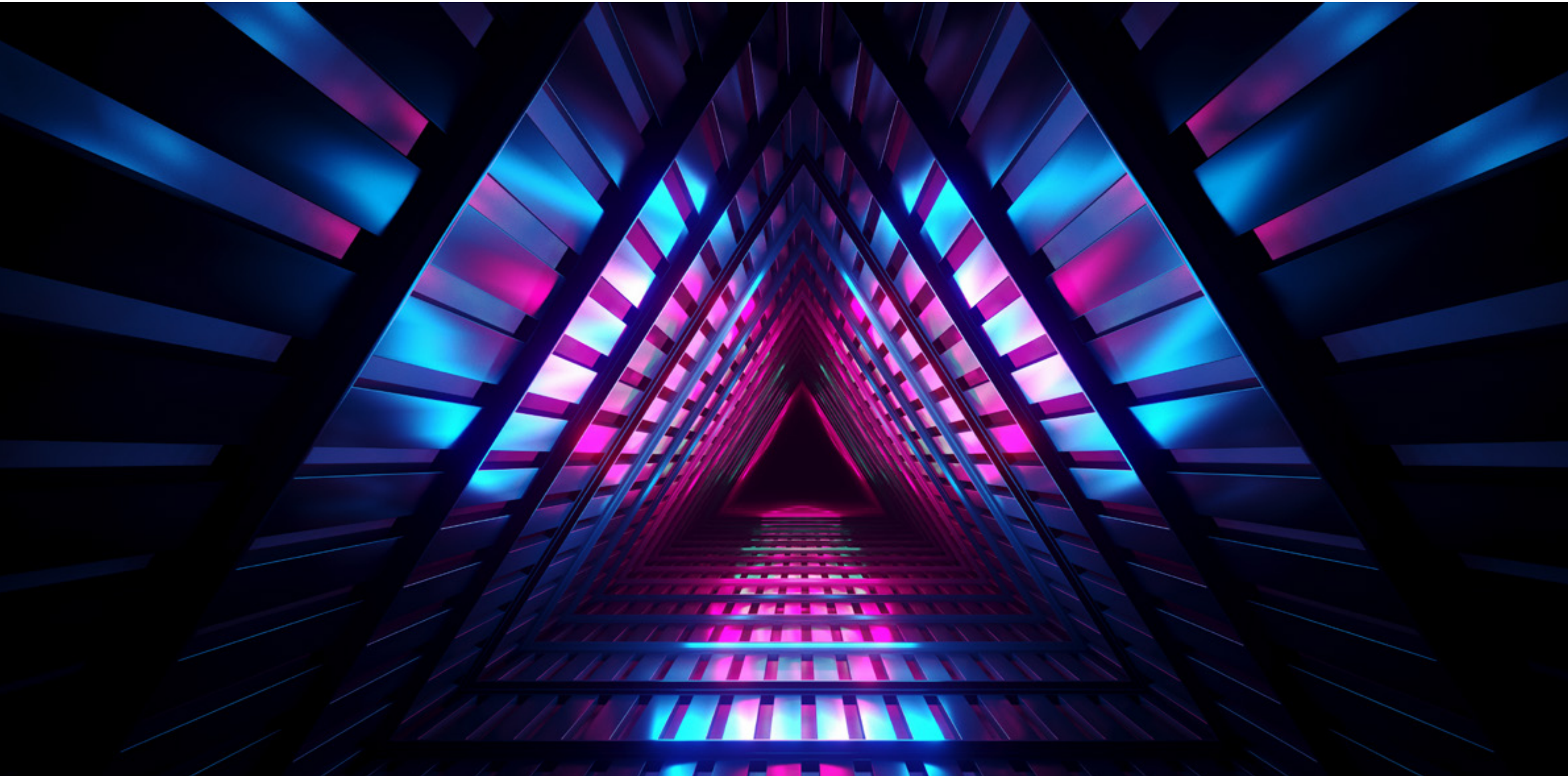


インテル® oneAPI を使用した ヘテロジニアス・プログラミング

複数のアーキテクチャーにわたる多様なワークロードで妥協のないパフォーマンスを提供

Nitya Hariharan インテル コーポレーション アプリケーション・エンジニア
Rama Kishan Malladi 同パフォーマンス・モデリング・エンジニア
Amarpal S. Kapoor 同テクニカル・コンサルティング・エンジニア
Kevin P O'Leary 同テクニカル・コンサルティング・エンジニア

今日のハードウェアが達成可能な最大パフォーマンスを引き出すには、ハードウェア機能を適切に使用することと、移植性が高く保守が容易な電力効率の良いコードのバランスをうまく取ることです。これらの要因は必ずしも連携して作用するとは限りません。ユーザーのニーズに応じて優先順位付けする必要があります。ユーザーにとって、アーキテクチャーごとに個別のコードベースを維持することは容易ではありません。スカラー、ベクトル、行列、および空間アーキテクチャーでシームレスに動作する標準化に準拠して簡素化されたプログラミング・モデルは、コードの再利用を高め、習得への投資を軽減することで開発者の生産性を高めます。

