intel software

THE PARALLEL UNIVERSE

インテルの AI ハードウェアとソフトウェアの最適化を活用して Llama を高速化

インテル® Arc™ GPU 上の TensorFlow* による転移学習

PyTorch* を使用して山火事を予測する

Issue
54
2023

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。

Unified Acceleration Foundation が設立されました！

すべての人々がオープンなアクセラレーテッド・コンピューティングの未来を形作ることができるように支援する、新たな手段が提供されたニュースをお知らせできることを嬉しく思います。9 月 19 日に、Linux* Foundation ファミリーの一部である Joint Development Foundation は、Unified Acceleration Foundation（英語）の設立を発表しました。これにより、多くの企業が協力して、真のマルチベンダー、マルチアーキテクチャー、アクセラレーテッド・コンピューティングのソフトウェア・サポートを提供できるようになります。UXL Foundation は oneAPI 仕様を引き継ぎ、世界中の複数の組織が協力して、オープンなアクセラレーテッド・コンピューティングの未来を形作ることができるように支援します。今後の号で、アクセラレーテッド・コンピューティングの可能性を広げる Foundation のメンバーからの記事を掲載できることを期待しています。

この夏にインテルがプレミアメンバーとして PyTorch Foundation に加入（英語）したことを記念して、本号では PyTorch* の記事を 2 つ掲載しました。1 つ目の、本号の注目記事「**インテルの AI ハードウェアとソフトウェアの最適化を活用して Llama を高速化**」では、大規模言語モデルの一般化を説明します。2 つ目の「**PyTorch* を使用して山火事を予測する**」では、転移学習を使用して正確な画像分類器を効率良く開発する方法を紹介します。

関連記事として、転移学習と勾配ブースティングに関する「**インテル® Arc™ GPU 上の TensorFlow* による転移学習**」と「**CPU 上での XGBoost、LightGBM、CatBoost 推論の高速化**」も掲載しました。

次に、エンタープライズ AI に関する 2 つの記事が続きます。Red Hat とインテルのチームの共著記事、「**AI でビジネスを拡大する**」では、インテル® アーキテクチャーで Red Hat* OpenShift* Data Science とオープンソースの AI テクノロジーを活用する方法を紹介します。この記事は、52 号の「エンタープライズ規模でのサプライチェーン最適化」の続編です。「**Microsoft* Azure* 上にセキュアな Kubeflow* パイプラインを構築**」では、クラウドでインテル® ソフトウェア・ガード・エクステンションズを活用してセキュアで高速化されたマシンラーニング・パイプラインをデプロイする方法を紹介します。

最後に、Fortran と OpenMP* を使用したヘテロジニアス並列処理に関する記事のパート 2 として、前号の「Fortran の DO CONCURRENT を使用したアクセラレーター・オフロード」の単純なエッジ検出の例を、より現実的な Sobel エッジ検出アルゴリズムを使用して拡張した「**OpenMP* ターゲットオフロードの事例**」で締めくくります。

AI とデータサイエンス、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、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
2023 年 10 月

ポッドキャスト
生成 AI は我々の成長、仕事、生活をどのように変えるのか
聴く（英語）

intel. software

将来に通用するコード

プロプライエタリーの壁を超えて成長

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、コードの可能性を広げましょう。

オープンスタンダードである oneAPI に根ざしたインテル® ソフトウェア開発ツールは、
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® ソフトウェア開発ツールの詳細 →

インテルの AI ハードウェアとソフトウェアの最適化を活用して Llama を高速化

大規模な言語モデルへのアクセスを一般化

Fan Zhao インテル コーポレーション AI ソフトウェア・エンジニアリング・マネージャー
Antony Vance Jeyaraj インテル コーポレーション クラウド・ソフトウェア・アーキテクト
Sree Ganesan インテル コーポレーション AI ソフトウェア・エンジニアリング・マネージャー
Susan Lansing インテル コーポレーション AI マーケティング・マネージャー
Kristina Kermanshahche インテル コーポレーション ソフトウェア・プロダクト・マネージャー
Radhika Rao インテル コーポレーション AI マーケティング・マネージャー
Patricia Mwove インテル コーポレーション クラウド・ソフトウェア・アーキテクト
Andres Rodriguez インテル コーポレーション フェロー

大規模言語モデル（LLM）へのアクセスをさらに一般化するため、Meta 社は Llama 2 をリリースしました。モデルをより広く利用できるようにすることで、AI コミュニティー全体で全世界に利益をもたらす取り組みが促進されるでしょう。LLM がテキストの生成、コンテンツの要約と翻訳、質問への応答、会話、および数学の問題を解くことや推論などのより複雑なタスクの実行において実証してきた優れた能力を考えると、LLM は、社会に利益をもたらす最も有望な AI テクノロジーの 1 つであると言えます。LLM には、新たな創造性と洞察力を引き出し、AI コミュニティーを刺激してテクノロジーを進歩させる可能性を秘めています。

Llama 2 は、開発者、研究者、組織が生成 AI を活用したツールとエクスペリエンスを構築するのを支援するように設計されています。Meta 社は、事前トレーニングおよび微調整済みの、70 億（7B）、130 億（13B）、および 700 億（70B）パラメーターの Llama 2 のモデルをリリースしました。Llama 2 で、Meta 社は微調整済みモデル全体に、中核となる 3 つの安全技術（教師あり安全微調整、ターゲットを絞った安全コンテキストの抽出、人間のフィードバックからの安全強化学習）を実装しました。これにより、Meta 社は安全実績を向上することができました。アクセスを一般化することにより、透過的かつオープンな方法で脆弱性を継続的に特定して軽減できるようになります。

インテルは、コミュニティーが Llama 2 のようなモデルを開発して実行できるように、競争力の高い魅力的なオプションを備えた AI ソリューションのポートフォリオを提供しています。インテルの豊富なハードウェア・ポートフォリオと、最適化されたオープン・ソフトウェアを組み合わせることにより、限定された計算リソースにアクセスするという課題を解決する代替手段が提供されます。この記事では、Habana® Gaudi®2 ディープラーニング・アクセラレーター、第 4 世代インテル® Xeon® スケーラブル・プロセッサー、インテル® Xeon® CPU マックス・シリーズ、およびインテル® データセンター GPU マックス・シリーズを含むインテルの AI ポートフォリオ上での、Llama 2 の 7B および 13B パラメーター・モデルの初期推論パフォーマンスを紹介します。ここで紹介する結果は、現在リリースされているソフトウェアのデフォルト設定のパフォーマンスであり、今後のリリースではさらなるパフォーマンスの向上が期待されます。現在、70B パラメーター・モデルにも取り組んでおり、近々コミュニティーに更新情報を提供する予定です。

Habana® Gaudi®2 ディープラーニング・アクセラレーター

Habana® Gaudi®2 は、ハイパフォーマンス、高効率のトレーニングと推論を提供するように設計されており、Llama や Llama 2 などの大規模言語モデルに特に適しています。LLM のメモリー要求を満たす（つまり、推論パフォーマンスを高速化する）ため、各 Habana® Gaudi®2 アクセラレーターは、96GB のオンチップ HBM2E を搭載しています。Habana® Gaudi®2 は、PyTorch* と DeepSpeed* を統合した、Habana SynapseAI* ソフトウェア・スイートにより、トレーニングと推論の両方をサポートしています。さらに、レイテンシーの影響を受けやすい推論アプリケーションに適した、HPU グラフ（英語）と DeepSpeed* 推論（英語）のサポートが最近 SynapseAI* に追加されました。2023 年第 3 四半期には、FP8 データ型のサポートを含む、さらなるソフトウェアの最適化が Habana® Gaudi®2 に提供される予定です。このアップデートにより、パフォーマンスの大幅な向上、スループットの向上、LLM 実行のレイテンシーの軽減が期待されます。

LLM のパフォーマンスを向上させるには、サーバー内とノード間の両方でネットワークのボトルネックを軽減する、柔軟かつ機敏なスケーラビリティーが必要です。すべての Habana® Gaudi®2 には、24 の 100GB イーサネット・ポートが統合されています。21 のポートをサーバー内の 8 つの Habana® Gaudi®2 の All-to-all 接続専用に、3 つのポートをスケールアウト専用にできます。このネットワーク構成は、サーバー内外の両方でスケールされたパフォーマンスを高速化するのに役立ちます。

Habana® Gaudi®2 は、最近公開された MLPerf* ベンチマーク（英語）に掲載された 384 の Habana® Gaudi®2 アクセラレーター上での 1750 億（175B）パラメーターの GPT-3* モデルのトレーニングで、大規模言語モデルでの優れたトレーニング・パフォーマンスを実証しました（詳細は、「MLCommons、AI でのインテルの強力な競争優位性を示す最新のベンチマーク結果を公開」を参照してください）。この実証されたパフォーマンスにより、Llama と Llama 2 のトレーニングと推論の両方で、Habana® Gaudi®2 は非常に効果的なソリューションとなります。

次に、単一の Habana® Gaudi®2 デバイスでの、バッチサイズ 1、出力トークン長 256、混合精度（BF16）を使用したさまざまな入力トークン長の Llama 2 7B および Llama 2 13B モデルの推論パフォーマンスを紹介します。パフォーマンス・メトリックは、（最初のトークンを除く）トークンあたりのレイテンシーです。推論の実行には、optimum-habana テキスト生成スクリプト（英語）を使用しました。Hugging Face の optimum-habana（英語）ライブラリーを使用すると、Habana® Gaudi® アクセラレーター向けにコード変更を最小限に抑えて、これらのモデルをシンプルかつ簡単にデプロイできます。**図 1** は、Habana® Gaudi®2 で入力トークン長 128 ~ 2K の推論を実行したレイテンシーが、7B モデルでトークンあたり 9.0 ~ 12.2 ミリ秒、13B モデルでトークンあたり 15.5 ~ 20.4 ミリ秒であることを示しています（ハードウェアとソフトウェアの構成の詳細は、この記事の最後に記載しています）。

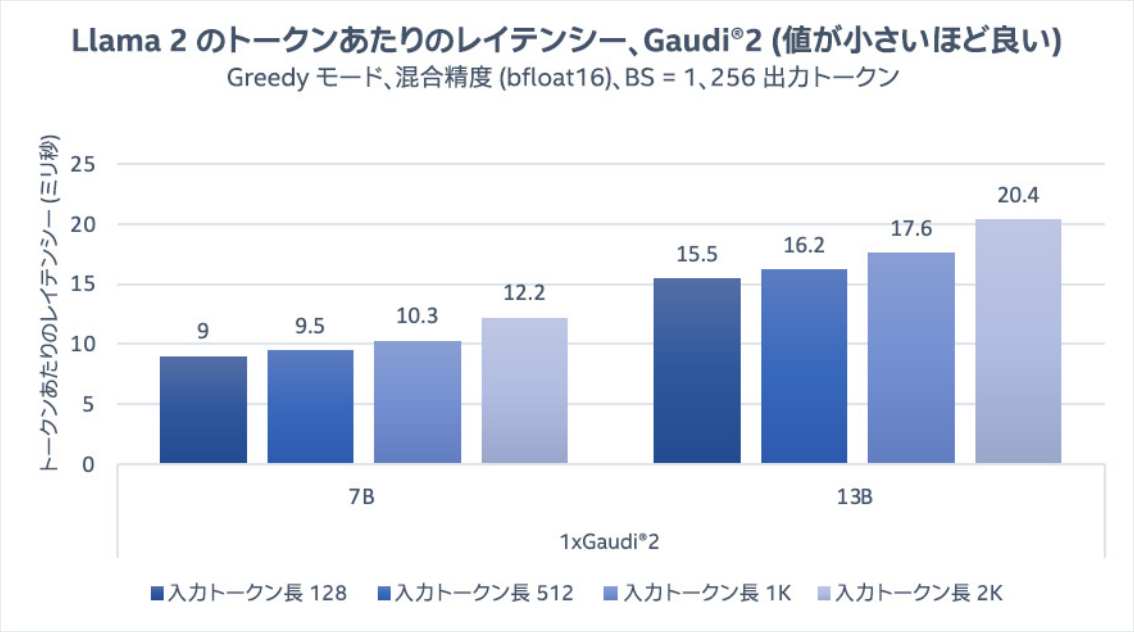

図 1. Habana® Gaudi®2 上の Llama 2 7B および 13B パラメーター・モデルの推論のパフォーマンス

Habana® Gaudi® プラットフォームで Llama 2 を使用して、生成 AI の旅を今すぐ始めることができます。Habana® Gaudi®2 にアクセスするには、インテル® デベロッパー・クラウド（英語）のインスタンスをサインアップするか、Habana® Gaudi®2 サーバー・インフラストラクチャーに関して Supermicro（英語）までお問い合わせください。

インテル® Xeon® スケーラブル・プロセッサー

第 4 世代インテル® Xeon® スケーラブル・プロセッサーには、インテル® アドバンスト・マトリクス・エクステンション（インテル® AMX）と呼ばれる AI 機能を大幅に高速化するアクセラレーターが内蔵されています。具体的には、すべてのコアに BF16 および INT8 GEMM（GEneral Matrix-matrix Multiplication、汎用行列乗算）アクセラレーターがあり、ディープラーニングのトレーニングと推論ワークロードを高速化します。さらに、インテル® Xeon® CPU マックス・シリーズでは、2 つのソケットで 128GB の高帯域幅メモリー（HBM2E）が搭載されており、ワークロードがメモリー帯域幅に制限されることが多い LLM で利点が得られます。

インテル® Xeon® プロセッサー向けのソフトウェア最適化はディープラーニング・フレームワーク（英語）にアップストリームされており、PyTorch*、TensorFlow*、DeepSpeed*、その他の AI ライブラリーのデフォルトのディストリビューションで利用できます。インテルは、PyTorch* 2.0（英語）の代表的な機能である torch.compile の CPU バックエンドの開発と最適化を主導しています。また、公式 PyTorch* ディストリビューションにアップストリームされる前でも、インテルの CPU 向けの高度な最適化を利用できるように PyTorch 向けインテル® エクステンション（英語）も提供しています。

大容量のメモリーが搭載されている第 4 世代インテル® Xeon® スケーラブル・プロセッサーは、単一ソケット内で低レイテンシーでの LLM 実行が可能であり、会話型 AI やテキスト要約アプリケーションに適用できます。この記事では、BF16 および INT8 で 1 つのソケットごとに 1 つのモデルを実行する場合のレイテンシーに注目しています。PyTorch 向けインテル® エクステンション（英語）は、INT8 精度モデルで適切な精度を確保する SmoothQuant（英語）をサポートしています。

LLM アプリケーションは高速リーダーの読み取り速度を満たす十分な速さでトークンを生成する必要があることから、調査するパフォーマンス・メトリックとしてトークンあたりのレイテンシー（各トークンの生成時間）を選択し、リファレンスとして熟練者の読み取り速度（トークンあたり 100 ミリ秒以下）を選択しました。**図 2** と **図 3** は、シングルソケットの第 4 世代インテル® Xeon® スケーラブル・プロセッサーで入力トークン長 32 ~ 2K の推論を実行したレイテンシーが、7B BF16 モデルおよび 13B INT8 モデルで 100 ミリ秒未満であることを示しています。

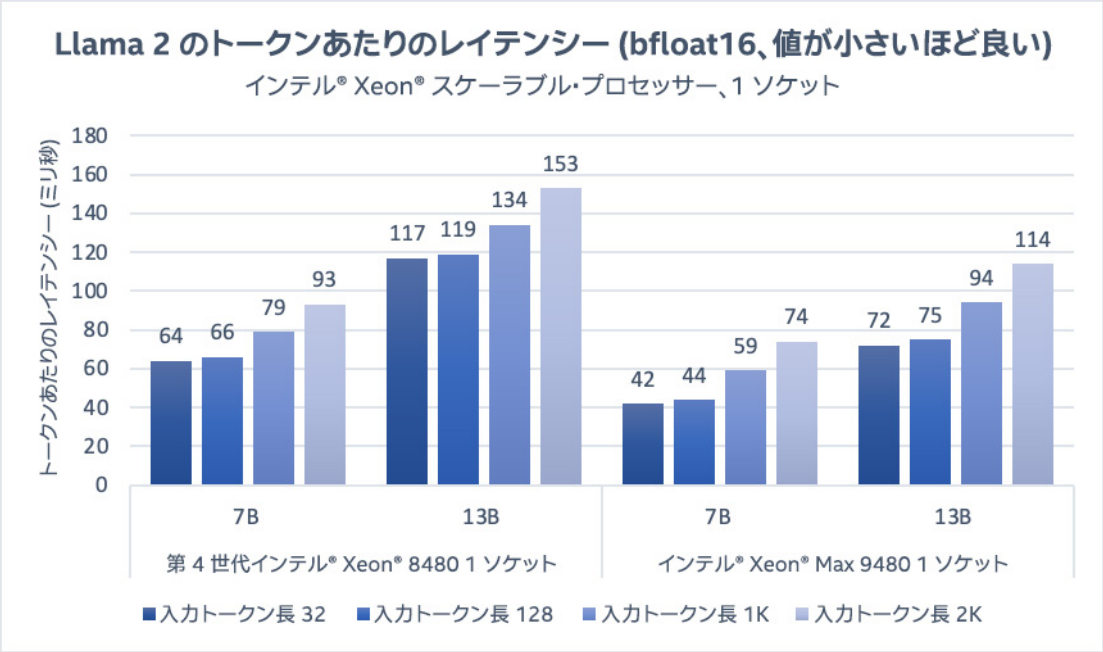

図 2. インテル® Xeon® スケーラブル・プロセッサー上の Llama 2 7B および 13B パラメーター・モデルの推論（bfloat16）のパフォーマンス

