intel software

THE PARALLEL UNIVERSE

AI PC により大規模な LLM 開発がデスクトップで可能に

低ビットの量子化されたオープン LLM リーダーボード

Intel Vision 2024 のエンタープライズ AI アート展

Issue 57 2024

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。

変化する AI ハードウェアとソフトウェアの状況

本号に入る前に、私の休暇中、前号の編集長を引き受けてくれた The Parallel Universe の名誉編集者 James Reinders 氏に感謝します。

MLCommons は最近、「新しい MLPerf* 推論ベンチマーク結果が生成 AI モデルの急速な成長を浮き彫りに」（英語）という記事を公開しました。インテルは初めてインテル® Gaudi® 2 アクセラレーターの結果を MLCommons に提出し、このアクセラレーターが AI アプリケーションに低コストの代替手段を提供する方法を実証しました。

「業界には、高性能で高効率のコンピュート・オプションによって、生成 AI エンタープライズ製品のギャップに対処するという明確なニーズがあります。最新の MLPerf* の結果は、企業や顧客が標準ネットワークとオープン・ソフトウェアを備えた、よりコスト効率が高くスケーラブルなシステムを求めており、生成 AI をより多くの顧客が利用しやすくする上で、インテル® Gaudi® アクセラレーターが市場にもたらす独自の価値を示しています。

インテル コーポレーション
データセンター AI 製品管理担当 VP & GM
Zane Ball
（出典（英語））

2024 年秋に発売予定のインテル® Gaudi® 3 AI アクセラレーターの MLPerf* の結果を見るのが楽しみです。

私は最近、AI エンドユーザーの観点から何ができるかを確認するため、ローエンドの AI PC を試しています。一方、同僚の Tony Mongkolsmai は、AI 開発者がローカルで何ができるかを確認するため、ハイエンドの AI PC を試しています。本号の特集記事「**AI PC により大規模な LLM 開発がデスクトップで可能に**」では、大規模なモデルを開発する際、リモートのデータセンターやクラウドシステムに替わって、ローカルシステムで便利かつ安全に本格的な開発を行う方法を紹介します。

続く一連の記事では、AI に関連するさまざまなトピックを取り上げます。「**低ビットの量子化されたオープン LLM リーダーボード**」では、特定のクライアントに対して高品質のモデルを見つける新しいツールについて説明します。「**Intel Vision 2024 のエンタープライズ AI アート展**」では、「展示会」を振り返り、同様のアート作品を生成するチュートリアルを提供します。この記事では、「アート」の定義は非常に緩やかですが、結果として得られる画像と実用的な応用は魅力的です。「**Transformers による GGUF モデルの高速化**」では、GGUF 形式（モデルの保存と処理効率を最適化する新しいバイナリー形式）を活用し、低ビット LLM 推論を高速化する方法を紹介します。「**llama.cpp を使用したインテル® GPU 上での LLM 実行**」では、新しい SYCL* バックエンドを使用して、インテル® GPU 上で llama.cpp（人気が高まっている軽量で高性能な LLM フレームワーク）を実行する方法を説明します。

そして、MatLogica 社 CTO の Dmitri Goloubentsev 氏による「**Python* と C++ 分析によるシミュレーションと後方伝播の高速化**」に関する記事を紹介し、最後に「**インテル® データ・ストリーミング・アクセラレーターを使用したメモリー帯域幅依存カーネルの高速化**」で、ソフトウェア・パイプラインに対する CPU とアクセラレーターのハイブリッド・アプローチについて説明します。

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
2024 年 7 月

AI PC により大規模な LLM 開発がデスクトップで可能に

700 億のパラメーターを持つ LLM をクラウドだけで開発しない

Tony Mongkolsmai インテル コーポレーション ソフトウェア・アーキテクト / テクニカル・エバンジェリスト

大規模言語モデル（LLM）の普及により、計算能力に対するニーズが高まっています。OpenAI、Anthropic などの強力なソリューションはクラウドで実行されますが、小規模なシステムで実行できるモデルも増えています。ローカルで実行できるかどうかは、計算能力とメモリー容量の両方に依存します。モデルには通常、モデルの大きさと計算の複雑さを示すパラメーター・サイズがあります。パラメーター・サイズが大きくなると、ユーザーは計算能力の要件だけでなく、メモリー要件の課題にも直面します。トレーニングや推論を効率良く実行するには、LLM をデバイス（通常は GPU）のメモリーにロードする必要があります。コンシューマー向け GPU のメモリーは最大 24GB に制限されており、その多くは 16GB 未満のメモリーを搭載しています。企業が計算能力の要件とモデル精度のバランスを取れるように、Meta の Llama* モデルと Microsoft の Phi モデルにはさまざまなパラメーター・サイズが用意されています。例えば、Hugging Face* では Llama* モデルが最も人気があります。Llama* 2 には 7B、13B、70B のサイズがあり、Llama* 3 には 8B と 70B のサイズがあります。7B、8B、および 13B モデルは、多くのハイエンドのコンシューマー GPU で量子化と最適化を使用して実行できます。70B モデルは通常、コンシューマー GPU には大きすぎます。

大きな LLM を使用してローカルで開発する

一般に私は、ソリューションを開発するためデータセンターやクラウドにログインすることは好みません。ローカルシステムで開発するほうがレイテンシーとセキュリティーの面では優れていますが、LLM のサイズと計算能力の要件がローカル開発の妨げとなります。この課題を解決する 1 つのオプションは、エンタープライズ・グレードの GPU を搭載したワークステーションを導入することですが、コストが高くなる可能性があります。幸い、インテル® Core™ Ultra プロセッサー・ソリューションを出荷しているスモール・フォームファクター・ベンダーが解決策を提供しています。

私は最近、インテル® Arc™ 統合 GPU（iGPU）内蔵のインテル® Core™ Ultra 155H プロセッサーを搭載した Asus* NUC 14 Pro システムを購入しました。このシステムの斬新な点は、iGPU がシステム RAM の最大半分を GPU メモリーとして使用できることです。最大 96GB の DDR5-5600 DRAM で構成すると、システムはワークステーションの数分の 1 のコストで最大の 70B Llama* モデルを実行できます。

システムの設定

ハードウェア・セットアップ

Asus* NUC 14 Pro のセットアップは簡単でした。48GB DDR5-5600 SODIMM DRAM モジュール 2 本を差し込み、M.2 20x80 NVMe* SSD を取り付けるだけでした。テストでは Windows* と Linux* をデュアルブートにしたいと考えており、Meta* Llama* 3 70B モデルは約 550GB のディスク容量を必要とするため、少なくとも 4TB の SSD が必要でした。ハードウェアのセットアップにかかった時間は 5 分未満です。

オペレーティング・システムのインストール

オペレーティング・システムのインストールも比較的簡単でした。まず、Windows* 11 をインストールしましたが、これにはハードウェア・イーサネット接続が必要でした。Wi-Fi* はそのままでは機能しないため、ネットワーク・インストールの回避策はありませんでした。Windows* をインストール後、Asus のウェブサイトから最新のドライバーをダウンロードしてインストールし、Windows* Update を行いました。

次に、Ubuntu* 22.04.4 をインストールしましたが、問題なく動作しました。Linux* を起動後、指示（英語）に従って最新のユーザーモード GPU ドライバーをインストールしました。

ヒント：Windows* と Linux* からアクセスできる比較的大きな共有ドライブを用意するとよいでしょう。Windows* と Linux* で LLM をテストする場合、モデルを共有ドライブに配置し、同じモデルが両方のパーティションでスペースを占有しないようにする必要があります。

LLM の実行

LLM のソフトウェア・スタックは急速に変化しており、さまざまな選択肢があります。ここでは、従来の Hugging Face*/PyTorch* ワークフローと、人気の llama.cpp オープンソース・フレームワークを使用して Llama* 3 70B をテストしますが、将来的にはより大規模なデータセンター・システムで実行することを想定しています。ここでは、Windows* のワークフローについて説明しますが、このプラットフォームとソフトウェア・スタックは Linux* でも動作します。

基本構成

モデル実行の AI 固有の部分に入る前に、Windows* の環境を準備する必要があります。いくつかのパッケージをインストールします。

1. Windows* Visual Studio* 2022 Community Edition
2. conda-forge（英語）
3. インテル® oneAPI ベース・ツールキット

PyTorch* で実行

PyTorch* を使用して LLM を実行するには、通常、Hugging Face*（英語）にあるオープンソース・ライブラリー、API、モデルを使用します。

PyTorch* と Hugging Face* API のインストール

PyTorch* 環境と必要な Hugging Face* ライブラリーをセットアップするには、`conda-forge` Miniforge プロンプトを起動し、次のコマンドを実行します。

```
# conda-forge で conda 環境を作成して有効化
conda create -n llm python=3.11 libuv
conda activiate llm
# PyTorch* および Hugging Face* ライブラリーとインテルの PyTorch* 向け LLM ライブラリーをインストール
pip install --pre --upgrade ipex-llm[xpu] --extra-index-url
https://pytorch-extension.intel.com/release-whl/stable/xpu/us/
```

ここでは、インテル® GPU 上で Hugging Face* API のより高性能な実装を提供するインテルの PyTorch* LLM ライブラリーを使用して依存関係をインストールしました。ライブラリーの依存関係ツリーは、PyTorch* とその他の必要な Hugging Face* ライブラリーもインストールします。

モデルの取得

ここでは、Hugging Face* からダウンロード可能な最新の Llama* 3 モデルである、Llama* 70B をテストします。Llama* モデルは Meta からの承認が必要なので、https://huggingface.co/meta-llama/Meta-Llama-3-70B（英語）からアクセスを申請し、ライセンスに同意する必要があります。承認されると、次の操作を実行できます。

```
# Hugging Face* CLI のインストール
pip install huggingface-cli
# CLI を使用して Hugging Face* にログイン
git clone https://huggingface.co/meta-llama/Meta-Llama-3-70B
```

前述のように、モデルはかなり大きく、551GB のディスク容量を消費するため、ダウンロードには時間がかかる場合があります。

推論の実行

PyTorch* と Hugging Face* API を使用して推論を実行するのは簡単です。ほかの GPU スクリプトとわずかに異なる部分があることが分かります (緑色でハイライトされた箇所)。ここでは、インポートにインテルのライブラリーが使用されており、`to` 関数のターゲットデバイスは「`cuda`」ではなく「`xpu`」です。

```
from ipex_llm.transformers import AutoModelForCausalLM

...

model = AutoModelForCausalLM.from_pretrained(model_path,
     load_in_4bit=True,
     optimize_model=True,
     trust_remote_code=True,
     use_cache=True)

# モデルを GPU アクセラレーターに移動
model = model.half().to('xpu')

# トークナイザーをロード
tokenizer = AutoTokenizer.from_pretrained(model_path, trust_remote_code=True)

...

# 予測トークンを生成
with torch.inference_mode():
     prompt = get_prompt(args.prompt, [], system_prompt=DEFAULT_SYSTEM_PROMPT)
     input_ids = tokenizer.encode(prompt, return_tensors="pt").to('xpu')

     # 推論を開始
     output = model.generate(input_ids,
            eos_token_id=terminators,
            max_new_tokens=args.n_predict)
     torch.xpu.synchronize()
     output = output.cpu()
     output_str = tokenizer.decode(output[0], skip_special_tokens=False)
```

完全なコードはこちら（英語）を参照してください。スクリプトの実行は簡単です。次に示すように、いくつかのパラメーター（モデル、プロンプト、出力サイズなど）を渡します。

```
python llama3.py --repo-id-or-model-path D:\models\Meta-Llama-3-70B-Instruct --prompt "Tell
me if or why AI is important to the future of humanity" --n-predict 51
```

```
(llm) C:\Users\tonym\dev>python llama3.py --repo-id-or-model-path D:\models\Meta-Llama-3-70B-Instruct --prompt "Tell me if or why AI is impor
tant to the future of humanity" --n-predict 512
C:\Users\tonym\miniforge3\envs\llm\Lib\site-packages\torchvision\io\image.py:13: UserWarning: Failed to load image Python extension: 'Could n
ot find module 'C:\Users\tonym\miniforge3\envs\llm\Lib\site-packages\torchvision\image.pyd' (or one of its dependencies). Try using the full
path with constructor syntax.'If you don't plan on using image functionality from `torchvision.io`, you can ignore this warning. Otherwise, t
here might be something wrong with your environment. Did you have `libjpeg` or `libpng` installed before building `torchvision` from source?
  warn(
2024-05-31 11:04:40,189 - INFO - intel_extension_for_pytorch auto imported
Loading checkpoint shards: 100%|████████████████████████████████████████████| 30/30 [00:02<00:00, 11.49it/s]
2024-05-31 11:04:43,305 - INFO - Converting the current model to sym_int4 format.....
Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned or trained.
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `attent
ion_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `attent
ion_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.
Inference time: 322.889187097549444 s
-------------------- Prompt --------------------
<|begin_of_text|><|start_header_id|>user<|end_header_id|>

Tell me if or why AI is important to the future of humanity<|eot_id|><|start_header_id|>assistant<|end_header_id|>


-------------------- Output (skip_special_tokens=False) --------------------
<|begin_of_text|><|begin_of_text|><|start_header_id|>user<|end_header_id|>

Tell me if or why AI is important to the future of humanity<|eot_id|><|start_header_id|>assistant<|end_header_id|>

What a great question! Artificial Intelligence (AI) is indeed crucial to the future of humanity, and here's why:

**1. Solving Complex Problems**: AI can help tackle some of humanity's most pressing challenges, such as:
        * Climate change: AI can analyze vast amounts of environmental data to identify patterns and predict climate-related events.
        * Healthcare: AI can aid in disease diagnosis, personalized medicine, and drug discovery.
        * Energy: AI can optimize energy consumption, predict energy demand, and improve energy storage.

**2. Improving Efficiency and Productivity**: AI can automate repetitive tasks, freeing humans to focus on creative, high-value work, leading
 to:
        * Increased productivity: AI can streamline processes, reducing time and resources spent on mundane tasks.
        * Enhanced customer experience: AI-powered chatbots and virtual assistants can provide 24/7 support.

**3. Enabling New Industries and Jobs**: AI is creating new opportunities and transforming existing ones, such as:
        * Autonomous vehicles: AI is driving the development of self-driving cars, trucks, and drones.
        * Cybersecurity: AI-powered systems can detect and respond to threats more effectively than traditional methods.
        * Data Science: AI has created a new field of data analysis and interpretation.

**4. Enhancing Human Life**: AI can improve the quality of life for individuals, particularly:
        * Elderly care: AI-powered systems can assist with daily tasks, provide companionship, and monitor health.
        * Accessibility: AI-driven tools can help people with disabilities, such as speech-to-text systems and image recognition.

**5. Space Exploration and Colonization**: AI will play a critical role in:
        * Spacecraft navigation: AI can optimize routes, detect anomalies, and make autonomous decisions.
        * Planetary colonization: AI can help establish and maintain human settlements on other planets.

**6. Cybersecurity and Defense**: AI can help protect against cyber threats, detect anomalies, and enhance national security:
        * Threat detection: AI-powered systems can identify potential threats and respond in real-time.
        * Autonomous defense: AI can enable autonomous defense systems to respond to threats without human intervention.

**7. Education and Skills Development**: AI can revolutionize education by:
        * Personalized learning: AI can tailor educational content to individual students' needs and abilities.
        * Skills development: AI can help identify skill gaps and provide targeted training recommendations.
```