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

oneAPI は、これらの利点を提供するインテル主導の業界イニシアチブです。標準およびオープン仕様に基づいており、**データ並列 C++ (DPC++)** (英語) 言語とドメイン・ライブラリー群が含まれます。oneAPI の目標は、業界のハードウェア・ベンダーがそれぞれの CPU やアクセラレーター向けに独自の互換性のある実装を開発することです。これにより、開発者は複数のアーキテクチャーとベンダーデバイスにわたって、単一の言語とライブラリー API でコーディングすることができます。

インテル製 CPU とアクセラレーター向けのインテルのベータ版 oneAPI 開発ツール実装には、**インテル® oneAPI ベース・ツールキット** (英語) に加えて、異なるユーザー向けのいくつかのドメイン固有のツールキット—**インテル® oneAPI HPC ツールキット** (英語)、**インテル® oneAPI IoT ツールキット** (英語)、**インテル® oneAPI DL フレームワーク・デベロッパー・ツールキット** (英語)、および**インテル® oneAPI レンダリング・ツールキット** (英語)—があります。

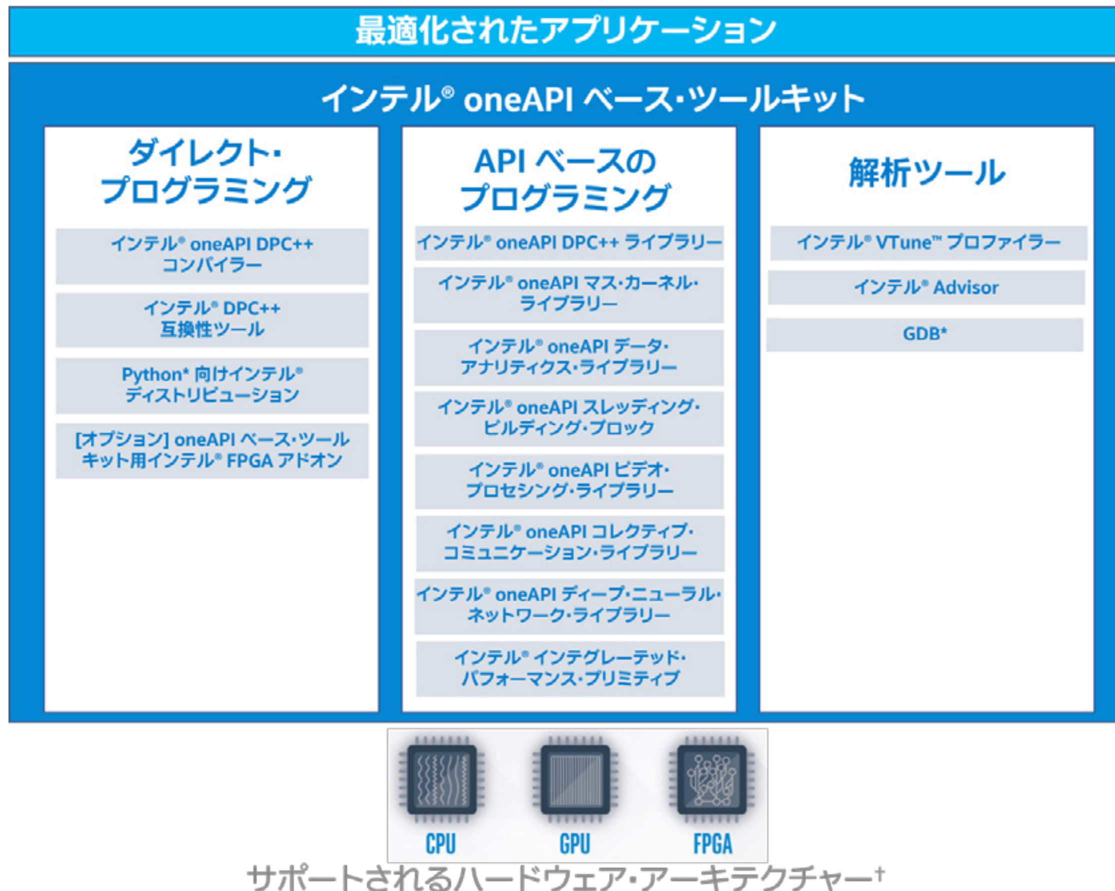
図 1 は、ベータ版**インテル® oneAPI** (英語) 製品とインテル® oneAPI ベース・ツールキットの各種レイヤーを示しています。インテル® oneAPI ベース・ツールキットには、インテル® oneAPI DPC++ コンパイラー、インテル® DPC++ 互換性ツール、複数の最適化ライブラリー、高度な解析およびデバッグツールが含まれます。さまざまなアーキテクチャーにわたる並列性は、Khronos Group の SYCL* をベースとした DPC++ 言語を使用して表現されます。DPC++ 言語は、最新の C++ 機能とインテル固有の拡張によりアーキテクチャーを効率良く使用します。コードは CPU で実行することも、利用可能なアクセラレーターへオフロードすることもできるため、コードの再利用が可能です。フォールバック・プロパティーにより、アクセラレーターが利用できない場合、コードは CPU で実行されます。ホストとアクセラレーターでの実行とメモリー依存関係は明確に定義されます。

