C++ Thrust アプリケーションを SYCL* と oneAPI DPC++ ライブラリー（oneDPL）へ移行する

oneAPI で C++ アプリケーションを現代化および高速化し、ベンダーへの依存を排除する

Pablo Reble インテル コーポレーション ソフトウェア・エンジニア

プログラマーはライブラリーが大好きです。単に怠け者だからというだけでなく、ライブラリーが生産性とパフォーマンスを向上するからです。よくあるパターンを最適化するのに時間を浪費し、ターゲット・ハードウェアが変更されるたびにまた同じことをするのは無意味です。パターンを特定する作業はすでに終わっているため、ISO C++ 標準テンプレート・ライブラリー（STL）のような事前定義の抽象化機能を使うべきです。これらはすぐに使えて、専門家によって最適化されています。

STL は CPU だけをターゲットにしているのであれば最適ですが、コンピューティングの世界はヘテロジニアスです。CPU だけでなく、それ以外の選択肢も必要です。NVIDIA* Thrust ライブラリーと oneAPI データ並列 C++ ライブラリー（oneDPL）は CPU 以外にも対応しています。この 2 つを比較してみましょう。どちらもオープンソース・プロジェクト[1,2] ですが、Thrust のデバイスサポートはプロプライエタリーなソフトウェア・コンポーネントに依存しています（CUB*/CUDA* を通じてのみ GPU をサポートしています）。つまり、ベンダー依存です。一方、oneDPL は SYCL* をベースにしており、複数のベンダーのアクセラレーターをサポートするように設計されています。[3]

[1] oneDPL リポジトリー：https://github.com/oneapi-src/oneDPL（英語）
[2] Thrust リポジトリー：https://github.com/NVIDIA/thrust（英語）
[3] 「SYCL* の事例：ISO C++ がヘテロジニアス・コンピューティングに十分でない理由」

Thrust からベンダーに依存しない oneDPL への移行を検討するのは当然と言えます。ここでは、2 つの簡単な例を使用して、ツールを使用するアプローチと手動でコーディングし直すアプローチの 2 つの補完的な移行戦略について見ていきます。CUDA* を SYCL* に移行する SYCLomatic ツールについては、皆さんすでにご存知かもしれません。[4] ツールを使用した移行には限界があり、特にテンプレート化されたコードでは、必ずしも機能的ではなく、生成されるコードの品質や保守性には限界があります。例えば、ラムダ関数や自動型推論のような新しい C++ の機能を導入する場合、手作業による編集やコードの現代化が通常必要になります。移行後のコードは、オープン・スタンダードに準拠した、最新の C++ に沿ったものになります。Thrust は登場から 10 年が経っており、そろそろコードをアップグレードする時期と言えます。いくつかのコード例を見てみましょう。

最初の例は、シーケンス中のいくつかの値をインプレース変換します。変換する値はマスクで定義します。この関数は、値を格納するストリームと、ステンシルシーケンスを格納するストリームの 2 つの入力ストリームを受け取ります。最初のシーケンスの各値は、2 番目の入力シーケンスのステンシル値が特定の条件（コード例では非ゼロのプレディケート）を満たす場合に変換（コード例では符号反転）されます。

```
// Thrust コード例 (ステージ 0)
thrust::transform_if(dev_inp,
                     dev_inp + 10,
                     dev_stencil,
                     dev_inp,
                     thrust::negate<int>(),
                     thrust::identity<int>());
```

oneDPL は標準 C++ アルゴリズムと密接に連携しています。STL はステンシルのオーバーロードをサポートしていないため、oneDPL も同様にサポートしていません。このパターンは別の方法で表現する必要があるため、SYCLomatic は、ドロップイン置換として代替案を提供しています。

```
// 移行後のコード (ステージ 1)
dpct::transform_if(oneapi::dpl::execution::make_device_policy(syclQueue),
                   dev_inp,
                   dev_inp + 10,
                   dev_stencil,
                   dev_inp,
                   std::negate<int>(),
                   dpl::identity<int>());  // identity は C++20 の機能
```