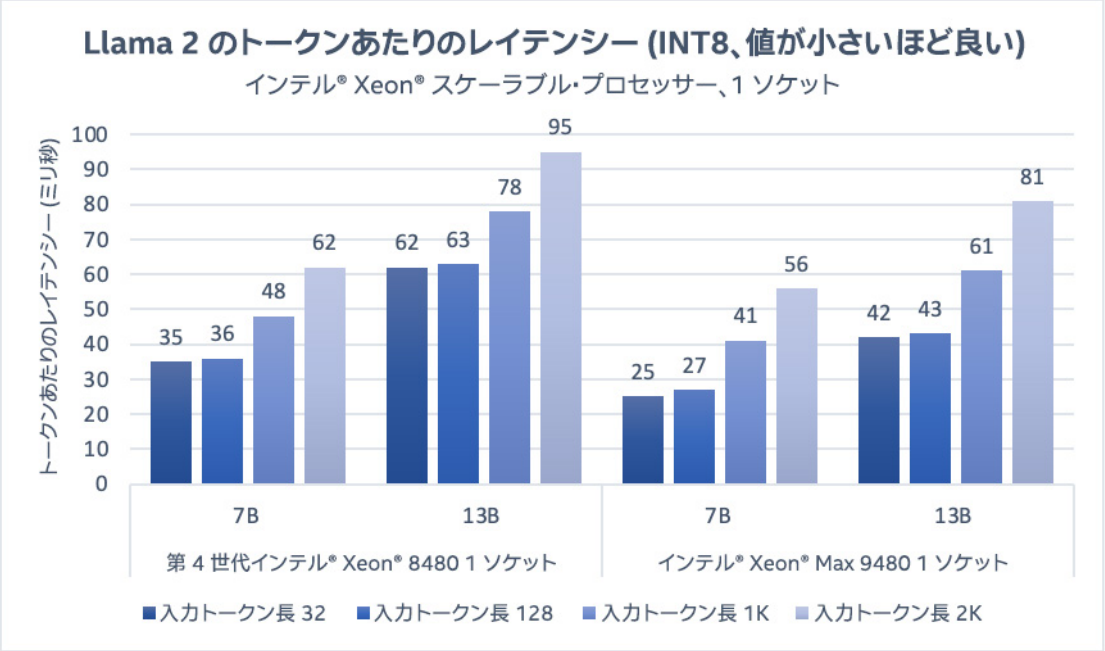

図 3. インテル® Xeon® スケーラブル・プロセッサー上の Llama 2 7B および 13B パラメーター・モデルの推論（INT8）のパフォーマンス

インテル® Xeon® CPU マックス・シリーズは、HBM2E の高帯域幅の恩恵を受け、両方のモデルのレイテンシーを軽減します。インテル® AMX アクセラレーションは、大きなバッチサイズでのスループットを向上させます。1 ソケットの第 4 世代インテル® Xeon® スケーラブル・プロセッサーで、7B および 13B パラメーター・モデルで 100 ミリ秒未満のレイテンシーを達成できます。ユーザーは、スループットを向上させ、複数クライアントの個々に対応できるように、各ソケットで 2 つの並列インスタンスを実行できます。あるいは、PyTorch 向けインテル® エクステンション（英語）と DeepSpeed*（英語）を活用して両方の第 4 世代インテル® Xeon® スケーラブル・プロセッサー上で推論を実行し、テンソル並列処理を使用してレイテンシーをさらに軽減したり、より大きなモデルをサポートすることもできます。

インテル® Xeon® プラットフォーム上での LLM と Llama 2 の実行に関する詳細は、こちら（英語）から入手できます。第 4 世代インテル® Xeon® スケーラブル・プロセッサーのクラウド・インスタンスは、AWS*、GCP* および Azure* でプレビュー版を利用できます。Ali Cloud では一般公開されています。インテルは、今後も PyTorch* および DeepSpeed* にソフトウェアの最適化を追加して、Llama 2 やその他の LLM をさらに高速化する予定です。

インテル® データセンター GPU マックス・シリーズ

インテル® データセンター GPU マックスは、並列計算、HPC、および AI ワークロードを高速化します。インテル® データセンター GPU マックス・シリーズは、インテルで最も高性能かつ高密度のディスクリート GPU であり、1,000 億個以上のトランジスターを 1 つのパッケージに搭載し、インテルの GPU 計算ビルディング・ブロックであるインテル® Xe コアを最大 128 個搭載しています。

インテル® データセンター GPU マックス・シリーズは、AI や HPC 分野で使用されるデータを多用するコンピューティング・モデルにおいて、画期的なパフォーマンスを発揮するように設計されています。

- ディスクリート SRAM テクノロジーに基づく 408MB の L2 キャッシュと 64MB の L1 キャッシュおよび最大 128GB の高帯域幅メモリー（HBM2E）。
- AI を強化するインテル® Xe マトリクス・エクステンション（インテル® XMX）は、シストリック・アレイを搭載し、単一のデバイスでベクトルと行列の機能を実現。

インテル® データセンター GPU マックス・シリーズ製品ファミリーは、生産性とパフォーマンスを最大限に引き出す、共通のオープンな標準ベースのプログラミング・モデルである oneAPI により統合されています。インテル® oneAPI ベース・ツールキットには、高度なコンパイラー、ライブラリー、プロファイラー、CUDA* コードから C++ with SYCL* への移行を支援するコード移行ツールが含まれています。

インテル® データセンター GPU マックス・シリーズ向けのソフトウェアの有効化と最適化は、PyTorch 向けインテル® エクステンション、TensorFlow 向けインテル® エクステンションおよび DeepSpeed 向けインテル® エクステンションなど、主要なフレームワーク向けのオープンソースの拡張を通じて提供されます。これらの拡張をアップストリームのフレームワークのリリースとともに使用することで、ユーザーはマシンラーニング・ワークフローのドロップイン・アクセラレーションを実現できます。

Llama 2 7B および 13B パラメーター・モデルの推論パフォーマンスはパッケージに 2 つの GPU（タイル）を搭載した 600W OAM デバイスで評価しましたが、推論の実行には 1 つのタイルのみ使用しました。**図 4** は、1 タイルのインテル® データセンター GPU マックスで入力トークン長 32 ~ 2K の推論を実行したレイテンシーが、7B モデルでトークンあたり 20 ミリ秒未満、13B モデルでトークンあたり 29.2 ~ 33.8 ミリ秒であることを示しています。ユーザーは、スループットを向上させ、複数のクライアントに別々に対応できるように、各タイルで 1 つずつ、2 つの並列インスタンスを実行できます。

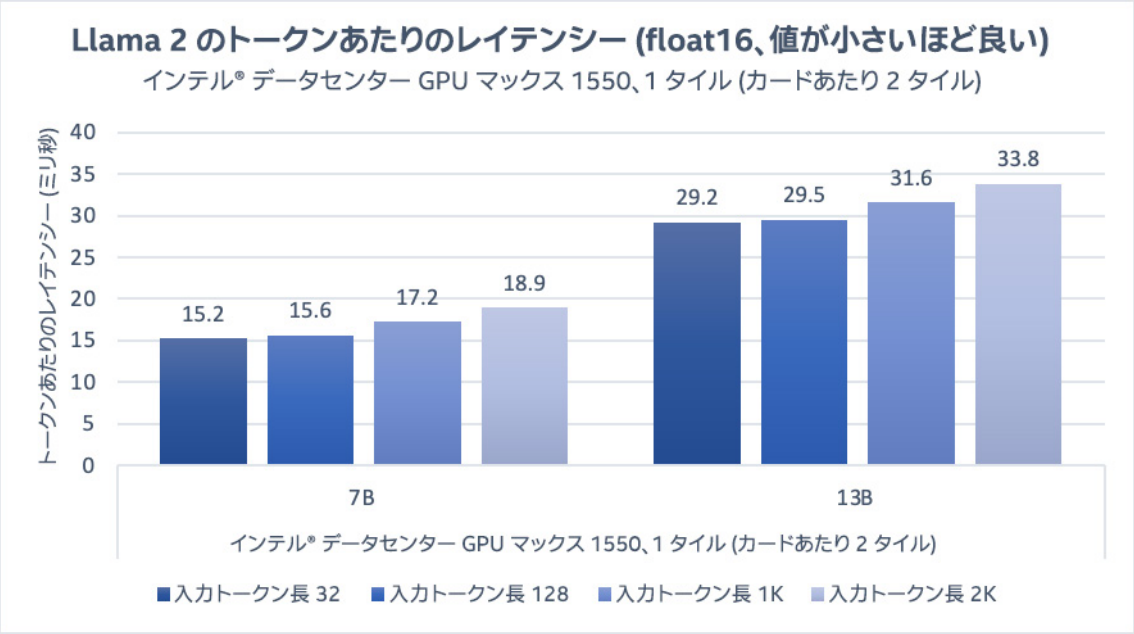

図 4. インテル® データセンター GPU マックス 1550 上の Llama 2 7B および 13B パラメーター・モデルの推論のパフォーマンス

インテル® データセンター GPU プラットフォーム上での LLM と Llama 2 の実行に関する詳細は、こちら（英語）から入手できます。インテル® デベロッパー・クラウドで、インテル® データセンター GPU マックスのクラウド・インスタンスを利用できます（記事執筆時点ではベータ版）。

インテルは、推論に加えて、Hugging Face Transformers（英語）、PEFT（英語）、Accelerate（英語）、および Optimum（英語）ライブラリーへの最適化のアップストリームによる微調整の高速化にも積極的に取り組んでいます。また、サポートされるインテルのプラットフォームにテキスト生成、コード生成、補完、要約などの典型的な LLM ベースのタスクを効率的にデプロイできる、Transformers 向けインテル® エクステンション（英語）のリファレンス・ワークフロー（英語）を提供しています。

まとめ

この記事では、Habana® Gaudi®2 ディープラーニング・アクセラレーター、第 4 世代インテル® Xeon® スケーラブル・プロセッサー、インテル® Xeon® CPU マックス・シリーズ、およびインテル® データセンター GPU マックス・シリーズを含む、インテルの AI ハードウェア・ポートフォリオ上での、Llama 2 の 7B および 13B パラメーター・モデルの初期推論パフォーマンスを紹介しました。ソフトウェア・リリースでは引き続き最適化を行っており、近々 LLM およびより大きな Llama 2 モデルについての更新情報を提供する予定です。

製品および性能に関する情報

Habana® Gaudi®2 ディープラーニング・アクセラレーター：測定は、8x Habana® Gaudi®2 HL-225H メザニンカード、2x インテル® Xeon® Platinum 8380 プロセッサー @ 2.30GHz および 1TB システムメモリーを搭載した HLS2- Gaudi®2 サーバー上で Habana SynapseAI* バージョン 1.10 および optimum-habana バージョン 1.6 を使用して行われました。パフォーマンスは 2023 年 7 月に測定されました。

第 4 世代インテル® Xeon® スケーラブル・プロセッサー：

- 第 4 世代インテル® Xeon® 8480：第 4 世代インテル® Xeon® Platinum 8480+ プロセッサー、2 ソケットシステム、112 コア（224 スレッド）、インテル® ターボ・ブースト・テクノロジー有効、インテル® ハイパースレッディング・テクノロジー有効、メモリー：16x32GB DDR5 4800MT/ 秒、ストレージ：953.9GB。OS：CentOS* Stream 8、カーネル：5.15.0-spr.bkc.pc.16.4.24.x86_64。バッチサイズ：1。1 ソケットで計測。PyTorch* nightly build0711。PyTorch 向けインテル® エクステンション tag v2.1.0.dev+cpu.llm。モデル：Llama 2 7B および 13B、データセット LAMBADA。トークン長：32/128/1024/2016（in）、32（out）。ビーム幅 4。精度：BF16 および INT8。2023 年 7 月 12 日に行われたインテル社内のテスト結果。
- インテル® Xeon® マックス 9480：インテル® Xeon® CPU マックス 9480 プロセッサー、2 ソケットシステム、112 コア（224 スレッド）、インテル® ターボ・ブースト・テクノロジー有効、インテル® ハイパースレッディング・テクノロジー有効、メモリー：16x64GB DDR5 4800MT/ 秒、8x16GB HBM2 3200 MT/ 秒、ストレージ：1.8TB。OS：CentOS* Stream 8、カーネル：5.19.0-0812.intel_next.1.x86_64+server。バッチサイズ：1。1 ソケットで測定。PyTorch* nightly build0711。PyTorch 向けインテル® エクステンション llm_feature_branch。モデル：Llama 2 7B および 13B、データセット LAMBADA。トークン長：32/128/1024/2016（in）、32（out）。ビーム幅 4。精度：BF16 および INT8。2023 年 7 月 12 日に行われたインテル社内のテスト結果。

インテル® データセンター GPU マックス・シリーズ：1 ノード、2x インテル® Xeon® Platinum 8480+ プロセッサー、56 コア、インテル® ターボ・ブースト・テクノロジー有効、インテル® ハイパースレッディング・テクノロジー有効、NUMA 2、合計メモリー 1024GB（16x64GB DDR5 4800MT/ 秒 [4800MT/ 秒]）。BIOS SE5C7411.86B.9525.D19.2303151347、マイクロコード 0x2b0001b0、1x イーサネット・コントローラー X710 10GBASE-T、1x1.8TB WDC WDS200T2B0B、1x931.5GB INTEL SSDPELKX010T8、Ubuntu* 22.04.2 LTS、5.15.0-76-generic、4x インテル® データセンター GPU マックス 1550（1 つの OAM GPU カードの 1 つのタイルのみ使用して測定）、IFWI PVC 2_1.23166、agama driver：agama-ci-devel-627.7、インテル® oneAPI ベース・ツールキット 2023.1、PyTorch* 2.0.1 + PyTorch 向けインテル® エクステンション v2.0.110+xpu（dev/LLM branch）。AMC ファームウェア・バージョン：6.5.0.0。モデル：Llama 2 7B および 13B、データセット LAMBADA。トークン長：32/128/1024/2016（in）、32（out）、Greedy 検索、精度：FP16。2023 年 7 月 7 日に行われたインテル社内のテスト結果。

インテル® Arc™ GPU 上の TensorFlow* による転移学習

インテルのコンシューマー向け GPU と Windows* Subsystem for Linux* 2 を使用した高速かつ簡単なトレーニングと推論

Christopher Lishka インテル コーポレーション AI フレームワーク・エンジニア
AG Ramesh インテル コーポレーション 主席エンジニア
Geetanjali Krishna インテル コーポレーション AI ソフトウェア・エンジニアリング・マネージャー

転移学習は、新しいデータセットで事前トレーニング済みモデルの使用を可能にするディープラーニング（DL）手法です。モデルの重みで学習した特徴が適切に適用されるように、新しいデータはオリジナルのデータに十分に類似している必要があります。この場合、モデルを完全にトレーニングする場合と比較して、新しいデータセットを使用してトレーニングする時間を大幅に短縮できます。これは、レイヤーの大部分で事前トレーニング済みの静的重みを残し、最後のモデルレイヤーを新しいデータセットでトレーニングされた新しい Dense レイヤーに置き換える、「ヘッドレスモデル」を使用しているためです。

インテル® Arc™ A シリーズ・ディスクリート GPU は、PC 上で DL ワークロードを迅速に実行する簡単な方法を提供し、TensorFlow* モデルと PyTorch* モデルの両方で動作します。この記事では、インテル® Arc™ GPU 上で TensorFlow 向けインテル® エクステンション（ITEX）（英語）を実行し、Windows* 上で事前に構築された ITEX Docker* イメージを使用してセットアップを簡素化します。この例では、ImageNet データセットで事前トレーニング済みの、TensorFlow* Hub の EfficientNetB0 モデルを使用します。新しいデータセットは、TensorFlow* Datasets にある「Stanford Dogs」です。これには ImageNet で提供されているものより正確な犬種のラベルが付けられた犬の画像が含まれています。Stanford Dogs 向けに DL ネットワークを調整するため、インテル® Arc™ A770 GPU を使用して迅速にトレーニングされる新しい Dense レイヤーを追加します。次に、完全にトレーニングした EfficientNetB0 と比較して、転移学習がどのくらい高速化されたかを確認します。

セットアップ

以前の記事、「インテル® Arc™ GPU で TensorFlow* Stable Diffusion を実行する」では、Windows* Subsystem for Linux* 2（WSL2）で実行する Ubuntu* コンテナをセットアップして Windows* ホスト上のインテル® Arc™ GPU にアクセスする方法を説明しました。この例では、WSL2 と Docker*、および事前に構築された ITEX Docker* イメージを使用して、ITEX プラグインのインストールを簡素化する方法を紹介します。この記事では、16GB のインテル® Arc™ A770 ディスクリート GPU カードを装着した第 13 世代インテル® Core™ i9 プロセッサー・ベースの PC を使用しました。Windows* 11 上の Docker* コンテナで実行する ITEX を使用します。

準備

まず、Windows* 11 に WSL2 と Ubuntu* 22.04 コンテナがインストールされている必要があります。インストール方法は、こちらを参照してください。Docker* Desktop on Windows* は、Docker* イメージをダウンロードし、WSL2 サブシステムを使用して Docker* コンテナを実行する簡単な方法を提供します。ITEX が設定済みのイメージを使用しますが、最初に Docker* を Windows* にインストールする必要があります。Docker* Desktop for Windows* を使用するか(インストール手順(英語))、WSL2 内で実行している Linux* に Docker* をインストールします(インストール例（英語))。

Docker* コンテナで Jupyter Notebook* を実行する

この例では、Ubuntu* (WSL2 内で実行) から Docker* を実行します。最初に、Ubuntu* WSL2 コンテナを起動します。Unix* シェルプロンプトから、ITEX 向けに事前に構築された Docker* イメージをプルします。

```
$ docker pull intel/intel-extension-for-tensorflow:xpu
```

ITEX Docker* イメージがダウンロードされていることを確認します。

```
$ docker images
REPOSITORY                              TAG       IMAGE ID       CREATED         SIZE
intel/intel-extension-for-tensorflow    xpu       2fc4b6a6fad7   8 days ago      7.42GB
```

次に、`docker run` コマンドを使用して、`intel/intel-extension-for-tensorflow:xpu` イメージでコンテナを開始します。

```
$ docker run  -ti   -p 9999:9999  --device /dev/dxg --mount type=bind,src=/usr/lib/wsl,dst=/usr/
lib/wsl -e LD_LIBRARY_PATH=/usr/lib/wsl/lib  intel/intel-extension-for-tensorflow:xpu
```