インテル® DPC++ 互換性ツールを使用して、CUDA* から DPC++ へコードを移行することもできます。このツールは、開発者が短期間で移行できるように支援し、通常はコードの 80 ~ 90% を自動的に移行します。

DPC++ に加えて、インテル® oneAPI HPC ツールキットは、コードを GPU へオフロードできる OpenMP* 5.0 機能をサポートします。ユーザーは、DPC++ を使用するか、既存の C/C++/Fortran コードのオフロード機能を使用することができます。API ベースのプログラミングは、インテル® GPU 向けに最適化されたライブラリー群 (例えば、**インテル® oneAPI マス・カーネル・ライブラリー** (英語)) によってサポートされます。

ベータ版インテル® oneAPI 製品は、**インテル® VTune™ プロファイラー**[1] と**インテル® Advisor**[2] でも新しい機能を提供しており、アクセラレーターへオフロードしたコードをデバッグしたり、オフロードコードのパフォーマンス・メトリックを確認できます。

インテル® Advisor
現代のハードウェア向けにコードを最適化
詳細



1 **ベータ版インテル® oneAPI ベース・ツールキットのコンポーネント**

この記事では、ヘテロジニアス・プログラミングを推進するベータ版インテル® oneAPI 製品を紹介します。インテル® oneAPI ソフトウェア・モデルについて述べ、コンパイルモデルとバイナリー生成手順を説明します。インテル® oneAPI は、すべてのアーキテクチャーに対応した単一のバイナリーを生成するため、コンパイルとリンク手順が通常のバイナリー生成方法と異なります。最後に、いくつかのサンプルプログラムを検証します。この記事では、「アクセラレーター」、「ターゲット」、「デバイス」という用語を同じ意味で使用していることに注意してください。

インテル® oneAPI ソフトウェア・モデル

SYCL* 仕様ベースのインテル® oneAPI ソフトウェア・モデルは、コード実行とメモリー使用の観点からホストとデバイス間の相互作用を表します。モデルは 4 つの部分で構成されます。

1. **プラットフォーム・モデル**は、ホストとデバイスを指定します。
2. **実行モデル**は、デバイスで実行されるコマンドキューとコマンドを指定します。
3. **メモリーモデル**は、ホストとデバイス間のメモリー使用を指定します。
4. **カーネルモデル**は、計算カーネルのターゲットをデバイスにします。

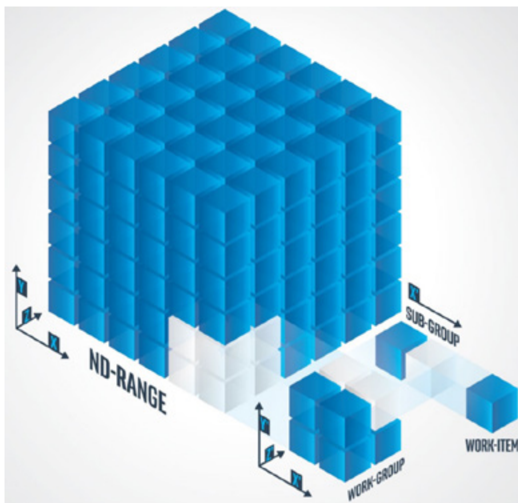
プラットフォーム・モデル

インテル® oneAPI プラットフォーム・モデルは、ホストと、互いにまたはホストと通信する複数のデバイスを指定します。ホストはデバイス上のカーネルの実行を制御し、複数のデバイスが存在する場合はそれらの間で調整します。各デバイスは、複数の計算ユニットを持つことができます。そして、各計算ユニットは、複数の処理要素を持つことができます。プラットフォームがインテル® oneAPI ソフトウェア・モデルの最小要件を満たしていれば、インテル® oneAPI 仕様は GPU、FPGA、ASIC など、複数のデバイスをサポートできます。通常、特定のオペレーティング･システム、特定の GNU* GCC バージョン、デバイスに必要な特定のドライバーがホストにインストールされている必要があります (プラットフォーム要件の詳細は、インテル® oneAPI の各コンポーネントのリリースノートを参照)。