ラムダ関数をカスタム・ファンクターとして使用し、標準 C++ コードを作成することで、さらに改善できます。この方法では、ステンシルシーケンスが一般的な入力シーケンスとは異なる方法で処理されます。プレディケートの評価はカスタム・ファンクターの内部で行われ、ライブラリーの内部に隠されることはありません。

```
// 手動編集後 (ステージ 2)
dpl::transform(dpl::execution::dpcpp_default,
               dev_inp,
               dev_inp + 10,
               dev_stencil,
               dev_inp,
               [&](const auto& input, const auto& mask)
                  {
                     return mask ? std::negate<>()(input) : input;
                  });
```

[4] SYCLomatic リポジトリー：https://github.com/oneapi-src/SYCLomatic（英語）。この記事で紹介した SYCLomatic の例は、説明を目的としたものであることに注意してください。インテル® DPC++ 互換性ツールなどの実装では、実際の結果は異なる場合があります。

これで、SYCL* デバイスをターゲットにするために必要なカスタム実行ポリシーを除いて、STL の動作のみを使用してパターンを表現できました。これは、Thrust と oneDPL を区別する重要な点です。Thrust には、入力イテレーターがアルゴリズムの実行場所を定義するデフォルトモードがあります。これは、実行パラメーターがイテレーターではなく、実行ポリシーによって定義される ISO C++ とは異なります。oneDPL は標準 C++ アルゴリズムと密接に連携し、カスタムポリシーを提供します。それを使って、アルゴリズムの実行場所を定義できます。便宜上、デフォルトの SYCL* デバイスをターゲットとする定義済みポリシーも用意されています。どちらのポリシーも、上記のステージ 1 およびステージ 2 のサンプルコードで使用されています。

次の移行例では、別のよくあるパターンであるソートについて見てみましょう。Thrust バージョンはキーでソートします。このアルゴリズムは 2 つの入力シーケンスを受け取ります。1 つは値を、もう 1 つはキーを保持しています。

```
// Thrust コード例 （ステージ 0）
thrust::stable_sort_by_key(dev_values, dev_values + 10, dev_keys);
```

値の入力シーケンスは、キーのシーケンスの要素を比較してソートされます。このパターンは既存の ISO C++ 関数にはないため、SYCLomatic は機能的な代替案を提供します。

```
// 移行後のコード （ステージ 1）
dpct::stable_sort(oneapi::dpl::execution::make_device_policy(syclQue),
                  dev_values,
                  dev_values + 10,
                  dev_keys);
```

カスタムのイテレーターとファンクターを使用することで、さらに改善できます。ZIP イテレーターを使用してキーと値のシーケンスを結びつけ、単一の反復処理空間を作成します。カスタム・ファンクターは、値とキーのペアでキー要素のみを比較します。その結果、若干冗長になりますが、カスタム・イテレーターのような汎用的な拡張を含む ISO C++ 関数のみを使用します。

```
// 手動編集後 （ステージ 2）
dpl::stable_sort(dpl::execution::dpcpp_default,
                 make_zip_iterator(dev_values, dev_keys),
                 make_zip_iterator(dev_values, dev_keys) + 10,
                 [](const auto& a, const auto& b)
                   {
                      return get<1>(a) < get<1>(b);
                   });
```

高速化された C++ STL アルゴリズムを実装したライブラリーは、開発者の生産性を大幅に向上できます。ツールによる移行ステップを手動コード編集で補完することで、保守性とコード品質を向上できます。将来の開発に向けて、範囲ベースの API と P2300R5 std::execution の提案（英語）は、現在の C++ の制限に対応し、この記事で説明したような並列パターンを表現する有力な代替手段です。ISO C++ でこれらの機能を実現するには、既存のアプリケーションを大幅に作り直す必要があります。今のところ、プロプライエタリーな言語やベンダー依存を避け、マルチベンダーのハードウェアに対応するには、可能な限り ISO C++ に従い、拡張機能を一般的なものにするのが最善です。