`docker run` コマンドのオプションは次のとおりです。

- `-p 9999:9999` は、Jupyter Notebook* を Windows* のブラウザーで使用できるように、Docker* コンテナ経由で標準ポートを渡します。
- `--device /dev/dxg` は、インテル® Arc™ A770 GPU にアクセスできるように、DirectX* ドライバーを Docker* コンテナに渡します。
- `--mount type=bind,src=/usr/lib/wsl,dst=/usr/lib/wsl -e LD_LIBRARY_PATH=/usr/lib/wsl/lib` は、WSL2 の共有ライブラリー・ディレクトリーを Docker* コンテナに渡します。

Docker* Desktop for Windows* を使用している場合は、PowerShell* またはコマンドプロンプトで上記の Docker* コマンドを直接実行できます。

ITEX Docker* コンテナを実行したら、Jupyter Notebook* を含むいくつかのパッケージを `pip` でインストールします。

```
root:/# pip install jupyter 'ipywidgets>=7.6.5' 'matplotlib>=3.3.4' scipy 'tensorflow_hub>=0.12.0'
'tensorflow-datasets>=4.4.0'
```

Jupyter Notebook* を開始します。

```
root:/# jupyter notebook --allow-root --ip 0.0.0.0 --port 9999
```

上記の Jupyter Notebook* コマンドを実行した後、Microsoft Edge* ブラウザーを使用して Jupyter Notebook * サーバーに接続します。Microsoft Edge* ブラウザーで、Jupyter* の出力でリストされた、次のような URL を開きます。

```
http://127.0.0.1:8888/?token=...
```

インテル® Arc™ A770 GPU を使用した転移学習

この例を Jupyter Notebook* で実行すると、トレーニング前、トレーニング中、トレーニング後に犬の画像がどのように分類されているか簡単に視覚化できます。各コードセクションをセルとして実行し、効果を確認します。各自の Notebook に従ってかまいません。

インポートする主なパッケージは、TensorFlow*、TensorFlow* Hub（標準の事前トレーニング済みモデルへのアクセスを提供）、および TensorFlow* Datasets（さまざまな用途に適した標準のトレーニングおよびテストセットを提供）です。また、ヘルパー関数で画像と分類の視覚化に使用するため、Matplotlib と NumPy* もインポートします。

```
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np
```

```
[…]
2023-08-03 16:44:35.519530: I itex/core/devices/gpu/itex_gpu_runtime.cc:129] Selected platform:
Intel(R) Level-Zero
2023-08-03 16:44:35.519589: I itex/core/devices/gpu/itex_gpu_runtime.cc:154] number of sub-devices is
zero, expose root device.
```

ITEX プラグインは、TensorFlow* パッケージのインポート中に自動的に検出されます。上記の出力メッセージで示されているように、インテルの GPU が存在する場合は自動的にデフォルトのデバイス（インテル® oneAPI レベルゼロを使用）に設定されます。

次に、環境に設定できるパラメーターをいくつか示します。

```
batch_size = 32
dataset_directory = '/tmp/efficienetB0_datasets'     # ダウンロードしたデータセットの場所
output_directory = '/tmp/efficienetB0_saved_model'  # モデルを保存する場所
```

この実行では、TensorFlow* Hub の EfficientNetB0 を使用します。最後のレイヤーを削除した重み付きの事前トレーニング済みレイヤーであるヘッドレスモデル（「特徴ベクトルモデル」とも呼ばれます）のみを使用します。各画像の解像度も指定する必要があります。ここでは、EfficientNetB0 を 224 x 224 ピクセルの画像でトレーニングします。これは、データセットの画像のサイズを変更するために使用されます。

TensorFlow* Hub の EfficientNet モデルのドキュメントはこちら（英語）から入手できます。分類モデルと特徴ベクトルモデル（ヘッドレスモデル）の両方へのリンクが含まれています。

```
# TensorFlow Hub の "efficientnet_b0" モデル
model_handle = "https://tfhub.dev/google/efficientnet/b0/classification/1"
feature_vector_handle = "https://tfhub.dev/google/efficientnet/b0/feature-vector/1"
image_size = 224
```

次のヘルパー関数、`show_predictions()` は、予測されたラベルと実際のラベル、およびバッチ内の最初の 10 個の画像を視覚的に表示します。正しくラベル付けされた画像は緑で表示されます。誤ってラベル付けされた画像は赤で表示され、括弧内に正しいラベルが表示されます。転移学習のトレーニング中に進行状況を確認するため、いくつかのポイントで `show_predictions()` を使用します。

では、データセットを設定しましょう。ここでは、TensorFlow* Datasets を使用して、ImageNet で提供されている

```
# ヘルパールーチンで正しい予測にビジュアル・インジケーターを追加してバッチの 10 の画像を表示
def show_predictions(image_batch, label_batch, class_names):

    predicted_batch = model.predict(image_batch)  # サンプルバッチを使用して予測
    predicted_id = np.argmax(predicted_batch, axis=-1)
    predicteds = [class_names[id] for id in predicted_id]

    actuals = [class_names[int(id)] for id in label_batch]

    print("Correct predictions are shown in green")
    print("Incorrect predictions are shown in red with the actual label in parenthesis")

    plt.figure(figsize=(10,9))  # 結果を表示
    plt.subplots_adjust(hspace=0.5)
    for n in range(0, 10):  # (batch_size - 2) を使用するとバッチ全体を出力
        plt.subplot(6,5,n+1)
        plt.imshow(image_batch[n])
        correct_prediction = actuals[n] == predicteds[n]
        color = "darkgreen" if correct_prediction else "crimson"
        title = predicteds[n].title() \
            if correct_prediction \
            else "{}\\n({})".format(predicteds[n], actuals[n])
        plt.title(title, fontsize=9, color=color)
        plt.axis('off')
    _ = plt.suptitle("Model predictions")
    plt.show()
```

ものより正確な犬種のラベルが付けられた犬の画像を含む「Stanford Dogs」データセットをロードします。このコードを初めて実行すると、上記で指定したディレクトリーにデータセットがダウンロードされます。完了するまで 5 ~ 10 分かかります。ダウンロードの状況は、プログレスバーで確認できます。同じ Docker* コンテナ内で続けて実行する場合は、キャッシュされたデータセットが使用されるため、高速に実行されます。

データセットが利用可能になると、トレーニング (75%) とテスト (25%) に分割されます。次に、それぞれキャッシュされ、バッチに分割され、プリフェッチに設定されます。トレーニング・データセットもシャッフルされるため、Dense レイヤーのトレーニング速度にランダム性が生じます。最後に、後で使用できるようにデータセットのクラス名が保存されます。

Stanford Dogs データセットはこちら（英語）から入手できます。このデータセットが選択された理由は、転移学習で 90% までトレーニングするのに数エポックかかる大きさでありながら、妥当な時間でダウンロードできるためです。TF-Datasets には、小規模なものから大規模なものまで、使用できるデータセットがほかにもあります。

```
[train_ds, test_ds], info = tfds.load("stanford_dogs",  # TensorFlow Datasets からデータセットをロード
                        data_dir=dataset_directory,
                        split=["train[:75%]", "train[:25%]"],
                        as_supervised=True,
                        shuffle_files=True,
                        with_info=True)

def preprocess_image(image, label):  # 画像を float32 に変換してモデルと一致するようにサイズ変更
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize_with_pad(image, image_size, image_size)
    return (image, label)

train_ds = train_ds.map(preprocess_image)
test_ds = test_ds.map(preprocess_image)

# ランダムになるようにトレーニング・データをシャッフル
train_ds = train_ds.cache()
train_ds = train_ds.shuffle(info.splits['train'].num_examples)
train_ds = train_ds.batch(batch_size)
train_ds = train_ds.prefetch(tf.data.AUTOTUNE)

# テストデータはシャッフル不要、キャッシュはバッチ後に完了
test_ds = test_ds.batch(batch_size)
test_ds = test_ds.cache()
test_ds = test_ds.prefetch(tf.data.AUTOTUNE)

class_names = info.features["label"].names  # データセットのクラス名を取得
```

次に、TensorFlow* Hub からヘッドレス EfficientNetB0 モデルをロードします。このモデルに含まれる、(ImageNet データセットでのトレーニングに基づく) 事前トレーニング済みの重みは変更しません。完全にトレーニングされる Dense レイヤーも追加します。このレイヤーのサイズは、データセットのクラスの数に合わせて設定されます。次にモデルがコンパイルされ、形状が出力されます。ヘッドレスモデルは、事前トレーニング済みの重みを含む多くのレイヤーで構成されているにもかかわらず、単一の KerasLayer として表示されることに注意してください。完全なモデルには 4,203,284 のパラメーターがありますが、そのうち 153,720 のパラメーターのみトレーニングされ、残りは変更されません。

```
# 特徴ベクトルレイヤーはトレーニングしない (すでにトレーニング済み)
feature_extractor_layer = hub.KerasLayer(feature_vector_handle,
                                         input_shape=(image_size, image_size, 3),
                                         trainable=False)
model = tf.keras.Sequential([
    feature_extractor_layer,
    tf.keras.layers.Dense(len(class_names))  # トレーニングする Dense レイヤーを追加
])

model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['acc'])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 1280)              4049564

 dense (Dense)               (None, 120)               153720

=================================================================
Total params: 4,203,284
Trainable params: 153,720
Non-trainable params: 4,049,564
_________________________________________________________________
```

トレーニングを開始する前に、トレーニングされていない Dense レイヤーを使用して、モデルが適切な犬種ラベルをどの程度予測できるか確認してみましょう。まず、テスト・データセットで推論を実行します。

```
with tf.device('/xpu:0'):  model.evaluate(test_ds, batch_size=batch_size)
```

```
94/94 [==============================] - 29s 158ms/step - loss: 2.4268 - acc: 0.0087
```

次に、ヘルパー関数を使用して、最初のバッチの最初の 10 枚の画像を表示します。

```
batch = next(iter(test_ds))  # テストに使用するデータセットのバッチを取得
image_batch, label_batch = batch
show_predictions(image_batch, label_batch, class_names)
```

```
1/1 [==============================] - 1s 834ms/step
      Correct predictions are shown in green
      Incorrect predictions are shown in red with the actual label in parenthesis
```

Dense レイヤーをトレーニングしていない場合、予想どおり、すべての予測は不正確で、赤で表示されています。上のラベルは誤った分類で、下のラベル（括弧内）は正しい予測です。

では、トレーニングを行ってみましょう。Dense レイヤーの重みのみ調整します。まず、ラベル予測に対する完全でないトレーニングの効果を後から確認できるように、1 エポックのみ実行します。使用されているデバイスは `/xpu:0`（つまり、ホスト PC にインストールされているインテル® Arc™ A770 GPU）であることに注意してください。

```
with tf.device('/xpu:0'):  model.fit(train_ds, epochs=1, shuffle=True, verbose=1)
```

```
282/282 [==============================] - 28s 85ms/step - loss: 0.8545 - acc: 0.6581
```

1 エポック後、転移学習を使用して Dense レイヤーのみトレーニングしただけで、精度は 65.81% になりました。Stanford Dogs を使用して EfficientNetB0 を最初から完全にトレーニングした場合、最初のエポック後の精度は約 1.99% にすぎません。この最初のエポックには約 28 秒かかることにも注意してください。これは、最初のエポックでは GPU をウォームアップする（データを GPU メモリーに転送して、後の実行のために保存する）必要があり、後のエポックよりも遅くなるためです。

では、最初のバッチの予測を見てみましょう。まず、テスト・データセットに推論を実行します。

```
with tf.device('/xpu:0'):  model.evaluate(test_ds, batch_size=batch_size)
```

```
94/94 [==============================] - 4s 41ms/step - loss: 0.3159 - acc: 0.8697
```

次に、ヘルパー関数を使用して予測を表示します。

```
show_predictions(image_batch, label_batch, class_names)
```

```
1/1 [==============================] - 0s 65ms/step
Correct predictions are shown in green
Incorrect predictions are shown in red with the actual label in parentheses
```

バッチの最初の 10 枚の画像のうち 9 枚が正しく予測されました。これは、テスト・データセットの推論の精度が 86.97% に向上したためです。トレーニングの精度を向上させると、より多くの画像が正しく予測されるようになります。実行では、トレーニング・データセットがシャッフルされ、トレーニングの精度にある程度のランダム性が生じるため、多少の予測ミスが発生する可能性があります。

では、さらに 2 つのトレーニング・エポックを実行してみましょう。

```
with tf.device('/xpu:0'):  model.fit(train_ds, epochs=2, shuffle=True, verbose=1)
```

```
Epoch 1/2
282/282 [==============================] - 12s 41ms/step - loss: 0.2812 - acc: 0.8581
Epoch 2/2
282/282 [==============================] - 12s 40ms/step - loss: 0.1967 - acc: 0.9029
```

わずか 3 エポックのトレーニングの後、転移学習の精度は 90.29% まで向上しました。インテル® Arc™ A770 GPU で Stanford Dogs を使用して EfficientNetB0 を最初から完全にトレーニングした場合、90% の精度に達するには約 30 エポックかかります。つまり、転移学習を使用すると、より速く高い精度に達することができます。

最初のエポックでデータが GPU メモリーにコピーされていたため、以降の 2 つのエポックはそれぞれインテル® Arc™ A770 GPU でわずか 12 秒で完了しました。Dense レイヤーのみトレーニングすればよいことも転移学習でエポックの完了が速くなる理由の 1 つです。インテル® Arc™ A770 GPU で EfficientNetB0 のすべてのレイヤーを完全にトレーニングすると、(最初のエポックの後) エポックごとに約 61 秒かかります。転移学習は、トレーニング時間の短縮 (収束までのエポック数の減少) とエポック数の短縮 (トレーニングするパラメーター数の減少) の両方を実現します。

インテル® Arc™ A770 GPU は、インテル® Core™ i9 CPU に比べて大幅なスピードアップを達成します。結果を比較したところ、GPU での転移学習のトレーニングは CPU よりも 10 倍以上高速でした。

では、わずか 3 エポックのトレーニング後に予測がどのようになるか見てみましょう。

```
with tf.device('/xpu:0'):  model.evaluate(test_ds, batch_size=batch_size)
```

```
94/94 [==============================] - 4s 42ms/step - loss: 0.1411 - acc: 0.9410
```

テスト・データセットの精度はすでに 94.10% に達しています。

```
show_predictions(image_batch, label_batch, class_names)
```

```
1/1 [==============================] - 0s 56ms/step
Correct predictions are shown in green
Incorrect predictions are shown in red with the actual label in parentheses
```

94.10% の精度で、テスト・データセットの最初の 10 枚の画像がすべて正しく予測されました。インテル® Arc™ A770 GPU で Stanford Dogs を使用して EfficientNetB0 を最初から完全にトレーニングすると、90% の精度に達するまで 30 エポックで約 31 分かかります。転移学習を使用すると、わずか数分で同じ結果を得ることができます。

PyTorch* を使用して山火事を予測する

CPU で転移学習を使用して正確な画像分類器を効率良く開発

Bob Chesebrough インテル コーポレーション シニア AI ソリューション・アーキテクト

この記事では、PyTorch* による転移学習（英語）を使用して、画像のディテールのみを使用し、空中写真が伝える火災の危険性に従って空中写真を分類します。MODIS 火災データセット（英語）は、2018 年から 2020 年までにカリフォルニア州で発生した実際の火災を記録しています。MODIS（Moderate Resolution Imaging Spectroradiometer、中解像度イメージング分光放射計）データセットには、過去の山火事の場所の詳細な情報が得られるように、特定の日付範囲の高解像度画像とラベル付けされた地図データが含まれています。次に、火災が発生した地域および近隣地域の 2016 年から 2017 年の 2 年間の（火災が発生する前の）画像をサンプリングします。転移学習を使用して、「Fire（火災発生）」および「NoFire（火災なし）」とラベル付けされた数百枚の画像を追加し、（空中写真で事前トレーニングされていない）事前トレーニング済み ResNet 18 モデルを適応させます。

空中写真で使用するために（もともと ImageNet データセットでトレーニングされた）事前トレーニング済みモデルを微調整することは、山火事を予測するという状況下で画像から有用な情報を抽出する効果的なアプローチです。深い層とスキップ接続を備えた ResNet アーキテクチャーは、物体認識や画像分類を含むさまざまなコンピューター・ビジョン（英語）タスクで効果的であることが実証されています。このアプローチを使用すると、正確なモデルを構築するために必要なのは、数百枚の画像と約 15 分の CPU 時間だけです。詳細は、続きをお読みください。

空中写真を使用したケーススタディー

私のアプローチは、2016 年から 2021 年までの、カリフォルニア州の火災が発生した地域と発生していない地域の空中写真を利用した予測のみに焦点を当てたバイナリー分類器を作成することでした。トレーニング・セットには、2016 年から 2017 年までの、重要な地域の火災発生前の空中写真を使用しました。評価セットには、2018 年から 2020 年までに撮影された同じ場所の画像（および 2021 年の画像を含む拡張セット）を使用しました。火災が発生した地域と発生していない地域の両方をサンプリングしました。山火事の可能性は、MODIS データセットから取得した既知の山火事の地域に基づいています。サクラメントの北の、太平洋からシエラネバダ山脈までの地域を選択しました。この地域では、過去に大規模な山火事が何度も発生しています。

データの収集

データの取得と前処理は、次の基本的な手順に従いました。最初に、Google Earth Engine*（英語）と JavaScript* プログラムを使用して MODIS 火災データと空中写真を収集しました。プロジェクトのスクリプトは ForestFirePrediction（英語）リポジトリーにあります。次に、米国農務省の USDA/NAIP/DOQQ データセットから地図を生成しました。最後に、NASA MODIS/006/MCD64A1 データセットから空中写真を抜き出しました。

MODIS により定義された、2018 年から 2020 年までに火災が発生した場所と発生していない場所を**図 1** と**図 2** に示します。赤は火災が発生した地域です。オレンジと水色のピンは使用した画像のサンプルの場所で、オレンジのピンは火災が発生した場所を、水色のピンは火災が発生していない場所を表します。各画像は約 60 平方マイル（約 155 平方キロメートル）の範囲をカバーします。

