AI PC で将来の AI 開発を試す

金融サービスのリスク計算における oneMKL 乱数ジェネレーター・デバイス API

OpenACC* API から OpenMP* API への移行

Issue
56
2024

目次

編集者からのメッセージ

James R. Reinders インテル コーポレーション エンジニア、The Parallel Universe 名誉編集長
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 12 冊の技術書の共著者です。最新の著書では、C++ with SYCL* の使用法を紹介しています。世界最速のコンピューター（TOP500 リストの 1 位）を含む、多くのスーパーコンピューターやソフトウェア開発ツールに多大な貢献を行ってきた強運の持ち主でもあります。

Henry の休暇中、本号の編集長を引き受けることになりました。

この数年、私は C++ with SYCL* に焦点を当てて並列コンピューティングの世界を旅してきました。インテル® コンパイラー・チームが C++ コンパイラーの SYCL* サポートに関して SYCL* 準拠に達したことは素晴らしいニュースです。チームの主要メンバーである Greg が、「**インテル® C++ コンパイラーが Khronos SYCL* 2020 準拠の最初のコンパイラーに**」で、このニュースを簡単に説明してくれました。

本号では、著者が行った手順を実際に試してみることができるハウツー記事をいくつか紹介しています。注目記事「**AI PC で将来の AI 開発を試す**」では、AI PC で OpenAI* Triton を実行します。「**OpenACC* API から OpenMP* API への移行**」では、OpenACC* コードを OpenMP* に移行する人向けに、移行方法とエクスペリエンスを詳しく説明します。SYCL* を素早く理解できるように、私が過去 1 年間に行ったいくつかの簡単な入門セッションをベースとした記事、「**1 時間（未満）で SYCL* を学ぶ**」も掲載しました。

財務リスク･シミュレーション向けの乱数に関する記事、「**金融サービスのリスク計算における oneMKL 乱数ジェネレーター・デバイス API**」も含まれています。高品質の乱数と優れた財務リスク･シミュレーションは容易でないテーマです。この記事では、重要なテクニックを紹介します。

「**トランスフォーマー向けインテル® エクステンションを利用した効率的な自然言語埋め込みモデル**」では、AI テクノロジーに焦点を当て、自然言語処理の埋め込みモデルのパフォーマンスを向上させる革新的なアプローチを紹介します。

注目記事では、Tony Mongkolsmai が彼のパーソナル AI PC で試した AI の楽しみ方を紹介しています。AI PC に搭載されているインテル® Core™ Ultra プロセッサー上で OpenAI* Triton を実行する彼の様子から、興味のある情報が得られることを願っています。

AI PC は、パーソナル・コンピューティングの継続的な進化における非常に現実的かつ重要な次のステップです。今では当たり前となったグラフィックスと Wi-Fi* がパーソナル・コンピューティング・デバイスでサポートされた当初、どちらも PC への追加機能であり、最初は導入を促進する多くの広告を目にしたものです。私は、グラフィックスや Wi-Fi* と同様に、本格的なレベルの AI サポートが一般的な PC に組み込まれるようになると確信しています。その理由は、グラフィックスや Wi-Fi* と同様に、ユーザー・エクスペリエンスがはるかに向上するからです。

AI および AI PC を対象とする今後の記事は、ハイパフォーマンスおよび並列のあらゆるものをカバーするという我々の継続的な取り組みの鍵を握る部分となるでしょう。我々は間違いなく、このエキサイティングで重要な AI PC 時代の初期段階にいます。

本号をお楽しみいただけることを期待しています。皆様からのフィードバックおよび貢献に感謝します。

James Reinders
2024 年 4 月

AI PC で将来の AI 開発を試す

Tony Mongkolsmai インテル コーポレーション

インテル® GPU 向けのオープンソース OpenAI* Triton バックエンドを構築して実行する方法

ディープラーニング（DL）AI ソリューションが大規模になるにつれて、AI 開発者が直面する最大の課題の 1 つが、パフォーマンスに優れたモデルを効率良く作成する方法です。これまで、AI モデルの開発者はカーネルを C++ で記述し、pybind を使用して Python* から実行する必要がありました。このため、AI 開発者は DL カーネルとモデルを理解するだけでなく、C++ とカーネル開発に関連する C++ テンソルの抽象化についても学ぶ必要がありましたが、この課題への取り組みとして、OpenAI* は、開発者が効率的な GPU コードを作成できるようにする、オープンソースのドメイン固有の Python* に似たプログラミング言語およびコンパイラーである Triton をリリースしました。

OpenAI* Triton の概要

Triton は、機能とパフォーマンスの点で C++ と Python* の中間層を提供します。目標は、DL 開発者が複数層のソフトウェア・スタックを実装することなく、最適化されたカーネルを構築できるようにすることです。Triton はネイティブ Python* ではありませんが、Python* コードと同じソースファイル内で開発できるため、コード管理がはるかに容易になります。

DL は本質的に大規模な並列計算を扱うため、Triton は個々の要素ではなくデータのブロックを扱うように設計されています。これにより、変数を定義することで、データのセットを表す言語の構文が簡素化されます。

```
result = x + y
```

例えば、上記のコードは `x` ベクトルと `y` ベクトルの要素単位の加算を行い、出力ベクトルを `result` 変数に書き込みます。

構文の簡素化以外の Triton の利点は、パフォーマンスです。Triton は言語であるとともにコンパイラーでもあるため、構文を特定のハードウェア設計にマップすることができます。ニューラル・ネットワークは大規模であり、実行するには大量のメモリー操作が必要です。Triton は、開発者とコンパイラー・バックエンドが、ネイティブ Python* で利用できるものよりも最適化されたキャッシュ階層とメモリーの使い方を言語の操作にマップできるよう意図的に定義されています。

これは Triton で実現される機能のほんの一部であり、ほかにも DL 開発者にとって不可欠な操作が含まれています。Triton の詳細は、OpenAI* の Triton リソース（英語）を参照してください。

インテル® Core™ Ultra プロセッサー上の Triton

OpenAI* はソフトウェア開発の課題を簡素化します。開発者は、**場所と時間に拘束されることなく**コードを開発したいと考えています。ワークステーション・レベルのラップトップを利用することもできますが、持ち運びに苦労するのが目に見えています。そこで、強力な統合 GPU とニューラル・プロセシング・ユニットを備えた、新しいインテル® Core™ Ultra プロセッサー・ベースのラップトップを選択し、新しい AI PC で Triton を試してみることにしました。

インテル® Core™ Ultra プロセッサー向け Triton のコンパイル

Triton はオープンソースですが、インテル® GPU のサポートは開発中であり、Triton のメイン・リポジトリーにまだアップストリームされていません。インテルは、ここで Triton バックエンドをオープンソースで（英語）開発しています。ほかのハードウェア・ベンダーと同様に、インテルは Triton のフォークからこのバックエンドを開発しており、この作業のアップストリームに取り組む予定です。GitHub* サイトには、このコードはインテル® データセンター GPU マックス・シリーズのカード上で開発およびテストされているという記述がありますが、任意のインテル® GPU 向けに oneAPI を使用して開発されたソフトウェアは、ほかのインテル® GPU でも動作するはずです。そこで、ソースからコードをビルドして、MSI* Prestige 16 AI Evo ラップトップでどのように動作するか確認してみることにしました（ちなみに、コンシューマーレベルのインテル® Arc™ A770 GPU 上で実行したところ、同様に動作しました）。

システムの設定

次の構成のシステムを使用しました。

- Ubuntu* 23.10、カーネル 6.5.0-17
- インテル® oneAPI ベース・ツールキット 2024.0.1（インストール手順（英語））
- Anaconda*（インストール手順（英語））

Linux* をインストールした後、`/etc/default/grub` を変更する必要があることに注意してください。

```
GRUB_CMDLINE_LINUX_DEFAULT=" quiet splash"
```

を次のように変更します。

```
GRUB_CMDLINE_LINUX_DEFAULT=" quiet splash i915.force_probe=7d55"
```

ここで、`7d55` は統合 GPU の PCI ID です。アップストリームの Ubuntu* 統合ドライバーは最新の iGPU の PCI ID をまだ認識しないため、この変更が必要になります。次に、

```
sudo update-grub
```

を実行した後、再起動します。

Triton の入手とビルド

GitHub* のビルド手順に従うのは難しいため、AI PC で Triton をビルドして実行するために私が実行したコマンドを以下に示します。最初に、環境を設定します。

```
# インテルの XPU 向け Triton バックエンドをチェックアウトしてディレクトリーを変更
git clone https://github.com/intel/intel-xpu-backend-for-triton.git -b llvm-target
cd intel-xpu-backend-for-triton

# Python* 3.10 を使用して conda 環境を作成し環境をアクティベート
conda create --name triton python=3.10
conda activate triton

# oneAPI のビルドおよびランタイム環境を設定
source /opt/intel/oneapi/setvars.sh

# Triton のビルド・インフラストラクチャーに応じてコンポーネントをインストールし
# Clang でビルドするように環境変数を設定
pip install ninja cmake wheel
export TRITON_BUILD_WITH_CLANG_LLD=true
```

Triton は現在 Python* 3.11 と 3.12 では動作しないため、Python* 3.10 を使用しました。`llvm-target` はコードのメインブランチであるため、ブランチを指定しないで実行することもできます。

リポジトリーの指示に従って、ビルドを行います。

```
./scripts/compile-triton.sh
```

残念ながら、このビルドでは CUDA* が見つからないというエラーが発生します。Ubuntu* 23.10 に CUDA* ツールキットをインストールしようとすると、Ubuntu* 23.10 ではそのまま使用できない依存ライブラリーが原因でエラーになります。この問題を回避するには、`/etc/apt/sources.list` を編集して次の行を追加し、Ubuntu* 23.10 APT ソースに 23.04 のリポジトリーを追加します。

```
deb http://archive.ubuntu.com/ubuntu/ lunar universe
```

次のコマンドを実行して、CUDA* をインストールできるようにします。

```
sudo apt update
```

ビルドを再実行すると、`clang` コンパイラーと `clang++` コンパイラーが見つからないという別のエラーが発生します。clang ベースのインテル® oneAPI DPC++ コンパイラーをインストールしているため、`PATH` 変数を設定して、インテル® DPC++ コンパイラーに含まれる clang を指定します。

```
export PATH=$PATH:/opt/intel/oneapi/compiler/latest/bin/compiler
```