Python* 出力には、モデル・チェックポイントのロード、モデルの量子化、モデルからのプロンプトと出力の表示など、Hugging Face* ユーザーが期待するすべての標準的な情報が表示されます。

llama.cpp で実行

llama.cpp（英語）プロジェクトは、純粋な C++ 実装を使用して LLM 推論を可能にします。さまざまな GPU とフレームワークのバックエンドをサポートしています。テストには SYCL* バックエンドを使用します。

ビルド

最初に、リポジトリーをクローンします。

```
git clone https://github.com/ggerganov/llama.cpp.git
```

次に、依存関係の cmake（英語）、mingw-w64（英語）、および インテル® oneAPI ベース・ツールキットをインストールします。依存関係をインストールしたら、SYCL* を使用して llama.cpp を簡単にビルドできます。

```
# oneAPI 環境の初期化
"C:\Program Files (x86)\Intel\oneAPI\setvars.bat" intel64
# llama.cpp ビルドの設定
cmake -B build -G "MinGW Makefiles" -DLLAMA_SYCL=ON -DCMAKE_C_COMPILER=icx
-DCMAKE_CXX_COMPILER=icx -DCMAKE_BUILD_TYPE=Release
# llama.cpp 実行ファイルのビルド
cmake --build build --config Release -j
```

バイナリーは `build/bin/main` に出力されます。

モデルの取得

llama.cpp プロジェクトは、LLM を高速にロードするため GGUF（GPT-Generated Unified Format、GPT によって生成された統一形式）を使用します。この形式にはセキュリティー上の懸念があるため、信頼できるソースからの GGUF ファイルを使用していることを確認してください。この記事では、Hugging Face* LMStudio（英語）リポジトリー（ここをクリックしてダウンロード）から、Llama* 3 70B モデルの INT4 量子化バージョンをダウンロードしました。ディスク上で 42.5GB あるため、モデルを実行すると、同程度の GPU メモリーが使用されます。

推論の実行

実行ファイルとモデルを用意できたら、モデルとプロンプトを使用して llama.cpp を呼び出し、すべてのモデルレイヤーを GPU にオフロードします。

```
build\bin\main.exe --model d:\models\Meta-Llama-3-70B-Instruct-Q4_K_M.gguf --prompt "Tell me
if or why AI is important to the future of humanity" --n-gpu-layers 999
```

Hugging Face*/PyTorch* ワークフローとは異なり、llama.cpp では事前に量子化されたバージョンのモデルをロードします。llama.cpp 出力には、いくつかのモデル・パラメーター、モデルのロード方法、デバイス上のモデルの特性が表示されます。モデルには 81 のレイヤーがあり、インテル® Core™ Ultra プロセッサー上の iGPU で実行されます。モデルは SYCL* ベースの計算パスで 81 のレイヤーを実行するため、40GB のメモリーを使用していることが分かります。最後に、Hugging Face*/PyTorch* 実装に似たプロンプトと出力が表示されます。

これが素晴らしい理由

いくつもの複雑なコンポーネントを組み込んだ、さまざまなアプリケーションやツールを構築してきた経験から、ローカルでできることが多ければ多いほど、開発プロセスを効率化できると実感しています。ワークフローを理解していれば、クラウドにモデルをデプロイするのは簡単です。

また、作業方法を理解していれば、エンタープライズ・インフラストラクチャーにもモデルを容易にデプロイできます。LLM をローカルで実行できることの素晴らしい点は、これらすべてを理解しなくても、すぐにテストできることです。いずれソリューションのデプロイが課題になりますが、ソリューションを設計している間は心配する必要がありません。

このソリューションのもう 1 つの利点は、小さなモデルであっても、クライアント・システムでは量子化を使用して実行できることが多いことです。これにより、モデルの精度が低下する可能性がありますが、メモリー容量を増やすことで、より高精度の量子化を使用してモデルをテストしたり、量子化されていないモデルを使用してモデルの機能を適切に判断できます。

LLM の設計、使用、統合に関連したソフトウェア開発は、増え続けるソフトウェア・ソリューションにとって優先事項です。これらのソリューションに対応する計算能力の要件とメモリー要件は、最高のコンシューマー・システムでさえも満たすことが難しく、できるだけローカルに保持したいという開発者の要望も考慮すると、開発者の生産性を最適化する上で大きな課題が生じます。

幸いなことに、96GB の DRAM で構成されたインテル® Arc™ 統合 GPU と小型フォームファクター PC を組み合わせることで、この課題を解決できます。このシステムでは、ハイエンドのコンシューマー向けディスクリート GPU でも実行できないモデルをローカルで実行できます。これらすべてが、約 1,200 米ドル（執筆時点）という非常にリーズナブルな価格で実現できます。何よりも、前述のソフトウェアは oneAPI 統合スタック上に構築されているため、ローカルで開発し、どこにでもデプロイできる利点があります。

intel® tiber™
Developer Cloud

コードの ROI を 高める

自分だけのクラウドで
快適に作業

複数の言語をサポートし、高速なヘテロジニアス・コンピューティングを実現する、単一のオープンなプログラミング・モデルで、コードの能力を引き出しましょう。オープン・スタンダードをベースとして構築されたインテル® Tiber™ デベロッパー・クラウドは、AI 開発を活性化し、優れたパフォーマンスを実現するための理想的なエンタープライズ環境を提供します。

エンタープライズ、クラウド、HPC、
AI などの分野の計算を高速化。

今すぐ開始

www.xlsoft.com/jp/products/intel/devcloud/

低ビットの量子化されたオープン LLM リーダーボード

クライアント向けに高品質のモデルを見つける新しいツール

Kaokao Lv、Wenjiao Yue、Wenhua Cheng、Jun Lin、Hanwen Chang、Tai Huang、Haihao Shen
インテル コーポレーション

Hugging Face* にはすでにリーダーボード（英語）がありますが、なぜ新しいリーダーボードを作成したのでしょうか？ それは、量子化の結果の比較は簡単ではないからです。主な問題は、ほとんどの量子化モデルで精度の結果が不足していることで、もう 1 つの問題は、Hugging Face* リーダーボードで特定のモデル名を検索すると多数のモデルが見つかることです。そのため、それらがマージされているか、ファインチューニングされているか、FP16 で量子化されているか、混合されているかを判断するには、多くの手作業が必要です。これは、モデルのデプロイメントに課題をもたらします。

そこで、量子化 LLM モデルに注目し、検索エンジンを強化することでこれらの問題に対処する、低精度量子化リーダーボード（英語）を開発しました。ユーザーは、アルゴリズム（AutoRound（英語）、GPTQ（英語）、AWQ（英語）、BitsAndBytes（英語）、および GGUF（英語））、計算のデータ型（int8、fp16、bf16 など）、重みのデータ型（fp4、int4、nf4 など）、モデルサイズ、二重量子化が有効かどうかで、量子化 LLM を即座に検索できます。低ビットの量子化されたオープン LLM リーダーボードは、特定のクライアントに効率良くデプロイできる高品質のモデルを見つけられる貴重なツールです。

量子化アプローチ

さまざまな量子化方法と重みおよび計算のデータ型にわたって LLM モデルのベンチマークを効果的に行うには、堅牢な量子化ツールが必要です。インテルのリーダーボードは、LLM 量子化サポートに Transformers 向けインテル® エクステンション（英語）を利用しています。このソリューションは、GPTQ（英語）や AWQ（英語）などのよく知られている重みのみの量子化方法をシームレスに統合するインターフェイスを備えた Transformers のような API を提供します。さらに、このツールには、低ビット LLM 推論用のインテルの AutoRound 量子化アルゴリズム（英語）が組み込まれています。Transformers 向けインテル® エクステンションの量子化機能は、オープンソースのモデル圧縮ツールであるインテル® ニューラル・コンプレッサー（英語）をベースに構築されています。

低ビットの量子化されたオープン LLM リーダーボード

このリーダーボードには 10 の異なるベンチマーク（ARC-c、ARC-e、Boolq、HellaSwag、Lambada_openai、MMLU、Openbookqa、Piqa、Truthfulqa_mc1、Winogrande）が含まれています。ランキングは、これらのベンチマークの平均スコアによって決定され、再ランキング時に特定のベンチマークを優先するオプションがあります。

私たちの評価では、AutoRound は一般的なさまざまなモデルで準可逆圧縮に近づいており、GPTQ や AWQ などのほかの方法よりも一貫して優れており、GGUF よりも優れた精度を示しています。Llama* 2 7b-chat と Mistral*-7b-instruct の精度を**図 1** と**図 2** に示します。