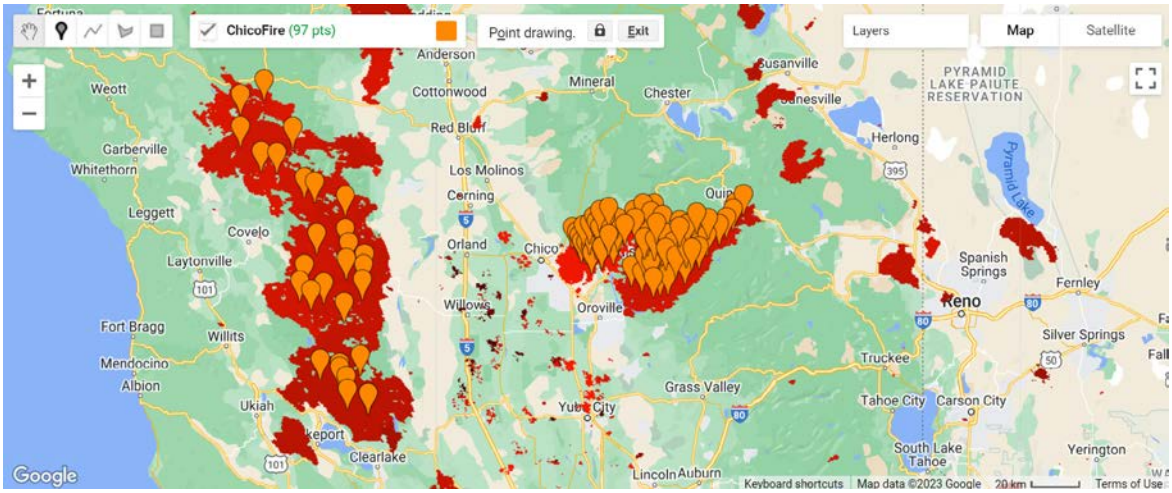

図 1. Google Earth Engine* と MODIS/006/MCD64A1 データセットを使用して山火事が発生した場所をサンプリング

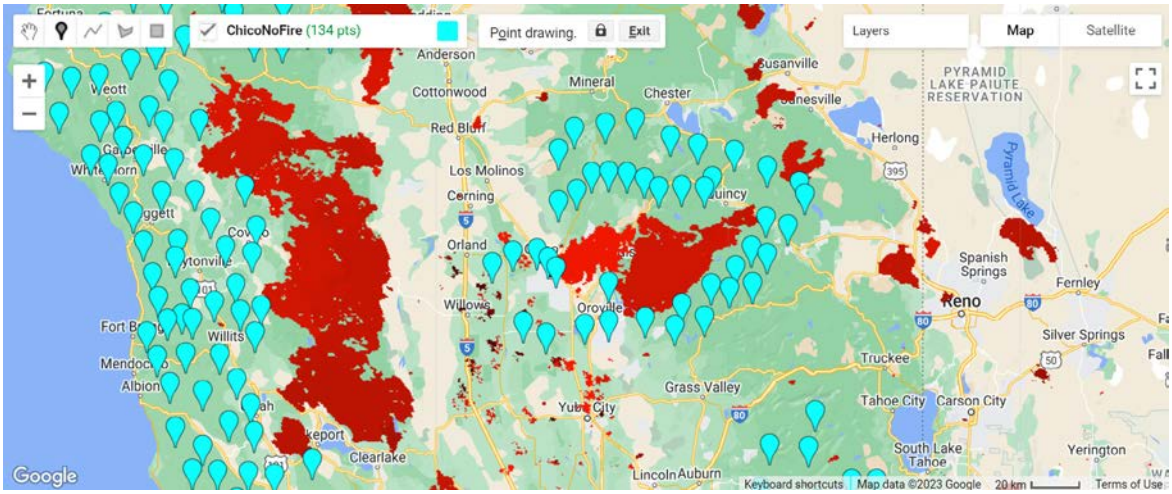

図 2. Google Earth Engine* と MODIS/006/MCD64A1 データセットを使用して山火事が発生していない場所をサンプリング

空中写真のサンプリングには、NAIP/DOQQ データセットを使用しました。例えば、**図 3** は、カリフォルニア州パラダイス付近の、大規模火災（2018 年）が発生する前の空中写真を示しています。

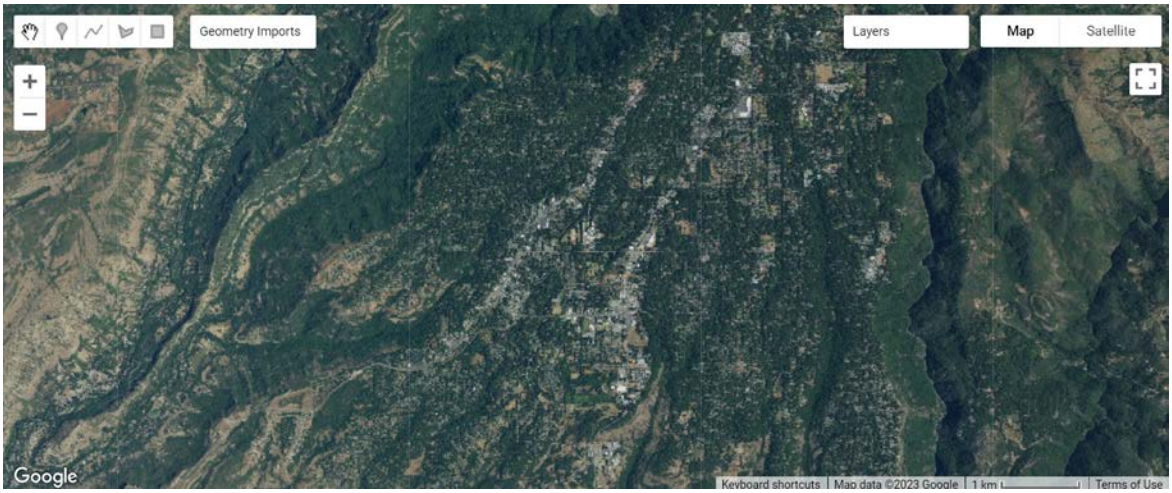

図 3. USDA/NAIP/DOQQ データセットの空中写真のサンプル (2018 年の山火事発生前のカリフォルニア州パラダイス)

火災が発生した地域のサンプルとして 106 枚の画像を、火災が発生していない地域のサンプルとして 111 枚の画像を使用しました（**表 1**）。転移学習を使用しているため、モデルを最初からトレーニングする場合よりもデータセットをはるかに小さくできます。画像のトレーニング、検証、テストの内訳は次のとおりです。

	トレーニング	検証	テスト
FIRE	87	9	10
NOFIRE	90	10	11

コード

モデルは ResNet-18 ベースです。ユーティリティー関数、トレーナークラス、モデルクラス、メトリクスクラスの 4 つの主要コードセクションを作成しました。さらに、最終的な混同行列とモデルの精度を表示するコードを追加しました。

インポート

```
import intel_extension_for_pytorch as ipex
import torch
import torch.nn as nn
import torchvision.models as models
from torch.utils.data import DataLoader
from torchvision import datasets, models, transforms
```

トレーニングと検証のデータセットを作成

トレーニング画像と検証画像の場所を定義し、各セット内の各画像の画像拡張を計算します。

```
num_physical_cores = psutil.cpu_count(logical=False)
data_dir = pathlib.Path(“./data/output/”)
TRAIN_DIR = data_dir / “train”
VALID_DIR = data_dir / “val”

…
```

トレーニングと検証セットのデータセット変換を定義

各画像で実行する一連の拡張を定義します。

```
…
img_transforms = {
    “train”: transforms.Compose(
        [
            transforms.RandomHorizontalFlip(),
            transforms.RandomVerticalFlip(),
            transforms.RandomRotation(45),
            transforms.ToTensor(),
            transforms.Normalize(*imagenet_stats),
    ),
…
}
…
```

モデルのクラスを定義

モデルは、ResNet18 ベースのディープ・ニューラル・ネットワークのバイナリー分類器です。

```
class FireFinder(nn.Module):
…
    def __init__(self, backbone=18, simple=True, dropout= .4):
        super(FireFinder, self).__init__()
        backbones = {
            18: models.resnet18,
        }
            fc = nn.Sequential(
                nn.Linear(self.network.fc.in_features, 256),
                nn.ReLU(),
                nn.Dropout(dropout),
                nn.Linear(256, 2)
```

Trainer クラスを定義

このステップでは、インテル® AI アナリティクス・ツールキットの一部である、PyTorch 向けインテル® エクステンション（英語）を使用します。

```
class Trainer:
…
        self.loss_fn = torch.nn.CrossEntropyLoss()
        self.ipx = ipx
        self.epochs = epochs
        if isinstance(optimizer, torch.optim.Adam):
            self.lr = 2e-3
        self.optimizer = optimizer(self.model.parameters(), lr=lr)

    def train(self):
        self.model.train()
        t_epoch_loss, t_epoch_acc = 0.0, 0.0
        start = time.time()
        for inputs, labels in tqdm(train_dataloader, desc="tr loop"):
            inputs, labels = inputs.to(self.device), labels.to(self.device)
            if self.ipx:
                inputs = inputs.to(memory_format=torch.channels_last)
            self.optimizer.zero_grad()
            loss, acc = self.forward_pass(inputs, labels)
            loss.backward()
            self.optimizer.step()
            t_epoch_loss += loss.item()
            t_epoch_acc += acc.item()
        return (t_epoch_loss, t_epoch_acc)
…
    def _to_ipx(self):
        self.model.train()
        self.model = self.model.to(memory_format=torch.channels_last)
        self.model, self.optimizer = ipex.optimize(
            self.model, optimizer=self.optimizer, dtype=torch.float32
        )
```

モデルをトレーニング

20 エポック、ドロップアウト率 0.33、学習率 0.02 でモデルをトレーニングします。

```
epochs = 20
ipx = True
dropout = .33
lr = .02

torch.set_num_threads(num_physical_cores)
os.environ["KMP_AFFINITY"] = "granularity=fine,compact,1,0"

start = time.time()
model = FireFinder(simple=simple, dropout= dropout)
trainer = Trainer(model, lr = lr, epochs=epochs, ipx=ipx)
tft = trainer.fine_tune(train_dataloader, valid_dataloader)
```

モデルの推論

モデルに対して推論するクラスと関数を定義します。

```
class ImageFolderWithPaths(datasets.ImageFolder):
…
def infer(model, data_path: str):
    transform = transforms.Compose(
        [transforms.ToTensor(), transforms.Normalize(*imagenet_stats)]
    )
    data = ImageFolderWithPaths(data_path, transform=transform)
    dataloader = DataLoader(data, batch_size=4)
…
```

次のコード例は、モデルを使用して画像をスコアリングする方法を示しています。

```
images, yhats, img_paths = infer(model, data_path="./data//test/")
```

モデルの精度

次の混同行列は、21 のサンプル中に 2 つの誤検出があることを示しています。モデルの全体的な精度は約 89.9% です。

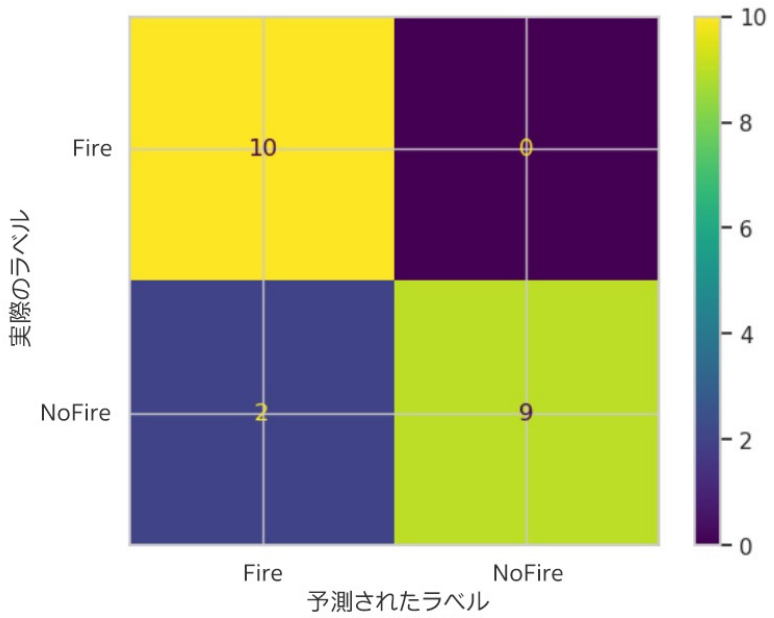

図 4 は、すべてのサンプル（トレーニング、検証、テスト）のモデルの予測を示しています。この図から、モデルが火災が発生する危険性のある地域を非常に正確に予測したことが分かります。緑のピンはモデルが火災が発生しないと予測したサンプルの場所で、赤のピンはモデルが火災が発生すると予想したサンプルの場所です。赤で表示されているのは、2018 年から 2020 年までに実際に火災が発生した地域です。

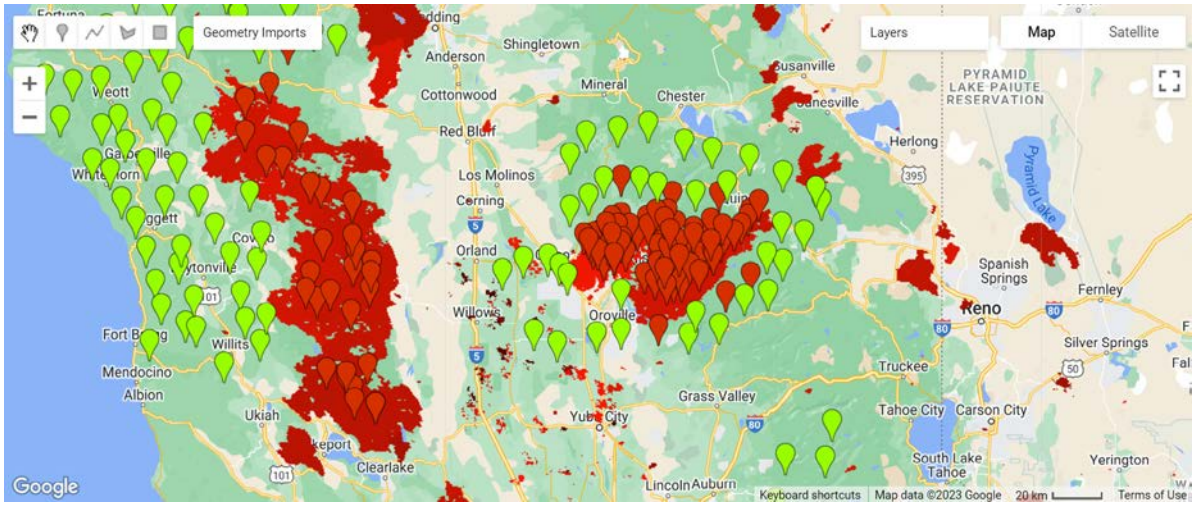

図 4. すべてのサンプルの推論結果

図 5 は、カリフォルニア州パラダイス付近の拡大図です。

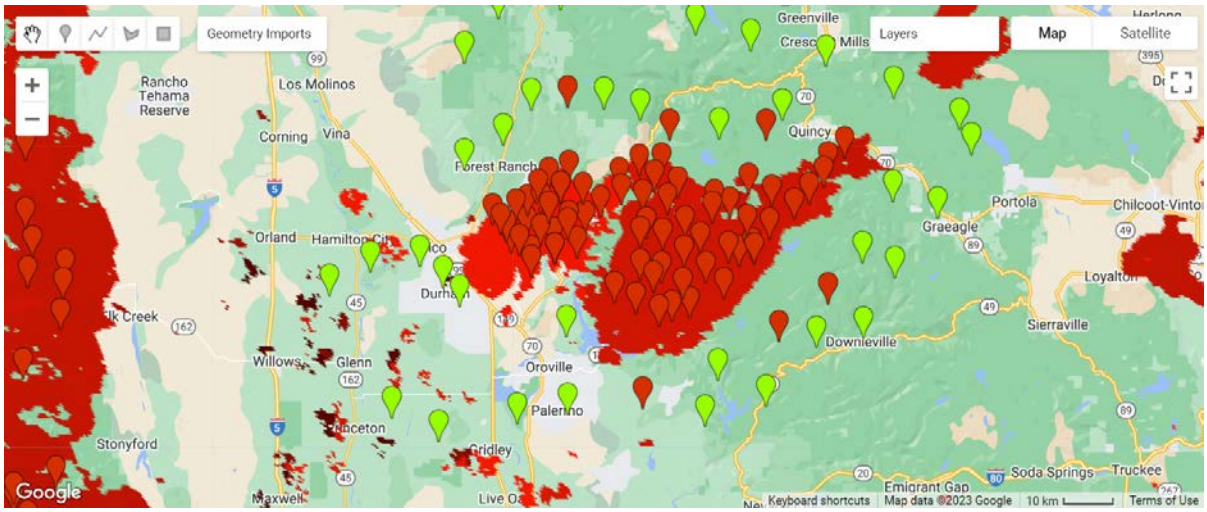

図 5. カリフォルニア州パラダイス付近のモデルの予測
(火災発生地域をほぼ正確に予測している)

まとめ

この記事では、PyTorch* が提供する画像解析機能とインテルの最適化機能を使用し、ResNet18 モデルをトレーニングおよびテストして、山火事を正確に予測する方法を説明しました。約 60 平方マイル(約 155 平方キロメートル)の範囲をカバーする空中写真をモデルに取り込むことにより、火災の有無を正確に予測できます。モデルの精度は現在約 89% ですが、反復回数を増やし、画像のセットを増やし、正則化に重点を置くことで、精度が向上する可能性があります。

この記事について議論したい方は、招待リンク（英語）をクリックしてインテル® DevHub Discord に参加してください。また、新しいインテル® デベロッパー・クラウド（英語）で、最新のインテルのハードウェア上でインテル® AI アナリティクス・ツールキットを試してみてください。

CPU 上での XGBoost、LightGBM、CatBoost 推論の高速化

インテル® oneAPI データ・アナリティクス・ライブラリー（インテル® oneDAL）で XGBoost、LightGBM、CatBoost 推論ワークロードを高速化