コンパイルスクリプトをもう一度実行します。

```
         ********************************************************************************
         Please be careful with folders in your working directory with the same
         name as your package as they may take precedence during imports.
         ********************************************************************************

  !!
    with strategy, WheelFile(wheel_path, "w") as wheel_obj:
  Building editable for triton (pyproject.toml) ... done
  Created wheel for triton: filename=triton-3.0.0-0.editable-cp310-cp310-linux_x86_64.whl size=3072 sha256=bf4fd604e7ba68134d
2d935f96a9e96d0b62865cbf74081b39653a47a1708867
  Stored in directory: /tmp/pip-ephem-wheel-cache-5z1lh_f8/wheels/98/85/b6/6ad462c1a1735d0b1af8dba5e03cd79040eac6c513748a5631
Successfully built triton
Installing collected packages: triton
  Attempting uninstall: triton
    Found existing installation: triton 3.0.0
    Uninstalling triton-3.0.0:
      Created temporary directory: /tmp/pip-uninstall-yqqxblnw
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/__editable__.triton-3.0.0.pth
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/__editable___triton_3_0_0_fin
der.py
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/__pycache__/__editable___trit
on_3_0_0_finder.cpython-310.pyc
      Created temporary directory: /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/~riton-3.0.0.dist-info
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/triton-3.0.0.dist-info/
      Successfully uninstalled triton-3.0.0

Successfully installed triton-3.0.0
Remote version of pip: 24.0
Local version of pip:  23.3.1
Was pip installed by pip? False
Removed build tracker: '/tmp/pip-build-tracker-ts1gk_qf'
(triton) tonym@prestige:/scratch/intel-xpu-backend-for-triton$
```

インテル® Core™ Ultra プロセッサー上での Triton のテスト

最初のテストは、`intel-xpu-backend-for-triton` コードベース内の `python/tutorials` ディレクトリーをチェックアウトすることです。基本的な機能を評価するため、このディレクトリーから最初のいくつかの例を実行しています。テストコードを見ると、使用するインテル® GPU を選択するため OpenAI* Triton リポジトリーからいくつかの変更が加えられていることが分かります。次のコードのデバイス指定を、

```
x = torch.rand(size, device='cuda')
y = torch.rand(size, device='cuda')
```

次のように変更します。

```
x = torch.rand(size, device='xpu')
y = torch.rand(size, device='xpu')
```

ランタイム環境の設定

新しい Triton ビルドを試すには環境に PyTorch* を追加する必要があるため、最初のステップとして PyTorch* 向けインテル® エクステンションをインストールします。

```
python -m pip install torch==2.1.0a0 torchvision==0.16.0a0 torchaudio==2.1.0a0 intel-extension-for-pytorch==2.1.10+xpu --extra-index-url https://pytorch-extension.intel.com/release-whl/stable/xpu/us/
```

チュートリアルの例では、いくつかの追加の Python* ライブラリーも必要です。次のように `pip` を使用して簡単にインストールできます。

```
pip install matplotlib pandas
```

実行例

最初のチュートリアルは、2 つのデータブロック間の単純なベクトル加算です。出力は以下のようになります。

```
(triton) tonym@prestige:/scratch/intel-xpu-backend-for-triton/python/tutorials$ python 01-vector-add.py
/home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/torchvision/io/image.py:13: UserWarning: Failed to loa
d image Python extension: ''If you don't plan on using image functionality from `torchvision.io`, you can ignore this
 warning. Otherwise, there might be something wrong with your environment. Did you have `libjpeg` or `libpng` install
ed before building `torchvision` from source?
  warn(
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'
tensor([1.3713, 1.3076, 0.4940,  ..., 0.8654, 1.1553, 0.4071], device='xpu:0')
tensor([1.3713, 1.3076, 0.4940,  ..., 0.8654, 1.1553, 0.4071], device='xpu:0')
The maximum difference between torch and triton is 0.0
vector-add-performance:
           size     Triton      Torch
0        4096.0   0.003057   0.003096
1        8192.0   0.006198   0.006193
2       16384.0   0.012383   0.012375
3       32768.0   0.024736   0.024673
4       65536.0   0.049506   0.049383
5      131072.0   0.098313   0.098653
6      262144.0   0.196897   0.196460
7      524288.0   0.388927   0.389345
8     1048576.0   0.769500   0.768111
9     2097152.0   1.496443   1.492719
10    4194304.0   2.858620   2.823484
11    8388608.0   5.277260   5.095062
12   16777216.0   9.116155   8.511318
13   33554432.0  14.523118  12.744209
14   67108864.0  15.574265  12.825385
15  134217728.0  14.566962  11.500764
```

出力は、データの正当性とベクトル加算における操作の相対パフォーマンスを測定したものです。ベクトルのサイズの増加とともに、Triton アプローチによりインテル® Core™ Ultra プロセッサーの統合 GPU のパフォーマンスが向上していることが分かります。

2 つ目のチュートリアルは、fused-softmax の実装です。softmax 関数は、ベクトルを結果の確率分布にマッピングするもので、ニューラル・ネットワークの最後の活性化関数として DL アルゴリズムでよく使用されます。softmax は入力ベクトルに対して大量の操作を実行するため、ベクトル化によりパフォーマンスが大幅に向上する可能性があります。fused-softmax 実装は、データアクセスをブロックして naïve 実装を改善することを目的としています。PyTorch* にはネイティブの fused-softmax 実装がすでに存在することに注意してください。つまり、この操作のハードウェアに最適化された C++ バージョンと比較することができます。チュートリアルを実行すると、次の出力が得られます。

```
(triton) tonym@prestige:/scratch/intel-xpu-backend-for-triton/python/tutorials$ python 02-fused-softmax.py
/home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/torchvision/io/image.py:13: UserWarning: Failed to loa
d image Python extension: ''If you don't plan on using image functionality from `torchvision.io`, you can ignore this
 warning. Otherwise, there might be something wrong with your environment. Did you have `libjpeg` or `libpng` install
ed before building `torchvision` from source?
  warn(
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'
/scratch/intel-xpu-backend-for-triton/python/tutorials/02-fused-softmax.py:184: UserWarning: The grad mode is detecte
d as torch.no_grad() is NOT enabled. In this mode on XPU, please expect NO graph and fusion optimization will be appl
ied.
 (Triggered internally at /build/intel-pytorch-extension/csrc/gpu/jit/fusion_pass.cpp:829.)
  ms, min_ms, max_ms = triton.testing.do_bench(lambda: naive_softmax(x), quantiles=quantiles)
softmax-performance:
          N    Triton  Torch (native)  Torch (jit)
0     256.0  0.508614        0.520325     0.485650
1     384.0  0.738521        0.779249     0.711792
2     512.0  0.976950        1.029912     0.916427
3     640.0  1.210549        1.273200     1.120450
4     768.0  1.441097        1.514099     1.302048
..      ...       ...             ...          ...
93  12160.0  8.014298       13.657079     3.345754
94  12288.0  8.042449       13.338677     3.417873
95  12416.0  8.148614       13.899843     3.363594
96  12544.0  8.153968       13.621084     3.355575
97  12672.0  8.267947       15.834839     4.628440

[98 rows x 4 columns]
```

ここで、softmax の 3 つの異なる実装のパフォーマンスを確認できます。最初の列は Triton バージョン、2 つ目の列は C++ 最適化バージョン、3 つ目の列は naïve バージョンです。インテル® Core™ Ultra プロセッサーでは、Triton バージョンは naïve バージョンよりも高速ですが、C++ ネイティブバージョンのコードよりも遅くなっています。Torch C++ 実装はすでに大幅に最適化されているため、この結果は当然のことです。しかし、Python* のみを使用した naïve 実装よりもはるかに高速なコードを作成できることは、開発者として興味深いことです。

まとめ

Triton フレームワークは、DL カーネル開発者から強く支持されています。単純な並列構文と、基盤となるハードウェア上のデータアクセスをすべて高水準の Python* コードから最適化する機能は、DL カーネル開発の簡素化に大いに役立ちます。このフレームワークをインテル® Core™ Ultra プロセッサー・ベースの AI PC と組み合わせることにより、必要なときに必要な場所で DL カーネル開発を行うための前途有望な道を切り拓くことができるでしょう。`intel-xpu-backend-for-triton` の開発は現在進行中です。このエキサイティングなハードウェアとソフトウェアの組み合わせの恩恵を受けられることを楽しみにしています。

intel 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

https://www.xlsoft.com/jp/products/intel/devcloud/services.html

金融サービスのリスク計算における oneMKL 乱数ジェネレーター・デバイス API

Andrey Fedorov インテル コーポレーション マス・アルゴリズム・エンジニア
Gennady Fedorov インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア
Vladimir Polin 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
Robert Mueller-Albrecht インテル コーポレーション プロダクト・マーケティング・エンジニア

柔軟な並列計算能力の必要性

バーゼル銀行監督委員会（BCBS）（英語）は、金融リスク軽減策の必要性の高まりにより、計算需要が 4 ~ 20 倍に増加すると予測しています。これには、気候変動、サプライチェーンの混乱、地政学的な変化に関連する、資産の変動を深く理解するための高度な市場予測アルゴリズムとリスク・シミュレーションが含まれます。

銀行および金融サービス業界（FSI）のソフトウェア開発者は、GPU ベースのアクセラレーションを備えたハイパフォーマンス・コンピューティング（HPC）環境を利用して、リスク・シミュレーション・モデルを実装しています。これらのワークロードは広範にデプロイされるため、さまざまなハードウェア・プラットフォーム構成で実行できることが重要です。

Unified Acceleration (UXL) Foundation（英語）の取り組みが支持され SYCL*（英語）の人気が高まるまでは、ワークロードを新しい環境にデプロイするたびにコードを移植してリファクタリングする必要があり、大きな課題と非効率性に直面していました。