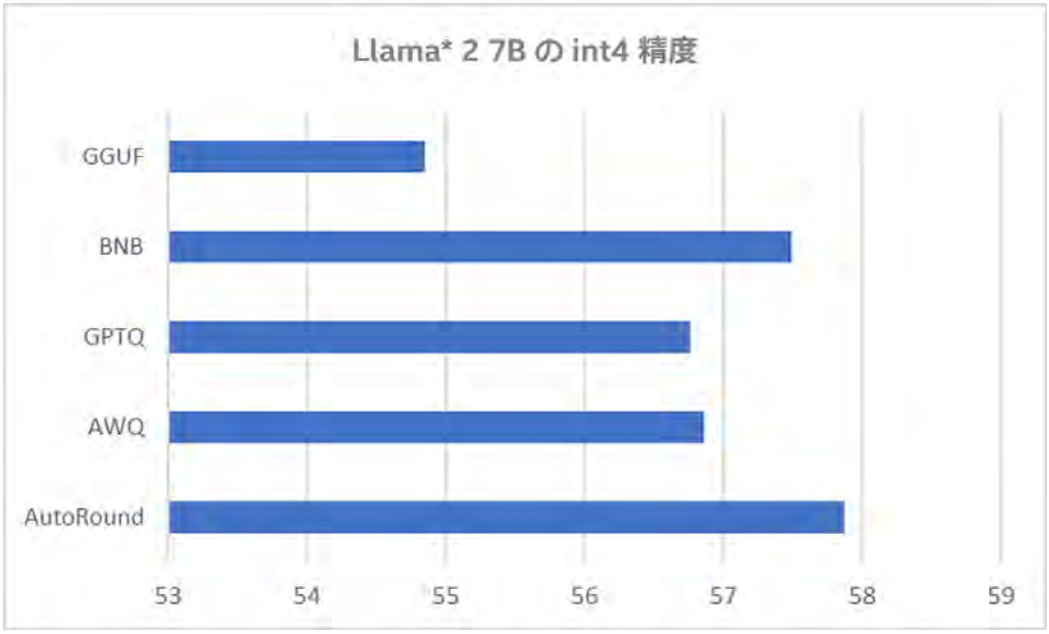


図 1. int4 Llama* 2 7B-chat の平均精度（値が大きいほうが良い）

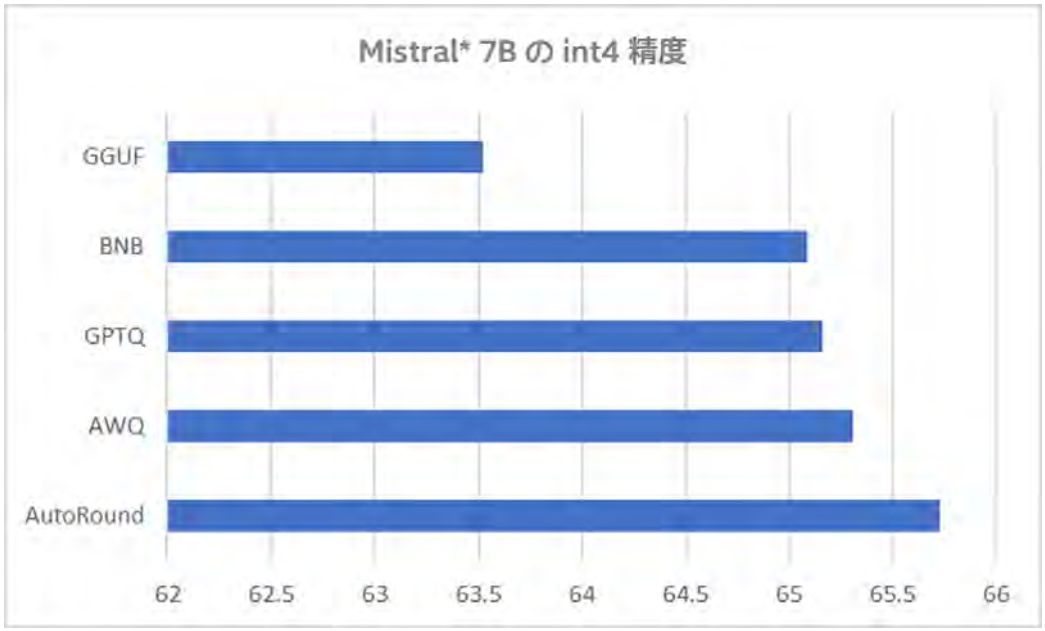


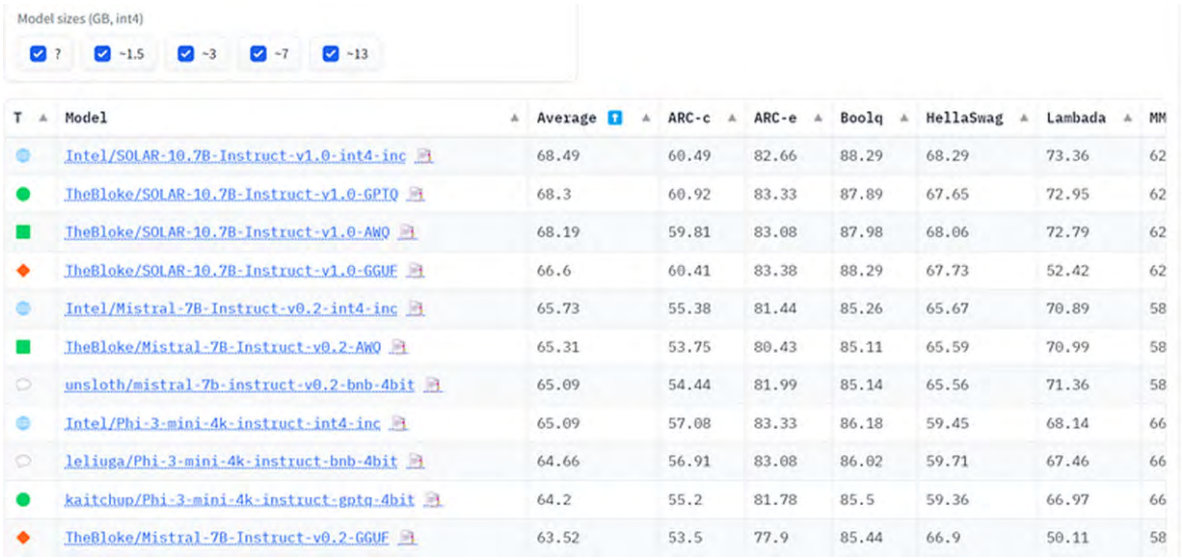
図 2. int4 Mistral*-7B-instruct-v0.2 の平均精度（値が大きいほうが良い）

また、13B AutoRound モデルと fp16 7B モデルを比較したところ、AutoRound はすべてのメトリックと平均で一貫して fp16 7B を上回っていることが分かりました（**表 1**）。これにより、低ビットの量子化された中規模 LLM を利用することで、モデルサイズが小さくても、半精度の小規模 LLM よりも優れたパフォーマンスを発揮できます。

	Average	Arc-c	Arc-e	boolq	lambada	mmlu	openbookqa	piqa	truthfulqa	winogrande
Baichuan2-7B-Chat	57.1	41.64	72.72	79.45	67.63	50.82	30.4	74.48	31.21	68.98
Baichuan2-13B-Chat AutoRound int4	60.46	47.35	74.92	82.11	71.32	55.78	31.2	75.84	36.35	73.01

表 1. fp16 7B モデルと 13B AutoRound モデルの精度（値が大きいほうが良い）

さらに、AutoRound は一般的なモデルのほとんどに対応できますが、ほかの量子化アルゴリズムには制限があります。これが、AutoRound モデルが利用しやすい理由です（**表 2**）。



T	Model	Average	ARC-c	ARC-e	Boolq	HellaSwag	Lambada	MM
	Intel/SOLAR-10.7B-Instruct-v1.0-int4-inc	68.49	60.49	82.66	88.29	68.29	73.36	62
	TheBloke/SOLAR-10.7B-Instruct-v1.0-GPTQ	68.3	60.92	83.33	87.89	67.65	72.95	62
	TheBloke/SOLAR-10.7B-Instruct-v1.0-AWQ	68.19	59.81	83.08	87.98	68.06	72.79	62
	TheBloke/SOLAR-10.7B-Instruct-v1.0-GGUF	66.6	60.41	83.38	88.29	67.73	52.42	62
	Intel/Mistral-7B-Instruct-v0.2-int4-inc	65.73	55.38	81.44	85.26	65.67	70.89	58
	TheBloke/Mistral-7B-Instruct-v0.2-AWQ	65.31	53.75	80.43	85.11	65.59	70.99	58
	unsloth/mistral-7b-instruct-v0.2-bnb-4bit	65.09	54.44	81.99	85.14	65.56	71.36	58
	Intel/Phi-3-mini-4k-instruct-int4-inc	65.09	57.08	83.33	86.18	59.45	68.14	66
	leliuga/Phi-3-mini-4k-instruct-bnb-4bit	64.66	56.91	83.08	86.02	59.71	67.46	66
	kaitchup/Phi-3-mini-4k-instruct-gptq-4bit	64.2	55.2	81.78	85.5	59.36	66.97	66
	TheBloke/Mistral-7B-Instruct-v0.2-GGUF	63.52	53.5	77.9	85.44	66.9	50.11	58

表 2. 低ビットの量子化されたオープン LLM リーダーボード（2024 年 5 月 11 日現在）

AutoRound は、AWQ などのアルゴリズムにはない lm_head の量子化もサポートしています。GPTQ や AWQ とは異なり、AutoRound は新しいモデルを自動的に受け入れることができます。最後に、AutoRound はさまざまなデータセットにわたるキャリブレーション機能を提供します。

量子化サンプルコード

次のサンプルコードは、Transformers 向けインテル® エクステンション（英語）の Transformer のような API を使用して AutoRound を適用し、LLM を量子化する方法を示します。

```
from transformers import AutoTokenizer
from intel_extension_for_transformers.transformers import AutoModelForCausalLM, AutoRoundConfig

model_name_or_path = "Intel/neural-chat-7b-v3-3"
prompt = "Once upon a time, a little girl"
tokenizer = AutoTokenizer.from_pretrained(model_name_or_path, trust_remote_code=True)
inputs = tokenizer(prompt, return_tensors="pt").input_ids

q_config = AutoRoundConfig(bits=4, tokenizer=tokenizer)
int4_model = AutoModelForCausalLM.from_pretrained(
                                    model_name_or_path,
                                    quantization_config=q_config,
                                     )
output = int4_model.generate(inputs, max_new_tokens=100, do_sample=True)
```

詳細については、Transformers 向けインテル® エクステンションの AutoRound サンプル（英語）を参照してください。

コラボレーションと今後の取り組み

ぜひ、リーダーボードを試して、量子化モデルをアップロードしてください。皆さんからのフィードバック、質問、コメントをお待ちしています。独自のモデルで試してみたい場合は、チケットを作成して、インテルのチームからどのようなサポートが得られるかをご確認ください。

将来的には、リーダーボードを拡張して、超低ビットの量子化されたオープン LLM をサポートする予定です。このトピックに興味がある場合は、お気軽にメールでお問い合わせください。繰り返しになりますが、皆さんからのフィードバックと貢献をお待ちしています。

Intel Vision 2024 のエンタープライズ AI アート展

インテル® Gaudi® AI アクセラレーターによる Stable Diffusion のリアルタイム・ファインチューニング

Srinarayan Srikanthan インテル コーポレーション
Preethi Venkatesh インテル コーポレーション

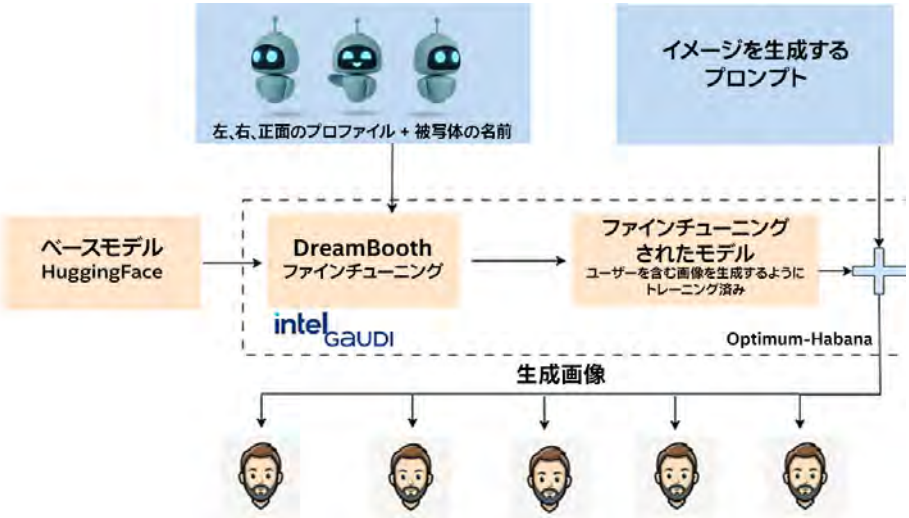
Intel Vision 2024 のエンタープライズ AI アート展では、参加者がインテル® Gaudi® AI アクセラレーターでオープンソースの Stable Diffusion モデルを実行して、独自のアート作品を作成するデモが実施されました。このデモの特筆すべき点は、参加者がリアルタイムで作品を作成する Few-Shot 学習の実践的な応用を通して、創造性と革新性を育むため企業環境で AI をどのように活用できるかを具体的に示したことです。

この記事では、このデモについて説明し、自分で実行する方法も紹介します。

デモの概要

Stable Diffusion に基づくカスタムアート作品を作成して、インテル® Gaudi® プロセッサーおよびソフトウェア・スタックの機能を紹介しました。デモでは、DreamBooth（英語）ファインチューニング手法を使用して、限られた画像セットで Dreamlike Diffusion（英語）と呼ばれる Stable Diffusion 1.5 モデルのバリエーションをファインチューニングし、最小限のデータでパーソナライズされたアートを生成できることを示しました。インテル® Gaudi® プロセッサーと optimum-habana（英語）により、必要な計算能力が提供され、高速なモデル・トレーニングが行われました。

ベースモデルは、Stable Diffusion 1.5 ベースの Dreamlike Diffusion 1.0 です。DreamBooth ファインチューニングは、Stable Diffusion のようなテキストから画像への変換モデルを、被写体のわずか数枚（3 ~ 5 枚）の画像でパーソナライズする方法です。このデモでは、Stable Diffusion 1.5 の DreamBooth ファインチューニングにより、独自の画像からカスタムアートを作成できるようにします。



結果として得られるファインチューニングされたモデルは、以下に示すようなユーザープロンプトに基づいて、ユニークなアート作品を生成できます。

プロンプト：a dreamlike vision of the universe swirling within close-up side portrait of solo gaudigeekatintel, fluid, constellations and nebulae, dreamlike art, fantasy, star trek aesthetic, vibrant pastel color aesthetic, concept art, sharp focus, flawless skin, pastel colors, digital painting, hd, dramatic lighting, trending in art station
（gaudigeekatintel の横顔をクローズアップした肖像画の中に渦巻く幻想的な宇宙、流体、星座と星雲、幻想的なアート、ファンタジー、スタートレックの美学、鮮やかなパステルカラーの美学、コンセプトアート、鮮明、完璧な肌、パステルカラー、デジタル絵画、HD、ドラマチックな照明、ArtStation のトレンド）

Intel Vision 2024 のデモの録画はこちら（英語）から視聴できます。

チュートリアル：独自の Stable Diffusion アートを作成する

DreamBooth ファインチューニング

デモでは、Stable Diffusion 1.5 の DreamBooth ファインチューニングを使用して、独自の画像（512 x 512 解像度にサイズ変更）のカスタムアートを作成します。4 つの環境変数を設定する必要があります。

1. **MODEL_NAME :** Dreamlike Diffusion 1.0 モデル（dreamlike-art/dreamlike-diffusion-1.0）に設定します。これは、dreamlike.art（英語）によって作成された高品質のアートでファインチューニングされた Stable Diffusion 1.5 です。
2. **INSTANCE_DIR :** 3 ~ 5 枚の入力画像を含むディレクトリーに設定します（例 : `/home/art_studio/gaudigeekatintel/`）。入力画像は、背景が無地で、適切な照明が当たり、肩の高さで 512 x 512 にサイズ変更された被写体の正面、右側面、左側面の画像です（以下のサンプル画像を参照）。被写体の参照または名前は、できるだけ一意である必要があります（例 :「gaudigeekatintel」）。ここでは、モデルのアーティファクトの作成にこの参照名を使用します。
3. **OUTPUT_DIR :** ファインチューニングされたモデルの保存先ディレクトリーに設定します（例 : `/home/art_studio/dd_model_gaudigeekatintel`）。
4. **CLASS_DATA_DIR :** トレーニングするクラスの画像を含むディレクトリーに設定します。このチュートリアルでは、DreamBooth ファインチューニングで人物画像を生成するために使用する男性と女性のサンプル画像をダウンロードします。Kaggle データセット（英語）から JPEG 形式の「男性」と「女性」の画像をそれぞれ約 50 枚選択し、`/home/art_studio/person` という名前のディレクトリーに保存します。