実行モデル

インテル® oneAPI 実行モデルは、ホストとデバイスでのコードの実行方法を指定します。ホスト実行モデルは、カーネルの実行とホストとデバイス間のデータ管理を調整するためコマンドグループを作成します。コマンドグループは、インオーダーまたはアウトオブオーダー・ポリシーで実行可能なキューに送信されます。次のコマンドが実行される前にデバイス上の更新データをホストが利用できるように、キュー内のコマンドを同期できます。

デバイス実行モデルは、アクセラレーターでの計算処理を指定します。実行範囲には、一連の要素 (1 次元または多次元のデータセット) が含まれており、**図 2** の 3 次元の場合に示すように ND-Range、work-group、sub-group、および work-item の階層に分割されます。



2 **ND-Range、work-group、sub-group、および work-item の関係**

これは、インテル拡張である sub-group を除いて SYCL* モデルに似ています。work-item は、カーネル内の最小実行単位です。work-group は、work-item 間のデータ共有方法を決定します。この階層レイアウトは、パフォーマンスを向上するために使用すべきメモリーの種類も決定します。例えば、work-item は通常デバイスメモリーに格納されている一時データを操作し、work-group はグローバルメモリーを操作します。sub-group は、ベクトルユニットを備えたハードウェア・リソースをサポートするために追加されました。これにより要素の並列実行が可能です。

図 2 から ND-Range 内の work-group または work-item の場所が重要であることは明らかです。これは、計算カーネル内で更新されるデータポイントを決定します。各 work-item が操作する ND-Range のインデックスは、nd_item クラスの組込み関数 (**`global_id`**、**`work_group_id`**、**`sub_group_id`**、および **`local_id`**) を使用して決定されます。

メモリーモデル

インテル® oneAPI メモリーモデルは、ホストとデバイスによるメモリー・オブジェクトの処理を定義します。これは、アプリケーションのニーズに応じて、ユーザーがメモリーの割り当て場所を決定するのに役立ちます。メモリー・オブジェクトは、タイプバッファーまたはイメージとして分類されます。メモリー・オブジェクトの場所とアクセスモードは、アクセサーを使用して指定できます。アクセサーは、ホスト上のオブジェクト、デバイス上のグローバルメモリー、デバイスのローカルメモリー、またはホスト上のイメージに対して異なるアクセスターゲットを提供します。アクセスタイプは、読み取り、書き込み、アトミック、または読み書きです。

統合共有メモリーモデルにより、ホストとデバイスは明示的なアクセサーを使用せずにメモリーを共有できます。イベントを使用する同期は、ホストとデバイス間の依存関係を管理します。ユーザーは、ホストまたはデバイスで更新されたデータをいつ再利用できるようにするか制御するイベントを明示的に指定するか、暗黙的にこの決定をランタイムまたはデバイスドライバーに任せることができます。

カーネル・プログラミング・モデル

インテル® oneAPI カーネル・プログラミング・モデルは、ホストとデバイスで実行されるコードを指定します。並列処理は自動では行われません。ユーザーが言語構造を使用して明示的に指定する必要があります。

DPC++ 言語は、ホスト側で C++11 以降の機能をサポートできるコンパイラーを必要とします。しかし、デバイスコードには、C++03 機能と、ラムダ式、可変引数テンプレート、右辺値参照、エイリアス・テンプレートなどの特定の C++11 機能が必要です。**`std::string`**、**`std::vector`**、および **`std::function`** サポートも必要です。仮想関数と仮想継承、例外処理、ランタイム型情報 (RTTI)、および new と delete 演算子を使用するオブジェクト管理を含むデバイスコードには特定の制限があります。

ユーザーは、さまざまな方法でホストコードとデバイスコードを区別できます。ラムダ式は、カーネルコードとホストコードを一致させることができます。ファンクターは、ホストコードを同じソースファイルの別の関数として保持できます。OpenCL* コードを移行するユーザー、またはホストとデバイスコード間に明示的なインターフェイスを必要とするユーザー向けに、カーネルクラスは必要なインターフェイスを提供します。

ユーザーは、3 つの異なる方法で並列処理を実装できます。

- **単一のタスク :** 単一の work-item でカーネル全体を実行します。
- **`parallel_for` 構造 :** 処理要素全体でタスクを分散します。
- **`parallel_for_work_group`:** `parallel_for_work_group` 構造は、work-group 全体でタスクを分散します。work-group 内の work-item は、バリアを使用して同期できます。

インテル® oneAPI コンパイルモデル

インテル® oneAPI コンパイルモデルは、ビルドとリンク手順で構成されます。生成されるバイナリーは、複数のアクセラレーターでデバイスコードの実行をサポートする必要があります。つまり、DPC++ コンパイラーとリンカーは、バイナリーを生成するため追加のコマンドを実行する必要があります。通常、ユーザーが意識することはありませんが、これはターゲット固有のバイナリーを生成するのに役立ちます。