モンテカルロ法は金融分野でよく知られた手法であり、シミュレーション・シナリオのシードとして高性能の乱数生成に依存しています。特にエキゾチック・オプションでは、予想投資利益率（ROI）が複雑すぎて直接計算できないことが多いため、オプションの価格付けに使用されます。アメリカン・オプション・モデルとヨーロピアン・オプション・モデルは、さまざまなオプション投資結果の確率予測に広く使用されている 2 つのモンテカルロ・シミュレーション手法です。

2020 年に C++ with SYCL* ベースの oneAPI マス・カーネル・ライブラリー（oneMKL）インターフェイス（英語）が登場して以来、oneAPI 仕様のこのコンポーネントとインテル® oneAPI マス・カーネル・ライブラリー（英語）のオープンソース拡張機能により、乱数ジェネレーター（RNG）ルーチン向けのデータ並列 C++ インターフェイス（英語）が提供され、よく使用される擬似乱数、準乱数、連続分布と離散分布を備えた非決定性生成器が実装されています。

ほかの oneMKL 関数ドメインと同様に、oneMKL RNG には実装の一部として次のものが含まれます。

1. 手動オフロード機能（乱数ジェネレーター・ホスト API）
2. デバイス機能 — SYCL* カーネルから直接呼び出し可能な関数群（乱数ジェネレーター・デバイス・ルーチン（英語））。

この記事では、oneMKL 乱数ジェネレーター（RNG）デバイス API を使用して、インテル® データセンター GPU 上での計算パフォーマンスをホスト API 実装の 2 倍に大幅に向上する方法を示します。

ヨーロピアン・オプション価格付けモデルとアメリカンオプション価格付けモデルのホスト API とデバイス API の RNG 関数呼び出しのパフォーマンスを測定して比較します。

さらに、モンテカルロ・アルゴリズムを使用したアメリカンオプション価格付けモデルを例として使用し、金融ワークロードを独自の CUDA* コードから SYCLomatic（英語）（またはインテル® DPC++ 互換性ツール）を使用してオープンなマルチプラットフォーム SYCL* プログラミング・モデルに移行する方法を詳しく説明します。

この記事で紹介する結果は、インテル® データセンター GPU Max 1550 上で oneAPI 2024.0 リリースを使用して得られたものです。

RNG デバイス API の基本

デバイス・インターフェイスの主な目的は、SYCL* カーネルから呼び出せるようにすることです。送信時間はアプリケーションの全体的な実行時間に大きな影響を与えるため、パフォーマンスが大幅に向上します。グローバルメモリー転送のコストを排除し、素早く乱数を取得し、同じカーネル内で乱数を処理します。次に例を示します。

ホスト API

```
auto engine = oneapi::mkl::rng::mrg32k3a(*stream, 1ull);
oneapi::mkl::rng::generate(oneapi::mkl::rng::gaussian<double>(0.0, 1.0),
                           engine, n, d_samples);

sycl_queue.parallel_for(
    sycl::nd_range<3>(sycl::range<1>(n_groups) * sycl::range<1>(n_items),
                      sycl::range<1>(n_items)),
    [=](sycl::nd_item<1> item_1) {
        post_processing_kernel(d_samples);
    }).wait();
```

デバイス API

```
sycl_queue.parallel_for(
    sycl::nd_range<3>(sycl::range<1>(n_groups) * sycl::range<1>(n_items),
                      sycl::range<1>(n_items)),
    [=](sycl::nd_item<1> item_1) {
        oneapi::mkl::rng::device::mrg32k3a<1> engine_device(1ull, n);
        oneapi::mkl::rng::device::gaussian<double> distr(0.0, 1.0);
        double rng_val = oneapi::mkl::rng::device::generate(distr, engine_device);

        post_processing_kernel(rng_val);
    }).wait();
```

上記のホスト API コードシーケンスは、CPU から GPU への呼び出しを開始する 2 つのインスタンスを示しています。

1. 少なくとも 1 つの SYCL* カーネルを含む `generate()` 関数呼び出し
2. `nd_range` および `post_processing_kernel` を含む、オフロードされた `parallel_for` ループカーネル

デバイス API コードシーケンスでは、これらがすべて単一のオフロードされた `parallel_for` SYCL* カーネル内に含まれており、CPU と GPU 間のデータ交換が最小限になります。

そのため、デバイス API を使用すると、実装に必要なカーネルの数がホスト API よりも少なくなります。RNG デバイス API は、GitHub* にあるオープンソース・プロジェクト、oneMKL インターフェイス（英語）の一部としても利用できます。

アメリカン・モンテカルロとヨーロピアン・モンテカルロのワークロードを別々に考えてみましょう。

アメリカン・モンテカルロ・オプション価格付けモデル

このベンチマークをインテル® GPU 上で実行するため、NVIDIA* 開発者コードサンプル GitHub* リポジトリー（英語）からオリジナルコードを取得しました。次に、SYCLomatic オープンソース・プロジェクトを使用して、ネイティブ CUDA* GPU コードを SYCL* に移行しました。このツールはインテル® oneAPI ベース・ツールキットに含まれています。Apache* 2.0 ライセンスの下で GitHub* から入手することもできます。SYCLomatic を使用すると、オリジナルの CUDA* コードを SYCL* に移植できます。一般的な CUDA* コードの約 95% が自動的に SYCL* コードに移行されます。

GitHub* リポジトリー（英語）でアメリカン・モンテカルロ・オプション価格付けモデルのサンプル・ソースコードを SYCL* に移行する手順を確認し、SYCL* 対応のサンプル・ソースコード・プロジェクトを調べてください。

プロセスを完了するため、手動でコードの一部を変更して、ターゲット・アーキテクチャーで必要なパフォーマンス・レベルになるようにチューニングしました。

さらに、SYCLomatic の移行が完了した後、ホスト・インターフェイス呼び出しを追加しました。使用する SYCL* カーネルの数を減らすため、次のカーネル（`generate_paths_kernel`）へのデバイス呼び出しを追加しました。この方法では、乱数が生成されるとすぐに使用されるため、必要なカーネルの数が減り、乱数を保存するためのメモリーが不要になります。

デバイス API インターフェイス機能を適用すると、従来のホスト・インターフェイス呼び出しと比較してパフォーマンスが最大 2.13 倍向上します。アプリケーション全体では、最大 15% のパフォーマンス向上を達成できました。

図 1 は、計算したパスの数に応じたアメリカン・モンテカルロ・ベンチマーク結果のパフォーマンスのスケーラビリティーを示しています。**RNG と次のカーネルのスピードアップ**は、RNG ホスト・インターフェイス呼び出しと `generate_paths_kernel` を個別に使用したホスト API バージョンと、RNG デバイス API と `generate_paths_kernel` を組み合わせて使用したバージョンを比較したパフォーマンスの向上を示しています。**ベンチマーク全体のスピードアップ**は、RNG ホスト API を使用した場合と、RNG デバイス API を使用した場合を比較したベンチマーク全体のパフォーマンスの向上を示しています。

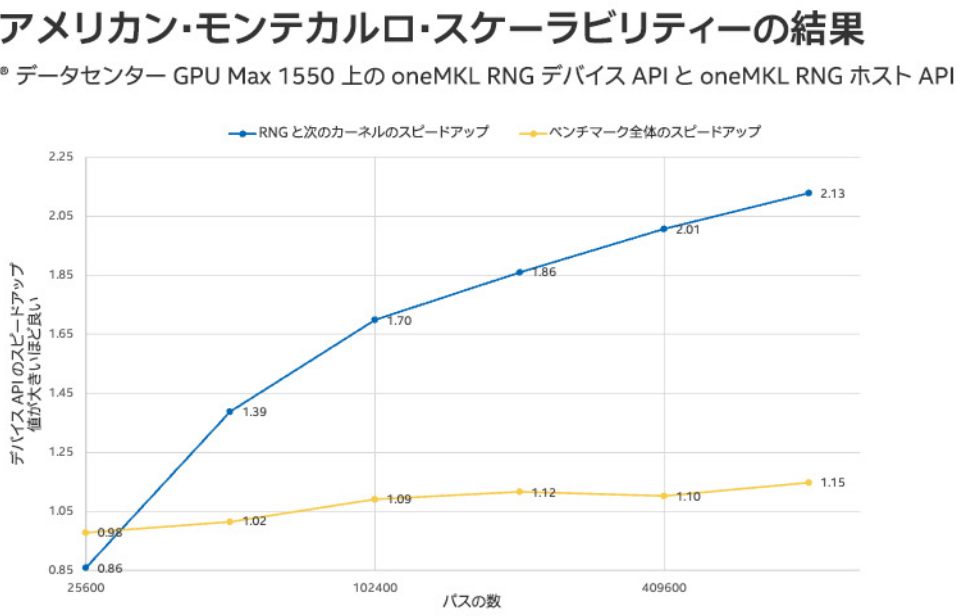

図 1. アメリカン・モンテカルロ・スケーラビリティーの結果

システム構成

テスト実施日：性能の測定結果は **2023 年 11 月 24 日現在のインテルの社内テストに基づいています。**
また、現在公開中のすべてのアップデートが適用されているとは限りません。
システム構成とワークロード設定：1 ノード、2x インテル® Xeon® Platinum 8480+ プロセッサー、56 コア、512GB。
Ubuntu* 22.04.2 LTS、カーネル : 5.15.47+prerelease70。インテル® oneAPI マス・カーネル・ライブラリー（インテル® oneMKL）2024.0。
インテル® DPC++ コンパイラー 2024.0.0（2024.0.0.20231017）。インテル® レベルゼロ、インテル® データセンター GPU Max 1550 1.3
[1.3.26690]。GPU : インテル® データセンター GPU Max 1550、GT 周波数 : 1.60GHz、1024 ユニット、2 タイル。

性能は、使用状況、構成、その他の要因によって異なります。

ヨーロピアン・モンテカルロ・オプション価格付けモデル

アメリカン・モンテカルロ・オプション・モデルのデバイス API とホスト API のパフォーマンスを確認したら、ヨーロピアン・モンテカルロ・オプション・モデルでも同じことを行います。oneAPI サンプル GitHub*（英語）で入手できるモンテカルロ・ヨーロピアン・オプション・サンプル（英語）のパフォーマンスを検討しましょう。

