インテル® oneAPI DPC++: OpenCL* および SYCL* テクノロジーとのカーネルと API の相互運用性

この記事は、インテル® デベロッパー・ゾーンに公開されている「Intel® oneAPI DPC++: Kernel and API interoperability with OpenCL* and SYCL* technology」の日本語参考訳です。

はじめに

この記事の内容

- OpenCL* C カーネルの取り込みと SYCL* での実行
- 純粋な SYCL* による単一ソースプログラムとの違い
- ヒント: 相互運用性機能、エラー処理、ビルドの推奨事項、精度の問題、計測
- 参考資料 (開発ツールとドキュメント)

この記事の対象者

- OpenCL* C 1.2 または OpenCL* C 2.0 で記述されているカーネルまたは SPIR-V 1.2 (Clang の OpenCL* 向け C++ (英語)) をターゲットとするカーネルがあるユーザー。SYCL* の取り込みを理解して、将来のアプリケーションを準備したいユーザー。
- 複数のヘテロジニアス計算 API をコードベースで使用したいユーザー。例えば、OpenCL* API と SYCL* プログラミングやほかの API を混在使用するユーザー。この環境では、次のことが可能です。
    - 同じプログラム内で OpenCL * API とほかの API を使用してメモリーを操作できます。
    - SYCL* プログラミングとオプションを理解することにより、定型コードを最小限に抑えることができます。

この記事の続きを読む前に、Tech.Decoded で 2019 年 12 月に公開されたインテル® oneAPI DPC++ 紹介ビデオのパート 1 (英語) およびパート 2 (英語) をご覧になることを推奨します (必須ではありません)。

必要条件

- インテル® oneAPI ベース・ツールキット DPC++ Beta3 以降がインストールされていること (2020 年の初めに一般公開)
    - インテル® CPU: ベータ版インテル® oneAPI: DPC++ (英語) の必要条件と同じです。
    - インテル® インテル® グラフィックス・テクノロジー: ベータ版インテル® oneAPI: DPC++ (英語) の必要条件と同じです。

ウォークスルー・ソースは、SYCL * 1.2.1 および OpenCL* C 1.2 の相互運用可能な実装で問題なく動作することを目的としています。インテル® oneAPI イニシアチブの目標は、ヘテロジニアス・コードの移植性を高めることです。