インテル® Gaudi® プロセッサー上でのファインチューニング

Gaudi Docker イメージを起動して、上記で作成したすべてのディレクトリーのボリュームをマウントします。

```
cd /home/art_studio
docker run -it --runtime=habana \
-e HABANA_VISIBLE_DEVICES=all \
-e OMPI_MCA_btl_vader_single_copy_mechanism=none \
--cap-add=sys_nice \
--net=host \
--ipc=host \
-v $(pwd):/home \
vault.habana.ai/gaudi-docker/1.15.1/ubuntu22.04/habanalabs/pytorch-installer-2.2.0:latest
```

optimum-habana をインストールします。

```
pip install optimum-habana==v1.11.0
pip install peft
```

コンテナ内のすべての環境変数を設定します。

```
export MODEL_NAME="dreamlike-art/dreamlike-diffusion-1.0"
export INSTANCE_DIR="/home/gaudigeekatintel/"
export OUTPUT_DIR="/home/dd_model_gaudigeekatintel"
export CLASS_DATA_DIR="/home/person"
```

GitHub* の optimum-habana リポジトリーをクローンします。

```
git clone -b v1.12.0 https://github.com/huggingface/optimum-habana.git
```

DreamBooth ファインチューニング機能を備えた optimum-habana のサポートが PR (英語) によって追加され、インテル® Gaudi® ベースのアーキテクチャーでファインチューニングを実行できるようになります。

```
cd optimum-habana
cd examples/stable-diffusion/training/
```

インテル® Gaudi® プロセッサーで AI モデルをトレーニングする場合、トレーニング・スクリプトを初めて実行すると、optimum-habana は時間をかけてグラフと呼ばれる複雑な計算マップを作成します。このグラフは、効率的な処理を保証するためハードウェアに合わせて調整されます。これは、特定のユースケースで 1 回だけ発生するウォームアップ・コストです。最初の被写体の 3 つの画像に対して 1 回だけ実行する必要があります。その後、グラフを使用して後続の被写体をファインチューニングできます。グラフを作成するには、環境変数 `PT_HPU_RECIPE_CACHE_CONFIG` を使用して `recipe_cache` を制御します。これには、カンマで区切られた次の 3 つの情報が必要です。

1. グラフの保存場所（マシン上のパス、例 : `/tmp/recipe_cache`）。
2. 使用後にグラフを削除するかどうか（保持する場合は `False`、削除する場合は `True`）。
3. キャッシュサイズの上限（メガバイト単位、例 : 1GB の場合は `1024`）。

例 :

```
PT_HPU_RECIPE_CACHE_CONFIG=/tmp/recipe_cache,False,1024 python <script.py>
```

これで、後で再利用できるように指定された場所にグラフが作成され保存されます。

次のパラメーターで `train_dreambooth.py` スクリプトを 5 ステップ実行し、グラフを作成します。この手順は、指定されたベアメタルシステムまたはコンテナで 1 回だけ実行する必要があることに注意してください。グラフがキャッシュに保存されたら、被写体が同じ場合でも、異なる場合でも、後続の実行でこのプロセスを繰り返す必要はありません。

```
PT_HPU_RECIPE_CACHE_CONFIG=/tmp/dld_recipe_cache,False,1024 \
python train_dreambooth.py  \
--pretrained_model_name_or_path=$MODEL_NAME \
--instance_data_dir $INSTANCE_DIR \
--output_dir=$OUTPUT_DIR \
--class_data_dir=$CLASS_DATA_DIR \
--with_prior_preservation \
--prior_loss_weight=1.0 \
--instance_prompt="gaudigeekatintel" \
--class_prompt="person" \
--train_batch_size=1 \
--gradient_accumulation_steps=1 \
--learning_rate=2e-6 \
--lr_scheduler="constant" \
--lr_warmup_steps=0 \
--max_train_steps=5 \
--gaudi_config_name Habana/stable-diffusion \
--train_text_encoder \
--center_crop \
--num_class_images=12 \
--seed=0 \
--prior_generation_precision bf16 full
```

次に、`train_dreambooth.py` スクリプトを 350 ステップ実行し、被写体の画像で完全にファインチューニングします。このステップでは、`/tmp/dld_recipe_cache` に保存されたグラフを再利用します。

```
PT_HPU_RECIPE_CACHE_CONFIG=/tmp/dld_recipe_cache,False,1024 \
python train_dreambooth.py  \
--pretrained_model_name_or_path=$MODEL_NAME \
--instance_data_dir $INSTANCE_DIR \
--output_dir=$OUTPUT_DIR \
--class_data_dir=$CLASS_DATA_DIR \
--with_prior_preservation \
--prior_loss_weight=1.0 \
--instance_prompt="gaudigeekatintel" \
--class_prompt="person" \
--train_batch_size=1 \
--gradient_accumulation_steps=1 \
--learning_rate=2e-6 \
--lr_scheduler="constant" \
--lr_warmup_steps=0 \
--max_train_steps=350 \
--gaudi_config_name Habana/stable-diffusion \
--train_text_encoder \
--center_crop \
--num_class_images=12 \
--seed=0 \
--prior_generation_precision bf16 full
```

これにより、保存されたグラフと最適なパラメーターを使用して、`INSTANCE_DIR` に保存された画像でモデルがファインチューニングされます。ファインチューニングされたモデルは `OUTPUT_DIR` に保存されます。保存されたグラフを使用したファインチューニングの実行には、1 枚のインテル® Gaudi® カードで約 3 ~ 4 分かかります。このデモ設定では 1 枚のカードを対象としていますが、現在 8 枚のカードで実行できるようにワークロードを最適化しています。

テキストプロンプトを使用してファインチューニングされた画像を生成する

このデモの出力例を以下に示します。次のサンプルスクリプトでは、optimum-habana の `diffusers` クラスを使用して、Stable Diffusion パイプラインとスケジューラーを実行します。ファインチューニングされたモデルを読み込み、入力プロンプトに基づいて肖像画を生成します。ここでは、「`portrait of solo gaudigeekatintel in a multiverse universe with planets, moons and solar flares, star trek, pastel colors, blue and purple tone background, dramatic lighting, trending in art station.`（惑星、月、太陽フレア、スタートレック、パステルカラー、青と紫のトーンの背景、ドラマチックな照明、ArtStation のトレンドを含む多元宇宙を背景とした gaudigeekatintel の肖像画）」というプロンプトを試します。独自の肖像画を作成するには、DreamBooth ファインチューニングで使用したインスタンス・プロンプトを入力してください。

```
import torch
import os
from optimum.habana.diffusers import GaudiStableDiffusionPipeline, GaudiDDIMScheduler

# スケジューラーの設定
scheduler = GaudiDDIMScheduler.from_pretrained(
     "dreamlike-art/dreamlike-diffusion-1.0",
     subfolder="scheduler"
)

# インテル(R) Gaudi(R) プロセッサーの Stable Diffusion パイプラインの使用
pipeline = GaudiStableDiffusionPipeline.from_pretrained(
     '/home/dd_model_gaudigeekatintel',
     scheduler=scheduler,
     use_habana=True,
     use_hpu_graphs=True,
     gaudi_config="Habana/stable-diffusion"
)

negative_prompt="easynegative, head covered, face covered, helmet, not indoors, no flashy
jewelry, not indian bride, two men, two women, two persons, two heads, two bodies, duplicate
person, multiple person, mirror reflection, eye color change, low quality, worst quality:1.4,
bad anatomy, bad composition, out of frame, ugly, old person with wrinkles, morbid,
mutilated, out of frame, extra fingers, mutated hands, poorly drawn hands, poorly drawn face,
mutation, deformed, ugly, blurry, bad anatomy, bad proportions, extra limbs, cloned face,
disfigured, out of frame, ugly, extra limbs, bad anatomy, gross proportions, malformed limbs,
missing arms, missing legs, extra arms, extra legs, mutated hands, fused fingers, too many
fingers, long neck,  watermark, signature, text, deformed nose, deformed lips"

# “gaudigeekatintel” の代わりに DreamBooth ファインチューニングの –-instance_prompt フラグで使用したインス
タンス・ラベルを使用

prompt=" portrait of solo gaudigeekatintel in a multiverse universe with planets, moons and
solar flares, star trek, pastel colors, blue and purple tone background, dramatic lighting,
trending in art station"

image = pipeline(prompt, height=536, width=960, negative_prompt= negative_prompt, num_
inference_steps=150, guidance_scale=7).images[0]

output_path= '/home/output/'
os.makedirs(output_path)
filename='gaudigeekatintel.png'
os.path.join(output_path, filename)
```

プロンプト：portrait of solo gaudigeekatintel in a multiverse universe with planets, moons and solar flares, star trek, pastel colors, blue and purple tone background, dramatic lighting, trending in art station
（惑星、月、太陽フレア、スタートレック、パステルカラー、青と紫のトーンの背景、ドラマチックな照明、ArtStation のトレンドを含む多元宇宙を背景とした gaudigeekatintel の肖像画）

追加のファインチューニングとプロンプトの例

プロンプト：create a captivating book cover of Patatintel as an astronaut in space, face not covered, bring illuminating large planet, dramatic lighting, 4k, trending in art station, sharp focus, flawless skin, photorealistic, dreamart, hd
（宇宙飛行士 Patatintel の魅惑的な本の表紙を作成、顔は覆われていない、照らされた大きな惑星、ドラマチックな照明、4K、ArtStation のトレンド、鮮やか、完璧な肌、フォトリアリスティック、幻想的、HD）

プロンプト：photorealistic close-up portrait of solo Christophatintel with suit, futuristic cityscape dominated with skyscraper with electronic gadgets, dramatic lighting, trending in art station
（スーツを着た Christophatintel の写真のようなクローズアップの肖像画、電子機器、ドラマチックな照明、ArtStation のトレンドを含む高層ビルが目立つ未来的な都市の風景）

プロンプト：portrait of solo Sriatintel in the interstellar space, dressed in the space suit from the movie interstellar, black hole, moons, universe, make it look like a movie poster, dramatic lighting, trending in art station
（星間空間にいる Sriatintel の肖像画、映画「インターステラー」の宇宙服を着ている、ブラックホール、月、宇宙、映画のポスターのように見せる、ドラマチックな照明、ArtStation のトレンド）

このチュートリアルは、インテル® Tiber™ デベロッパー・クラウド[1] または AWS* のインテル® Gaudi® AI アクセラレーター・インスタンスで試すことができます。詳細は、インテルの AI フレームワーク（英語）サポートまでお問い合わせください。

[1] 旧インテル® デベロッパー・クラウド

[**訳者注：**2024 年 8 月現在、インテル® Tiber™ デベロッパー・クラウドでは、インテル® Gaudi® 2 AI アクセラレーターへのアクセスを Enterprise ELA プランの利用者向けに承認制で提供しています。]

Transformers による GGUF モデルの高速化

インテル® プラットフォームでのパフォーマンスとメモリー使用量の改善

Bo Dong、Jun Lin、Zhentao Yu、Zhenzhong Xu、Yu Luo、Hanwen Chang、Haihao Shen
インテル コーポレーション

GGUF（GPT-Generated Unified Format、GPT によって生成された統一形式）は、ファイル内のテンソルとメタデータを素早く検査できる新しいバイナリー形式です（**図 1**）。これは言語モデルファイル形式の大きな飛躍であり、GPT などの大規模言語モデル（LLM）の保存と処理の効率を最適化します。PyTorch* モデルを GGUF 形式に変換するのは簡単です。

図 1. GGUF 形式（出典（英語））

Hugging Face* Transformers は最近、Transformers PR (英語)で GGUF をサポートしました。Transformers は、PyTorch* を使用して推論を実行する前に、GGUF モデルを FP32 に逆量子化します。使い方は簡単で、**図 2** に示すように、`from_pretrained` で `gguf_file` パラメーターを指定するだけです。

```
from transformers import AutoTokenizer, TextStreamer, AutoModelForCausalLM

model_name = "TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF"
gguf_file = "tinyllama-1.1b-chat-v1.0.Q4_0.gguf"
model = AutoModelForCausalLM.from_pretrained(model_name, gguf_file = gguf_file)
```

図 2. Hugging Face* Transformers での GGUF モデルの使用

Transformers 向けインテル® エクステンション（英語）は低ビットの LLM 推論を高速化します。Hugging Face* Transformers を拡張し、インテル® プラットフォーム上でパフォーマンスを向上します。幅広いモデルで GGUF 推論をサポートしており、使い方も簡単です（**図 3**）。

```
from intel_extension_for_transformers.transformers import AutoModelForCausalLM
from transformers import AutoTokenizer, TextStreamer


model_name = "TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF"
gguf_file = "tinyllama-1.1b-chat-v1.0.Q4_0.gguf"
model = AutoModelForCausalLM.from_pretrained(model_name, gguf_file = gguf_file)
```