このユースケースでも、デバイス API を使用するメリットがあるか見てみましょう。

アメリカン・モンテカルロ・モデルとの違いは、リポジトリーのコードがすでにデバイス API を使用していることです。そのため、パフォーマンスを比較できるようにするには、ホスト API 実装を追加する必要があります。ホスト API と 1 年間のオプション価格付けモデルを追加するため、`option_years` 変数を `1` に設定します。この検討の後、デバイス呼び出しを別の SYCL* カーネルとしてのホスト呼び出しに簡単に置換できます。ただし、生成された数値を保存するメモリーが必要です。

これらの変更を適用すると、オリジナルのサンプルの構成で ~100 000 000 000 個の倍精度数を保存するには、GPU 上のメモリーが不足していることが分かりました。デバイス API を使用する欠点の 1 つは、CPU と比較して GPU では利用できるメモリーが限られているため、大規模なデータセットでは高度な共有メモリー管理が必要になることです。

デバイス API とホスト API の使用を除いて、リファレンス例を根本的に変更しないようにするため、計算する資産価格のパスの数を 16,000 に減らすことにしました。この変更により、コードを大幅に変更することなく、デバイス API 実装を使用してコードを正常に実行できるようになりました。

ヨーロピアン・オプション価格付け予測モデルのデバイス API とホスト API のパフォーマンスを比較すると、この場合も GPU デバイス API の使用により実行速度が向上していることが分かります。

図 2 は、RNG デバイス API を使用することにより、RNG ホスト API と比較してヨーロピアン・モンテカルロ・ベンチマーク全体のパフォーマンスが 3.69 倍向上する様子を示しています。

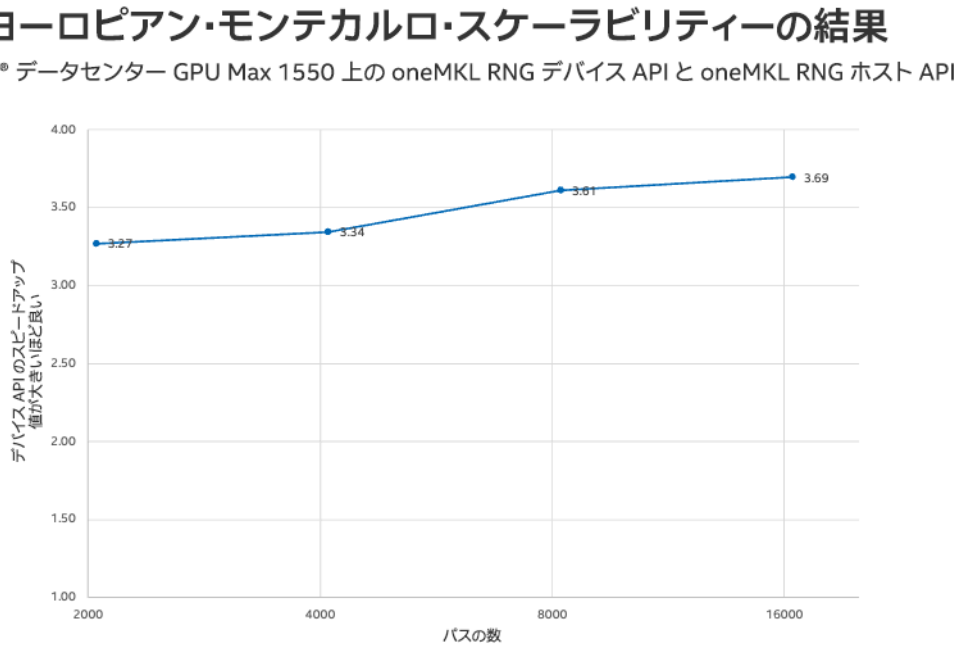

図 2. ヨーロピアン・モンテカルロ・スケーラビリティーの結果

システム構成

テスト実施日：性能の測定結果は **2023 年 11 月 24 日現在のインテルの社内テストに基づいています。**
また、現在公開中のすべてのアップデートが適用されているとは限りません。
システム構成とワークロード設定：1 ノード、2x インテル® Xeon® Platinum 8480+ プロセッサー、56 コア、512GB。
Ubuntu* 22.04.2 LTS、カーネル：5.15.47+prerelease70。インテル® oneAPI マス・カーネル・ライブラリー（インテル® oneMKL）2024.0。
インテル® DPC++ コンパイラー 2024.0.0（2024.0.0.20231017）。インテル® レベルゼロ、インテル® データセンター GPU Max 1550 1.3 [1.3.26690]。GPU：インテル® データセンター GPU Max 1550、GT 周波数：1.60GHz、1024 ユニット、2 タイル。

性能は、使用状況、構成、その他の要因によって異なります。

oneMKL RNG デバイス API ルーチンの活用

デバイス API を使用すると、ユーザーはホスト・インターフェイスと比較して本質的なパフォーマンス向上を達成できますが、特定のドメインの実装に関する多くのコーディングと知識が必要になります。デバイス API は、ユーザーが SYCL* 並列ルーチンのパラメーターを制御できる、柔軟な API でもあります。

この記事の結果は、重要なパフォーマンス・パスで乱数の計算を行う（金融以外の）ほかのアプリケーションにも適用できます。

課題に対処することは、インテル® コンパイラーと oneMKL ソフトウェア開発チームにとって日常的なプロセスです。インテル® ソフトウェアをユーザーにとって良いものにするため、我々は効率的なアルゴリズムと優れた最適化手法を常に模索しています。

この記事に興味を持たれた方は、まず、インテル® oneAPI マス・カーネル・ライブラリー、oneAPI マス・カーネル・ライブラリー（oneMKL）インターフェイス・プロジェクト (英語)、SYCLomatic（英語）を参照してみてください。

関連情報

- インテル® oneAPI マス・カーネル・ライブラリー
- oneAPI マス・カーネル・ライブラリー（oneMKL）インターフェイス（英語）
- SYCLomatic（英語）
- インテル® DPC++ 互換性ツール
- Unified Acceleration（UXL）Foundation（英語）

OpenACC* API から OpenMP* API への移行

Harald Servat、Shiquan Su、および Tobias Kloeffel インテル コーポレーション
Ron Caplan Predictive Science, Inc.
Junyi Cheng コロラド大学ボルダー校

はじめに

OpenACC* と OpenMP* はどちらもアクセラレーターへのオフロードをサポートしています。OpenACC* が早期に利用可能になったことは計算ワークロードに GPU デバイスを使用するユーザーにとって良いニュースと言えますが、OpenACC* がサポートするデバイスベンダーはまだ限られています。現在は OpenMP* オフロードをサポートするアクセラレーター・ベンダーの数が増えていることから、アクセラレーターへのオフロードに OpenMP* を採用する需要が高まっています。

OpenACC* から OpenMP* API へのインテル® アプリケーション移行ツール（英語）は、OpenACC* ベースのアプリケーションの OpenMP* への移行を支援するオープンソース・プロジェクトです（IWOMP の論文（英語）を参照）。このツールは、C/C++ および Fortran アプリケーションのソースを解析し、OpenACC* 構造を識別して、可能な場合は意味的に同等の OpenMP*（5.0 以降）構造を提案します。構造間の記述的 / 規範的な違いを含む、さまざまな理由により、すべての OpenACC* 構造を OpenMP* に変換できるわけではありません。

特に、パフォーマンス・チューニングは考慮されないので注意が必要です。このツールは意味的に正しい移行コードを提供することのみに焦点を当てており、パフォーマンスが最適化されたバージョンの移行コードを提供することには焦点を当てていないため、パフォーマンスの最適化は後の段階で行うことになります。OpenACC* 構造を変換できない場合、ツールは移行レポートにメッセージを生成します。2 つの言語間のマッピングの詳細は、IWOMP の論文で説明されています。アプリケーションのソースが移行されたら、開発者は OpenMP* 5.0 準拠のコンパイラーを選択し、ターゲット・アーキテクチャー向けにアプリケーションをコンパイルするだけです。

次のセクションで、POT3D と GEM という 2 つの移行の成功事例について説明します。その後、いくつかのコメントを述べて、この記事のまとめとします。

アプリケーション移行の例

移行ツールを実証するため、2 つのアプリケーション、POT3D と GEM を選択しました。詳細は、**表 1** を参照してください。これらのアプリケーションを移行した後、インテル® HPC ツールキット 2024.0.1 のコンパイラーとライブラリーを使用してコンパイルしました。両方のアプリケーションを、2 ソケットのインテル® Xeon® Platinum 8480+ プロセッサーと 4 つのインテル® データセンター GPU Max 1550 を含む単一ノード上で実行しました。各 GPU は 2 つの計算タイルで構成され、ノードの計算タイルは合計 8 つになります。対応するリポジトリーで提供されているリファレンス出力と比較して実行結果を検証しました。

			OpenACC* 構造 / 節の数				
アプリケーション	**言語**	**行数**	計算	データ	アトミック	非同期	API 呼び出し
POT3D	Fortran	~11k	33	33	2	23	0
GEM	Fortran	~18k	34	69	118	0	1

表 1. アプリケーションの特性

注：
対象となる計算構造：`kernels`、`loop`、`parallel` および `loop`。
対象となるデータ構造：`enter/exit data`、`host_data`、および update。
対象となるアトミック構造：`atomic`。
対象となる非同期構造：`async` および `wait`。

POT3D

POT3D（英語）（commit-id: 5e8ee69f92860a372d5746462199ddfa702bbac2）は、観測された光球磁場を境界条件として使用して太陽コロナ磁場を近似するポテンシャル場の解を計算する Fortran コードです。コードは MPI を使用して並列化されますが、GPU アクセラレーションには OpenACC* と Fortran 標準 `do concurrent` 構文の 2 つの代替手段があります。この記事では、OpenACC* バージョンを使用します。

