。
   e. 前述のステップはメインループを形成し、4 組ずつ（インデックス {0、16、32、48}、{1、17、33、49}、...）と対応するループ / ブロック ID（0、64、128、192、...）の中から 16 個の最大値を見つけるため、すべての入力要素に対して繰り返されます。

2. **リダクションとターゲットブロック：**メインループの後、インテル® AVX-512 レジスターに対する 1 回のリダクションで、メインブロックの 16 個の値の中から最大値を見つけます。リダクション・ステップで得た最大値は、インテル® AVX-512 レジスターの 16 の場所のどこに最大値があるかを 16 ビットのマスクで特定するために使用されます。16 ビットの結果マスクの各ビットは、インテル® AVX-512 ブロックインデックス・レジスターの int32 要素の選択を表します。この int32 値は、最大値が発生したブロック ID を表します。そして、最大値が 1 つの場所で発生したか、複数の場所で発生したかによって、以下の 2 つの方法で最大値を含む最初のブロックを特定します。
3. **単一インスタンス：**この場合、最大値はデータセットの中で一度だけ見つかっており、16 ビットの結果マスクのうち 1 つのビットがセットされています。これにより、インテル® AVX-512 ブロックインデックス・レジスターから対応する int32 値が特定されます。例えば、マスクが 0000 0000 1000 0000 の場合、インテル® AVX-512 インデックス・レジスターの 7 番目の int32 要素の値は、最大値が見つかったブロック ID を表しています。

   ブロックへのオフセット:この実装の特徴の 1 つは、ブロック全体（`N_UNROLL` の大きさ）を読み取らなくても、`vmaxps` 命令の固有のパターンを使用して対象インデックスを特定できることです。ここでは、$\mathtt{Blk}_{id}$ を 16 ビットの結果マスクに設定されている 1 ビットに対応するインテル® AVX-512 インデックス・レジスターからの int32 値、$\mathtt{M}_x$ を 16 ビットの結果マスクに 1 ビットが設定されているビット位置（0 ~ 15）とします。**図 2** から、`vmaxps` は 16 要素のオフセットにある値を比較していることが分かります。したがって、最大値のインデックスは `N_UNROLL` が 64 の場合、($\mathtt{Blk}_{id} + \mathtt{M}_x$)、($\mathtt{Blk}_{id} + \mathtt{M}_x + 16$)、($\mathtt{Blk}_{id} + \mathtt{M}_x + 32$)、($\mathtt{Blk}_{id} + \mathtt{M}_x + 48$) のいずれかで発生していると考えられます。
4. **重複：**結果マスクに含まれる 1 ビットの数が 1 より大きい場合、データセットの複数の場所で最大値が発生しています。この場合、マスクに設定された 1 ビットに対応するすべての int32 値のうち、最小のものをインテル® AVX-512 インデックス・レジスターから見つけます。最小値のインデックスを再びインテル® AVX-512 インデックス・レジスターと比較し、最大値がアンロールされたブロック内で複数回発生しているかどうかを確認します。複数回発生している場合は、ブロック全体を 16 要素のチャンクで読み込み、最大値が検出された最初のインデックスを特定します。そうでない場合は、ステップ 3 で説明した方法を使用します。

図 2 と **3** は、インテル® AVX-512 レジスターにおけるメインブロックの計算とリダクション・シーケンスを視覚的に表現したものです。**コードリスト 2** は、このアルゴリズムの実装を示しています。

