# 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！