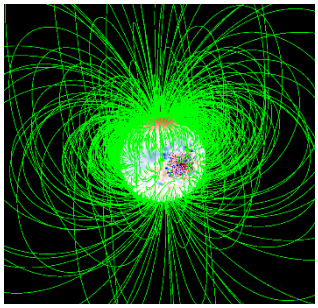

図 1. POT3D はポテンシャル場ソリューションにより磁力線をトレースします。表面の放射状の磁場が赤と青のカラーマップとして示されています。

次のコマンドを実行してコードを変換しました。

```
${PATH_TO_TRANSLATOR}/intel-application-migration-tool-for-openacc-to-openmp src -async=none
```

ここで、`src` はアプリケーションのソース・ディレクトリーを指します。`-async=none` はソースコード内に存在する非同期 OpenACC* ステートメント（`wait` および `async` を含む）を無視してコードを変換します。OpenMP* は、これらのステートメントの正確なマッピングを提供しません。幸いなことに、開発者によれば、非同期はこのアプリケーションにほとんど利益をもたらしません。

移行に関する注意事項

移行レポートに表示されるコメントと警告のうち、アプリケーションが `!$acc set device_num(iprocsh)` を使って、使用する OpenACC* デバイスを指定していることに注目してください。OpenMP* は、この動作を模倣する API 呼び出し（`omp_set_default_device`）と環境変数（`OMP_DEFAULT_DEVICE`）を提供します。インテル® MPI は、MPI ランクを GPU デバイス（またはタイル）にピニングする環境変数（`I_MPI_OFFLOAD_PIN`）も提供します。そのため、MPI ランク（GPU タイルのバインド）に関しては、構造の変換を無視して、インテル® MPI のインフラストラクチャーに依存しました。

変換に関して言えば、多くのデータ割り当てに `!$acc data create` という注釈が付いていることが分かります。これは、対応するデータをアクセラレーターのアドレス空間に割り当てる指示であり、`!$omp target enter data` を使用して変換されます。最も時間のかかる領域（`cgsolve`）には、`!$omp target teams loop` に変換された `!$acc parallel loop` 節が含まれていることも分かりました。興味深いことに、元の構造では `present(X)` 節または `default(present)` 節を明示的に使用して、オフロードランタイムでアクセラレーターのアドレス空間に指定された変数が存在するかどうかをチェックしています。これらの節はそれぞれ、`present` マップタイプ修飾子または `defaultmap` 節を使用して `map` に変換されます。ホストとアクセラレーターのアドレス空間の内容を同期するため、境界交換や I/O の実行前後に `!$acc update` などの節もあります。

対応する Makefile は、IFX 向けの MPI コンパイラー・ラッパー（`mpiifx`）を使用するように手動で調整され、OpenMP* オフロード構造（`-fiopenmp -fopenmp-targets=spir64`）と HDF5 の依存関係を処理するために必要なオプションが追加されました。

アプリケーションの実行に関しては、アプリケーション・リポジトリーに含まれる small テストを使用し、環境変数 `I_MPI_ADJUST_BARRIER=1 I_MPI_OFFLOAD_PIN=1 I_MPI_OFFLOAD_PRINT_TOPOLOGY=1 I_MPI_DEBUG=3` を設定してバイナリーを（`mpirun` 経由で）呼び出しました。達成されたパフォーマンスを**図 2** に示します。アプリケーションは 4 MPI ランク（各 GPU で 1 つのコンピューティング・タイルを使用）までは理想に近いスケーラビリティーを示しますが、8 MPI ランク（各 GPU で 2 つの計算タイルを使用）になるとパフォーマンスが低下します。このパフォーマンス低下の根本的な理由を考察するため、アプリケーション開発者といくつかのアイデアを議論しました。

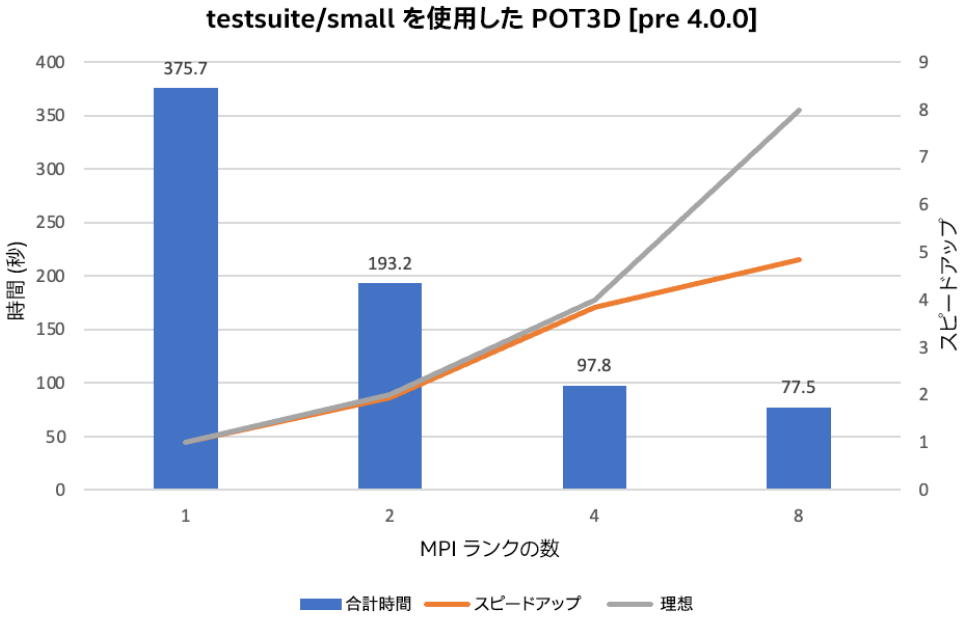

図 2. ターゲットシステム上で POT3D を実行したときに達成されたパフォーマンス

GEM

GEM（英語）は、1990 年代から Fortran で記述された、物理場効果を考慮したオープンソースの Particle-In-Cell（PIC）法のシミュレーション・コードです。ITER（英語）のような、磁気的に閉じ込められた核融合装置における輸送を研究することを目的としています。

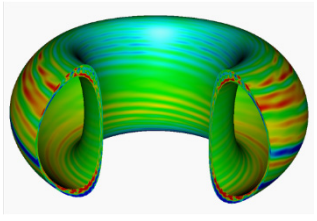

図 3. GEM は、磁気平衡でドーナツ状に移動するプラズマ粒子をシミュレートします。

GEM には時間のかかるタスクが 2 つあります。

1. 入れ子のループ（リダクションあり / なし）
2. 物理場の行列ソルバー

どちらも GPU で高速化できる可能性がありますが、開発者はコードの保守性を高める標準 LAPACK を使用してソルバーを実装することを好むため、高速化に OpenACC* を使用していたのは入れ子のループのみです。まず、移行ツールを利用して OpenACC* 構造を OpenMP* に変換します。次に、`!$omp dispatch` 構造により並列かつ高速な LAPACK 実装を提供する oneAPI マス・カーネル・ライブラリー（oneMKL）を使用するように、変換されたコードを手動で変更します。後者の変換はこの記事では取り上げません。

アプリケーションのファイル構造は単純で、すべてのソースファイルが同じフォルダーに含まれています。コードを変換するコマンドは次のとおりです。

```
${PATH_TO_TRANSLATOR}/intel-application-migration-tool-for-openacc-to-openmp *.f90
```

変換後、IFX Fortran コンパイラーで OpenMP* オフロード機能を有効にするため、OpenACC* のコンパイラー・オプションを `-DOPENACC2OPENMP_ORIGINAL_OPENMP -fiopenmp -fopenmp-targets=spir64` に置換しました。オリジナルコードでは、GPU に送るには小さすぎるいくつかのループを OpenACC* が処理するときに複数のスレッドを起動する OpenMP* 構文も適用しているため、`-DOPENACC2OPENMP_ORIGINAL_OPENMP` コンパイラー・オプションを追加しました。これらは移行ツールにより保持されますが、このプリプロセッサー変数で保護されます。

移行に関する注意事項

移行中に見つかった特異点を述べておきます。

1. データ管理
2. ループ構造での並列処理

データ管理

GPU と CPU 間のデータ移動に関して、GEM は非構造化データアプローチを使用します。このアプローチにより、コードがホストメモリー上にデータを割り当てた（または割り当て解除した）直後に、ホストとデバイス間のデータマッピングを確立する柔軟性が提供されます（データ割り当て時に `!$acc enter data create(X)` 構造を使用します）。この構造は、移行ツールにより `!$omp target enter data(X)` として変換されます。

ほかのアプリケーション・モジュールがデバイスデータにアクセスする必要がある場合、OpenACC* の `present(X)` 節に依存します。この節はマップタイプ修飾子（`map(present,alloc:X)`）に変換されます。見た目は同じではありませんが、この 2 つは同じセマンティクスを表現します。ただし、OpenMP* では、割り当てが行われない場合でも、存在チェックをマップタイプ（この場合は `alloc`）で行う必要があります。

さらに、アプリケーション・コードは `acc_is_present(X)` API を利用して、GPU アドレス空間にデータが存在しない潜在的なケースを識別します。この API ルーチンは移行ツールで変換されず、ツールはこの誤変換に関する警告を生成します。

ループ構造での並列処理

`!$acc loop` で高速化される 2 つの主な並列領域を特定します。1 つ目の領域は、`!$acc atomic` 構造を使用していくつかのリダクションを実装します。これらの実装は対応する OpenMP* に変換されます。2 つ目の領域は、以下のコードで表されます。内部に `control-flow (if)` 構造があることに注意してください。このループには入れ子のループ構造が含まれていますが、完全な入れ子のループではないため、`collapse` 節を使用することはできません。

```
!$acc parallel present(xp,s_buf,ipsend,s_displ,s_counts)
!$omp target teams map(present,alloc:xp,s_buf,ipsend,s_displ,s_counts)
!$acc loop independent  private(i,isrt,iend,isbuf)
!$omp loop order(concurrent) private(i,isrt,iend,isbuf)
    DO i=0,nvp-1
       IF( s_counts(i) .GT. 0 ) THEN
          isrt = s_displ(i)+1
          iend = s_displ(i)+s_counts(i)
          !$acc loop independent
          !$omp loop order(concurrent)
          DO isbuf=isrt,iend
             s_buf(isbuf) = xp(ipsend(isbuf))
          END DO
          !$acc end loop
          !$omp end loop
       END IF
    END DO
!$omp end loop
!$acc end parallel
!$omp end target teams
```