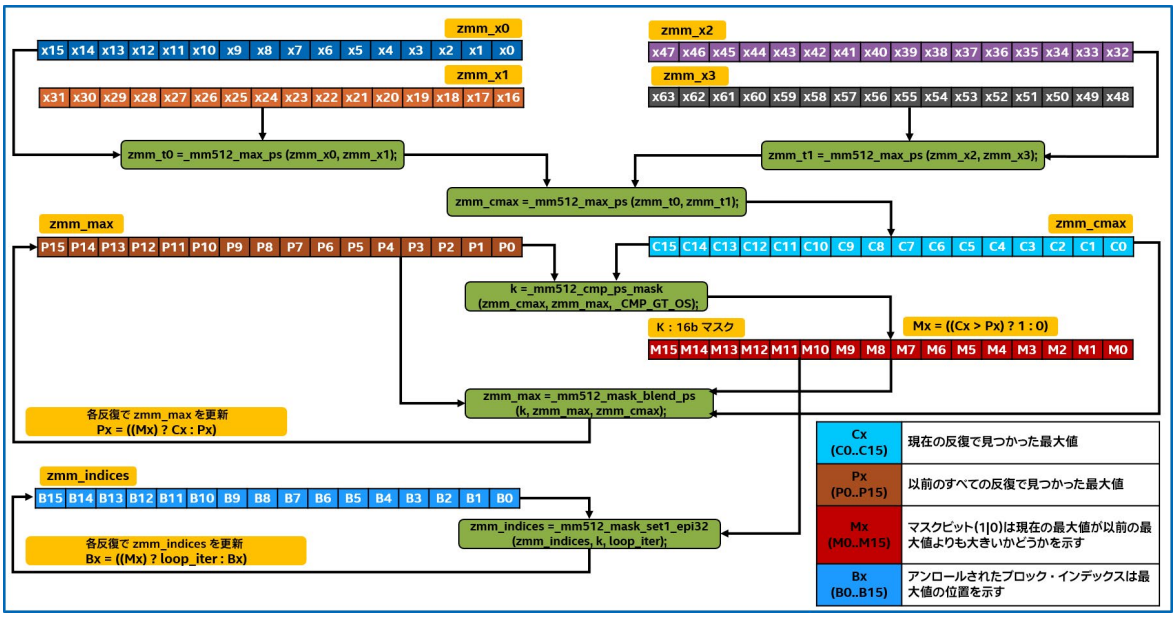


図 2. Maxloc のインテル® AVX-512 + インデックス追跡実装

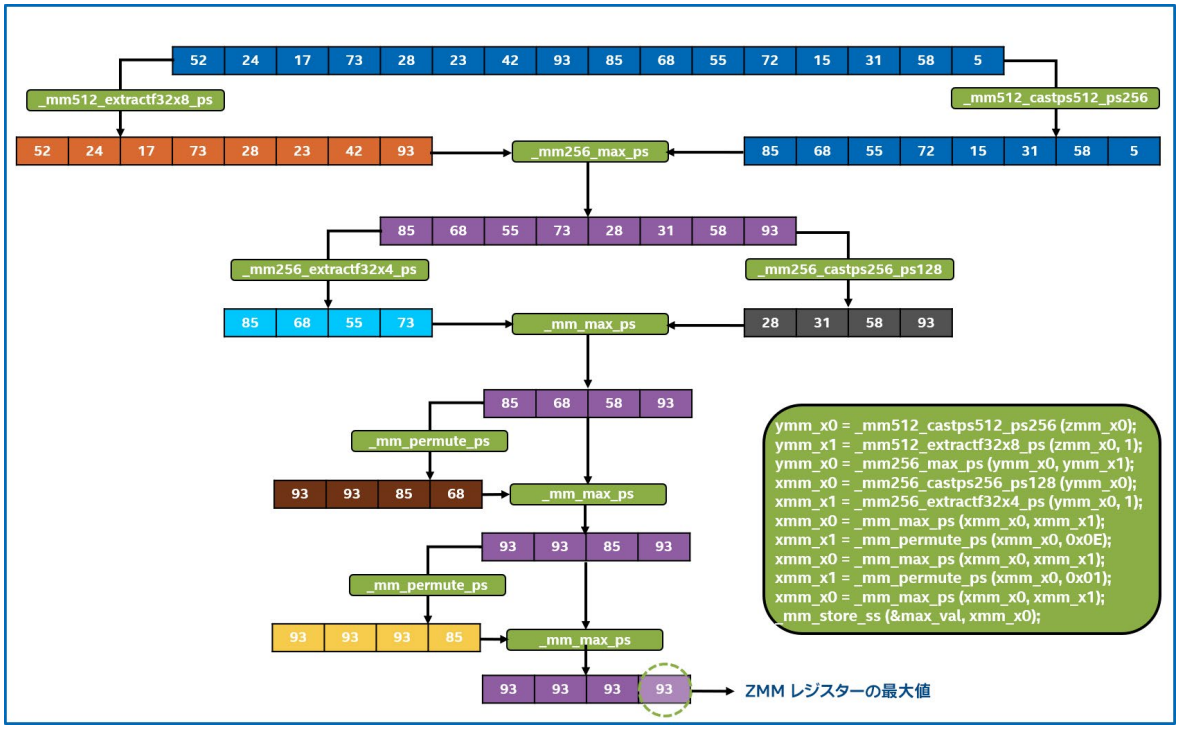


図 3. インテル® AVX-512/ZMM レジスターをレデュースして最大値を特定する命令シーケンス

```
1  int max_idx_tracking (int *p_n, float *p_x)
2  {
3    __m512 zmm_x0, zmm_x1, zmm_x2, zmm_x3, zmm_x4, zmm_x5, zmm_x6, zmm_x7;
4    __m512 zmm_curr_max, zmm_tmp0, zmm_tmp1, zmm_tmp2, zmm_tmp3, zmm_tmp4, zmm_tmp5;
5    __m256 ymm_x0, ymm_x1;
6    __m128 xmm_x0, xmm_x1;
7    __mmask16 m0;
8    __m512 zmm_max      = _mm512_set1_ps(FLT_MIN);
9    __m512i zmm_indices = _mm512_setzero_epi32();
10
11   int n = *p_n;
12   int offset, block_idx;
13   float max_val;
14
15   for (int j=0; j<n; j+=N_UNROLL) {
16     zmm_x0   = _mm512_loadu_ps(p_x);
17     zmm_x1   = _mm512_loadu_ps(p_x+16);
18     zmm_x2   = _mm512_loadu_ps(p_x+32);
19     zmm_x3   = _mm512_loadu_ps(p_x+48);
20     zmm_tmp0 = _mm512_max_ps(zmm_x0, zmm_x1);
21     zmm_tmp1 = _mm512_max_ps(zmm_x2, zmm_x3);
22 #if N_UNROLL > 64
23     zmm_x4   = _mm512_loadu_ps(p_x+64);
24     zmm_x5   = _mm512_loadu_ps(p_x+80);
25     zmm_x6   = _mm512_loadu_ps(p_x+96);
26     zmm_x7   = _mm512_loadu_ps(p_x+112);
27     zmm_tmp2 = _mm512_max_ps(zmm_x4, zmm_x5);
28     zmm_tmp3 = _mm512_max_ps(zmm_x6, zmm_x7);
29     zmm_tmp4 = _mm512_max_ps(zmm_tmp0, zmm_tmp1);
30     zmm_tmp5 = _mm512_max_ps(zmm_tmp2, zmm_tmp3);
31
32     zmm_curr_max = _mm512_max_ps(zmm_tmp4, zmm_tmp5);
33 #else