Nikolay Petrov インテル コーポレーション AI ソフトウェア・エンジニアリング・マネージャー
Dmitry Razdoburdin インテル コーポレーション AI フレームワーク・エンジニア

最近は生成 AI の人気が高まっていますが、決定木で勾配ブースティングを使用することが表形式データを処理する最適な方法であることに変わりはありません。ほかの多くの技術やニューラル・ネットワークと比較した場合でも、より高い精度を提供します。XGBoost、LightGBM、および CatBoost 勾配ブースティングは、現実世界のさまざまな問題の解決、研究、Kaggle コンペティションで幅広く利用されています。これらのフレームワークは、デフォルトの設定でも優れたパフォーマンスを提供しますが、推論スピードには向上の余地があります。推論はマシンラーニング・ワークフローの最も重要なステージであることを考慮すると、パフォーマンス向上によってもたらされる利点は少なくありません。

次の例は、品質を損なうことなく、より高速な予測を得るためにモデルを変換する方法を示しています（**表 1**）。

```
import daal4py as d4p
d4p_model = d4p.mb.convert_model(xgb_model)
d4p_prediction = d4p_model.predict(test_data)
```

Dataset	n_estimators	daal.2023.2.0 vs xgb	daal.2023.2.0 vs lgbm	daal.2023.2.0 vs catboost
abalone	1000	3.28	4.93	7.94
airline-ohe	1000	8.38	13.76	2.50
higgs1m	100	3.85	13.66	5.06
higgs1m	300	3.67	12.97	3.37
higgs1m	1000	3.31	26.41	2.64
higgs1m	3000	3.40	26.14	2.38
higgs1m	10000	3.95	36.21	2.34
higgs1m	30000	3.85	44.28	2.21
mlsr	200	4.04	18.03	1.78
mortgage1Q	100	8.22	7.96	4.14
plasticc	60	2.63	6.21	3.71
santander	10000	2.58	11.58	1.75
airline	100	5.10	12.72	4.73
bosch	100	15.35	20.27	4.99
covtype	100	2.36	7.05	2.86
epsion	100	41.97	51.50	6.81
fraud	100	5.03	8.78	5.85
higgs	100	6.50	15.25	4.62
year_prediction_msd	100	12.27	14.79	6.60
geomean		**5.24**	**15.11**	**3.63**

表 1. daal4py の推論パフォーマンスと XGBoost、LightGBM、および CatBoost の比較。
色分けは各列の最良のケースと最悪のケースを示し、値が大きいほどパフォーマンスが優れていることを示します。

oneDAL のアップデート

推論の高速化に関する前回の記事 (43 号の「XGBoost と LightGBM 推論パフォーマンスの向上」) から数年が経過し、以下のような変更と改善が行われました。

1. API の簡素化および勾配ブースティング・フレームワークとの連携
2. CatBoost モデルのサポート
3. 欠損値のサポート
4. パフォーマンスの向上

API の簡素化および勾配ブースティング・フレームワークとの連携

メソッドが `convert_model()` と `predict()` のみになり、最小限の変更で既存のコードに簡単に統合できます。

一般的な XGBoost

```
from xgboost import xgb

xgb_model = xgb.fit(X_train, y_train)



xgb_prediction = xgb_model.predict(test_data)
```

daal4py モデルビルダー

```
from xgboost import xgb

xgb_model = xgb.fit(X_train, y_train)

from daal4py import d4p
d4p_model = d4p.mb.convert_model(xgb_model)

d4p_prediction = d4p_model.predict(test_data)
```

トレーニング済みモデルを daal4py に変換することもできます。

XGBoost :

```
d4p_model = d4p.mb.convert_model(xgb_model)
```

LightGBM :

```
d4p_model = d4p.mb.convert_model(lgb_model)
```

CatBoost :

```
d4p_model = d4p.mb.convert_model(cb_model)
```

scikit-learn* 形式の推定器もサポートされました。

```
from daal4py.sklearn.ensemble import GBTDAALRegressor
reg = xgb.XGBRegressor()
reg.fit(X, y)
d4p_predt = GBTDAALRegressor.convert_model(reg).predict(X)
```

アップデートされた API では、XGBoost、LightGBM、および CatBoost モデルをすべて 1 カ所で使用できます。メイン・フレームワークの場合と同様に、分類と予測の両方に同じ `predict()` メソッドを使用することもできます。詳細は、ドキュメント（英語）を参照してください。

CatBoost モデルのサポート

daal4py に CatBoost モデルのサポートを追加すると、勾配ブースティング・タスクの処理におけるライブラリーの汎用性と効率を大きく向上させました。CatBoost(Categorical Boosting の略)は、卓越したスピードとパフォーマンスで知られていますが、daal4py アクセラレーションを使用することで、さらに高速に処理できます（注 : カテゴリー特徴は推論ではサポートされていません)。CatBoost のサポートにより、daal4py は、最も一般的な 3 つの勾配ブースティング・フレームワークをサポートしました。

欠損値のサポート

欠損値は、現実世界のデータセットではよく発生します。欠損値は、ヒューマンエラー、データ収集の問題、観察メカニズムの制限など、さまざまな理由で発生する可能性があります。欠損値を無視したり不適切に処理すると、偏った解析が行われ、最終的にマシンラーニング・モデルのパフォーマンスが低下します。

欠損値のサポートは daal4py 2023.2 バージョンで追加されました。欠損値を含むデータでトレーニングされたモデルを使用したり、欠損値を含むデータを推論に使用することができます。データ・サイエンティストやエンジニアのデータ準備プロセスが合理化されるとともに、より正確で堅牢な予測が可能になります。

パフォーマンスの向上

前回のパフォーマンス比較以降、多くの最適化が oneDAL に追加されました（**表 2**）。

Datasets	n_estimators	**daal.2023.1.0** vs xgb	**daal.2023.2.0** vs xgb	**daal.2023.2.0** vs **daal.2023.1.0**
abalone	1000	2.09	3.28	1.57
airline-ohe	1000	8.44	8.38	0.99
higgs1m	100	3.33	3.85	1.16
higgs1m	300	3.35	3.67	1.10
letters	1000	0.90	1.87	2.07
mortgage1Q	100	7.25	8.22	1.13
airline	100	4.42	5.10	1.15
covtype	100	2.13	2.36	1.11
epsion	100	41.02	41.97	1.02
fraud	100	4.35	5.03	1.16
higgs	100	5.67	6.50	1.15
year_prediction_msd	100	10.73	12.27	1.14
geomean		**4.75**	**5.71**	**1.20**

表 2. 2023.1 および XGBoost と比較した 2023.2 バージョンのパフォーマンスの向上

すべてのテストは、AWS* EC2 c6i.12xlarge インスタンス（24 コアのインテル® Xeon® Platinum 8375C プロセッサーを含む）で動作している scikit-learn_bench（英語）と、Python* 3.11、XGBoost 1.7.4、LightGBM 3.3.5、CatBoost 1.2、daal4py 2023.2.1 を使用して実施されました。

新しいケースやその他の改善対象の拡大に、ぜひご協力ください。

daal4py の入手方法

daal4py は oneDAL（英語）ライブラリーの Python* インターフェイスで、PyPI*、conda-forge、および conda メインチャネルで利用できます。完全にオープンソースのプロジェクトであり、GitHub* リポジトリー（英語）経由での問題報告、リクエスト、貢献を受け付けています。PyPI* からライブラリーをインストールするには、次のコマンドを実行します。

```
pip install daal4py
```

conda-forge の場合は次のとおりです。

```
conda install -c conda-forge daal4py --override-channels
```

これらの結果は、コードに簡単な変更を加えることにより、現在のインテルのハードウェアで勾配ブースティング・データ解析タスクを大幅に高速化できることを示しています。これらのパフォーマンスの向上により、品質を犠牲にすることなく、コンピューティング・コストを低く抑え、より迅速に結果が得られるようになります。

AI でビジネスを拡大する

Red Hat* OpenShift* Data Science とインテル® アーキテクチャーを使用してオープンソースの AI テクノロジーを活用

Ted Jones Red Hat シニア・ソリューション・アーキテクト
Karl Eklund Red Hat 主席アーキテクト
Paulina Olszewska インテル コーポレーション クラウド・ソリューション・エンジニア
Piotr Grabuszynski インテル コーポレーション クラウド・ソリューション・エンジニア

AI は、未来のテクノロジーではなく、すでに現実のものとなっています。小売、銀行と金融、セキュリティー、ガバナンス、医療、産業は、AI が日常的に導入されている分野のほんの一部です。AI が生成したコンテンツと人間が生成したコンテンツの見分けがつかなくなる時代がそこまでやってきています。自然言語処理（NLP）は、機械と人間の間の対話から 2 人の人間の間の対話に近いレベルまで進化しています。この進化により、政府、起業家、金融および医療分野の企業に新たな機会が訪れています。

さまざまなマシンラーニング（ML）タスクが容易になるように、インテルと Red Hat は新しいソリューションを開発しています。データマイニング、モデルのデプロイ、トレーニング、プロセスの自動化、データ転送はすべて、Red Hat* OpenShift* Data Science などの製品により強化されます。パフォーマンス向上のため、インテルは、インテル® アドバンスト・マトリクス・エクステンション（インテル® AMX）を含む新しいテクノロジーとアクセラレーターを提供しています。

問題ステートメント

AI は、さまざまな方法で活用できるビジネス上の利点を企業にもたらします。NLP は、大規模なテキストデータを効率良く解析することで、不正検出に役立ちます。感情分析を実行して、医療をサポートすることもできます。NLP により、より自然な形式で人間と機械の対話が可能になります。この記事では、企業に関する質問に答える AI モデルの簡単な使用法を示します。インテリジェント・アシスタントを使用すると、応答時間が短縮され、顧客対応に必要な人員の数が減り、コストが削減され、顧客の利便性が向上します。

アプローチ

目標は、製品、価格、販売、企業に関する顧客の質問に答えることです。次の例は、オープンソースのコンポーネントを使用して、顧客と対話できるチャットボットをセットアップする簡単な方法を示しています。いくつかのプログラミングと開発を行うことで、ビジネスをサポートする、完全に機能する実用的なアプリケーションに拡張および変換できます。

この目標を達成するため、ここでは BERT（Bidirectional Encoder Representations from Transformers の略）とオープンソースの BERT 質問応答 Python* デモ（英語）を使用しています。次の例は、より迅速な応答を提供するため、Red Hat* OpenShift* Data Science プラットフォームと OpenVINO™ ツールキットの Operator を、インテル® AMX を搭載した第 4 世代インテル® Xeon® スケーラブル・プロセッサー上で実行しています。

テクノロジー

Red Hat* OpenShift* Data Science

Red Hat* OpenShift* Data Science（英語）は、インテリジェント・アプリケーションのデータ・サイエンティストやプログラマー向けのサービスであり、セルフマネージドまたはマネージド・クラウド・プラットフォームとして利用できます。ML モデルを実際の環境にデプロイする前に、迅速に開発、トレーニング、テストできる、完全にサポートされた環境を提供します。オンプレミス、パブリッククラウド、データセンター、エッジのいずれであっても、コンテナ上の本番環境で ML モデルをデプロイし、Red Hat* OpenShift* Data Science からほかのプラットフォームに簡単にエクスポートできます。

ML に Red Hat* OpenShift* Data Science を使用すると、多くの利点があります。このプラットフォームには、Jupyter Notebook*、TensorFlow*、PyTorch*、OpenVINO™ など、データ・サイエンティストがワークフローで使用するさまざまな商用パートナーとオープンソースのツールおよびフレームワークが含まれています。Red Hat* OpenShift* Data Science は、安全でスケーラブルな環境を提供します。

インテル® AMX

インテル® AMX は、CPU 上のディープラーニングのトレーニングと推論のパフォーマンスを向上させる新しい内蔵アクセラレーターです。AI の処理が高速化され、パフォーマンスが最大 3 倍向上します。自然言語処理、推奨システム、画像認識などのワークロードに最適です。第 4 世代インテル® Xeon® スケーラブル・プロセッサー上のパフォーマンスは、インテル® oneAPI ベース・ツールキットの一部であるインテル® 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）を使用して微調整でき、TensorFlow* と PyTorch* AI フレームワーク、およびインテル® ディストリビューションの OpenVINO™ ツールキットに統合できます。これらのツールキットは Red Hat* OpenShift* Data Science で使用できます。

実装例

この例では、Super Shoes! という商店を説明する簡単なウェブサイトをベースとして使用します（**図 1**）。小売業者は、独自のウェブサイトを使用したり、会社に関する重要な情報を含む文書を作成することができます。次の例は、質問に回答できる簡単なチャットボットを開発し、顧客の観点から応答時間を短縮する方法を示しています。

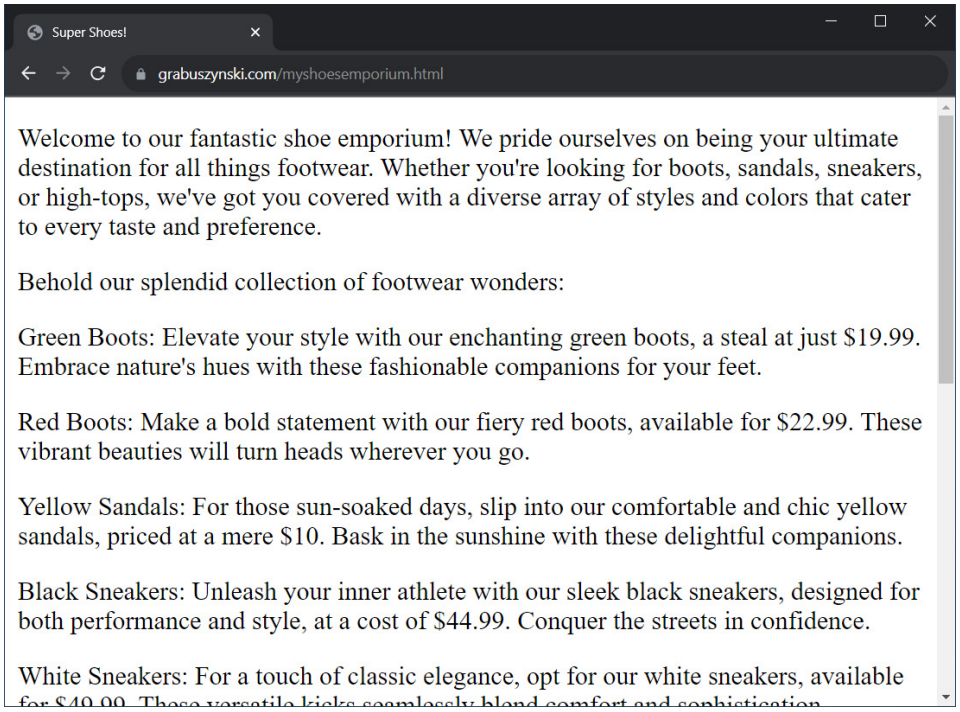

図 1. 例で使用したウェブサイトの一部

チャットボットの作成を開始するには、Red Hat* OpenShift* クラスターのイメージ・レジストリーを準備します（英語）。永続ボリューム（OpenShift* Data Foundation など）を提供するストレージと、インテル® AMX をサポートする CPU を検出する Node Feature Discovery Operator も必要です。カーネルは実行時にインテル® AMX を検出するため、インテル® AMX を個別に有効にして構成する必要はありません。ノードのラベルをチェックして、インテル® AMX が利用可能であることを確認します。

```
feature.node.kubernetes.io/cpu-cpuid.AMXBF16=true
feature.node.kubernetes.io/cpu-cpuid.AMXINT8=true
feature.node.kubernetes.io/cpu-cpuid.AMXTILE=true
```

Red Hat* OpenShift* Data Science と OpenVINO™ ツールキットの Operator は、OpenShift* Container Platform ウェブコンソールから利用できる OperatorHub からインストールする必要があります（**図 2**）。デフォルト設定で十分ですが、正しいインストールの順序（最初に OpenShift* Data Science、次に OpenVINO™ ツールキットの Operator）を守ってください。次に、redhat-ods-applications プロジェクトの [Installed Operators] タブに両方のオペレーターがリストされていること、ステータスが Succeeded になっていることを確認します。

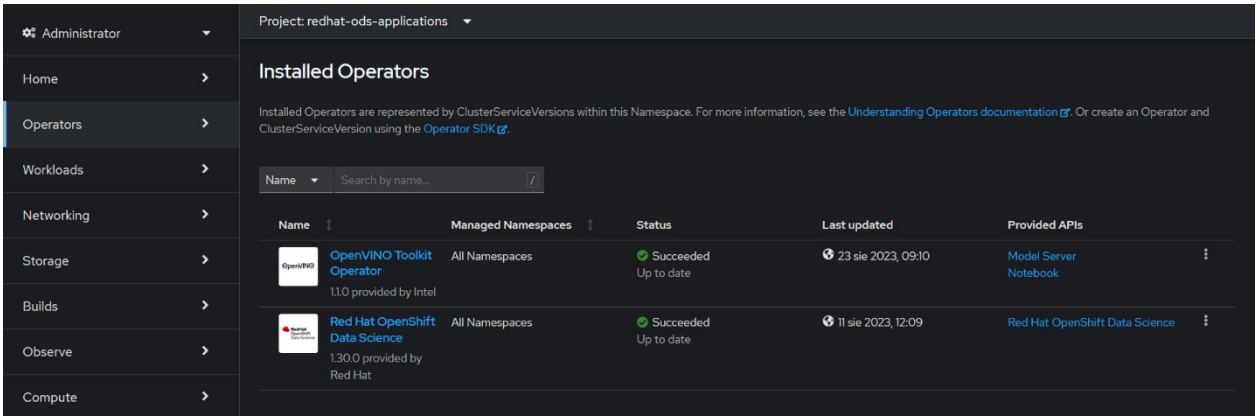

図 2. インストールされたオペレーターのステータスを示す OpenShift* コンソールのスクリーンショット

OpenVINO™ ツールキットの Operator には追加の構成が必要です。redhat-ods-applications プロジェクトが選択されていることを確認します。OpenVINO™ ツールキットの Operator をクリックし、**[Notebook]** タブを選択します。デフォルト設定で新しいノートブックを作成します。これらの手順が完了したら、メインメニューから **[Builds] > [Builds]** に移動して、openvino-notebooks-v2022.3-1 ビルドが完了していることを確認します。