ホストコードのコンパイルは、標準の x86 アーキテクチャー向けの方法で行われます。アクセラレーター向けのバイナリーの生成は、1 つまたは複数のアクセラレーターに加えて、各アクセラレーターに固有の最適化をサポートする必要があるため、より複雑です。このアクセラレーター・バイナリーは、ファットバイナリーと呼ばれ、次のものが含まれます。

- **SPIR*-V (Standard Portable Intermediate Representation) 中間表現 :** デバイスに依存せず、コンパイル中にデバイス固有のバイナリーを生成します。
- **ターゲット固有バイナリー :** コンパイル時に生成されます。インテル® oneAPI は複数のアクセラレーターをサポートするため、複数のコード形式が作成されます。

clang ドライバー、ホストとデバイスの DPC++ コンパイラー、標準 Linux* (`ld`) または Windows* (`link.exe`) リンカー、およびファット・オブジェクト・ファイルを生成するツールなど、さまざまなツールがこれらのコード表現を生成します。実行時にインテル® oneAPI ランタイム環境は、ファットバイナリー内にデバイス固有のイメージがないかチェックし、利用可能な場合はそれを実行します。イメージが見つからない場合は、SPIR*-V イメージを使用してターゲット固有のイメージを生成します。

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

インテル® oneAPI プログラミングの例

このセクションでは、ベータ版インテル® oneAPI DPC++ コンパイラー、OpenMP* デバイスオフロード、インテル® DPC++ 互換性ツールのサンプルコードを紹介します。

DPC++ コードの記述

DPC++ コードを記述するには、DPC++ 言語の API と構文を使用します。**リスト 1** は、C++ (CPU) コードから DPC++ (ホストとアクセラレーター) コードへの変換例を示します。このサンプルコードは、GitHub* で公開されている Högbom CLEAN* アルゴリズムの実装です [4]。このアルゴリズムは、画像内の最大値を見つけて、観測の点広がり関数で畳み込まれたこの点光源の小さなゲインを減算して、最大値がしきい値よりも小さくなるまでこれを繰り返します。実装には 2 つの関数 **findPeak** と **subtractPSF** があります。これらは **リスト 1** と **2** に示すように、C++ から DPC++ へ移行する必要があります。

リスト 1. subtractPSF コードのベースラインと DPC++ 実装

```
subtractPSF C++ code: baseline
for (int y = starty; y <= stopy; ++y) {
  lhsIdx = y * residualWidth + startx;
  rhsIdx = (y - diffy) * psfWidth + (startx - diffx);
  for (int x = startx; x <= stopx; ++x, lhsIdx++, rhsIdx++)
    residual[lhsIdx] -= MUL * psf[rhsIdx];
}
```

```
subtractPSF DPC++ code
sycl::gpu_selector device_selector;
sycl::queue d_queue(device_selector);
sycl::buffer<float, 1>  a_device(psf.data(), psf.size());
sycl::buffer<float, 1>  b_device(residual.data(), residual.size());
unsigned long size_par = (stopy - starty);
sycl::range<1> a_size{size_par};
auto offset = sycl::id<1>(starty);

d_queue.submit([&](sycl::handler &cgh) {
  auto a_in  = a_device.get_access<sycl::access::mode::read>(cgh);
  auto b_res = b_device.get_access<sycl::access::mode::write>(cgh);
  cgh.parallel_for<class T>(a_size,offset,[=](sycl::id<1> idx) {
    int y = idx[0];
    int lhsIdx = y * residualWidth + startx;
    int rhsIdx = (y - diffy) * psfWidth + (startx - diffx);
    for (int x = startx; x < stopx; ++x, lhsIdx++, rhsIdx++) {
      b_res[lhsIdx] -= MUL * a_in[rhsIdx];
    }
  });
});
d_queue.wait_and_throw();
```

C/C++ から DPC++ へ移行するには、次のようなコード変更が必要です。

- **特定のデバイスのデバイスキューを追加** (デバイスセレクター API を使用)
- **デバイス上でバッファーを作成 / アクセス** (`sycl::buffer/get_access API` を使用)
- `parallel _ for` **を呼び出して**計算カーネルをスポーン / 実行
- **カーネル実行の完了を待機** (およびオプションで例外をキャッチ)
- **インテル® DPC++ コンパイラーとオプションを使用 :** `dpcpp -std=c++11 -O2 -lsycl -lOpenCL`

リスト 2 に `findPeak` 関数のコード変更を示します。ハードウェアの並列処理能力をうまく引き出せるように、DPC++ コードは `local_work_size`、`global_id/local_id`、`workgroup`、および OpenCL* や OpenMP* の構造に似た多くの API をサポートしています。