34     zmm_curr_max = _mm512_max_ps(zmm_tmp0, zmm_tmp1);
35 #endif
36
37     m0 = _mm512_cmp_ps_mask(zmm_curr_max, zmm_max, _CMP_GT_OS);
38     if (m0) {
39       zmm_max      = _mm512_mask_blend_ps(m0, zmm_max, zmm_curr_max);
40       zmm_indices  = _mm512_mask_set1_epi32(zmm_indices, m0, j);
41     }
42     p_x += N_UNROLL;
43   }
44
45   p_x = p_x-n;
46
47   // reduction tree on zmm register to find max. value
48   ymm_x0 = _mm512_castps512_ps256(zmm_max);
49   ymm_x1 = _mm512_extractf32x8_ps(zmm_max, 1);
50   ymm_x0 = _mm256_max_ps(ymm_x0, ymm_x1);
51
52   xmm_x0 = _mm256_castps256_ps128(ymm_x0);
53   xmm_x1 = _mm256_extractf32x4_ps(ymm_x0, 1);
54   xmm_x0 = _mm_max_ps(xmm_x0, xmm_x1);
55
56   xmm_x1 = _mm_permute_ps(xmm_x0, 0x0E);
57   xmm_x0 = _mm_max_ps(xmm_x0, xmm_x1);
58
59   xmm_x1 = _mm_permute_ps(xmm_x0, 0x01);
60   xmm_x0 = _mm_max_ps(xmm_x0, xmm_x1);
61   _mm_store_ss(&max_val, xmm_x0);
62
63
64   m0 = _mm512_cmp_ps_mask(_mm512_broadcastss_ps(xmm_x0), zmm_max, _CMP_EQ_OQ);
65   int pop_cnt = _mm_popcnt_u32(_cvtmask16_u32(m0));
66   if (pop_cnt == 1) {
67     _mm512_mask_compressstoreu_epi32(&block_idx, m0, zmm_indices);
68     offset = _bit_scan_forward(m0);
69   } else {
70     __m256i ymm_i0, ymm_i1;
71     __m128i xmm_i0, xmm_i1;
72
73     zmm_indices = _mm512_mask_blend_epi32(m0, _mm512_set1_epi32(INT_MAX), zmm_indices);
74     // reduction tree on zmm register to find min. index value