右上のメニューを使用して Red Hat* OpenShift* Data Science ダッシュボードに移動します（**図 3**）。

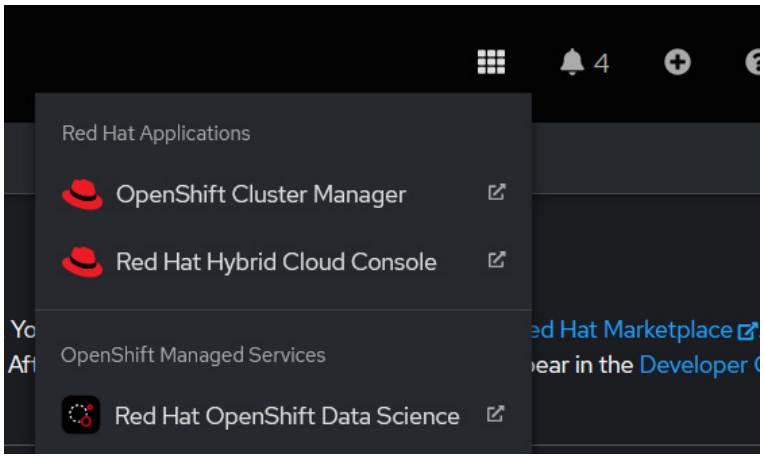

図 3. OpenShift* Managed Services のメニュー

Red Hat* OpenShift* Data Science インターフェイスで、**[Applications] > [Enabled]** タブを選択します。OpenVINO™ ツールキット v2022.3 イメージを使用して Jupyter Notebook* アプリケーションを起動します。コンテナのサイズを **[Large]** に変更します（**図 4**）。

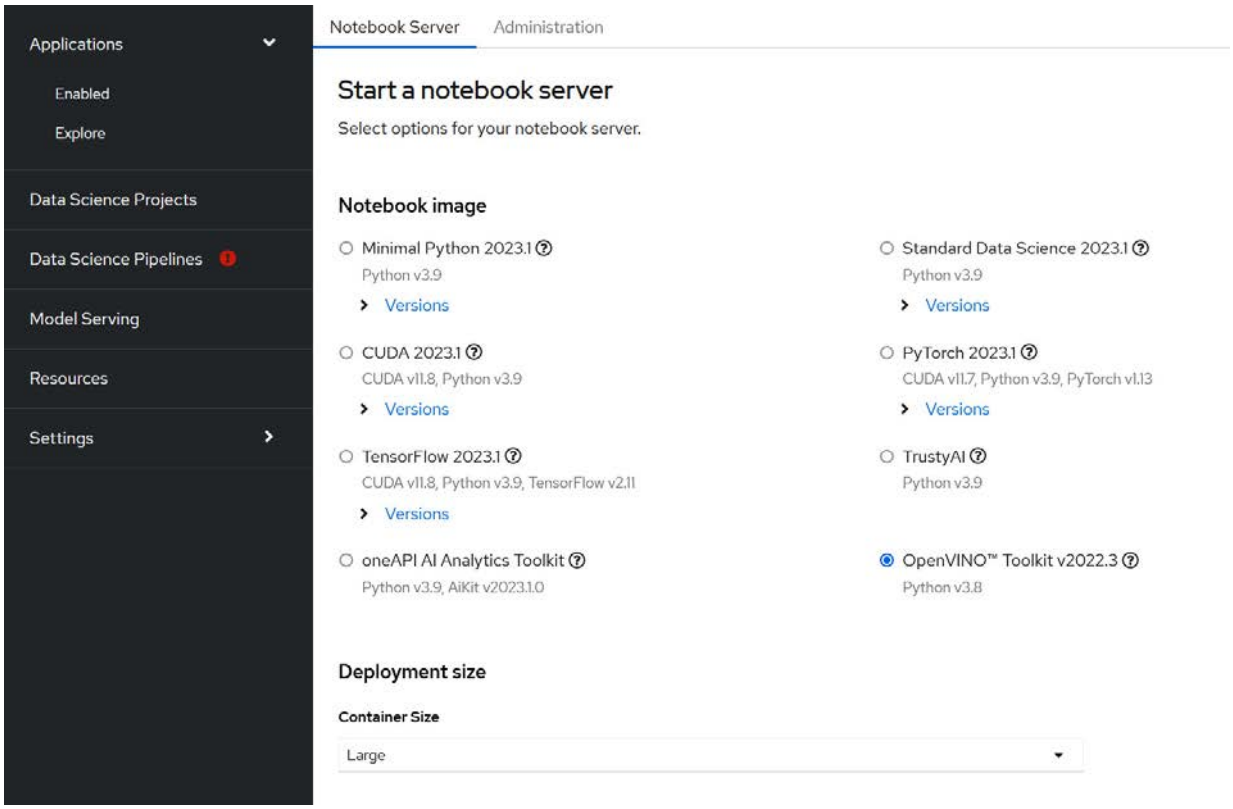

図 4. Red Hat* OpenShift* Data Science ダッシュボードで Jupyter Notebook* を起動

ランチャーウィンドウで **[Terminal]** を選択します（**図 5**）。

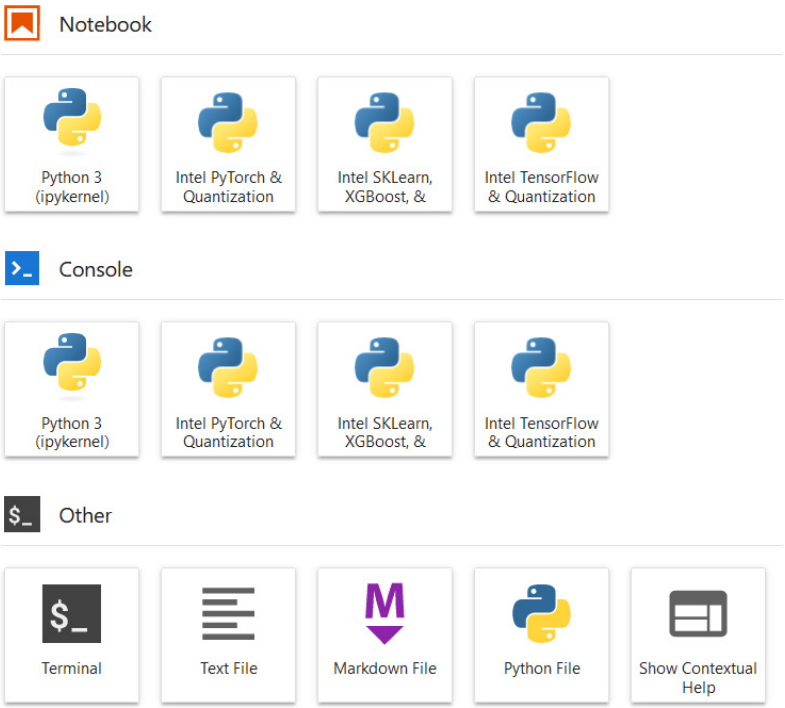

図 5. ランチャーウィンドウ

demos（英語）のソースコードを含む Open Model Zoo GitHub* リポジトリーをクローンします。

```
git clone --recurse-submodules
https://github.com/openvinotoolkit/open _ model _ zoo.git
```

コンパイルされたすべての命令を表示し、インテル® AMX の使用を制御するには、`ONEDNN_VERBOSE` 環境変数を 1 に設定します。

```
export ONEDNN_VERBOSE=1
```

前述したように、この例は BERT 質問応答 Python* デモをベースとしています。ただし、BERT-small の代わりに BERT-large を使用します。

`open_model_zoo/demos/bert_question_answering_demo/python` ディレクトリーには、デモでサポートされているモデルのリストを含む `models.lst` ファイルがあります。モデルをダウンロードするには、次のコマンドでこのファイルを指定します。

```
omz_downloader --list models.lst
```

Red Hat* OpenShift* Data Science を使用する最も重要な利点の 1 つは、必要なツール（上記で使用したモデル・ダウンローダー（英語）など）と Python* パッケージがすべて組込まれていることです。OpenVINO™ ツールキットの Operator のおかげで、コンポーネントのインストールおよび開発環境の準備という、長く退屈でエラーが発生しやすいプロセスをスキップできます。

モデルがダウンロードできたら、最初のチャットボットを実行する準備は完了です。スクリプトを呼び出してみましょう。

```
python3 bert_question_answering_demo.py --vocab=/opt/app-root/src/open_model_zoo/
demos/bert_question_answering_demo/python/intel/bert-large-uncased-whole-word-
masking-squad-int8-0001/vocab.txt --model=/opt/app-root/src/open_model_zoo/
demos/bert_question_answering_demo/python/intel/bert-large-uncased-whole-word-
masking-squad-int8-0001/FP32-INT8/bert-large-uncased-whole-word-masking-squad-
int8-0001.xml --input_names="input_ids,attention_mask,token_type_ids" --output_
names="output_s,output_e" --input="https://grabuszynski.com/myshoesemporium.html" -c
```

サンプルページを入力として渡す代わりに、小売業者は自社のウェブサイトのサンプルを使用できます。現在、チャットボットは会社とその製品に関する質問に答えることができます（**図 6**）。

`ONEDNN_VERBOSE` を 1 に設定した場合、インテル® AMX 命令が使用されていることを確認する `avx_512_core_amx` がログに表示されます。

```
        Type a question (empty string to exit): What is the price of green boots?
[ INFO ]                Show top 3 answers
[ INFO ] Answer: $19.99
         Score: 0.98
         Context: Green Boots: Elevate your style with our enchanting green boots, a steal at just $19.99
[ INFO ] Answer: $49.99
         Score: 0.03
         Context: Hiking Boots Sale: Calling all adventure enthusiasts! Our robust hiking boots are now available at
a special price of $49.99

        Type a question (empty string to exit): What are the opening hours?
[ INFO ]                Show top 3 answers
[ INFO ] Answer: 8AM and remain wide open until 9PM
         Score: 0.64
         Context:  Our doors open at 8AM and remain wide open until 9PM
[ INFO ] Answer: Welcome to our fantastic shoe emporium!
         Score: 0.00
         Context:       Welcome to our fantastic shoe emporium! We pride ourselves on being your ultimate destination
 for all things footwear

        Type a question (empty string to exit): []
```

図 6. 質問の例

```
onednn_verbose,info,cpu,isa:Intel AVX-512 with float16, Intel DL Boost and bfloat16
support and Intel AMX with bfloat16 and 8-bit integer support
```

この例を拡張することもできます。モデルはモデルサーバーを使用して提供できます。使いやすくユーザーフレンドリーなインターフェイスを提供するアプリケーションは、このプロジェクトの特長となるでしょう。

まとめ

Red Hat とインテルは、AI の新しいソリューション、テクノロジー、改善に常に取り組んでいます。この記事では、Red Hat* OpenShift* Data Science と OpenVINO™ ツールキットおよびインテル® AMX アクセラレーションを組み合わせることで、小売業界をどのように変えることができるかを説明しました。これらの最先端のテクノロジーをチャットボット・ソリューションで BERT モデルとともに使用することで、小売業者は次のことが可能になります。

- カスタマーサービスのコストを削減する
- 応答時間の短縮により顧客満足度を向上させる
- 潜在的に売上を伸ばす

Red Hat* OpenShift* Data Science とインテル® AMX アクセラレーションの強力な組み合わせにより、小売業者は大量のデータを迅速に処理して解析することができます。その結果として、より適切な意思決定と最適化された運用が可能になります。最終的に、この技術の融合により、小売業者は急速に変化する市場環境で競争力を維持するために必要なリソースを得ることができます。

intel software

コードの境界を乗り越える

oneAPI 向けインテル® デベロッパー・クラウドで 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 SYCL* 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）>

Microsoft* Azure* 上にセキュアな Kubeflow* パイプラインを構築

仮想マシン上でインテル® ソフトウェア・ガード・エクステンションズを活用してセキュアで高速化されたマシンラーニング・パイプラインをデプロイ

Kelli Belcher インテル コーポレーション AI ソフトウェア・ソリューション・エンジニア

多くのマシンラーニング・アプリケーションは、基礎となるコードとデータの機密性と整合性を確保する必要があります。これまで保存中 (ストレージ内) または転送中 (ネットワーク経由での転送) のデータの暗号化は注目されていましたが、最近まで使用中（メインメモリー内）のデータのセキュリティーは注目されていませんでした。インテル® ソフトウェア・ガード・エクステンションズ（インテル® SGX）（英語）は、アプリケーション・コードとデータを安全に処理および保存できる一連の手順を提供します。インテル® SGX は、CPU 内にトラステッド・エグゼキューション (信頼できる実行) 環境を作成することにより、コンテナからのユーザーレベルのコードが、エンクレーブと呼ばれるプライベートなメモリー領域を割り当てて、アプリケーション・コードを実行できるようにします。

Microsoft* Azure* Confidential Computing プラットフォームを使用すると、インテル® SGX が提供するセキュリティーと機密性を活用して、Windows* と Linux* の両方の仮想マシン（VM）をデプロイできます。このチュートリアルでは、Azure* Kubernetes* Services（AKS）クラスター上にインテル® SGX のノードをセットアップする方法を示します。次に、スケーラブルなマシンラーニング・パイプラインの構築とデプロイに使用できる Kubernetes* 向けのマシンラーニング・ツールキット、Kubeflow* をインストールします。最後に、XGBoost 向けインテル® オプティマイゼーション（英語）、インテル® oneAPI データ・アナリティクス・ライブラリー（インテル® oneDAL）、および scikit-learn 向けインテル® エクステンション（英語）を使用してモデルのトレーニングと推論を高速化する方法を説明します。

この最適化モジュールは、Amazon Web Services* (AWS*)、Microsoft* Azure*、Google Cloud Platform* (GCP*) などの主要なクラウド・プロバイダー上でインテルにより最適化された AI ソリューションの構築とデプロイを容易にするために設計された、クラウドネイティブのオープンソース・リファレンス・アーキテクチャーのセットである Microsoft* Azure* 向けインテル® クラウド・オプティマイゼーション・モジュール (英語) の一部です。各モジュール (リファレンス・アーキテクチャー) には、必要な手順とソースコードがすべて含まれています。

この最適化モジュールのクラウド・ソリューション・アーキテクチャーでは、インテル® SGX VM を中核コンポーネントとする AKS クラスターを使用します。Kubeflow* パイプラインがコンテナ化された Python* コンポーネントを構築できるように、Azure* Container Registry を AKS クラスターにアタッチします。これらの Azure* リソースは、Azure* リソースグループで管理されます。次に、Kubeflow* ソフトウェア・レイヤーを AKS クラスターにインストールします。Kubeflow* パイプラインを実行すると、各パイプライン Pod がインテル® SGX ノードに割り当てられます。クラウド・ソリューションのアーキテクチャーを**図 1** に示します。

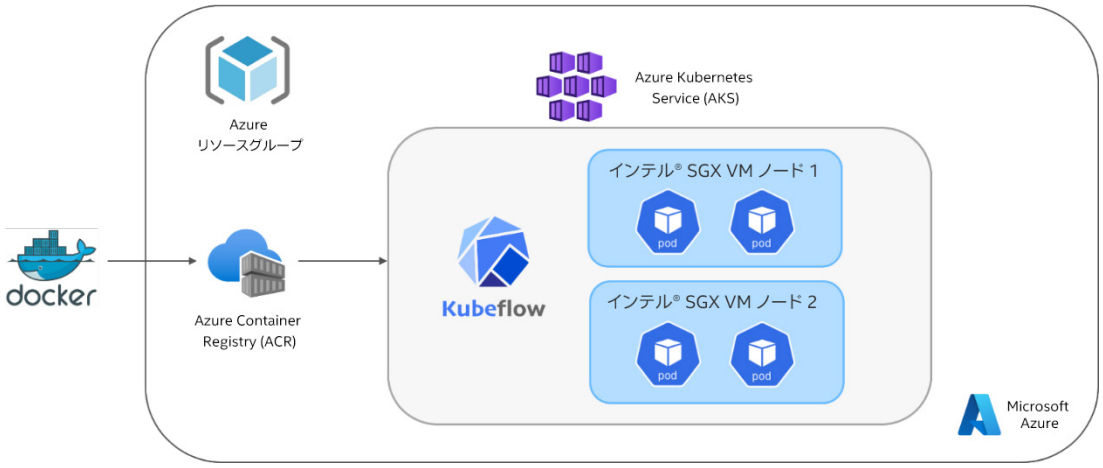

図 1. Kubeflow* パイプライン・アーキテクチャーの略図 (著者による画像)

準備

このチュートリアルを開始する前に、前提条件 (英語) をダウンロードしてインストールしていることを確認してください。次に、新しいターミナルウィンドウで次のコマンドを実行し、Azure* コマンドライン・インターフェイスを使用して Microsoft* Azure* アカウントにログインします。

```
az login
```

次に、ソリューションの Azure* リソースを保持するリソースグループを作成します。リソースグループの名前を `intel-aks-kubeflow` にして、場所を `eastus` に設定します。

```
# Set the names of the Resource Group and Location
export RG=intel-aks-kubeflow
export LOC=eastus

# Create the Azure Resource Group
az group create -n $RG -l $LOC
```

Confidential Computing ノードを使用して AKS クラスターをセットアップするには、最初にシステム・ノード・プールを作成し、Confidential Computing アドオンを有効にします。Confidential Computing アドオンは、対象の各 VM ノードがインテル® SGX 向けの Azure* デバイスプラグイン Pod のコピーを実行するようにクラスターの DaemonSet を設定します。次のコマンドは、Dv5 シリーズ（第 3 世代インテル® Xeon® プロセッサー）の標準 VM を使用してノードプールをプロビジョニングします。これは、`CoreDNS` や `metrics-server` などの AKS システム Pod をホストするノードです。次のコマンドはさらに、クラスターのマネージド ID を有効にして、標準の Azure* Load Balancer をプロビジョニングします。

```
# Set the name of the AKS cluster
export AKS=aks-intel-sgx-kubeflow

# Create the AKS system node pool
az aks create --name $AKS \
--resource-group $RG \
--node-count 1 \
--node-vm-size Standard_D4_v5 \
--enable-addons confcom \
--enable-managed-identity \
--generate-ssh-keys -l $LOC \
--load-balancer-sku standard
```