移行の結果として得られる最初の OpenMP* 実装は、いくつかの手動の変更を加えてコンパイルして実行できます。ランクごとに 1 つの GPU タイルを使用して 4 MPI ランクで実行した場合のパフォーマンスを**図 4** に示します。この結果は開発者の予想と一致しています。

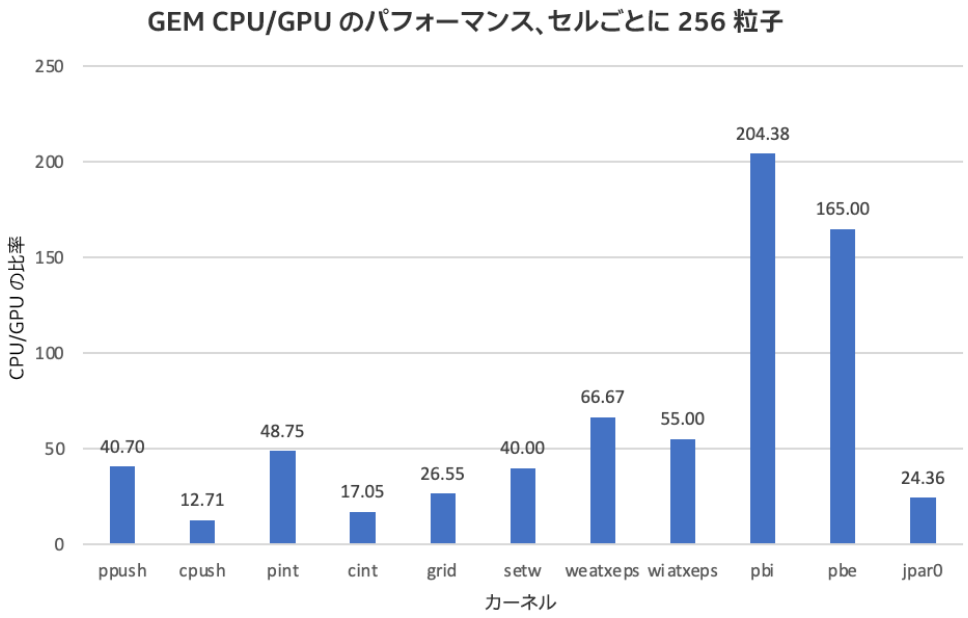

図 4. 主要なオフロードされたカーネルでの GEM アプリケーションの CPU と GPU の比率

まとめ

OpenACC* から OpenMP* API へのインテル® アプリケーション移行ツールは、OpenACC* コードの OpenMP* への移行に役立つこと、OpenMP* 5.0 をサポートしている場合、ユーザーはさまざまなハードウェア / ソフトウェア上でこれらのコードを実行できることを示しました。2 つのプログラミング・モデル間の暗黙的な違いにより、すべての OpenACC* アプリケーションを完璧に変換することはできませんが、このツールは移行作業の優れた開始点となるでしょう。現在、GEM 開発者は OpenACC* 移行済みコードのリポジトリー・フォーク（英語）を作成しています。この記事の公開時点で、リポジトリーに変更を加えることについて、POT3D 開発者と話し合いを行っています。

可能性のある将来の機能として、ユーザーのフィードバックに基づいて、対応する OpenMP*（存在する場合）への OpenACC* API 呼び出しの変換を追加しています。また、OpenACC* と OpenMP* の異なる非同期メカニズムに代わるマッピングを提供することにも取り組んでいます。最後に、オリジナルコード内の CPU OpenMP* ディレクティブを特定するためのいくつかの取り組みが行われており、CPU と GPU の両方をサポートする変換に OpenMP* メタディレクティブを使用することを検討しています。パフォーマンスの点では、多種多様なハードウェアとソフトウェアが利用可能であるため、パフォーマンスが最も高い変換を保証することはできません。パフォーマンス・ツールを使用してソースコードをチューニングすることを強く推奨します。

トランスフォーマー向けインテル® エクステンションを利用した効率的な自然言語埋め込みモデル

検索拡張生成の効率化

Yuwen Zhou、Zhenzhong Xu、Xin He、Bo Dong、Wenxin Zhang、および Haihao Shen
インテル コーポレーション

自然言語埋め込みの概要

自然言語埋め込みは、自然言語処理（NLP）に不可欠なもので、テキストを意味情報をキャプチャーするベクトルとして表します。これらの埋め込みは計算操作に必要な数値入力を提供するため、さまざまな下流の NLP タスクにとって重要です。

数ある埋め込みモデルの中でも、BGE（BAAI General Embedding）（英語）は効率が非常に優れています。small（英語）および base（英語）バージョンは、速度と効率のバランスが良く、テキスト埋め込みの理想的な選択肢となっています。BGE はベクトル・データベースとシームレスに統合され、テキスト埋め込み以外でも、その可能性をさらに拡大しています。

この記事では、BGE small モデルの速度と精度を大幅に向上するオープンソース・ツールのトランスフォーマー向けインテル® エクステンション（英語）を利用して、埋め込みモデルのパフォーマンスを向上させる革新的なアプローチを紹介します。

INT8 静的なトレーニング後の量子化

静的なトレーニング後の量子化（PTQ）は、追加のトレーニング段階を量子化する効果的なアプローチです。モデルの量子化パラメーター（スケール、ゼロ点など）を決定するには、代表的なデータセットを使用したキャリブレーションが必要です。bge-small-en-v1.5（英語）に精度を考慮した自動チューニングを備えた PTQ を適用して、最適な量子化モデルを生成します。以下のコード例は、トレーニング後の量子化を活用して BGE small モデルを最適化する方法を示しています。CQADupStack（英語）データセットをキャリブレーションに使用し、MTEB（英語）STS タスクを評価ベンチマークとして使用します。完全なコードはこちらから（英語）入手できます（ドキュメントは、readme（英語）を参照してください）。

```
from intel_extension_for_transformers.transformers import metrics, objectives,
QuantizationConfig
from intel_extension_for_transformers.transformers.trainer import NLPTrainer
# Replace transformers.Trainer with NLPTrainer
# trainer = transformers.Trainer(.....)
trainer = NLPTrainer(.....)
metric = metrics.Metric(
    name="eval_accuracy", is_relative=True, criterion=0.01
)
objective = objectives.performance
q_config = QuantizationConfig(
    approach="PostTrainingStatic",
    metrics=[metric],
    objectives=[objective]
)
model = trainer.quantize(quant_config=q_config, eval_func=mteb_sts_eval)
```

BGE で高速な NLP 推論のパフォーマンスを引き出す

このセクションでは、精度を損なうことなくこれらの BGE モデルの推論を高速化するように設計されたハイパフォーマンス NLP バックエンドを紹介します。軽量のベアメタル推論バックエンドの Apple Neural Engine*(ANE) を活用して、圧縮 NLP モデルのパフォーマンスを引き出します。ハードウェアとソフトウェアの両方の最適化を活用して、パフォーマンスを最大化します。

開発のプロセスを合理化するため、トランスフォーマー向けインテル® エクステンションで、Hugging Face の使い慣れたトランスフォーマー API と使いやすいモデル圧縮ツールを拡張します。このシームレスな統合により、ユーザーは ANE の機能を活用し、NLP モデルを最適化して推論を高速化し、生産性を向上させることができます。

開始方法を次に示します。

```
from transformers import AutoTokenizer
from intel_extension_for_transformers.transformers import AutoModel

sentences_batch = ['sentence-1', 'sentence-2', 'sentence-3', 'sentence-4']
tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-small-en-v1.5')
encoded_input = tokenizer(sentences_batch,
                          padding=True,
                          truncation=True,
                          max_length=512,
                          return_tensors="np")

engine_input = [encoded_input['input_ids'], encoded_input['token_type_ids'],
                encoded_input['attention_mask']]

model = AutoModel.from_pretrained('./model_and_tokenizer/int8-model.onnx',
                                  use_embedding_runtime=True)
sentence_embeddings = model.generate(engine_input)['last_hidden_state:0']
print("Sentence embeddings:", sentence_embeddings)
```

パフォーマンスの測定

MTEB STS 上で最適な量子化 BGE モデルを測定しました。すべてのモデルの精度相対損失は 1% 以内です(**表 1**)。

埋め込みレイテンシーとして、1 ソケット、24 コア / インスタンス、シーケンス長 = 512 の 1 つのインスタンス、バッチサイズ = 1 を使用して 1 つの文をエンコードする平均ミリ秒も測定しました（**図 1**)。

モデル	MTEB STS 平均 (10 データセット)	
	PyTorch* FP32	ITREX INT8
BAAI/bge-small-en-v1.5	81.59	81.43
BAAI/bge-base-en-v1.5	82.39	82.19
BAAI/bge-large-en-v1.5	83.11	82.82