75     ymm_i0 = _mm512_castsi512_si256(zmm_indices);
76     ymm_i1 = _mm512_extracti32x8_epi32(zmm_indices, 1);
77     ymm_i0 = _mm256_min_epi32(ymm_i0, ymm_i1);
78
79     xmm_i0 = _mm256_castsi256_si128(ymm_i0);
80     xmm_i1 = _mm256_extracti32x4_epi32(ymm_i0, 1);
81     xmm_i0 = _mm_min_epi32(xmm_i0, xmm_i1);
82
83     xmm_i1 = _mm_shuffle_epi32(xmm_i0, 0x0E);
84     xmm_i0 = _mm_min_epi32(xmm_i0, xmm_i1);
85
86     xmm_i1 = _mm_shuffle_epi32(xmm_i0, 0x01);
87     xmm_i0 = _mm_min_epi32(xmm_i0, xmm_i1);
88     _mm_store_ss((float *)&block_idx, _mm_castsi128_ps(xmm_i0));
89
90     __mmask16 m1 = _mm512_cmp_epi32_mask(
91                      _mm512_broadcastd_epi32(xmm_i0), zmm_indices, _MM_CMPINT_EQ);
92     if (_mm_popcnt_u32(_cvtmask16_u32(m1)) > 1) {
93       p_x += block_idx;
94       zmm_max = _mm512_broadcastss_ps(xmm_x0);
95
96       for (int j=0; j<N_UNROLL; j+=NUM_ELEMS_IN_REG) {
97         zmm_x0 = _mm512_loadu_ps(p_x);
98         m0     = _mm512_cmp_ps_mask(zmm_x0, zmm_max, _CMP_EQ_OQ);
99         if (m0) {
100          return (block_idx + j + _bit_scan_forward(m0));
101        }
102        p_x += NUM_ELEMS_IN_REG;
103      }
104    }
105    offset = _bit_scan_forward(m1);
106  }
107
108  for (int i=0; i<(N_UNROLL/NUM_ELEMS_IN_REG); i++) {
109    if (max_val == p_x[block_idx+offset+(i*NUM_ELEMS_IN_REG)]) {
110      return block_idx + offset + (i*NUM_ELEMS_IN_REG);
111    }
112  }
113 }
```