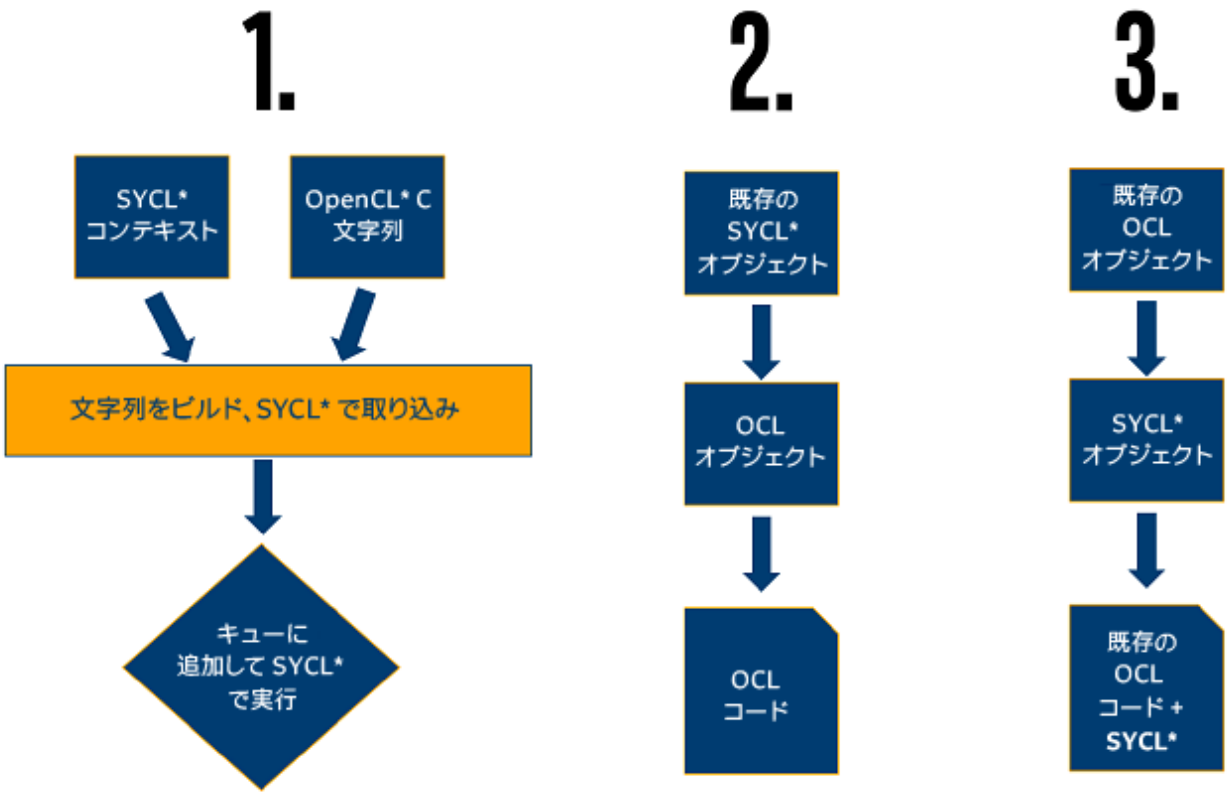
テスト・プラットフォームの構成

- インテル® Core™ i7-6770HQ プロセッサー、Linux* (Ubuntu* 18.04.4)
  - インテル® Iris® Pro グラフィックス 580
- OpenCL* 2.1 NEO 20.01.15264 (OpenCL* ドライバー向けインテル® グラフィックス計算ランタイム (英語))
- OpenCL* ICD ローダー・ライブラリー 2.2.12

Windows* およびほかのハードウェア・プラットフォームも利用できますが、このチュートリアルではテストしていません。

相互運用性のケース

OpenCL* と SYCL* プログラミングの相互運用性として 3 つのケースを考えます。



この記事では、最初のケースを説明します。このケースでは、開発者が既存の OpenCL* C カーネルに多くの投資を行っています。既存のヘテロジニアス・アプリケーションの開発者向け SYSL* ランタイム機能の入門編として利用してください。

ケース 2 およびケース 3 を可能にする API 機能は、この記事の後半で説明しています。

コードの比較

SYCL* プログラムを使用して OpenCL* C カーネルの取り込みを説明します。次に、比較のために、SYCL* のみの実装を示します。プログラムは、同じ SYCL*/OpenCL* 'gpu' ターゲットデバイスで実行します。

OpenCL* C カーネル取り込みプログラム

```
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
using namespace cl::sycl;

int main()
{
        const size_t szKernelData = 32;
        std::array<float, szKernelData> kernelData;
        kernelData.fill(-99.f);
        range<1> r(szKernelData);
        queue q{gpu_selector()};
        program p(q.get_context());
        p.build_with_source(R"CLC( kernel void sinf_test(global float* data)
{
                                data[get_global_id(0)] =
sin(get_global_id(0)*2*M_PI_F/get_global_size(0)) ;
                        } )CLC", "-cl-std=CL1.2");
        {
                buffer<float, 1> b(kernelData.data(), r);

                q.submit([&](handler& cgh) {
                        auto b_accessor =
b.get_access<access::mode::read_write>(cgh);
                        cgh.set_args(b_accessor);
                        cgh.parallel_for(r, p.get_kernel("sinf_test"));
                });
        }
        for(auto& elem : kernelData)
                std::cout << std::defaultfloat << elem << " " <<
std::hexfloat << elem << std::endl;
        return 0;
}
```

純粋な SYCL* プログラム

```
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
using namespace cl::sycl;

int main()
{
        const size_t szKernelData = 32;
        const float M_PI_F = static_cast<const float>(M_PI);
        std::array<float, szKernelData> kernelData;
        kernelData.fill(-99.f);
        range<1> r(szKernelData);
        queue q{gpu_selector()};
        program p(q.get_context());
        {
                buffer<float, 1> b(kernelData.data(), r);

                q.submit([&](handler& cgh) {
                        auto b_accessor =
b.get_access<access::mode::read_write>(cgh);
                        cgh.parallel_for(r, [=](nd_item<1> item) {
```

```
                                                b_accessor[item.get_global_id(0)] =
sin(item.get_global_id(0)*2*M_PI_F/item.get_global_range()[0]);
                                });
                        });
                }
                for(auto& elem : kernelData)
                        std::cout << std::defaultfloat << elem << " " <<
std::hexfloat << elem << std::endl;
                return 0;
}
```

OpenCL* C カーネル取り込みサンプルをビルドするには、次のコマンドを実行します。

```
dpcpp -fsycl-unnamed-lambda ingest.cpp -std=c++17 -o ingest
```

サンプルを実行するには、次のコマンドを実行します。

```
./ingest
```

純粋な SYSL* サンプルをビルドするには、次のコマンドを実行します。

```
dpcpp -fsycl-unnamed-lambda pure.cpp -std=c++17 -o pure
```

サンプルを実行するには、次のコマンドを実行します。

```
./pure
```

注: *-fsycl-unnamed-labmda* は、インテル® oneAPI ベースキット Beta04 以降の dpcpp ドライバーでデフォルト指定されるコンパイラー・オプションの一部です。ほかの SYCL* コンパイラーおよびランタイムでは、このコンパイラー・オプションがデフォルトで指定されない場合があります。