図 3. Transformers 向けインテル® エクステンション**（英語）での GGUF モデルの使用**

現在、Transformers は Llama* や Mistral* など、50 を超える一般的な LLM をサポートしています（**表 1**）。

	LLM	中国語モデル	コーディング・モデル	音声モデル
モデル	Meta-Llama-3-8B-Instruct, TinyLlama-1.1B, LLaMA2-tB, LLaMA2-13B, LLaMA2-70B, LLaMA-7B, LLaMA-13B, , Solar-10.7B, Mistral-7B, Mistral-7B-Instruct-v0.2, Mixtral-8x7B, , GPT-J-6B, GPT-NeoX-20B, Dolly-v2-3B, MPT-7B, MPT-30B, Falcon-7B, Falcon-40B, BLOOM-7B, OPT-125m, OPT-1.3B, OPT-13B, , phi-2, phi-1_5 phi-1, phi-3-128k, phi-3-48k, StableLM-2-1_6B, StableLM-3B, StableLM-2-12B, gemma-2b-it , gemma-7b, Neural-Chat-7B-v3-1, Neural-Chat-7B-v3-2,	Qwen-7B, Qwen-14B, Qwen1.5-7B, Qwen1.5-0.5B, ChatGLM-6B, ChatGLM2-6B, ChatGLM3-6B, Baichuan-13B-Chat,Baichuan2-13B-Chat,Baichuan2-7B-Chat	Magicoder-6.7B, StarCoder-1B, StarCoder-3B, StarCoder-15.5B, CodeLlama-7b	Whisper-tiny, Whisper-base Whisper-small Whisper-medium Whisper-large

表 1. サポートされる LLM

セットアップは簡単です。最初に、Neural Speed をインストールします。

```
git clone https://github.com/intel/neural-speed.git
cd neural_speed
python setup.py install
```

次に、Transformers 向けインテル® エクステンション（英語）をインストールします。

```
git clone https://github.com/intel/intel-extension-for-transformers.git
cd intel_extension_for_transformers
python setup.py install
```

最後に、Transformers をインストールします。

```
pip install transformers
```

図 4 のコードを使用して簡単にテキストを生成できます。

```python
from transformers import AutoTokenizer, TextStreamer
from intel_extension_for_transformers.transformers import AutoModelForCausalLM

# Specify the GGUF repo on the Hugginface
model_name = "TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF"
# Download the the specific gguf model file from the above repo
gguf_file = "tinyllama-1.1b-chat-v1.0.Q4_0.gguf"
# make sure you are granted to access this model on the Huggingface.
tokenizer_name = "TinyLlama/TinyLlama-1.1B-Chat-v1.0"
prompt = "Once upon a time, there existed a little girl,"
tokenizer = AutoTokenizer.from_pretrained(tokenizer_name, trust_remote_code=True)
inputs = tokenizer(prompt, return_tensors="pt").input_ids
streamer = TextStreamer(tokenizer)
model = AutoModelForCausalLM.from_pretrained(model_name, gguf_file = gguf_file)
outputs = model.generate(inputs, streamer=streamer, max_new_tokens=300)
```

図 4. テキストを生成するサンプルコード

インテル® Xeon® Platinum プロセッサー / インテル® Core™ Ultra 5 プロセッサー 125H ベースのシステム（最大 4400MHz、最小 400MHz、合計メモリー 32GB（8 x 4GB LPDDR5、[7467 MT/s]）、Ubuntu* 13.2.0–9ubuntu1）で Transformers のパフォーマンス比較を実施しました。メモリーが小さいため、比較は TinyLlama-1.1B-Chat（英語）に限定されました。Transformers 向けインテル® エクステンションは、Transformers と比較して優れたレイテンシーとメモリー使用量を提供しました（**表 2**）。

TinyLlama in=32, out=32	最初のトークンのレイテンシー	次のトークンのレイテンシー	メモリー使用量	Lambada 精度
Transformers 向けインテル® エクステンション	275ms	12.15ms	641MB	59.07
Hugging Face* Transformers	302ms	88.7ms	4416MB	58.9

表 2. パフォーマンス、メモリー使用量、精度の比較

Transformers 向けインテル® エクステンションは、わずかなコード変更で、メモリー制約のあるシステムでトークンのレイテンシーを最小限に抑える優れた方法です。カーネルは 1 ~ 8 ビットをサポートします。StreamingLLM や Tensor Parallelism などのより高度な機能もあるため、Transformers 向けインテル® エクステンション（英語）を確認することを推奨します。

intel. software

私の見たところ...

コードに改良の余地があるようです

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、コードのパフォーマンスを強化しましょう。オープンな標準ベースの oneAPI は、クロスアーキテクチャーのライブラリー、コンパイラー、ツールを提供し、コードをより多くのハードウェアに対応させ、優れたパフォーマンスを実現します。

oneAPI

oneAPI を活用して、エンタープライズ、クラウド、HPC、AI などの分野の計算を高速化。

今すぐダウンロード

llama.cpp を使用したインテル® GPU 上での LLM 実行

新しい SYCL* バックエンドを活用する

Zhang Jianyu、Meng Hengyu、Hu Ying、Luo Yu,、Duan Xiaoping、Majumder Abhilash
インテル コーポレーション

オープンソース・プロジェクトの llama.cpp（英語）は、人気が高まっている軽量の LLM フレームワークです。高いパフォーマンスとカスタマイズ性により、開発者、研究者、愛好家の活気に満ちたダイナミックなコミュニティーに成長しました。開始から約 1 年が経過した現在、GitHub* プロジェクトには 600 人以上の貢献者、52,000 個のスター、1,500 個のリリース、7,400 個のフォークがあります。最近のコードマージにより、サーバー向けおよびコンシューマー向けのインテル® GPU を含む、より多くのハードウェアがサポートされました。インテル® GPU のほかに、CPU（x86 および ARM）とその他のベンダーの GPU がサポートされています。

オリジナルの実装は Georgi Gerganov 氏によって作成されました。このプロジェクトは主に教育を目的としており、マシンラーニング用のテンソル・ライブラリーである ggml ライブラリー（英語）の新機能を開発するためのものです。

最近の更新により、インテルはより多くのデバイスで推論を可能にし、「あらゆるところに AI」をより多くのユーザーに提供しています。llama.cpp は C で記述されているため高速であり、ほかにも魅力的な機能を備えています。

- 16 ビット浮動小数点サポート
- 整数量子化サポート（4 ビット、5 ビット、8 ビットなど）
- サードパーティーの依存関係なし
- 実行時のメモリー割り当てなし

インテル® GPU 向けの SYCL* バックエンド

ggml には、さまざまなハードウェアをサポートして最適化するいくつかのバックエンドがあります。SYCL* バックエンドの開発には、さまざまなベンダーの GPU をサポートする oneAPI（英語）の SYCL*（ダイレクト・プログラミング言語）と oneMKL（高性能 BLAS ライブラリー）が利用されています。SYCL* は、ハードウェア・アクセラレーターの生産性を向上させるプログラミング・モデルです。これは、純粋な C++17 をベースとした、単一ソースの組込みのドメイン固有言語です。

SYCL* バックエンドは、すべてのインテル® GPU をサポートしており、次のデバイスで検証済みです。

- インテル® データセンター GPU マックス・シリーズおよびインテル® データセンター GPU フレックス・シリーズ
- インテル® Arc™ ディスクリート GPU
- インテル® Core™ Ultra プロセッサーに内蔵されたインテル® Arc™ GPU
- 第 11、12、13 世代インテル® Core™ プロセッサーに内蔵された iGPU

llama.cpp がインテル® GPU をサポートしたことで、何百万ものコンシューマー・デバイスが Llama* で推論を実行できるようになりました。OpenCL*（CLBlast）バックエンドと比較すると、SYCL* バックエンドはインテル® GPU でのパフォーマンスが大幅に向上しています。また、将来的には CPU や AI アクセラレーターを搭載したほかのプロセッサーなど、より多くのデバイスもサポートされる予定です。SYCL* バックエンドの使用方法については、「llama.cpp for SYCL」（英語）を参照してください。

SYCL* バックエンドを使用してインテル® GPU 上で LLM を実行する

詳細なガイドは、「llama.cpp for SYCL」（英語）を参照してください。SYCL* と oneAPI でサポートされているすべてのインテル® GPU で実行できます。サーバーおよびクラウドユーザーは、インテル® データセンター GPU マックス・シリーズおよびインテル® データセンター GPU フレックス・シリーズで実行できます。クライアント・ユーザーは、インテル® Arc™ GPU またはインテル® Core™ プロセッサーに内蔵されている iGPU で試すことができます。第 11 世代インテル® Core™ プロセッサー以降に内蔵されている iGPU をテストしました。古い iGPU でも動作しますが、パフォーマンスは低くなります。

唯一の制約はメモリーです。iGPU はホストの共有メモリーを使用し、dGPU は独自のメモリーを使用します。llama2–7b-Q4 モデルでは、80 以上の EU（第 11 世代インテル® Core™ プロセッサー以降）を搭載した iGPU を使用し、4.5GB を超える共有メモリーを有効にすることを推奨します（ホストメモリーの合計は 16GB 以上で、メモリーの半分を iGPU に割り当てることができます）。

インテル® GPU ドライバーのインストール

Linux* と Windows*（WSL 2）の両方がサポートされています。Linux* の場合、開発とテストに使用された Ubuntu* 22.04 を推奨します。

Linux*:

```
sudo usermod -aG render username
sudo usermod -aG video username
sudo apt install clinfo
sudo clinfo -l
```

出力例 :

```
Platform #0: Intel(R) OpenCL Graphics -- Device #0: Intel(R) Arc(TM) A770 Graphics
```

または

```
Platform #0: Intel(R) OpenCL HD Graphics -- Device #0: Intel(R) Iris(R) Xe Graphics \
[0x9a49\]
```

Windows* :「インテル® GPU ドライバーのインストール」を参照してください。

oneAPI ランタイムを有効にする

最初に、インテル® oneAPI ベース・ツールキットをインストールして、SYCL* コンパイラーと oneMKL を入手します。次に、oneAPI ランタイムを有効にします。

- Linux* : `source /opt/intel/oneapi/setvars.sh`
- Windows* : `"C:\Program Files (x86)\Intel\oneAPI\setvars.bat" intel64`

`sycl-ls` を実行して、レベルゼロデバイスが 1 つ以上存在することを確認します。`[ext_oneapi_level_zero:gpu:0]` のように、少なくとも 1 つの GPU が存在することを確認します。

ワンクリックでビルドします。

- Linux* : `./examples/sycl/build.sh`
- Windows* : `examples\sycl\win-build-sycl.bat`

上記のスクリプトには、oneAPI ランタイムを有効にするコマンドが含まれています。

ワンクリックでサンプルを実行する

llama-2–7b.Q4_0.gguf（英語）をダウンロードして、`models` フォルダーに保存します。

- Linux* : `./examples/sycl/run-llama2.sh`
- Windows* : `examples\sycl\win-run-llama2.bat`

上記のスクリプトには、oneAPI ランタイムを有効にするコマンドが含まれています。レベルゼロ GPU の ID が `0` でない場合は、スクリプト内のデバイス ID を変更してください。デバイス ID は以下のコマンドで確認できます。

- Linux* : `./build/bin/ls-sycl-device または ./build/bin/main`
- Windows* : `build\bin\ls-sycl-device.exe または build\bin\main.exe`

まとめ

llama.cpp の SYCL* バックエンドは、LLM 開発者とユーザーがすべてのインテル® GPU を利用できるようにします。インテルのラップトップの場合は iGPU が、ゲーミング PC の場合はインテル® Arc™ GPU が、クラウド VM の場合はインテル® データセンター GPU マックス・シリーズまたはインテル® データセンター GPU フレックス・シリーズが搭載されているかを確認してください。そして搭載されている場合は、インテル® GPU で llama.cpp による LLM の魔法のような機能をお楽しみください。インテル® GPU でさらに多くの機能と最適化を利用できるようにするため、SYCL* バックエンドを試して貢献してください。llama.cpp プロジェクトから、クロスプラットフォーム開発向けの oneAPI プログラミング・モデルを学ぶことができます。

Python* と C++ 分析によるシミュレーションと後方伝播の高速化

言語間の障壁を超えるグラフ・コンパイラー

Dmitri Goloubentsev MatLogica 最高技術責任者

AADC（Algorithmic Adjoint Differentiation Compiler、自動随伴微分コンパイラー）は、計算最適化の最前線に立っています。元々は、計算集約型タスクを効率良く処理できることから C++ 向けに設計されましたが、現在では当初の目的を超えて、ハイパフォーマンス・コンピューティングを必要とするシミュレーションや推論タスクなど、幅広いアプリケーションを高速化しています。演算子のオーバーロードを使用して有向非巡回グラフ（DAG）を効果的に抽出する AADC は、計算グラフが膨大な数のノードで構成される環境で優れた性能を発揮します。高密度テンソル演算を処理する一般的なマシンラーニング（ML）やディープ・ニューラル・ネットワーク・フレームワークとは異なり、AADC は主にスカラー演算で構成される非常に大規模な計算グラフを迅速にコンパイルすることに特化しており、複雑で大量の計算を迅速に処理する必要があるドメインに不可欠なツールです。

特に Python* では、主に ML およびディープラーニング（DL）アプリケーションのニーズによって計算ツールが形成されてきました。これらのアプリケーションは、テンソル演算の処理に優れたフレームワークを中心としています。代表的な例として、ニューラル・ネットワーク計算を処理する TensorFlow* や PyTorch* などがあります。ただし、主要なドメイン以外に適用する場合、これらのフレームワークには固有の制限があります。その 1 つは、メモリー内に完全な DAG を格納する必要があることです。