コードリスト 2. Maxloc のインテル® AVX-512 実装

パフォーマンスの評価

以下の実験プロトコルを使用して、インテル® AVX-512 Maxloc 実装と**コードリスト 1** に示したベースライン実装を比較します。

- ベースライン実装は、GCC（8.4.1）と Clang（11.0.0）で `-O3 -march=icelake-server -mprefer-vector-width=512` オプションを使用してコンパイルされています。インテル® AVX-512 実装には、ICC（v19.1.3.304）を使用しています。
- 入力データは、すべてのベンチマークで同じで、昇順の値で構成されています（つまり、最大値は最後のインデックスの位置にあります）。ランダムな値を使用しても、パフォーマンスに違いは見られませんでした。
- データが異なるキャッシュ階層（L1D、L2、L3）に収まる場合のパフォーマンスを評価できるように、1,024 要素から 4,194,304 要素（FP32 要素で 16 MB）の範囲の入力サイズを使用しました。
- ベンチマークには、ソケットあたり 38 コア、48KB L1D キャッシュ、1280KB L2、ソケットあたり 57MB L3 が搭載されたインテル® Xeon® Platinum 8368 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）を使用しました。
- 1 つのコアに固定された 1 つのスレッドのパフォーマンスを測定して、各問題サイズで 100 回の反復を実行した場合の平均経過時間を示しています。メモリーは 64B でアライメントされた境界に割り当てられています。
- インテル® AVX-512 実装では、メインブロックを 128 個の要素で明示的にアンロールしています。

表 2 は、GCC、Clang、および ICC でコンパイルされた明示的なインテル® AVX-512 実装によって生成された命令シーケンスを示しています（スペースの都合上、メインブロックのみ）。**図 4** は GCC、Clang、明示的なインテル® AVX-512 実装の Maxloc のパフォーマンス、**図 5** は GCC に対するインテル® AVX-512 実装のスピードアップです。パフォーマンスのグラフから、次のことが分かります。

1. ベースライン実装では、コンパイラー（GCC、Clang）の自動ベクトル化能力によってパフォーマンスが決まります。**表 2** のオブジェクト・コードの逆アセンブリーから、GCC も Clang も計算をベクトル化できず、1 つの FP32 要素を処理するスカラー命令（`vmovss`、`vucomiss`、`vmaxss`、`cmov`）を使用しています。そのため、命令名には「single, single-precision（単一、単精度）」を表す「`ss`」というサフィックスが付いています。興味深いことに、Clang はループを 8 つの要素でアンロールしているにもかかわらず、GCC と同じ単一のスカラー命令を発行して、8 つの要素をシリアルに処理しています。
2. 一方、インテル® AVX-512 実装では、512 ビット幅の ZMM レジスターに格納された 16 個の FP32 要素を、インテル® AVX-512 命令で処理しています（**表 2** の 3 列目）。インテル® AVX-512 命令は、**コードリスト 2** の組込み関数にマッピングされます。
3. Maxloc のインテル® AVX-512 実装は、優れたパフォーマンスを発揮します（**図 5**）。L1、L2、L3 キャッシュに収まるサイズの平均的なスピードアップは、それぞれ 18 倍、40 倍、13 倍です。L3 キャッシュでは、データをレジスターにロードする際のレイテンシーが増加するため、インテル® AVX-512 のベクトル化によるパフォーマンス向上は減少します。GCC と Clang では、SIMD を使用していないことが、キャッシュの種類にかかわらず、すべての問題サイズでのパフォーマンス低下に反映されています。Clang は 8 要素でアンロールしているにもかかわらず、その性能は GCC と同じです。