リスト 2. `findPeak` コードのベースライン (上) と DPC++ (下) 実装。`clPeak` は値と位置データの構造体です。work-group の同時実行は、グローバル ID とローカル ID、および work-group 内の複数のスレッド (work-item) 間のバリア同期を使用して行います。この parallel_for 実行の結果をレデュースして (リストでは示していません)、work-group 全体の最大値と位置を特定します。

```
findPeak: Baseline
maxVal = 0.0;
maxPos = 0;
const size_t size = image.size();
for (size_t i = 0; i < size; ++i) {
   if (abs(image[i]) > abs(maxVal)) {
      maxVal = image[i];
      maxPos = i;
   }
}
```

```
findPeak: DPC++
auto bufacc = buf.get_access<sycl::access::mode::read>(cgh);
auto resacc = res.get_access<sycl::access::mode::read_write>(cgh);
sycl::accessor<clPeak, 1, sycl::access::mode::read_write,
               sycl::access::target::local>
               local_res(sycl::range<1>(local_size), cgh);
cgh.parallel_for<class ex1>
                (sycl::nd_range<1>(sycl::range<1>(global_size),
                 sycl::range<1>(local_size)),[=](sycl::nd_item<1> item)
{
  size_t global_id = item.get_global_id(0);
  size_t local_id  = item.get_local_id(0);
  size_t local_dim = item.get_local_range(0);
  size_t group_id  = item.get_group(0);

  local_res[local_id].val = 0.0;
  local_res[local_id].pos = 0;

  if(fabs(bufacc[global_id]) > fabs(local_res[local_id].val)){
    local_res[local_id].val = bufacc[global_id];
    local_res[local_id].pos = global_id;
  }

  item.barrier(sycl::access::fence_space::local_space);
  if (local_id == 0) {
    resacc[group_id].val = 0.0;
    resacc[group_id].pos = 0;
    for (int i=0; i  < local_dim; i++) {
      if(fabs(local_res[i].val) > fabs(resacc[group_id].val)){
        resacc[group_id].val = local_res[i].val;
        resacc[group_id].pos = local_res[i].pos;
      }
    }
  }
});
```

OpenMP* オフロードサポート

ベータ版インテル® oneAPI HPC ツールキットは、OpenMP* オフロードサポートを提供します。これにより、ユーザーは OpenMP* デバイスオフロード機能を利用できます。OpenMP* プラグマを使用して C++ で記述されたオープンソースの Jacobi コード[3] を見てみましょう。このサンプルコードは、メインの反復ステップで次の処理を行います。

- Jacobi 更新を計算
- 古い解と新しい解の差異を計算
- 古い解を更新
- 残差を計算

反復コードを**リスト 3** に示します。

リスト 3. OpenMP* プラグマを使用するサンプル Jacobi ソルバー

```
// Iterate M times
#pragma omp parallel private (i, t)
{
#pragma omp for { // Jacobi update on xnew using b and x }
#pragma omp for reduction (+:d) { // Calculate difference d }
#pragma omp for { //overwrite x with x_new }
#pragma omp for reduction (+:r) { // Calculate residual r }
}
```

リスト 4 は、**`omp target`** 節を使用してデバイス環境に転送するデータを指定する、更新後のコードです。この節は、**`to`**、**`from`**、**`tofrom`**、または **`alloc`** のいずれかのデータ修飾子とともに使用できます。配列 **`b`** は変更されないため、**`to`** を使用します。また、**`x`** と **`xnew`** はオフロード・ディレクティブの前に初期化され、デバイス環境内で更新されるため、**`tofrom`** を使用します。リダクション変数 **`d`** と **`r`** も各反復中に設定および更新され、**`tofrom map`** 節を持ちます。

リスト 4. OpenMP* オフロードプラグマを使用する更新後のサンプル Jacobi ソルバー

```
// Iterate M times
#pragma omp target data map(tofrom:x,xnew) map(to:n,b) map(tofrom:d,r)
{
#pragma omp parallel for { // Jacobi update }
#pragma omp parallel for reduction (+:d) { // Calculate difference d }
#pragma omp parallel for { // Overwrite x_old with x_new }
#pragma omp parallel for reduction (+:r) { // Calculate residual r }
}
```

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

インテル® oneAPI コンパイラーでオフロード・ターゲット・コードをコンパイルするには、以下を設定する必要があります。

- コンパイラー・パスに関連する**環境変数**
- **関連する**ライブラリー
- **各種**コンポーネント

これらの環境変数へのパスは、ユーザーマシンのインテル® oneAPI 設定に依存します。ここでは、仕様の使いやすさを実証するため、どのマシンでも同様であるコンパイルプロセスを見てみます。コードをコンパイルするには、次のように LLVM ベースの **icx** または **icpc -qnextgen** コンパイラーを使用します。

```
$ icpc -qnextgen -fiopenmp -fopenmp-targets=spir64 -D__STRICT_ANSI__ jacobi.cpp
-o jacobi
```