多くの計算タスク、特に金融におけるシミュレーションや随伴微分法を含むタスクでは、大きなグラフを素早くコンパイルする能力がパフォーマンスに直接影響します。これらのコンテキストでは、ジャストインタイム（JIT）コンパイルが重要になりますが、JIT の利点は、コンパイルが高速でなければ十分に発揮されません。コンパイルが遅いと、JIT によってもたらされるはずのランタイム・パフォーマンスの向上が打ち消される可能性があります。

AADC アーキテクチャー

AADC は、ストリーミング・グラフ・コンパイラー・フレームワークを実装することで、高性能な計算タスクの厳しいニーズを満たすように設計されています。この革新的なアーキテクチャーは、ユーザープログラムの実行時に x64 インテル® AVX2/ インテル® AVX-512 マシンコード命令を動的に生成することで機能します。このメカニズムは、ベクトル化されたカーネルをコンパイルして、入力から出力までユーザー関数を複製するだけでなく、随伴または後方伝播も管理します。これにより、AADC は小さなメモリー・フットプリントを維持しながら、高いパフォーマンスを実現できます。**図 1** はこれを図解したものです。

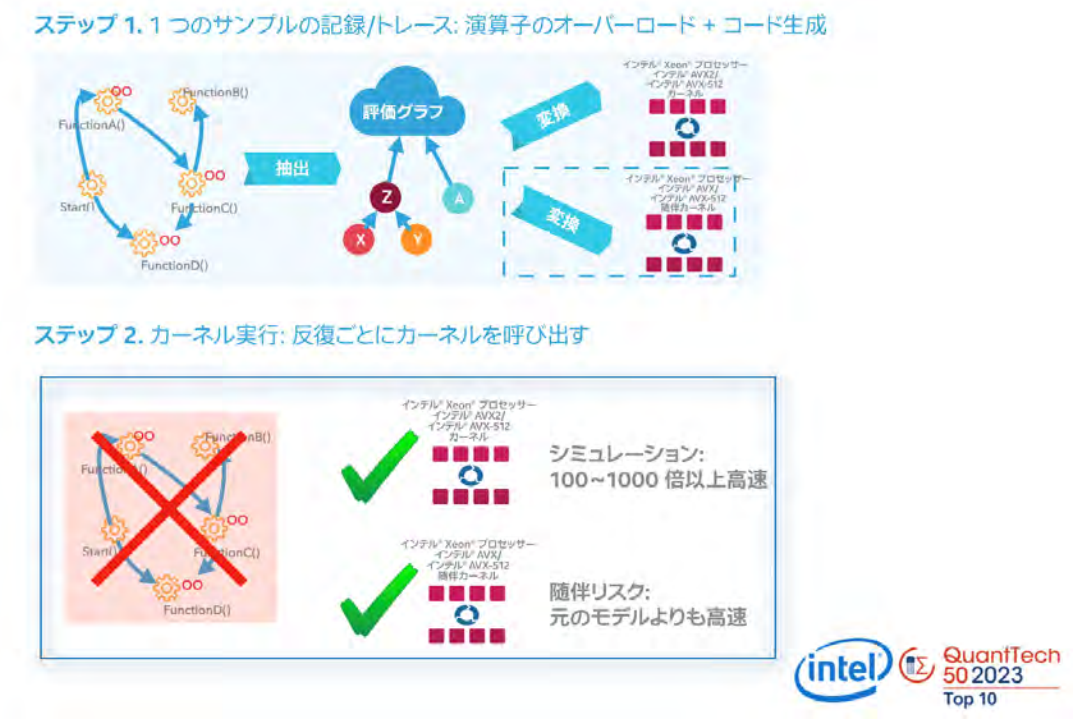


図 1. AADC

AADC 設計の中核は、基本操作を直接トレースし、プログラムの実行時にマシンレベルの命令に再コンパイルする機能です。このプロセスにより、高レベルのプログラミング言語の非効率性が排除されます。操作を x64 マシンコードに直接変換することで、AADC はマルチスレッド、インテル® AVX2、インテル® AVX-512 などの最新のプロセッサー機能を活用して、計算タスクを最大限の効率で実行できます。

AADC の重要なアーキテクチャー上の利点は、高度なコード折り畳みおよび圧縮技術です。従来の自動微分ツールは、大量の中間データを格納する必要があるため、メモリー使用量が大きくなりがちです。これに対し、AADC は高度なアルゴリズムを使用して、計算グラフと中間結果の格納に必要なメモリーを最小限に抑えます。これは、コードをインテリジェントに折り畳み、圧縮することで実現され、メモリー要件が軽減されるだけでなく、計算中に処理および移動するデータ量が軽減されるため、実行速度が向上します。

AADC は、C++ と Python* の両方の分析で計算をトレースする「アクティブタイプ」を採用し、さまざまなプログラミング環境間で計算操作の一貫した正確な記録を維持します。これにより、すべての計算が、言語に関係なくキャプチャーされ、最適化されます。ユーザーは、トレースをトリガーし、入力データの 1 つのインスタンスで JIT コンパイルを使用することで、このプロセスを開始します。コンパイルされた DAG は、複数のサンプルを処理するカーネルとして機能し、高レベルのプログラミング構造のオーバーヘッドなしで、低レベルのマシンコード速度で実行されます。

ケーススタディー：XVA パフォーマンスの強化

信用評価調整（XVA）計算は、金融市場における信用リスクと評価の管理に不可欠です。これらの計算はリソースを大量に消費することで有名で、リスクと価値を正確に評価するには堅牢なシミュレーションと価格設定モデルが必要です。このデモでは、MatLogica AADC を XVA シミュレーション・フレームワークに統合することで実現されるパフォーマンスの向上を示します。問題の XVA 計算では、Python* と C++ の組み合わせが使用され、Python* はオーケストレーション・レイヤーとして機能し、QuantLib ライブラリー（英語）をバックエンドとしてシミュレーション・モデルを価格設定関数に接続します。

元の Python* コードは必要な機能を備えていますが、ベクトルまたはマルチコア実行に対応していません。その結果、パフォーマンスは最適ではなく、リスク計算なしで 1 回の評価の実行を完了するのに約 35 秒かかります。これは、迅速な再計算や大規模なシミュレーションを必要とするシナリオでは実用的ではありません。AADC を組み込むことで、同じ計算タスクが大幅に改善されます。AADC は、既存の Python* および C++ コードから単一のモンテカルロ・パスの DAG を抽出し、インテル® AVX2/ インテル® AVX-512 とマルチスレッドを活用するカーネルを作成します。

AADC を使用すると、評価だけでなくすべての 1 次リスク感度を含め、XVA シミュレーションの実行時間が 35 秒から 1 秒へと大幅に短縮されます。比較については、xVA-QL-Original（英語）と xVA-QL-Example（英語）Jupyter* ノートブックを参照してください。

インテル® AVX-512 命令とマルチスレッドを使用すると、パフォーマンスがさらに向上し、シミュレーションの効率がさらに高まります（**図 2**）。これらの機能強化は、スピードと精度が意思決定と財務結果に直接影響する金融分野では非常に重要です。

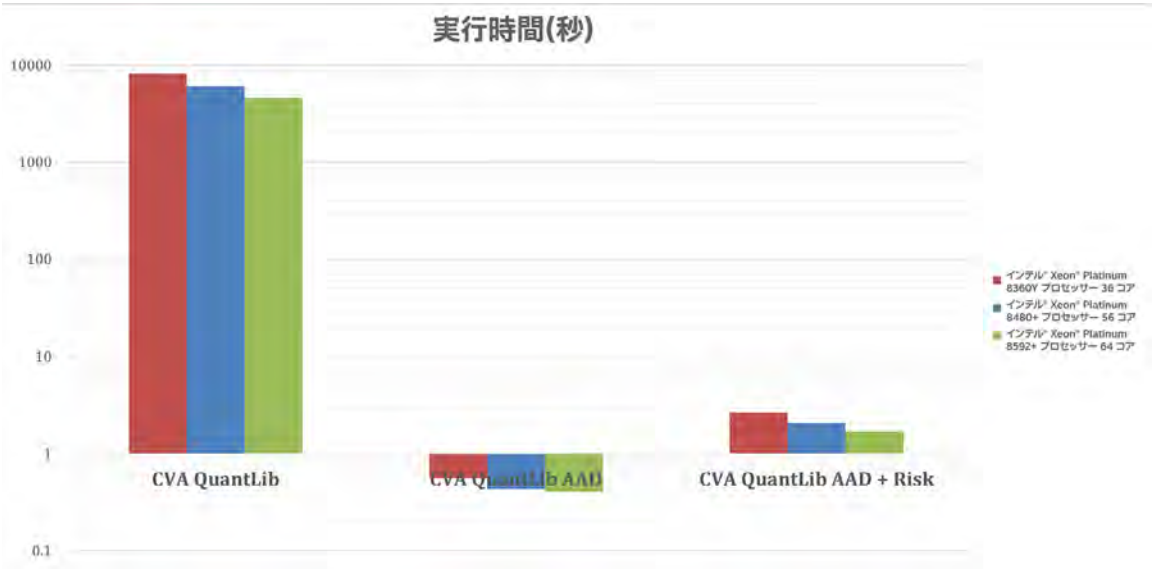


図 2. パフォーマンスの向上

C++ ライブラリーのインストルメンテーション

AADC を利用する主な技術要件の 1 つは、ユーザープログラムをトレースして DAG を抽出するのに不可欠な「アクティブ」な `idouble` 型を使用するため、基礎となる C++ ライブラリーをインストルメンテーションすることです。これは通常、単純な検索と置換操作で実現できますが、多くの場合、煩わしい C++ コンパイルエラーが発生します。これらのエラーは管理可能であり、スムーズな統合と操作を確実にするには対処する必要があります。このプロセスを容易にするため、AADC には、必要な修正を自動的に行うように設計された LLVM/Clang ベースのツールが含まれています。このツールは、変更プロセスを合理化し、必要な手作業を減らし、インストルメンテーション・フェーズでエラーが発生する可能性を最小限に抑えるのに役立ちます。

コード内の分岐の処理

AADC の記録 / 再生パターンが正しく機能するには、コード内の分岐がミスしないことが不可欠です。分岐ミスは、記録された計算に不整合をもたらし、任意の入力に対して記録された実行を適用する能力を制限します。この設定は線形問題では機能しますが、条件文を含むアルゴリズムでは困難な場合があります。

実際には、`if (x < K) return 0; else return (x-K);` などの文は、条件分岐を排除するため `max(0, x-K)` に変換する必要があります。AADC は、すべての分岐ミスを追跡し、記録に分岐ミスがない場合は一貫性があると見なされ、任意の入力に対して安全に使用できます。さらに、AADC は、ユーザーコード内で必要な変更の正確な場所を特定できるため、開発者はコードを最適化してコンパイラーとの互換性を高めることができます。

QuantLib およびほかのライブラリーとの互換性

オープンソースの QuantLib ライブラリーを使用したデモでは、AADC の記録要件と互換性を持たせるため修正が必要でした。しかし、QuantLib 内のすべてのインストルメント・タイプが現在サポートされているわけではなく、事前の修正なしに AADC を直接適用できる範囲には限界があります。これは、引き続きより多くの分析を完全微分可能なコードと互換性のある形式に変換する必要があることを示しています。

まとめ

ここで紹介したケーススタディーは、ベクトル化と効率的なコンパイル手法、および随伴微分法を活用することで、AADC がいかに計算金融タスクを変革できるかを明確に示しています。実行時間の劇的な短縮は、より複雑な分析を迅速に実行する能力とともに、AADC を金融機関や計量アナリストの重要な武器として位置付けています（**図 3**）。この実用的なデモは、AADC の能力を明確に示すだけでなく、金融工学やそれ以外の分野でパフォーマンス・ベンチマークを再定義する可能性を示しています。