すでに Azure* Container Registry をセットアップしている場合は、パラメーター `--attach-acr <registry-name>` を追加することでクラスターにアタッチできます。

システム・ノード・プールがデプロイされたら、インテル® SGX ノードプールをクラスターに追加します。次のコマンドは、Azure* DCSv3 シリーズから 2 つの 4 コアのインテル® SGX ノードをプロビジョニングします。キー `intelvm` と値 `sgx` を使用して、このノードプールにノードラベルを追加しています。このキー / 値のペアは、Kubernetes* `nodeSelector` で参照され、Kubeflow* パイプライン Pod をインテル® SGX ノードに割り当てます。

```
az aks nodepool add --name intelsgx \
--resource-group $RG \
--cluster-name $AKS \
--node-vm-size Standard_DC4s_v3 \
--node-count 2 \
--labels intelvm=sgx
```

Confidential ノードプールが設定されたら、次のコマンドを使用してクラスターのアクセス認証情報を取得し、ローカルの `.kube/config` ファイルにマージします。

```
az aks get-credentials -n $AKS -g $RG
```

次のコマンドでクラスターの認証情報が正しく設定されているかどうかを確認できます。AKS クラスターの名前が返されるはずです。

```
kubectl config current-context
```

インテル® SGX VM ノードが正常に作成されたことを確認するには、次のコマンドを実行します。

```
kubectl get nodes
```

`aks-intelsgx` という名前で始まる 2 つのエージェント・ノードが実行されていることが分かるはずです。DaemonSet が正常に作成されたことを確認するには、次のコマンドを実行します。

```
kubectl get pods -A
```

`kube-system` 名前空間で、`sgxplugin` という名前で始まる 2 つの Pod が実行されていることが分かるはずです。上記の Pod とノードプールが実行中であることを確認できたら、AKS クラスターで機密アプリケーションを実行する準備は完了です。

Kubeflow* のインストール

AKS クラスターに Kubeflow* をインストールするには、まず、Kubeflow* Manifests GitHub* リポジトリーをクローンします。

```
git clone https://github.com/kubeflow/manifests.git
```

カレント・ディレクトリーを、新しくクローンしたマニフェストのディレクトリーに変更します。

```
cd manifests
```

オプションのステップとして、次のコマンドを使用して、Kubeflow* ダッシュボードにアクセスするときに使用する（デフォルトの）パスワードを変更できます。

```
python3 -c 'from passlib.hash import bcrypt; import getpass;
print(bcrypt.using(rounds=12, ident="2y").hash(getpass.getpass()))'
```

`dex` ディレクトリーの `config-map.yaml` に移動して、新しいパスワードを設定ファイルの 22 行目付近のハッシュ値に設定します。

```
nano common/dex/base/config-map.yaml
```

```
    staticPasswords:
    - email: user@example.com
      hash:
```

次に、Istio* Ingress Gateway を `ClusterIP` から `LoadBalancer` に変更します。ブラウザーからダッシュボードにアクセスするために使用できる外部 IP アドレスが構成されます。`common/istio-1-16/istio-install/base/patches/service.yaml` に移動して、7 行目付近の `spec` の `type` を `LoadBalancer` に変更します。

```
apiVersion: v1
kind: Service
metadata:
  name: istio-ingressgateway
  namespace: istio-system
spec:
  type: LoadBalancer
```

AKS クラスターの場合は、Istio Webhook から AKS アドミッション・エンフォーサーを無効にする必要もあります。Istio の `install.yaml` に移動して、2694 行目付近に次のアノテーションを追加します。

```
nano common/istio-1-16/istio-install/base/install.yaml
```

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: istio-sidecar-injector
  annotations:
    admissions.enforcer/disabled: 'true'
  labels:
```

次に、Istio Gateway を更新して Transport Layer Security(TLS)プロトコルを構成します。HTTPS 経由でダッシュボードにアクセスできるようになります。`kf-istio-resources.yaml` に移動して、ファイルの最後、14 行目付近に次の内容を追加します。

```
nano common/istio-1-16/kubeflow-istio-resources/base/kf-istio-resources.yaml
```

```
    tls:
      httpsRedirect: true
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - "*"
    tls:
      mode: SIMPLE
      privateKey: /etc/istio/ingressgateway-certs/tls.key
      serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
```

これで、Kubeflow* をインストールする準備ができました。ここでは、`kustomize` を使用して、単一コマンドでコンポーネントをインストールします。コンポーネントを個別にインストール（英語）することもできます。

```
while ! kustomize build example | awk '!/well-defined/' | kubectl apply -f -; do echo
"Retrying to apply resources"; sleep 10; done
```

すべてのコンポーネントのインストールが完了するまでしばらくかかります。一部のコンポーネントは最初の試行で失敗する可能性があります。これは、Kubernetes* と kubectl 固有の動作です（例えば、CRD の準備ができた後に CR を作成する必要があります）。解決策は、コマンドが成功するまで再実行することです。コンポーネントのインストールが完了したら、次のコマンドを実行してすべての Pod が実行されていることを確認します。

```
kubectl get pods -A
```

オプション : Kubeflow* の新しいパスワードを作成した場合は、`dex` Pod を再起動して、新しいパスワードを使用していることを確認します。

```
kubectl rollout restart deployment dex -n auth
```

最後に、Istio ロードバランサーの外部 IP アドレスを使用して、TLS プロトコルの自己署名証明書を作成します。外部 IP アドレスを取得するには、次のコマンドを実行します。

```
kubectl get svc -n istio-system
```

Istio 証明書を作成して、次の内容をコピーします。

```
nano certificate.yaml
```

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: istio-ingressgateway-certs
  namespace: istio-system
spec:
  secretName: istio-ingressgateway-certs
  ipAddresses:
    - <Istio IP address>
  isCA: true
  issuerRef:
    name: kubeflow-self-signing-issuer
    kind: ClusterIssuer
    group: cert-manager.io
```

次に、証明書を適用します。

```
kubectl apply -f certificate.yaml
```

証明書が正常に作成されたことを確認します。

```
kubectl get certificate -n istio-system
```

これで、Kubeflow* ダッシュボードを起動する準備ができました。ダッシュボードにログインするには、Istio の IP アドレスをブラウザーに入力します。ダッシュボードに初めてアクセスした場合、自己署名証明書を使用しているために接続に関する警告が表示されることがあります。SSL CA 証明書がある場合は、その証明書に置き換えることもできます。自己署名証明書を使用する場合は、画面の表示に従います。dex のログイン画面が表示されます。ユーザー名とパスワードを入力します。Kubeflow* のデフォルトのユーザー名は `user@example.com`、デフォルトのパスワードは `12341234` です。

Kubeflow* ダッシュボードにログインしたら、Kubeflow* パイプラインをデプロイする準備は完了です。このモジュールの Kubeflow* パイプラインは、インテルと Accenture が協力して開発したローン不履行リスク予測 AI リファレンス・キット（英語）から派生したものです。コードはリファクタリングにより強化され、モジュール性と Kubeflow* パイプラインへの適合性が向上しました。このパイプラインは、借り手のローン不履行のリスクを予測する XGBoost モデルを構築し、XGBoost 向けインテル® オプティマイゼーション（英語）とインテル® oneDAL を使用してモデルのトレーニングと推論を高速化します。パイプライン全体のグラフを**図 2** に示します。

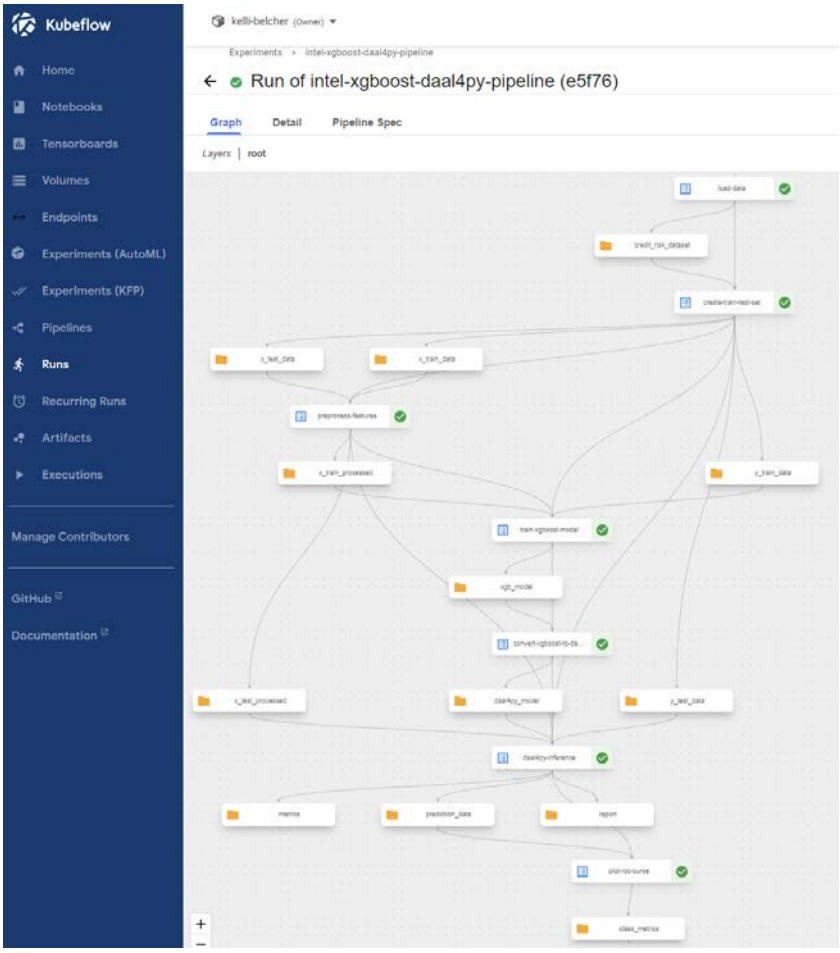

図 2. XGBoost Kubeflow* パイプラインのグラフ（著者による画像）

Kubeflow* パイプラインは、次の 7 つのコンポーネントで構成されます。

- **データのロード：**パイプライン実行パラメーターで指定された URL からデータセット（`credit_risk_dataset.csv`）をロードし、合成データの拡張を実行します。Kaggle の Credit Risk データセットをダウンロード(英語)します。
- **トレーニング・セットとテストセットの作成：**モデル評価のために、データを約 75:25 の比率でトレーニング・セットとテストセットに分割します。
- **特徴の前処理：**one-hot エンコーディングを使用してトレーニング・セットとテストセットのカテゴリー特徴を変換し、欠損値のインピュテーションを行い、数値特徴をべき乗変換します。
- **XGBoost モデルのトレーニング：**このコンポーネントは、XGBoost 向けインテル® オプティマイゼーションで提供されるアクセラレーションを活用してモデルをトレーニングします。
- **XGBoost モデルの daal4py への変換：**XGBoost モデルを推論に最適化された daal4py 分類器に変換します。
- **daal4py 推論：**推論に最適化された daal4py 分類器を使用して予測を計算し、モデルのパフォーマンスを評価します。各クラスの適合率（精度）、再現率、F1 スコア、モデルの曲線下面積（AUC）および精度スコアの出力概要が返されます。
- **受信者動作特性（ROC）曲線のプロット：**テストデータに対してモデル検証を実行し、ROC 曲線のプロットを生成します。

パイプラインのコードは、GitHub* リポジトリーの src（英語）フォルダー内にある `intel-xgboost-daal4py-pipeline-azure.py` です。Python* スクリプトを実行する前に、Kubeflow* パイプライン SDK のバージョン 2.0 以降がインストールされていることを確認します。SDK のバージョンを更新するには、次のコマンドを使用します。

```
pip install -U kfp
```

各パイプライン Pod がインテル® SGX ノードに確実に割り当てられるように、Kubernetes* の `nodeSelector` を使用します。`nodeSelector` は、Pod をスケジュールするときに、ラベルのキーと値のペアが一致するノードを検索します。次のコードは、インテル® SGX ノードプールに追加したノードラベルを使用して、データ前処理パイプライン・タスク、`preprocess_features_op` をインテル® SGX ノードに割り当てます。

```
from kfp import kubernetes

kubernetes.add_node_selector(task = preprocess_features_op,
      label_key = 'intelvm', label_value = 'sgx')
```

パイプラインのコンポーネント

パイプラインの最初のコンポーネント、`load_data` は、パイプライン実行パラメーターで指定された URL から Credit Risk データセットをダウンロードし、指定されたサイズまでデータを合成的に拡張します。新しいデータセットは、Kubeflow* MinIO ボリュームに出力アーティファクトとして保存されます。このデータセットは、コンポーネントの次のステップ、`create_train_test_set` に読み込まれます。このコンポーネントは、モデル評価のためにデータを約 75:25 に分割します。トレーニング･セットとテストセット、`X_train`、`y_train`、`X_test`、および `y_test` は、出力データセット・アーティファクトとして保存されます。

preprocess_features コンポーネントでは、XGBoost モデル向けのデータを準備するため、データ前処理パイプラインを作成します。このコンポーネントは、MinIO ストレージから X_train と X_test ファイルをロードし、one-hot エンコーディングを使用してカテゴリー特徴を変換し、欠損値のインピュテーションを行い、数値特徴をべき乗変換します。

次のコンポーネントは、XGBoost モデルをトレーニングします。インテルの CPU で XGBoost を使用すると、コードを変更することなく、oneAPI のソフトウェア・アクセラレーションを活用できます。サイズ 100 万の増分トレーニング更新の初期テストで、XGBoost 向けインテル® オプティマイゼーション v1.4.2 は、XGBoost v0.81 と比較して最大 1.54 倍のスピードアップを達成しました（ローン不履行リスク予測リファレンス・キットのパフォーマンス結果（英語）は、GitHub* で確認できます）。次のコードが train_xgboost_model コンポーネントに実装されています。

```
# Create the XGBoost DMatrix, which is optimized for memory efficiency and training speed
dtrain = xgb.DMatrix(X_train.values, y_train.values)

# Define model parameters
params = {"objective": "binary:logistic",
     "eval_metric": "logloss",
     "nthread": 4,   # num_cpu
     "tree_method": "hist",
     "learning_rate": 0.02,
     "max_depth": 10,
     "min_child_weight": 6,
     "n_jobs": 4,   # num_cpu
     "verbosity": 1}

# Train initial XGBoost model
clf = xgb.train(params = params, dtrain = dtrain, num_boost_round = 500)
```

モデルの予測速度をさらに最適化するため、トレーニング済みの XGBoost モデルを、convert_xgboost_to_daal4py コンポーネントで推論に最適化された daal4py 分類器に変換します。daal4py は oneDAL の Python* API です。サイズ 100 万のバッチ推論のテストで、oneDAL は最大 4.44 倍のスピードアップを達成しました。XGBoost モデルの daal4py への変換は、わずか 1 行のコードで済みます。

```
# Convert XGBoost model to daal4py
daal_model = d4p.get_gbt_model_from_xgboost(clf)
```

次に、daal4py_inference コンポーネントで、daal4py モデルを使用してテストセットでのモデルのパフォーマンスを評価します。

```
# Compute both class labels and probabilities
daal_prediction = d4p.gbt_classification_prediction(
     nClasses = 2,
     resultsToEvaluate = "computeClassLabels|computeClassProbabilities"
).compute(X_test, daal_model)
```

`gbt_classification_prediction` メソッドを呼び出して、バイナリークラスのラベルと確率の両方を計算します（上記のコードを参照）。この関数を使用して対数確率を計算することもできます。このコンポーネントは、CSV 形式の分類レポートと、2 つの Kubeflow* メトリクス・アーティファクト（曲線下面積とモデルの精度）を返します。これらのアーティファクトは、`metrics` アーティファクトの **[Visualization]** タブで確認できます。

パイプラインの最後のコンポーネント、`plot_roc_curve` は、確率と `true` クラスラベルを含む予測データをロードし、scikit-learn 向けインテル® エクステンション（英語）の CPU 高速化バージョンを使用して ROC 曲線を計算します。パイプラインの実行が終了すると、結果が Kubeflow* ClassificationMetric アーティファクトとして保存されます。このアーティファクトは、`roc_curve_daal4py` アーティファクトの **[Visualization]** タブで確認できます（**図 3**）。

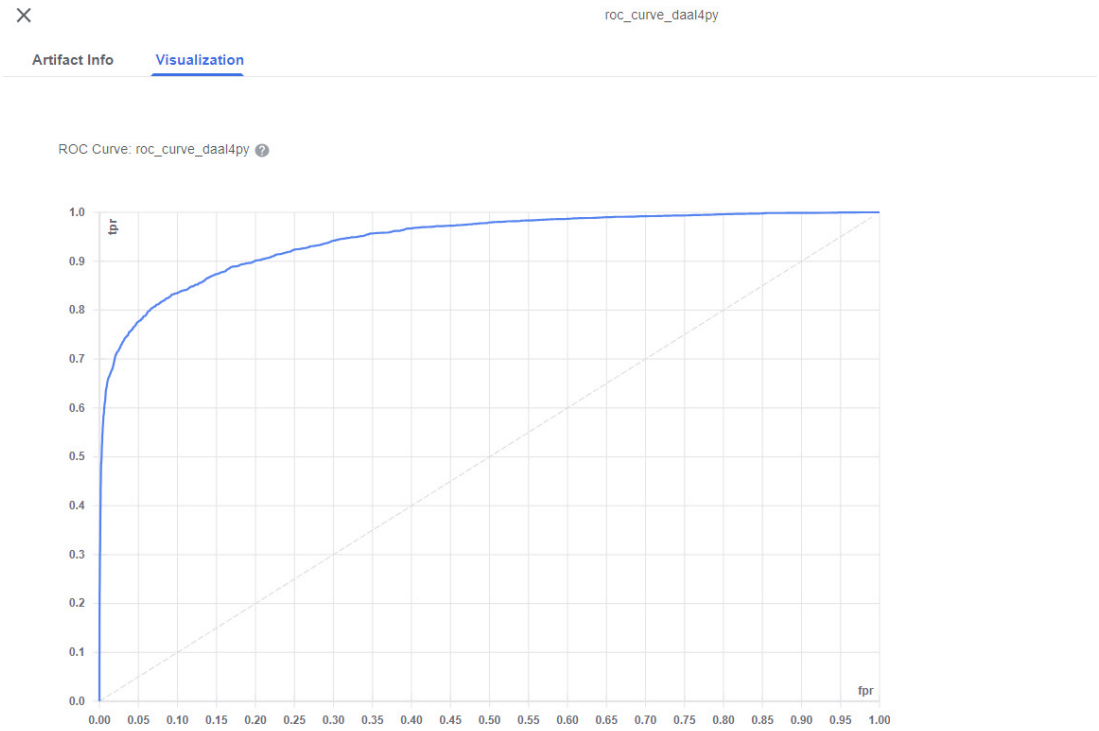

図 3. 分類器の ROC 曲線

次のステップ

完全なソースコード（英語）は、GitHub* にあります。実装のサポートが必要な場合は、こちらのサイト（英語）からお問い合わせください。インテル® クラウド・オプティマイゼーション・モジュール（英語）で詳細を確認したり、インテル® DevHub Discord（英語）サーバーでほかの開発者とチャットで交流することができます。