表 1. 埋め込みモデルの FP32 と INT8 の精度

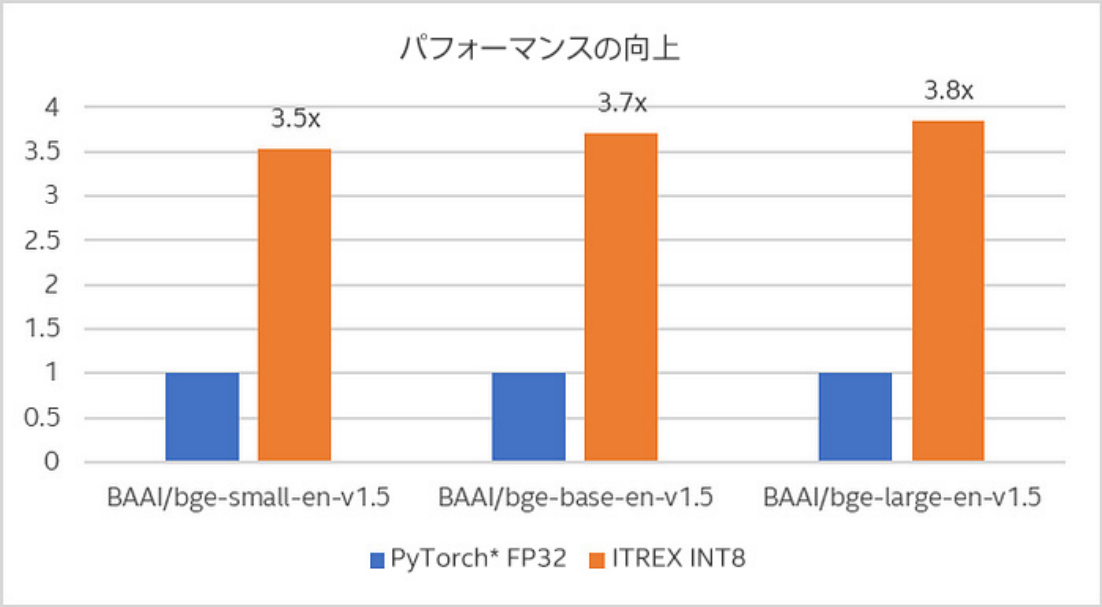

図 1. INT8 埋め込みモデルのパフォーマンスの向上

ハードウェア構成：インテル® Xeon® Platinum 8480+ プロセッサー、2 ソケット（ソケットあたり 56 コア）、2048GB RAM（16 スロット /128GB/4800MHz）、インテル® ハイパースレッディング・テクノロジー有効。OS : Ubuntu* 22.04.2LTS。**ソフトウェア構成：**Python* 3.9、NumPy* 1.26.3、ONNX* Runtime 1.13.1、ONNX* 1.13.1、Torch 2.1.0+cpu、Transformers 4.36.2。**テスト実施日：**2024 年 1 月 26 日。

最適化された BGE モデルを使用したチャットボットの構築

トランスフォーマー向けインテル® エクステンションで LangChain 埋め込み API を拡張し、次に示すようにユーザーが量子化された BGE モデルをロードできるようにしました。

```
from intel_extension_for_transformers.langchain.embeddings import
HuggingFaceBgeEmbeddings
embed_model = HuggingFaceBgeEmbeddings(model_name="/path/to/quantized/bge/model"
```

さらに、トランスフォーマー向けインテル® エクステンションの一部である NeuralChat と呼ばれるカスタマイズ可能なチャットボット・フレームワークを導入しました。このフレームワークを使用すると、複数のアーキテクチャー（インテル® Xeon® スケーラブル・プロセッサーやインテル® Gaudi® AI アクセラレーターなど）でチャットボットを迅速に構築できます。以下の例は、量子化された BGE small モデルを使用して知識検索向けのチャットボット・アプリケーションを開発する方法を示しています。

```
from intel_extension_for_transformers.neural_chat import plugins
from intel_extension_for_transformers.neural_chat import build_chatbot
from intel_extension_for_transformers.neural_chat import PipelineConfig

plugins.retrieval.enable = True
plugins.retrieval.args["input_path"] = "/path/to/docs"
plugins.retrieval.args["embedding_model"] = "/path/to/quantized/bge/model"
pipeline_config = PipelineConfig(model_name_or_path="facebook/opt-125m",
plugins=plugins)
chatbot = build_chatbot(pipeline_config)
response = chatbot.predict(query="What is Intel extension for transformers?")
```

まとめ

優れたパフォーマンスを達成するため、トランスフォーマー向けインテル® エクステンションを使用した埋め込みモデルの量子化と最適化の有効性を説明しました。品質を犠牲にすることなく推論の効率をさらに向上するため、ほかのモデル圧縮技術（プルーニングなど）も検討しています。ほかのインテル® AI ツールも試してみてください。最新の最適化に関する情報を受け取りたい場合は、トランスフォーマー向けインテル® エクステンションのリポジトリーに星を追加してください。プルリクエストを作成したり、リポジトリーに問題を送ることもできます。ご質問がある場合はお気軽にお問い合わせください。

1 時間（未満）で SYCL* を学ぶ

James Reinders インテル コーポレーション エンジニア

この記事では、C++ with SYCL* でプログラミングするために知っておくべき重要なことを紹介します。

ここでは最低限の要点のみ説明します。SYCL* に関する 500 ページの本に書かれていることをすべて伝えようとしているわけではありませんのでご安心ください。

基本を学んだ後、必要に応じてさらなる調査に使用できる小さなプログラムについて説明します。

SYCL* とは？

C++ with SYCL* を使用すると、ベンダー（NVIDIA、AMD、インテルなど）やアーキテクチャー（GPU、CPU、FPGA、DSP など）に関係なく、C++ プログラムからアクセラレーターを使用できるようになります。そのためには、SYCL* をサポートする C++ コンパイラーと、アクセラレーターをサポートするランタイム（ほとんどの場合、ベンダーの OpenCL* ランタイムなどのドライバー）が必要です。SYCL* 2020（現在の標準規格）では、OpenCL* 以外もサポートしています。このおかげで、SYCL* の実装では、NVIDIA の PTX、AMD の ROCm*/HIP、多くのベンダーの OpenCL*、インテルの SPIR-V*、複数ベンダーの OpenMP* など、さまざまな方法でハードウェアへの最適なパスを見つけることができます。インテル® コンパイラーはこれらのパスのいくつかを使用します。ハイデルベルク大学主導の SYCL* コンパイラー・プロジェクト（AdaptiveCpp）は、多くの革新的なパスを拓いたことでよく知られています。さまざまなアクセラレーターの SYCL* サポートを取得するための、多くのオプションがあります。

SYCL* は C++

SYCL* は C++ 向けに設計されていて、C++ プログラマーにとって非常に使いやすいものです。最小限の C++ の知識があれば、SYCL* を学習して使用できます。

示された手順に従う

C++ の知識を最大限に活用するには、tinyurl.com/learnSYCLnow（英語）の「Learn SYCL in an Hour (Maybe Less)（1 時間（未満）で SYCL* を学ぶ）」の手順に従ってください。この手順では、インテル® Tiber™ デベロッパー・クラウドにアクセスして、複数の GPU と必要なソフトウェアがインストールされた設定済みのシステムを使用する方法を説明しています。GitHub* からコード例を取得する方法についての情報も含まれています。

SYCL* には 3 つの鍵がある

SYCL* は、3 つの問題を解決する鍵として、次の機能を提供します。

1. 実行時に利用可能なアクセレーターの確認。
2. アクセラレーターとのデータの共有。
3. アクセラレーターへの計算のオフロード。

SYCL* には、オフロード計算の C++ エラー処理のサポートや、リダクション操作のビルトインサポートなど、便利な多くの追加機能が用意されています。これらの機能は、3 つの鍵をよく理解した後、必要に応じて学習すると良いでしょう。

アクセラレーターの検索 / 選択

アクセラレーターを検索 / 選択する際の目標は、アクセラレーターへの接続を取得して、データの共有とコードのオフロードができるようにすることです。SYCL* 用語では、これはキューを取得することを意味します。

キューは特定のアクセラレーターに接続します。キューは好きなだけ作成できます。必要に応じて、異なるキューを同じアクセラレーターに紐づけることもできます。サンプルプログラムでは、マシン上のすべてのアクセラレーターへのハンドルをキューの配列に埋め込みます。つまり、1 つだけ取得できる場合もあれば、複数取得できる場合もあります。インテル® Tiber™ デベロッパー・クラウドの手順に従うと、4 つ得られるでしょう（少なくともこの記事の執筆時点ではそうなっています）。

SYCL* は、実行時に利用可能なアクセラレーターを検索して選択する多くの制御を提供します。単純なコードから始めます。

```
sycl::queue q;
```

これで、すべてのデータ共有とオフロードに使用するハンドル `q` が得られます。この単純なケースでは、SYCL* ランタイムは単純にアクセラレーターを選択します。

私は通常、早い段階で名前空間 `sycl` を使用する `sycl::` キーワードを削除しますが、ここでは SYCL* を使用している部分が分かるように残しています。

SYCL* には常に利用可能なデバイスがあることに注意することが重要です。これは、常に動作する単純なプログラムを作成するときに非常に役に立ちます。アクセラレーターがないシステムでは、ホスト（私がこれまで見たすべての実装では CPU）が使用されます。

接続したデバイスを知りたい場合は、名前を出力します。

```
std::cout << “Running on “
          << q.get_device().get_info<sycl::info::device::name>();
```

アクセラレーターとのデータの共有

SYCL* ではデータの共有は簡単です。`malloc` に似たメモリー割り当てで USM（統合共有メモリー）を使用でき、その方法で割り当てられたメモリーはホストとアクセラレーター間で共有されます。通常は、ハードウェアで USM をサポートしているアクセラレーターでのみサポートされます。最新の GPU、CPU、FPGA は USM をサポートしているため、特に問題はありません。SYCL* は、ホストとアクセラレーター間で共有される明示的なバッファーもサポートしていますが、通常のポインターがホストとアクセラレーター間で動作することは許可していません。現時点では、バッファーを使用する場合を除いて、USM を使用することを推奨します。

サンプルプログラムでは円周率の桁を計算するジョブにバッファーを使用します。

コードは次のようになります。

```
        std::array<int, 200> d4;
        sycl::buffer outD4(d4);     // SYCL* バッファー
```

USM を使用するように変更する場合は、d4 の前の 2 行をコメントアウトして、次のコードを使用します。

```
        // SYCL* USM の割り当て
        auto d4 = (int *)sycl::malloc_shared( sizeof(int)*200, myQueue2 );
```

USM はポインターでのみアクセスできるため、アクセサー（outAccessor および myD4）を削除し、宣言をこれらのマクロに置換します。

```
        #define outAccessor d4
        #define myD4 d4
```

アクセラレーターへの計算のオフロード

コードを記述して、コードをアクセラレーターで実行するように指定します。Hello World! をアクセラレーターで実行する単純なコードを次に示します。

```
        q.submit([&](sycl::handler& cg) {|
                    auto os = sycl::stream{128, 128, cg};
                    cg.single_task(
                                        [=]() { os << “Hello World!\n”; });
                });
```

submit は、オフロードする操作があることを示します。single_task は、実行する単一の操作（ここでは Hello World! の出力）を指定します。single_task はオフロードする関数を指定できますが、ほとんどの場合、（このケースで行ったように）C++ ラムダ関数を使用してインラインで関数を指定します。