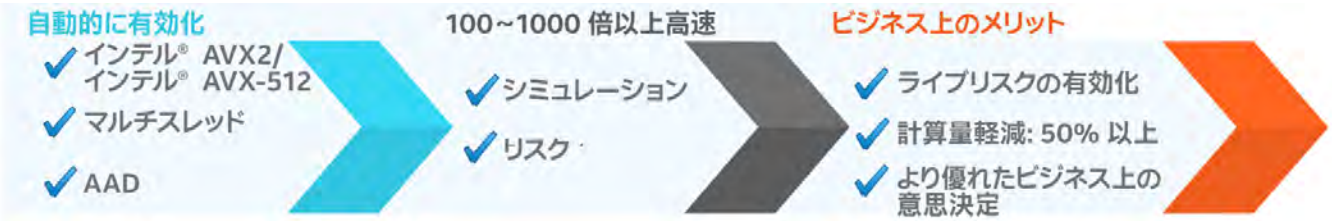
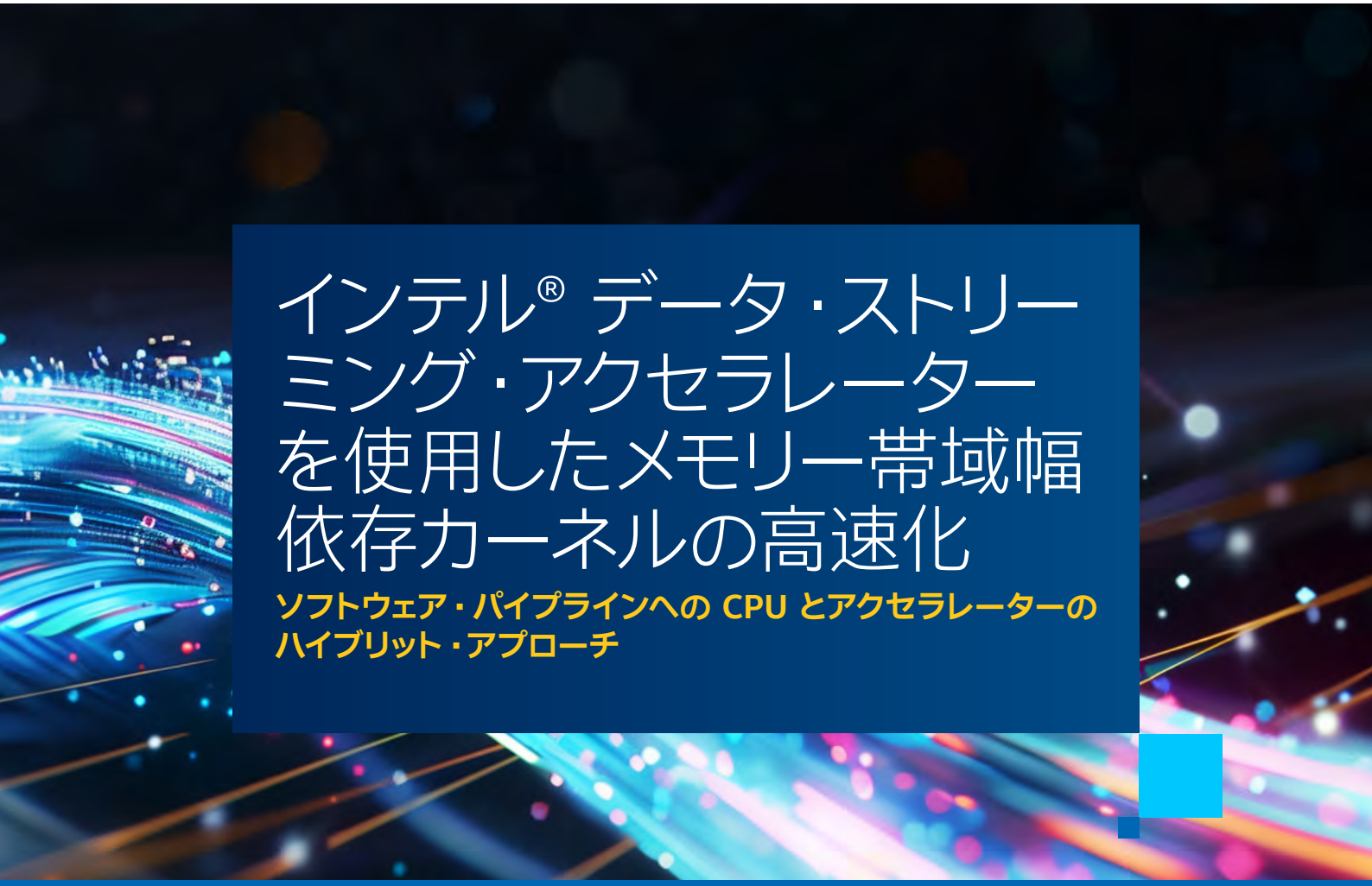


図 3. 技術的な側面からビジネス上のメリットまで



インテル® データ・ストリーミング・アクセラレーターを使用したメモリー帯域幅依存カーネルの高速化

ソフトウェア・パイプラインへの CPU とアクセラレーターのハイブリット・アプローチ

Vamsi Sripathi インテル コーポレーション ソフトウェア・イネーブリング & 最適化エンジニア

インテル® データ・ストリーミング・アクセラレーター（インテル® DSA）は、第 4 世代インテル® Xeon® スケーラブル・プロセッサー以降で利用できる、高性能なデータコピーおよび変換アクセラレーターです。メモリーコピー、比較、フィルなどのさまざまなデータ操作をサポートします。インテル® DSA は、作業キューと記述子を使用して CPU ソフトウェアと対話します。記述子には、操作のタイプ、ソースアドレスとデスティネーション・アドレス、データ長など、目的の操作に関する情報が含まれています。詳細については、The Parallel Universe 第 53 号の「ダイレクト・メモリー・アクセスの先へ：インテル® データ・ストリーミング・アクセラレーターによるデータセンター・コストの削減」を参照してください。

多くの科学および商用アプリケーションは、最適なパフォーマンスを実現するため、高性能な CPU と高いメモリー帯域幅を必要とします。CPU は世代を重ねるごとに、コアの追加、SIMD 幅の増加、新しい ISA 拡張などのマイクロアーキテクチャー機能によって性能が大幅に向上していますが、DRAM 帯域幅の向上はこれまで CPU の改良に追従できていませんでした。DRAM からデータをフェッチする速度が CPU 実行ユニットのストールを防ぐのに十分ではないため、実際のアプリケーションでは期待される性能向上が得られない「メモリーウォール」と呼ばれる現象に長年悩まされてきました。

CPU には、ロード / ストアキューとスーパーキュー形式のアーキテクチャー・バッファーと、メモリーウォールの効果を軽減するハードウェア・プリフェッチャーがありますが、各 CPU コアがハードウェア・キューに保持できるエントリーの数は限られているため、コア数が少ない場合はメモリー帯域幅への影響は限定的です。DRAM からデータをフェッチする必要がある場合、各メモリー要求が DRAM レイテンシーによりバッファースロットを長時間占有するため、キューがボトルネックになります。この対策として、利用可能な DRAM 帯域幅を完全に飽和させるため、より多くの CPU コアを使用して同時にメモリーアクセス要求を生成するのが一般的です。ただし、コア数が少ない状態で高いメモリー帯域幅を実現することは依然として困難です。

この記事では、インテル® DSA の長所を CPU と組み合わせて補完し、メモリー依存カーネル（つまり、メモリー・サブシステムが演算のデータオペランドを CPU コアに供給できる速度によってパフォーマンスが決まる操作）を高速化する手法について説明します。

パフォーマンスの測定には、標準の STREAM ベンチマーク（英語）を使用し、CPU のメモリー・パフォーマンスの特性を把握します。STREAM は、Copy、Scale、Add、Triad の 4 つのカーネルで構成されており（**表 1**）、これらのカーネルはすべてメモリー依存です。

カーネル	演算	リードバイト数（キャッシュをバイパスするストアを使用）	ライトバイト数	FLOPS
COPY	A[i] = B[i]	8	8	0
SCALE	A[i] = scalar × B[i]	8	8	1
ADD	C[i] = A[i] + B[i]	16	8	1
TRIAD	C[i] = A[i] + scalar × B[i]	16	8	2

表 1. STREAM カーネルの特性

インテル® DSA は強力なデータコピー / 変換エンジンとして機能しますが、データオペランドに対する乗算、加算、積和演算（FMA : Fused Multiply-Add）などの算術演算の実行はサポートしていません。そのため、インテル® DSA だけでは、メモリーと演算が混在するアプリケーション・カーネルを実行できません。ただし、インテル® DSA は、演算によって生成されるデスティネーション・データの場所（DRAM または CPU 上の最終レベルキャッシュ（LLC））を制御する独自の機能を提供します。例えば、DRAM にあるソースバッファーから、DRAM（CPU キャッシュをバイパス）または LLC にあるデスティネーションにデータをコピーできます。

インテル® DSA のキャッシュ書き込み機能を、DRAM から LLC へのプロキシー・ハードウェア・プリフェッチ・エンジンとして使用し、CPU コアで演算を処理できます。このソリューションは、インテル® DSA のデータ転送（DRAM から LLC へ）と、LLC から CPU レジスターへの非同期コピーによる CPU 計算を効率的にオーバーラップすることで機能します。デフォルトでは、インテル® DSA は LLC の約 14MB の部分に書き込むことができます（第 4 世代インテル® Xeon® スケーラブル・プロセッサー上の 15 ウェイの LLC のうち 2 ウェイ、有効サイズ = 2/15 * LLC のサイズ = 14MB）。**図 1** は、CPU のみのアプローチと CPU + インテル® DSA のハイブリッド・ワークフローの高レベルの違いを示しています。

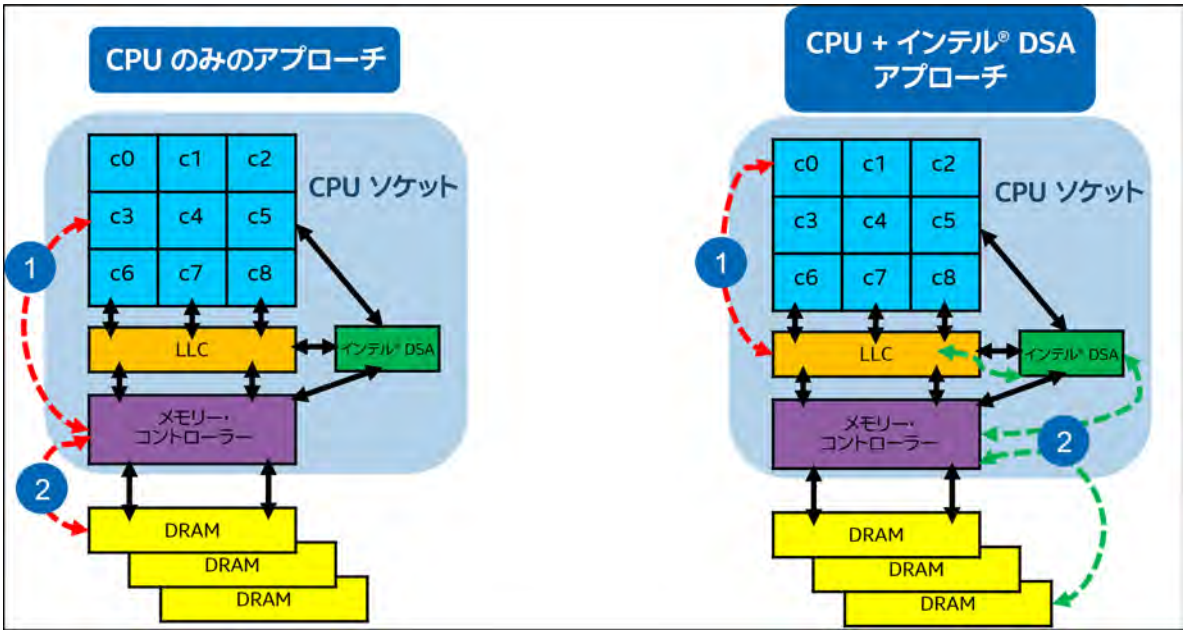


図 1. CPU のみのアプローチと CPU + インテル® DSA のハイブリッド実装の高レベルの違い

CPU のみのアプローチでは、ステップ 1 と 2 は、DRAM 上のデータに対する CPU コアからのロード / ストア要求を指し、メモリー・コントローラーに送られます。CPU + インテル® DSA アプローチのステップ 1 では、CPU は DRAM ではなく LLC からデータをロードします。ステップ 2 では、インテル® DSA は DRAM（メモリー・コントローラー経由）から LLC へのデータ転送を開始します。ステップ 1 と 2 はパイプライン化され、非同期的に実行されるため、CPU が T_x の時点で読み取る必要のあるデータは、$T_{(x-1)}$ でインテル® DSA によって LLC にコピー済みです。つまり、インテル® DSA が同時に次の反復のデータを DRAM からフェッチして LLC に書き込む一方で、CPU は現在の反復のデータを LLC から読み取って演算を実行します。ソフトウェア・パイプラインにより、すべてのインテル® DSA エンジンが効率良く使用されるようになります。

図 2 は、DRAM からデータ配列を読み取るなどの基本的な CPU 操作に対する CPU + インテル® DSA のハイブリッド・パイプライン・アプローチのワークフローを示しています。この例では、各 CPU スレッドに対して、キューの深さを 4 に設定し、各エントリーに入力バッファーの 1MB のデータを保持しています。つまり、CPU が LLC から 4MB を読み取っている間に、インテル® DSA は DRAM から次の 4MB のチャンクをフェッチします。