ベースライン(GCC)	ベースライン(Clang)	インテル® AVX-512(ICC)
20: mov %rcx,%rdx 23: vmovss (%rsi,%rdx,4),%xmm1 28: lea 0x1(%rdx),%rcx 2c: vcomiss %xmm0,%xmm1 30: vmaxss %xmm0,%xmm1,%xmm0 34: cmova %edx,%eax 37: cmp %rdx,%rdi 3a: jne 20 <ref_max+0x20>	40: vmovss (%rsi,%rdx,4),%xmm1 45: vmovss 0x4(%rsi,%rdx,4),%xmm2 4b: vucomiss %xmm0,%xmm1 4f: cmova %edx,%eax 52: vmaxss %xmm0,%xmm1,%xmm0 56: lea 0x1(%rdx),%edi 59: vucomiss %xmm0,%xmm2 5d: cmovbe %eax,%edi 60: vmaxss %xmm0,%xmm2,%xmm0 64: vmovss 0x8(%rsi,%rdx,4),%xmm1 6a: lea 0x2(%rdx),%eax 6d: vucomiss %xmm0,%xmm1 71: cmovbe %edi,%eax 74: vmaxss %xmm0,%xmm1,%xmm0 78: vmovss 0xc(%rsi,%rdx,4),%xmm1 7e: lea 0x3(%rdx),%edi 81: vucomiss %xmm0,%xmm1 85: cmovbe %eax,%edi 88: vmovss 0x10(%rsi,%rdx,4),%xmm2 8e: vmaxss %xmm0,%xmm1,%xmm0 92: lea 0x4(%rdx),%eax 95: vucomiss %xmm0,%xmm2 99: cmovbe %edi,%eax 9c: vmaxss %xmm0,%xmm2,%xmm0 a0: vmovss 0x14(%rsi,%rdx,4),%xmm1 a6: lea 0x5(%rdx),%edi a9: vucomiss %xmm0,%xmm1 ad: cmovbe %eax,%edi b0: vmaxss %xmm0,%xmm1,%xmm0 b4: vmovss 0x18(%rsi,%rdx,4),%xmm1 ba: lea 0x6(%rdx),%ecx bd: vucomiss %xmm0,%xmm1 c1: cmovbe %edi,%ecx c4: vmaxss %xmm0,%xmm1,%xmm0 c8: vmovss 0x1c(%rsi,%rdx,4),%xmm1 ce: lea 0x7(%rdx),%eax d1: vucomiss %xmm0,%xmm1 d5: cmovbe %ecx,%eax d8: vmaxss %xmm0,%xmm1,%xmm0 dc: add $0x8,%rdx e0: cmp %rdx,%r8 e3: jne 40 <ref_max+0x40>	20: vmovups (%rsi),%zmm0 26: vmovups 0x80(%rsi),%zmm3 2d: vmovups 0x100(%rsi),%zmm4 34: vmovups 0x180(%rsi),%zmm5 3b: vmaxps 0x40(%rsi),%zmm0,%zmm6 42: vmaxps 0xc0(%rsi),%zmm3,%zmm7 49: vmaxps x140(%rsi),%zmm4,%zmm8 50: vmaxps x1c0(%rsi),%zmm5,%zmm9 57: vmaxps %zmm7,%zmm6,%zmm10 5d: vmaxps %zmm9,%zmm8,%zmm11 63: vmaxps zmm11,%zmm10,%zmm12 69: add $0x200,%rsi 70: vcmpgtps %zmm2,%zmm12,%k1 77: vpbroadcastd %eax,%zmm1{%k1} 7d: add $0x80,%eax 82: vblendmps %zmm12,%zmm2,%zmm2{%k1} 88: cmp %edx,%eax 8a: jl 20 <max_idx_tracking+0x20>