せっかく並列化によるパフォーマンス向上のためにアクセラレーターを使用しているのですから、もう少し複雑な処理を行ってみましょう。SYCL* はカーネルを並列に呼び出すプログラミング・スタイルを重視していることを知っておいてください。これは CUDA* や OpenCL* で使用されているプログラミング・スタイルと同じです。アイデアは単純です。1 つのデータを操作する単純なシリアルコードのカーネルを作成し、カーネルが各データ要素で別々に呼び出されるようにして、カーネルを並列に呼び出します。

サンプルコードでは実際にブラー処理を並列で行っています。このコードを理解することは難しくありません。コードを理解したら、SYCL* が少しずつ理解できるようになります。ここでは、単純に Hello World! を並列で実行するようにして、並列処理を行う簡単な例を示します。

```
// this is the entire program

#include <sycl/sycl.hpp>
int main(int argc, char* argv[]) {
  sycl::queue q;
  std::cout << “Running on “
            << q.get_device().get_info<sycl::info::device::name>()
            << “\n”;
  q.submit([&](sycl::handler& cg) {
             auto os = sycl::stream{1024, 1024, cg};
             cg.parallel_for(10, [=](sycl::id<1> myid)
                 {
                   os << “Hello World! My ID is “ << myid << “\n”;
                 });

         });
}
```

私のラップトップで、WSL で実行すると、次のように出力されました。

```
Running on Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz
Hello World! My ID is {5}
Hello World! My ID is {0}
Hello World! My ID is {7}
Hello World! My ID is {2}
Hello World! My ID is {1}
Hello World! My ID is {8}
Hello World! My ID is {6}
Hello World! My ID is {9}
Hello World! My ID is {3}
Hello World! My ID is {4}
```

サンプルプログラム

このプログラムを開始点として、さまざまな実験を行うことができます。キューはアクセラレーターに接続し、キューを使用してデータ共有を設定するか計算をオフロードすることに注意してください。サンプルプログラムを理解し、実験とさらなる学習を行うための手がかりとしては、これで十分でしょう。

オンラインで提供しているサンプルプログラムは私が作成したものです。基本を示すことを目的としており、効果的な並列プログラムを作成することは全く考慮していません。世界中のほかの SYCL* 学習リソースはすべて、よく考慮された並列プログラミングの例を示そうとしていると私は思っています。そこで、ほかとは異なることをして、楽しくプログラミングを始めるきっかけを示せないかと考えました。

サンプルプログラムは 3 つの異なるジョブを実行します。各ジョブは、アクセラレーターが利用可能であれば 3 つの異なるアクセラレーターのうちの 1 つで実行されます。利用できない場合は、ホスト上ですべて実行されます。

簡単に変更および理解できるさまざまなことを示すために、このコードを作成しました。最初のコードとして、非常に価値のあるものだと考えています。Exercise_02_... サブディレクトリーに移動すると、コードの一部に、例の 1 つを共有するため、バッファーの使用と USM の使用を切り替える `#ifdef` ディレクティブが含まれていることに気付かれるでしょう。

役立つテクニックを学びながらコーディングを楽しんでもらえるように、次の 2 つの注目すべきことをコードに追加しました。

- キューの配列を作成し、見つかったすべてのアクセラレーターを配列にロードします。次に、3 つの異なるジョブのために 1 つ目、2 つ目、または 3 つ目のアクセラレーターを取得します（実行時に見つかった実際のアクセラレーターの数を法とします）。コードが複雑に見えても驚かないでください。`sycl::queue q;` でキューを作成すること以外、コードは何も変わっていません。これにより、実行時にお気に入りのアクセラレーターを選択し、必要に応じてカスタム・アルゴリズムに一致させることができます。SYCL* に精通していなくてもさまざまなことができます。
- プロファイルを選択してキューを設定します（通常は利用できると思いますが、もう少し手を加えてサポートするデバイスでのみプロファイルを使用するロジックを追加しない限り、プログラムを実行できる場所は制限されます）。プロファイル・オプションを使用すると、アクセラレーター上での実際の実行時間に関する情報を収集できます。この情報は、カーネル自体のチューニングを行っている場合にほかのコードにより引き起こされるノイズを減らすことに役立つため、実時間よりも価値があります。カーネルのタイミングと実時間のタイミングを組み合わせると、アプリケーションをチューニングするときに優れたデータが得られます。

SYCL* に慣れたら、sycl.tech (英語) にリストされているリソース (私が共著した本、サンプルやチュートリアルなど) からさらに多くのことを学ぶことができるでしょう。

まとめ

この記事では 2 つの重要なことを学びました。(1) SYCL* は、C++ からアクセラレーターを使用するための 3 つの問題に対処しています。(2) 最初の SYCL* プログラムとして、時間を忘れて変更と実験に夢中になれる小さなプログラムがあります。

作業に適したツールを使用できるように、できるだけ多くのプログラミング・モデル / 言語を知ることは非常に価値があると私は信じています。アクセラレーターを使用する C++ を記述する場合、並列処理を表現するカーネルスタイルがアルゴリズムにとって理にかなっていて、アプリケーションをベンダーやアーキテクチャー間で高度に移植できるようにしたいのであれば、SYCL* を知っていることは非常に有益です。

SYCL* を学ぶことは難しいことではありません。効果的な並列プログラミングをマスターすること ... それは別の問題です。😊

コーディングを楽しみましょう！

インテル® C++ コンパイラーが Khronos SYCL* 2020 準拠の最初のコンパイラーに

Greg Lueck インテル コーポレーション インテル® C++ コンパイラー・チーム 主席エンジニア

インテル® C++ コンパイラーが Khronos SYCL* 2020 準拠の最初のコンパイラーに

GPU アクセラレーターが、AI から HPC、画像処理に至るまで、さまざまな並列ワークロードの重要なソリューションとなっていることは皆さんご存じでしょう。最近まで、これらのプラットフォーム向けのプログラミング・モデルは、アプリケーション開発者を単一ベンダーのハードウェアに縛り付ける固有の言語により支配されてきました。Khronos Group は、GPU やその他のオフロード・アクセラレーターをプログラミングする、新しいマルチベンダーのオープンな標準である SYCL* を定義することにより、この問題を解決しました。SYCL* は最新の C++ プログラミング機能をベースとしており、あらゆるベンダーの GPU デバイスや、フィールド・プログラマブル・ゲート・アレイ（FPGA）などのほかのアクセラレーター・デバイスをターゲットにできる API を提供します。

Khronos Group が 2021 年の初めに SYCL* 2020 仕様を承認した後、各ベンダーは仕様への完全準拠を達成するために懸命に取り組んできました。インテルは最近、インテル® oneAPI DPC++/C++ コンパイラーの 2024.1 リリースで準拠を達成した最初のベンダーになりました。

Khronos Group は、準拠であると主張するベンダーの言葉をただ受け入れているのではありません。SYCL* の仕様の公開に加えて、Khronos Group は包括的な SYCL* 準拠テストスイートも提供しています。準拠であると主張するには、ベンダーはまずコンパイラーがこのテストスイートに合格したことを証明する必要があります。この厳しさがアプリケーション・コードをあるベンダーのコンパイラーから別のベンダーのコンパイラーに移植する際に役立っていることを考えると、業界にとって有益な厳しさと言えるでしょう。

C++ with SYCL* で記述されたアプリケーションは異なるベンダーのハードウェア間で移植可能ですが、その責任はコンパイラーとアプリケーション開発者が負うことになります。根本的な課題は、GPU の機能がベンダーごとに異なり、アクセラレーターの種類（GPU や FPGA など）によっても異なることです。これらの違いを明らかにしないと、ほとんどの場合、コードのパフォーマンスが低下したり、ベンダーによりパフォーマンスが大幅に異なることが歴史的に証明されています。

SYCL* はこれらの違いを隠すのではなく、これらの違いをプログラマーに明らかにします。そしてアプリケーション開発者が、異なるベンダー間での移植性が重要かどうかを判断します。開発者が単一ベンダーのハードウェア上でのみアプリケーションを実行する場合は、そのハードウェアでサポートされている機能を単純に使用できます。開発者が移植可能なアプリケーションを作成したい場合は、SYCL* 言語の豊富なクエリーメカニズムを使用して各デバイスで提供される機能を条件付きで利用することにより、各ハードウェア・デバイスのパフォーマンスを最大限に引き出すことができます。SYCL* の利点は、さまざまな種類のアクセラレーターで使用できる共通のプログラミング言語を提供しながら、各アクセラレーターに固有の機能を利用できることです。

初の SYCL* 準拠コンパイラーの発表により、開発者はこの移植性を利用できるようになりました。アプリケーション開発者は、コードが仕様に準拠するほかのコンパイラーに移植可能であることを保証しながら、SYCL* 2020 仕様のフルセットの機能を利用できます。

アクセラレーターの世界で、SYCL* コンパイラーがどのような意味を持っているのかを問う必要があります。まず、SYCL* をサポートする C++ コンパイラーです。C++ コンパイラーは、OpenMP* などの別の標準の、ほかの並列化やオフロードもサポートしていることがあります。アクセラレーターへのデータと計算のオフロードを制御する追加のサポートを備えた C++ でプログラミングしていると知っておくことが重要です。次に、準拠であることがコンパイラーとランタイムの組み合わせにより証明されていることです。Khronos のウェブサイトには、厳しい提出、レビュー、受理を通過した準拠製品が詳細な情報とともにリストされています（リンクはこの記事の最後を参照）。

最後に、SYCL* と、この最初の SYCL* 準拠コンパイラーについて詳しく学ぶための優れたリソースのリストを紹介します。

SYCL* の仕様は、khronos.org/sycl（英語）から入手できます。

準拠製品のリストは、khronos.org/conformance/adopters/conformant-products/sycl（英語）に掲載されています。

インテル® oneAPI DPC++/C++ コンパイラーは、www.xlsoft.com/jp/products/intel/compilers/dpcpp/ から入手できます。

SYCL* の学習には、tinyurl.com/book-SYCL（英語）の無料の電子書籍と sycl.tech（英語）のリソースが役立ちます。

intel software

THE PARALLEL UNIVERSE