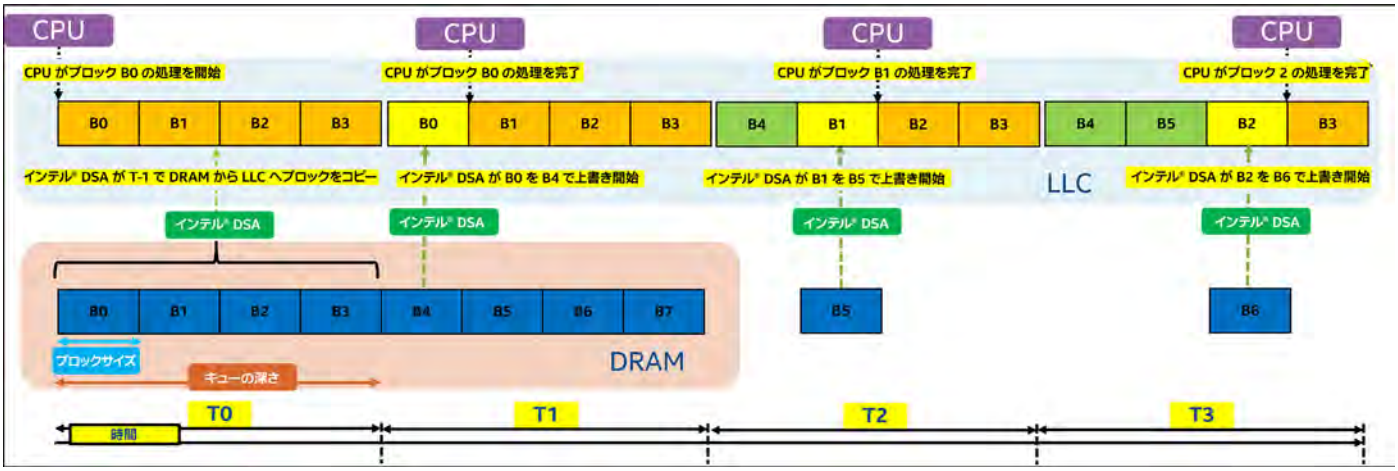


図 2. CPU + インテル® DSA のハイブリッド実装で 1 つの CPU コアで 1 つのバッファーにアクセスするデータ操作のパイプラインの例

インテル® Data Mover Library（インテル® DML）（英語）は、インテル® DSA を使用してデータ操作を実行する C/C++ API を提供するオープンソース・ライブラリーです。インテル® DML を使用して、インテル® DSA を初期化し、作業記述子を作成し、STREAM カーネルのジョブステータスを送信および照会します。**図 3** に例を示します。Triad 操作は OpenMP* スレッドを使用して並列化されます。各スレッドはデータバッファーを均等に分割し、キューのようなメカニズムを使用してスレッドチャンクをさらにブロックに分割します。各 CPU スレッドは、非同期コピー操作をインテル® DSA に送信し、完了するまで待機してからデータにアクセスして Triad 操作を計算します。ループの中で、非同期コピー操作は、将来の反復（`QUEUE_DEPTH`）でアクセスされるデータに対して送信されます。**図 4** は、インテル® DML を使用してデスティネーションを LLC（`DML_FLAG_PREFETCH_CACHE`）にする非同期コピー操作のコードを示しています。

```
void dsa_omp_triad(long long *p_n, double *p_src1, double *p_src2, double *p_dst,
                   double *p_tmp1, double *p_tmp2,
                   dml_job_t **p_dml_jobs_t1, dml_job_t **p_dml_jobs_t2)
{
  long long n = *p_n;

#pragma omp parallel
  {
    int nthrs = omp_get_num_threads();
    int ithr  = omp_get_thread_num();

    long long chunk = n/nthrs;
    long long tail  = n - (chunk*nthrs);
    long long start = ithr * chunk;
    if ((tail) && (ithr == nthrs-1)) {
      chunk += tail;
    }

    double *p_t_src1 = p_src1 + start;
    double *p_t_src2 = p_src2 + start;
    double *p_t_dst  = p_dst  + start;

    double *p_t_tmp1 = p_tmp1 + (ithr * ((BLK_SIZE_IN_BYTES*QUEUE_DEPTH)/sizeof(double)));
    double *p_t_tmp2 = p_tmp2 + (ithr * ((BLK_SIZE_IN_BYTES*QUEUE_DEPTH)/sizeof(double)));

    dml_job_t **p_t_dml_jobs_t1 = p_dml_jobs_t1 + (ithr*QUEUE_DEPTH);
    dml_job_t **p_t_dml_jobs_t2 = p_dml_jobs_t2 + (ithr*QUEUE_DEPTH);

    long long blk_elems = BLK_SIZE_IN_BYTES/sizeof(double);
    long long num_blks  = chunk/blk_elems;
    tail = chunk - (blk_elems*num_blks);

    for (int i=0; i<QUEUE_DEPTH; i++) {
      async_copy(&blk_elems, p_t_src1+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp1+(i*blk_elems), p_t_dml_jobs_t1[i]);
      async_copy(&blk_elems, p_t_src2+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp2+(i*blk_elems), p_t_dml_jobs_t2[i]);
    }

    long long tmp_elems = QUEUE_DEPTH * blk_elems;
    seq_triad(&tmp_elems, p_t_src1, p_t_src2, p_t_dst);

    int i;
    for (i=QUEUE_DEPTH; i<(num_blks-QUEUE_DEPTH); i++) {
      dml_wait_job(p_t_dml_jobs_t1[i%QUEUE_DEPTH]);
      dml_wait_job(p_t_dml_jobs_t2[i%QUEUE_DEPTH]);

      seq_triad(&blk_elems, p_t_tmp1+((i%QUEUE_DEPTH)*blk_elems), p_t_tmp2+((i%QUEUE_DEPTH)*blk_elems), p_t_dst+(i*blk_elems));

      async_copy(&blk_elems, p_t_src1+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp1+((i%QUEUE_DEPTH)*blk_elems), p_t_dml_jobs_t1[i%QUEUE_DEPTH]);
      async_copy(&blk_elems, p_t_src2+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp2+((i%QUEUE_DEPTH)*blk_elems), p_t_dml_jobs_t2[i%QUEUE_DEPTH]);
    }

    for (int j=i; j<(i+QUEUE_DEPTH); j++) {
      dml_wait_job(p_t_dml_jobs_t1[j%QUEUE_DEPTH]);
      dml_wait_job(p_t_dml_jobs_t2[j%QUEUE_DEPTH]);
      seq_triad(&blk_elems, p_t_tmp1+((j%QUEUE_DEPTH)*blk_elems), p_t_tmp2+((j%QUEUE_DEPTH)*blk_elems), p_t_dst+(j*blk_elems));
    }

    if (tail) {
      seq_triad(&tail, p_t_src1+(num_blks*blk_elems),  p_t_src2+(num_blks*blk_elems),  p_t_dst+(num_blks*blk_elems));
    }
  }
}
```

図 3. OpenMP* とインテル® DML を使用した STREAM Triad カーネルの CPU + インテル® DSA のハイブリッド実装

```
void async_copy(long long *p_n, double *p_src, double *p_dst, dml_job_t *p_dml_job)
{
  dml_status_t status;

  p_dml_job->operation              = DML_OP_MEM_MOVE;
  p_dml_job->flags                  = DML_FLAG_COPY_ONLY|DML_FLAG_PREFETCH_CACHE;
  p_dml_job->source_first_ptr       = (void *)p_src;
  p_dml_job->destination_first_ptr  = (void *)p_dst;
  p_dml_job->source_length          = (*p_n)*sizeof(double);
  p_dml_job->destination_length     = (*p_n)*sizeof(double);

  status = dml_submit_job(p_dml_job);

  if (status) {
    printf ("\tdml_submit_job status failed, status = %u\n", status); fflush(0);
  }
}
```

図 4. インテル® DML を使用した LLC への非同期コピー

標準の STREAM カーネルに加えて、多くのアプリケーション・ドメインでよく使用されるベクトルドット積演算（`res += a[i] × b[i];` 読み取りのみ、書き込みなし）のベンチマークも行いました。カーネルのベンチマークは次のように行いました。

- 各入力バッファーのサイズは 1GB、Scale と Dot のメモリー・フットプリントは 2GB、Triad は 3GB にしました。各カーネルを 100 回実行して、最高のパフォーマンスを結果としました。
- CPU のみの実装では、CPU コアで OpenMP* により並列化された操作を実行しました。キャッシュをバイパスするストア / 非テンポラルなストア（`vmovntpd`）を使用しました。
- インテル® DSA + CPU 実装では、インテル® DSA を使用して DRAM から LLC に入力バッファーをフェッチしました。CPU による非テンポラルなストアを使用して、メモリー内のデスティネーション・バッファーを直接更新しました。

図 5 は、第 4 世代インテル® Xeon® スケーラブル・プロセッサー（56 コア、8 チャネル DDR5@4800MT/s、理論上のピーク帯域幅は 8 チャネル x 8 バイト x 4.8GT/s = 307GB/s）のさまざまなコア数でのパフォーマンスを示しています。CPU のみの実装と比較したインテル® DSA + CPU のスピードアップは、ヒートマップで表示しています。

CPU コア数	Triad、リード 2:ライト 1 (GB/s)			ドット積、100% リード (GB/s)			Scale、リード 1:ライト 1 (GB/s)		
	CPU のみ	インテル® DSA + CPU	スピードアップ	CPU のみ	インテル® DSA + CPU	スピードアップ	CPU のみ	インテル® DSA + CPU	スピードアップ
1	20.09	37.23	1.85	15.78	32.55	2.06	21.13	33.51	1.59
2	39.8	71.97	1.81	31.49	65.48	2.08	41.71	66.37	1.59
3	57.83	98.46	1.70	46.23	82.31	1.78	60.62	96.89	1.60
4	74.3	115.77	1.56	60.55	99.8	1.65	77.84	121.48	1.56
5	91.94	133.41	1.45	74.72	120.74	1.62	96.02	137.47	1.43
6	108.56	154.55	1.42	89.92	129.4	1.44	112.36	160.99	1.43
7	124.21	173.53	1.40	104.82	141.94	1.35	127.46	180.40	1.42
8	138.96	189	1.36	118.11	156.85	1.33	140.70	198.91	1.41
9	153.02	189.66	1.24	131.33	169.62	1.29	151.79	214.96	1.42
10	167.11	191.17	1.14	144.75	187.08	1.29	162.44	223.37	1.38
11	176.66	201.54	1.14	157.29	198.32	1.26	172.00	231.38	1.35
12	186.78	208.3	1.12	170.52	208.94	1.23	179.69	228.60	1.27
13	196.31	212.49	1.08	178.94	217.53	1.22	186.15	229.21	1.23
14	204.55	217.95	1.07	190.3	225.45	1.18	192.15	227.13	1.18
15	211.51	221.73	1.05	203.22	234.85	1.16	197.61	228.88	1.16
16	218.38	223.77	1.02	215.32	242.56	1.13	201.79	228.96	1.13
17	223.51	226.84	1.01	224.49	242.84	1.08	205.78	228.72	1.11
18	228.37	226.48	0.99	229.66	242.89	1.06	209.17	227.17	1.09
19	232.59	227.97	0.98	235.9	241.71	1.02	212.55	225.18	1.06
20	234.98	229.38	0.98	245.3	243.05	0.99	215.23	225.72	1.05

図 5. CPU のみの実装と CPU + インテル® DSA のハイブリッド実装のパフォーマンス比較

前述のように、1 コアの CPU メモリー・パフォーマンスはアーキテクチャーのロード / ストアバッファーのサイズによって決まりますが、インテル® DSA にはこの制限はありません。したがって、CPU + インテル® DSA ハイブリッド方式では、インテル® DSA がパイプライン方式でデータを DRAM から LLC にコピーするため、CPU によって読み取られるデータはすべて LLC でヒットします。LLC ヒット・レイテンシーは DRAM アクセス・レイテンシーよりも短いため、メモリー要求がロード / ストアキューに留まる時間が短縮され、その結果、1 コアの帯域幅は、カーネルタイプに応じて 1.6 倍～ 2 倍に増加します。

コア数が 3 以下の場合、CPU が LLC からデータを読み出しても、LLC から 1 ブロックのデータを処理する時間は、インテル® DSA が DRAM から 1 ブロックをフェッチする時間よりも長くなり、DRAM レイテンシーが完全に隠蔽されるため、少ないコア数で高いゲインが得られます。ワークフローに CPU コアを追加すると、LLC からの CPU リード帯域幅はインテル® DSA よりも速くなり、CPU はインテル® DSA が LLC へのコピーを完了するのを待機することになります。この問題を軽減するため、使用するコア数に応じて 2 つの異なる実装を使用します。コア数が多い場合、インテル® DSA のキューで未処理の要求が多くなると大きな競合が発生します。そこで、CPU がインテル® DSA とともにメモリー要求も処理するような別の実装に切り替えます。Triad と Dot では、CPU も使用して入力バッファーの 1 つ（`src1`）をフェッチし、インテル® DSA がもう 1 つの入力バッファー（`src2`）をフェッチします。この実装は、コア数が多い場合に次のような利点があります。第 1 に、インテル® DSA が DRAM から LLC へのコピーを完了するのを待機する代わりに、CPU を使用できます。第 2 に、インテル® DSA は DRAM から LLC にバッファーを 1 つのみフェッチすればよくなるため、未処理のインテル® DSA 要求の数が半分になり、インテル® DSA キューの競合が減少します。全体として、CPU + インテル® DSA ハイブリッド方式のパフォーマンス上の利点は、コア数が増えるにつれて小さくなります。

結論として、STREAM ベンチマークは、インテル® DSA が少ない CPU コアを使用してパフォーマンスを向上できることを示しています。例えば、Scale カーネルでは、インテル® DSA を使用して 9 コアで、20 コアの CPU のみの実装と同じパフォーマンスを達成できます。Triad カーネルでは、インテル® DSA を使用して 8 コアで、12 コアの CPU のみの実装と同じパフォーマンスを達成できます。Dot カーネルでは、インテル® DSA を使用して 5 コアで、8 コアの CPU のみの実装と同じパフォーマンスを達成します。インテル® DSA と組み合わせる CPU の数が少ないほど、メモリー・パフォーマンスが向上するため、これはヘテロジニアス・アプリケーションの高速化に役立ちます。計算依存とメモリー依存のカーネルが混在するヘテロジニアス・アプリケーションでは、計算カーネルにより多くの CPU を割り当てて、全体的なパフォーマンスを向上できます。シングルスレッドのメモリー帯域幅依存のアプリケーションも高速化できます。単一の CPU では帯域幅を完全に飽和させることはできないため、アムダールの法則の影響を軽減するという観点から、インテル® DSA はワークロードのシーケンシャルな部分の時間を短縮するのに使用できます。

intel software

THE PARALLEL UNIVERSE