OpenMP* ターゲット オフロードの事例

ISO Fortran がヘテロジニアス・コンピューティングに十分でない理由

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
Ron Green インテル コーポレーション コンパイラー・エンジニアリング・マネージャー
Nawal Copty インテル コーポレーション シニア・ソフトウェア・エンジニア

以前の記事では、Fortran プログラムからアクセラレーターへの計算のオフロードについて紹介しました。

- Fortran、oneMKL、OpenMP* を使用して LU 因数分解を高速化
- Fortran と OpenMP* でヘテロジニアス・プログラミングの課題を解決
- oneMKL と OpenMP* ターゲットオフロードで線形システムを解く
- Fortran の DO CONCURRENT を使用したアクセラレーター・オフロード

この記事では、アクセラレーターへのオフロードに関して、Fortran `DO CONCURRENT` 文と OpenMP* `target` 構文の長所と短所を説明します。

`DO CONCURRENT` 構文は ISO Fortran 2008 で追加された機能で、`DO CONCURRENT` ループの反復が独立していて、任意の順序で実行できる（訳注：すなわち、並列に実行できる）ことをコンパイラーに通知またはアサートします。`DO CONCURRENT` ループはシーケンシャルに、あるいは並列に実行でき、OpenMP* バックエンドを使用して `DO CONCURRENT` ループをアクセラレーターにオフロードすることもできます。アクセラレーター・オフロード機能は 2013 年の OpenMP* バージョン 4.0 で追加されました。OpenMP* `target` ディレクティブを使用すると、プログラマーは、アクセラレーターで実行するコードの領域や、ホスト・プロセッサーとアクセラレーター間で転送するデータを指定できます。

アクセラレーター・オフロードのどちらのアプローチも、標準準拠のコンパイラーを備えた任意のシステムに移植できます。`DO CONCURRENT` は、簡潔な ISO Fortran 構文であるという長所があります。ただし、ISO Fortran には ISO C++ と同じいくつかの制限があります（編集者注：ヘテロジニアス並列処理に関する ISO C++ の制限については、「SYCL* の事例」を参照してください）。デバイスや不連続メモリーの概念がないため、制御フローをアクセラレーターに転送したり、ホストとデバイス間のデータ転送を制御する標準的な方法はありません。OpenMP* は、これらの制限に対処します。OpenMP* は ISO 標準ではありませんが、25 年以上にわたり成熟してきたオープンな業界標準です。OpenMP* `target` ディレクティブは冗長であり、プログラマーがコンパイラーに並列処理であることを説明する必要がありますが、次のコード例から分かるように、ホストとデバイス間のデータ転送を細かく制御し、並列領域を集約して効率を向上させることができます。

前号の記事「Fortran の DO CONCURRENT を使用したアクセラレーター・オフロード」では、単純なフィルターをバイナリーイメージに適用したエッジ検出について説明しました。今回は、より現実的なエッジ検出アルゴリズムを比較に使用します。Fortran `DO CONCURRENT` と OpenMP* `target` ディレクティブを使用して Sobel アルゴリズムを実装します。このアルゴリズムは、オリジナルの画像の各ピクセルに水平フィルターと垂直フィルターを適用して、変換された画像内に急激なピクセル強度の変化を抽出します。

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

水平　　垂直

変換前と変換後の例を**図 1** に示します。各ピクセルの演算は独立しているため、アルゴリズムは高度にデータ並列です。

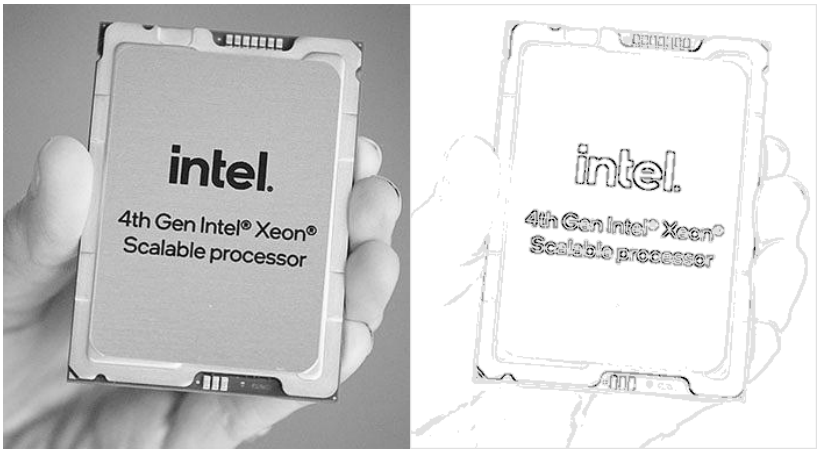

図 1. Sobel エッジ検出

Sobel アルゴリズムは通常、画像のスムージング、画像のエッジ検出、エッジのハイライトの、順に行う 3 つのステップで実装されます（編集者注：読者の方々は、以前の記事「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」などから、私が単純なアルゴリズムよりも同期とデータの依存関係を強調できるマルチステップのアルゴリズムを支持していることをご存知でしょう）。これらのステップは、3 つの Fortran `DO CONCURRENT` ループを使用して簡単にコーディングできます（**図 2**）。各ループでは、画像のサイズに応じて大幅なデータ並列処理が行われます。この例では 2D 画像を想定していますが、アルゴリズムはボリューム画像にも拡張できます。

`DO CONCURRENT` 構文は、**DO** 構文の異種形式にすぎません。`DO CONCURRENT` を初めて見た場合でも、ほとんどの Fortran プログラマーは、この例が二重に入れ子にされた **DO** ループのように画像の列と行を反復していることを理解できるでしょう。`DO CONCURRENT` ループには、一部の反復をマスクするプレディケートや変数のスコープを定義する追加の節を含めることができます（例えば、**図 2** の 2 つ目のループにはリダクション操作が含まれています）。

ピクセル値を含むデータ構造には赤、緑、青のチャネル（フィールド）がありますが、画像は通常、Sobel エッジ検出の前にグレースケールに変換されます。グレースケール画像ではチャネルが等価であるため、計算量が減ります。そのため、Sobel 実装の各ステップでは 1 つのチャネルのみ操作しています。計算量が減るだけではありません。次に示すように、ホストとデバイスのメモリー間で転送する必要があるデータの量も減ります。

```
gh     = reshape([-1,  0,  1, -2,  0,  2, -1,  0,  1], [3, 3])
gv     = reshape([-1, -2, -1,  0,  0,  0,  1,  2,  1], [3, 3])
smooth = reshape([ 1,  2,  1,  2,  4,  2,  1,  2,  1], [3, 3])

! 画像をスムージングしてノイズを削減
do concurrent (c = 2:img_width - 1, r = 2:img_height - 1)
    image_soa%red(r, c) = sum(image_soa%blue(r-1:r+1, c-1:c+1) * smooth) / 16
enddo

! Sobel エッジ検出を実行
do concurrent (c = 2:img_width - 1, r = 2:img_height - 1) reduce(max: max_gradient)
    image_soa%green(r, c) = abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gh)) + &
                            abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gv))
    max_gradient = max(max_gradient, image_soa%green(r, c))
enddo

! 勾配しきい値に基づいてエッジをハイライト
do concurrent (c = 1:img_width, r = 1:img_height)
    if (image_soa%green(r, c) >= 0.5 * max_gradient) then
        image_soa%green(r, c) = 0
    else
        image_soa%green(r, c) = 255
    endif
    image_soa%red(r, c)  = image_soa%green(r, c)
    image_soa%blue(r, c) = image_soa%green(r, c)
enddo
```

図 2. Fortran `DO CONCURRENT` ループ（青でハイライト表示）を使用して実装した Sobel エッジ検出。オフロードカーネルは緑でハイライト表示しています。完全なコード（`sobel_do_concurrent.F90`）は GitHub*（英語）から入手できます。

インテル® Fortran コンパイラーは、OpenMP* バックエンドを使用して `DO CONCURRENT` ループをアクセラレーターにオフロードできます。次のコマンドを使用して、「pvc」デバイス（インテル® データセンター GPU マックス）向けの事前コンパイルと、OpenMP* バックエンドによるアクセラレーター・オフロードのサンプルプログラムをビルドします。

```
$ ifx ppm_image_io.F90 sobel_do_concurrent.F90 -o sobel_dc_gpu -qopenmp \
>     -fopenmp-targets=spir64_gen -fopenmp-target-do-concurrent \
>     -Xopenmp-target-backend "-device pvc"
```

最初のソースファイル（ppm_image_io.F90（英語））は、画像 I/O を処理するユーティリティー・モジュールです。2 つ目のソースファイル（sobel_do_concurrent.F90（英語））には**図 2** のコードが含まれています。PPM 形式の 8K（7,680 x 8,404）解像度の画像（64,542,720 ピクセル x 4 バイト / ピクセル = 258,170,880 バイト）で実行ファイルを実行します。

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./sobel_dc_gpu -i xeon_4gen_8k.ppm -o xeon_8k_edges.ppm >& edge_detect_do_conc.out
$ grep Moving edge_detect_do_conc.out
Libomptarget --> Moving 258170880 bytes (hst:0x000014c199af22c0) -> (tgt:0xff00000005c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c189af1300) -> (tgt:0xff00000015400000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c179af0340) -> (tgt:0xff00000024c00000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000024c00000) -> (hst:0x000014c179af0340)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000015400000) -> (hst:0x000014c189af1300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000005c00000) -> (hst:0x000014c199af22c0)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c199af22c0) -> (tgt:0xff00000005c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c189af1300) -> (tgt:0xff00000015400000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c179af0340) -> (tgt:0xff00000024c00000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000024c00000) -> (hst:0x000014c179af0340)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000015400000) -> (hst:0x000014c189af1300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000005c00000) -> (hst:0x000014c199af22c0)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c199af22c0) -> (tgt:0xff00000036400000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c189af1300) -> (tgt:0xff00000045c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c179af0340) -> (tgt:0xff00000055400000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000055400000) -> (hst:0x000014c179af0340)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000045c00000) -> (hst:0x000014c189af1300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000036400000) -> (hst:0x000014c199af22c0)
```

ホスト（`hst`）とターゲット（`tgt`）デバイスのメモリー間で転送される赤、緑、青の画像チャネルを色分けしました。**図 2** では、3 つのセクションで 3 つの `DO CONCURRENT` ループが示されています（ターゲットデバイスにマップされる配列の Fortran 配列記述子、またはドープベクトルは示されていません。ドープベクトルは小さいため、このデータ移動は無視できます。同様に、3 x 3 フィルター行列も示されていません）。ホストとデバイス間のデータ転送は暗黙的に処理されます。プログラマーにとっては便利ですが、必ずしも効率が良いとは限りません。例えば、**図 2** の最初の `DO CONCURRENT` ループは、青のチャネルのみ読み取り、赤のチャネルのみ書き込みます。そのため、最初の `DO CONCURRENT` ループで必要なのは、1 つの `hst` → `tgt` 転送と 1 つの `tgt` → `hst` 転送のみです。しかし、実際には、3 つのチャネルがすべてデバイスに転送され、ホストに戻されています。

その結果、大量の不要なデータ移動が行われています。不連続メモリー間のデータ移動には時間と電力がかかるため、ホストとデバイス間のデータ転送を最小限に抑えることは、ヘテロジニアス並列処理のパフォーマンスにとって重要です。残念なことに、ISO Fortran 2018 および 2023 標準では、データ移動を制御したり、データが読み取り専用か書き込み専用かをランタイムに伝える言語構造が提供されていません。

幸いなことに、OpenMP* target オフロード API（英語）では、Sobel 実装の 3 つのステップを 1 つのターゲット・データ・マップ領域に集約する方法が提供されていて、必要な場合にのみデータが転送されます（**図 3**）。

```
!$omp target data map(tofrom: image_soa%blue(1:img_height, 1:img_width),  &
!$omp                          image_soa%green(1:img_height, 1:img_width), &
!$omp                          image_soa%red(1:img_height, 1:img_width))   &
!$omp             map(to: gh(1:3, 1:3), gv(1:3, 1:3), smooth(1:3, 1:3))

!$omp target teams distribute parallel do collapse(2)
do c = 2, img_width - 1
    do r = 2, img_height - 1
        ! 画像をスムージングしてノイズを削減
        image_soa%red(r, c) = sum(image_soa%blue(r-1:r+1, c-1:c+1) * smooth) / 16
    enddo
enddo
!$omp end target teams distribute parallel do

!$omp target teams distribute parallel do reduction(max: max_gradient) collapse(2)
do c = 2, img_width - 1
    do r = 2, img_height - 1
        ! Sobel エッジ検出を実行
        image_soa%green(r, c) = abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gh)) + &
                                abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gv))
        max_gradient = max(max_gradient, image_soa%green(r, c))
    enddo
enddo
!$omp end target teams distribute parallel do

!$omp target update from(max_gradient)

!$omp target teams distribute parallel do collapse(2)
do c = 1, img_width
    do r = 1, img_height
        ! 勾配しきい値に基づいてエッジをハイライト
        if (image_soa%green(r, c) >= 0.5 * max_gradient) then
            image_soa%green(r, c) = 0
        else
            image_soa%green(r, c) = 255
        endif
        image_soa%red(r, c)  = image_soa%green(r, c)
        image_soa%blue(r, c) = image_soa%green(r, c)
    enddo
enddo
!$omp end target teams distribute parallel do

!$omp end target data
```

図 3. OpenMP* target オフロード・ディレクティブ（青でハイライト表示）を使用して実装した Sobel エッジ検出。完全なコード（sobel_omp_target.F90）は GitHub*（英語）から入手できます。

図 3 の OpenMP* 実装を次のようにコンパイルして実行しました。

```
$ ifx ppm_image_io.F90 sobel_omp_target.F90 -o sobel_omp_gpu -qopenmp \
>     -fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device pvc"
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./sobel_omp_gpu -i xeon_4gen_8k.ppm -o xeon_8k_edges.ppm >& edge_detect_openmp.out
$ grep Moving edge_detect_openmp.out
Libomptarget --> Moving 258170880 bytes (hst:0x0000150220dfc340) -> (tgt:0xff00000005c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x0000150230dfd300) -> (tgt:0xff00000015400000)
Libomptarget --> Moving 258170880 bytes (hst:0x0000150240dfe2c0) -> (tgt:0xff00000024c00000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000024c00000) -> (hst:0x0000150240dfe2c0)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000015400000) -> (hst:0x0000150230dfd300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000005c00000) -> (hst:0x0000150220dfc340)
```

小さなドープベクトルとフィルター行列を削除して、ホスト（`hst`）とターゲット（`tgt`）デバイスのメモリー間で転送される赤、緑、青の画像チャネルを色分けしました。各画像チャネルは各方向に 1 回のみコピーされるようになり、`DO CONCURRENT` 実装よりもデータ転送が大幅に少なくなりました。ごのように、OpenMP* を使用すると、現在の ISO Fortran よりもデータ移動を細かく制御できます。

OpenMP* `target` オフロードは、大きな画像で Sobel エッジ検出を計算する場合にも、`DO CONCURRENT` よりも優れたパフォーマンスを提供します（**表 1**）。`DO CONCURRENT`（**図 2**）と OpenMP* `target`（**図 3**）の例は、ホスト CPU では同等のパフォーマンスが得られていますが（0.1 秒）、OpenMP* `target` で GPU にオフロードすると最高のパフォーマンスが得られています（0.05 秒）。一方、`DO CONCURRENT` の GPU でのパフォーマンスは、不要なデータ転送のため逆に低下しています（0.18 秒）。

	OpenMP	DO CONCURRENT
シーケンシャル	0.37	0.37
並列(CPU)	0.1	0.1
並列(GPU)	0.05	0.18

表 1. 8K（7,680 x 8,404）解像度の画像の OpenMP* `target` と Fortran `DO CONCURRENT` のパフォーマンスの比較（時間はすべて秒）。シーケンシャル・ベースラインは、OpenMP* を有効にしないでコンパイルされた OpenMP*（sobel_omp_target.F90）および `DO CONCURRENT`（sobel_do_concurrent.F90）の例です。CPU と GPU はそれぞれ、インテル® Xeon® Platinum 8480+ とインテル® データセンター GPU マックス 1100。

`DO CONCURRENT` は暗黙的な並列ループを表現するための便利な構文を提供しますが、ISO Fortran で不連続メモリーの概念、およびホストとデバイス間のデータ転送を制御する構文が利用できるようになるまでは、ヘテロジニアス並列処理には Fortran と OpenMP* を組み合わせることが最善と言えるでしょう。

ソースコードとテストイメージは、GitHub*（英語）から入手できます。最新のインテルのハードウェアとソフトウェアを利用可能な無料のインテル® デベロッパー・クラウド（英語）で、Fortran `DO CONCURRENT` と OpenMP* アクセラレーター・オフロードの実験を行うことができます。

intel software

THE PARALLEL UNIVERSE