-D__STRICT__ANSI__ オプションは、GCC 7.x 以上のシステムとの互換性を保証します。**spir64** オプションは、コードのターゲットに依存しない表現を参照して、リンク時または実行時にターゲット固有のコードに移行します。コードを実行するには、次のコマンドを実行します。

```
$ export OMP_TARGET_OFFLOAD="MANDATORY"
$ export LIBOMPTARGET_DEBUG=1
$./jacobi
```

OMP_TARGET_OFFLOAD の **MANDATORY** は、オフロードを GPU で実行する必要があることを示します。デフォルトでは、オフロードを CPU と GPU で実行できることを示す **DEFAULT** に設定されます。**LIBOMPTARGET_DEBUG** オプションを設定すると、デバッグに役立つオフロードランタイム情報が提供されます。

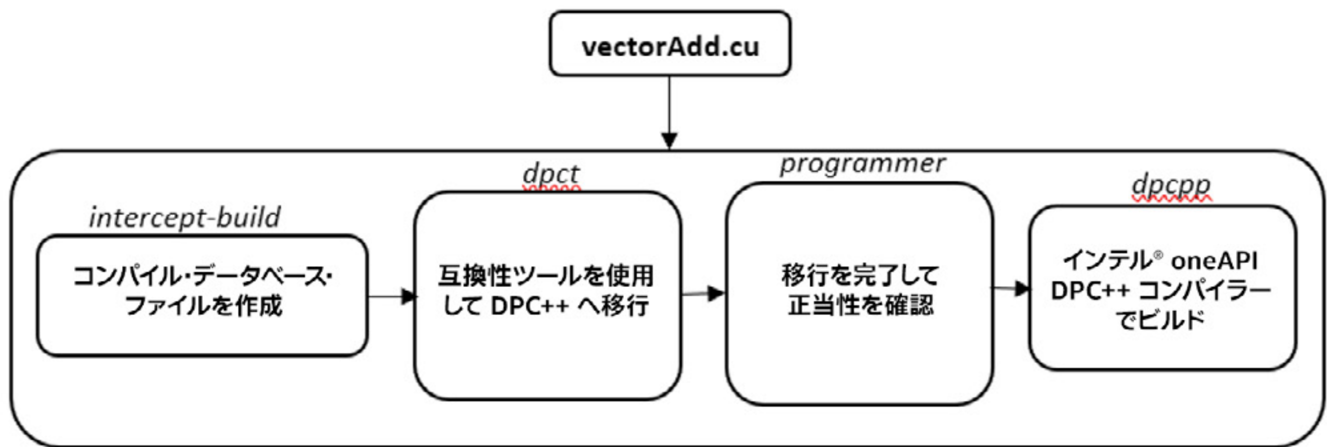
この OpenMP* オフロードサポートの例は C/C++ プログラム向けですが、Fortran オフロードもサポートされています。そのため、Fortran コードベースの HPC ユーザーも GPU 上でコードを実行できます。

インテル® DPC++ 互換性ツール

インテル® DPC++ 互換性ツールは、インテル® oneAPI ベース・ツールキットの一部として提供されているコマンドラインベースのコード移行ツールです。その主な役割は、既存の CUDA* ソースを DPC++ に移行することです。自動移行できないソース位置は適切なエラーと警告で示されます。また、インテル® DPC++ 互換性ツールは、ユーザーの判断が必要なソース位置にコメントを追加します。

図 3 は、CUDA* ソースコードから DPC++ へ移行する際の典型的なワークフローです。インテル® DPC++ 互換性ツールは、現在 Linux* および Windows* オペレーティング・システムをサポートしています。この記事では Linux* 環境を例に説明します。インテル® DPC++ 互換性ツールは、CUDA* SDK に含まれるヘッダーファイルを必要とします。移行プロセスを実証するため、ここでは CUDA* SDK 10.1 の VectorAdd サンプルを使用します。このサンプルは、通常以下の場所にあります。

```
$ ls /usr/local/cuda-10.1/samples/0_Simple/vectorAdd
```



3 **既存の CUDA* アプリケーションを移行するための推奨ワークフロー**

`VectorAdd` は約 150 行からなる単一ソースのサンプルです。このサンプルの CUDA* カーネル・デバイス・コードは、配列 **`A`** と **`B`** のベクトル加算を行い、結果を配列 **`C`** に格納します。

ここに示すコマンド、パス、および手順は、この記事の執筆時点のものであることに注意してください。製品の最終バージョンでは変更される可能性があります。

インテル® DPC++ 互換性ツールを使用するように環境を初期化するには、次のコマンドを実行します。

```
$ source /opt/intel/inteloneapi/setvars.sh
```