表 2. GCC、Clang、および ICC で生成された Maxloc コードの逆アセンブリー

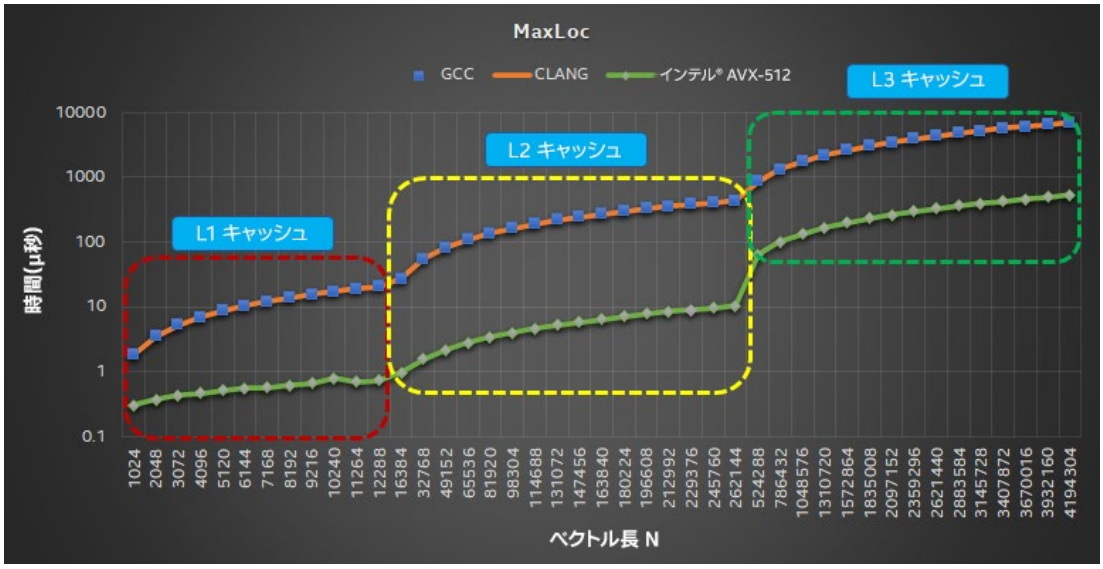


図 4. GCC、Clang、および明示的にインテル® AVX-512 のベクトル化を使用して ICC で生成された Maxloc のパフォーマンス

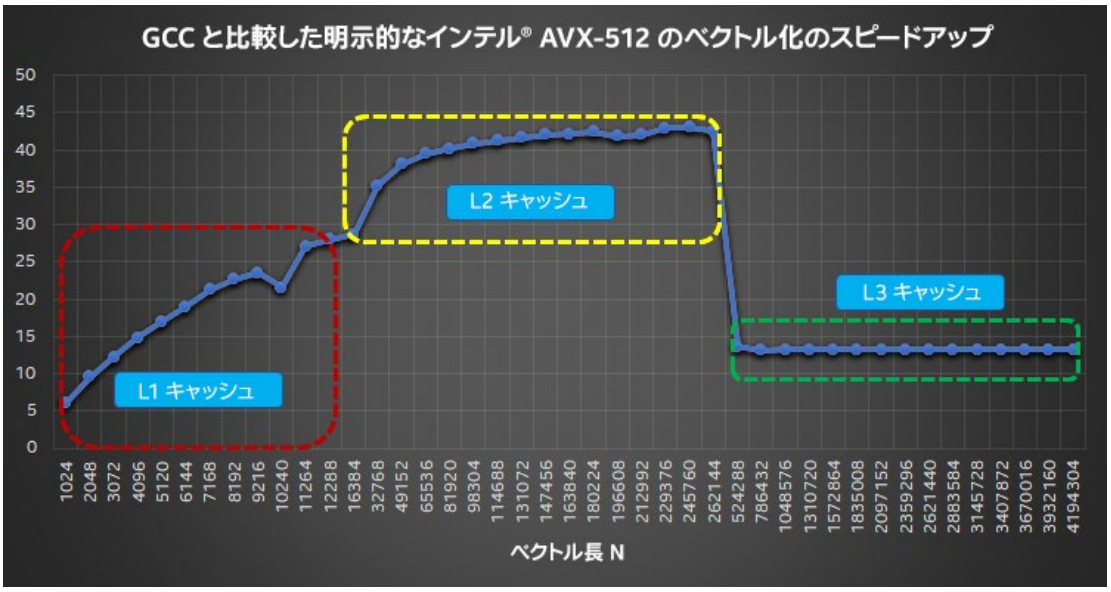


図 5. GCC と比較した明示的なインテル® AVX-512 のベクトル化によるスピードアップ

まとめ

これらの結果は、インテル® AVX-512 命令を Maxloc 演算に適用した場合のパフォーマンスの優位性を示しています。インテル® AVX-512 の明示的なベクトル化は、GCC や Clang よりも優れたパフォーマンスを実現します。この記事が、計算量の多いコードでインテル® AVX-512 の可能性を探る動機となることを願っています。

intel software

THE PARALLEL UNIVERSE

JPN/2112/PDF/XL/SPI/ND