`setvars.sh` スクリプトは、インテル® DPC++ 互換性ツールだけでなく、インテル® oneAPI ベース・ツールキットのすべてのツールを利用できるように環境を初期化します。

ここでは、**リスト 5** に示す CUDA* Makefile の簡易バージョンを使用します。

リスト 5. CUDA* コードを DPC++ に移行する Makefile

```
$ cat Makefile
CC=/usr/local/cuda-10.1/bin/nvcc
CCFLAGS=-m64 -gencode arch=compute_50,code=sm_50 -gencode
arch=compute_52,code=sm_52 -gencode arch=compute_60,code=sm_60 -gencode
arch=compute_61,code=sm_61 -gencode arch=compute_70,code=sm_70 -gencode
arch=compute_75,code=sm_75 -gencode arch=compute_75,code=compute_75
OBJECT=vectoradd.o
SOURCE=vectoradd.cu
main:
	$(CC) $(CCFLAGS) -o $(OBJECT) -c $(SOURCE)
clean:
	rm -rf vectoradd.o
```

次の手順は、Makefile の実行時に発行されるコマンドをインターセプトして、JSON 形式のコンパイル・データベース・ファイルに格納します。インテル® DPC++ 互換性ツールには、これを行うため **`intercept-build`** と呼ばれるユーティリティーが用意されています。次のコマンドで起動します。

```
$ intercept-build make
```

そして、実際の変換処理を実行します。

```
$ dpct -p compile_commands.json --in-root=. --out-root=dpct_output vectorAdd.cu
```

`--in-root` と **`--out-root`** オプションは、ユーザー・プログラム・ソースの場所と移行後の DPC++ コードの出力先を設定します。この処理は、**`./dpct_output/vectorAdd.dp.cpp`** を生成します。

ベクトル加算が統合 GPU で実行されるように、デフォルトのキューへ送信する代わりに明示的に GPU キューを指定します。サポートされるプラットフォームの一覧と各プラットフォームのデバイスの一覧は、**`get_platforms()`** と **`platform.get_devices()`** を呼び出して取得できます。ターゲットデバイスを検出すると、統合 GPU 用のキューが構築され、このキューにベクトル加算カーネルがディスパッチされます。この手法は、複数の独立したカーネルを同じホスト / ノードに接続されたさまざまなターゲットデバイスへオフロードする際にも使用できます。

次に、変更した DPC++ コードをコンパイルします。

```
$ dpcpp -std=c++11 -I=/usr/local/cuda-10.1/samples/common/inc
vectorAdd.dp.cpp -lOpenCL
```

結果バイナリーを呼び出し、**リスト 6** に示すようにベクトル加算が統合 GPU 上で実行されることを確認します。

リスト 6. 移行した DPC++ コードの統合 GPU での実行結果

```
$ ./a.out
[Vector addition of 5000000 elements]
Platform: Intel(R) OpenCL HD Graphics
 Device: Intel(R) Gen9 HD Graphics NEO
Platform: Intel(R) OpenCL
 Device: Intel(R) Core(TM) i7-7567U CPU @ 3.50GHz
Platform: SYCL host platform
 Device: SYCL host device
Copy input data from the host memory to the device
kernel launch with 19532 blocks of 256 threads
Running on :Intel(R) Gen9 HD Graphics NEO
Copy output data from the device to the host memory
Test PASSED
```

ツールに関する詳細は、次のヘルプオプションで確認できます。

```
$ intercept-build -h
$ dpct -h
```

複数のアーキテクチャーにわたる多様なワークロードで妥協のないパフォーマンスを提供

この記事では、oneAPI とベータ版インテル® oneAPI ツールキットについて紹介し、インテル® oneAPI ベース・ツールキットのコンポーネントの概要を説明しました。ベータ版には、HPC、AI、解析、ディープラーニング、IoT、およびビデオ解析アプリケーションの oneAPI への移行を支援する各種ツールキットが用意されています。『DPC++ プログラミング・ガイド』は、アクセラレーター・パフォーマンスを最適化するさまざまな構造に関する詳細を提供します。記事で紹介した OpenMP* の例は C++ プログラム向けですが、GPU オフロードは C や Fortran でもサポートされます。oneAPI は、コードを移行してさまざまなアクセラレーターで実行するのに必要なソフトウェア・エコシステムを提供します。

参考資料

1. **インテル® VTune™ プロファイラー**
2. **インテル® Advisor**
3. **OpenMP* を使用した Jacobi ソルバー (英語)**
4. **https://github.com/ATNF/askap-benchmarks/tree/master/tHogbomCleanOMP (英語)**

HIGHLIGHTS

oneAPI に関するお客様の声

世界中の企業、大学、機関が oneAPI エコシステムをサポートしており、その輪は拡大しています。

この記事の続きはこちら (英語) でご覧になれます。>