OpenCL* C カーネル取り込みプログラムの確認

32 の float 型の std::array をインスタンス化します。プログラムでエラーが発生した場合は、エクストリーム値 (-99.f) を設定します。

```
const size_t szKernelData = 32;
std::array<float, szKernelData> kernelData;
kernelData.fill(-99.f);
```

カーネルの起動時にバッファーのサイズとワークアイテムのグローバル数を設定する 1 次元の cl::sycl::range が作成されます。std::array のサイズに設定されます。

```
range<1> r(szKernelData);
```

cl::sycl::queue オブジェクトが作成され、*gpu_selector()* で提供されるデフォルトの GPU デバイスに関連付けられます。チュートリアル・システムでは、次のように設定されます。

- プラットフォーム: *Intel(R) OpenCL* HD Graphics*
- デバイス: *Intel(R) Gen9 HD Graphics NEO*
- デバイスバージョン: *OpenCL* 2.1 NEO*
- ドライバーバージョン: *20.06.15619*

```
queue q{gpu_selector()};
```

cl::sycl::queue と同じコンテキストで *cl::sycl::program* が作成されます。

```
program p(q.get_context());
```

OpenCL* C カーネルプログラムをビルドします。このサンプルでは、raw 文字列を示す R" 表記を文字列の前に使用しています。また、CLC(および)CLC 区切り文字を使用して複数行の raw 文字列をキャプチャーします。文字列は OpenCL* C カーネルを表します。2 つ目の引数は、OpenCL* C カーネル・ビルド・オプションのトグルリストで、OpenCL* C コンパイラーが OpenCL* C 1.2 標準のカーネルをコンパイルするように指定しています。*CL 1.2* コンパイルモードはデフォルトのモードであり、指定する必要はありませんが、ここでは完全を期すために指定しています。最新のインテルの実装で OpenCL* C カーネルプログラムをビルドする場合は、*-cl-std=CL2.0* も使用されます。

```
p.build_with_source(R"CLC( kernel void sinf_test(global float* data) {
                                data[get_global_id(0)] =
sin(get_global_id(0)*2*M_PI_F/get_global_size(0)) ;
                          } )CLC", "-cl-std=CL1.2");
```

取り込みオプション

- カーネルおよびビルドオプションは、ハードコードする代わりにディスクから読み取って取り込むこともできます。既存の OpenCL* C ソースベースで作業する場合は、このアプローチが一般的です。
- 実行時にカーネルを完全に生成することもできます。
- *-D* ビルドオプションを指定してプリプロセッサー・マクロ定義を使用し、実行時にカーネルを部分的に生成することもできます。
    - この「メタプログラミング」と呼ばれる実行時の部分的なカーネル生成は、多くの OpenCL* プロジェクトで一般的に行われています。

サンプルカーネルは、すべてのワークアイテムのストライドで正弦値を検出します。各ストライドは (2 * pi/ワークアイテムの総数) です。

```
kernel void sinf_test(global float* data) {
    data[get_global_id(0)] =
sin(get_global_id(0)*2*M_PI_F/get_global_size(0)) ;
}
```

プログラムは左中括弧から新しい範囲に入り、*cl::sycl::buffer* オブジェクトの作成と破棄を行います。このバッファーは 1 次元の単精度浮動小数点データで構成されます。この範囲で重要なのは、*cl::sycl::buffer<float, 1>* オブジェクトが右中括弧で破棄されることです。その結果として、ホストの可視メモリーを後続のホスト・プログラム・セクションで使用できるようになります。*delete* (C++) *free* (C) *clRelease* (OpenCL* API) などのメモリー・ストア・クリーンアップ呼び出しやメモリー解放呼び出しは、バッファー・オブジェクトの破棄後に行います。

```
        {
                buffer<float, 1> b(kernelData.data(), r);
...
        }
```

参照渡しラムダ関数によるキュー送信が定義されています。ラムダ関数には、後でキューに追加される操作に渡される SYCL* コマンド・グループ・ハンドラー (*cl::sycl::handler* オブジェクト) があります。

```
q.submit([&](handler& cgh) {
    auto b_accessor = b.get_access<access::mode::read_write>(cgh);
    cgh.set_args(b_accessor);
    cgh.parallel_for(r, p.get_kernel("sinf_test"));
});
```

SYCL* アクセサーは、*cl::sycl::buffer<>::get_access* メンバー関数を使用して *cl::sycl::buffer<float, 1>* オブジェクトから作成されます。アクセサーは、*cl::sycl::handler::set_args* 関数を使用して OpenCL* C カーネル関数の最初の (この場合のみ) 引数にバインドされます。

```
auto b_accessor = b.get_access<access::mode::read_write>(cgh);
cgh.set_args(b_accessor);
```

カーネルの実行は、*cl::sycl::handler::parallel_for* 関数を使用してスケジュールされます。合計ワークアイテムのサイズ (NDRange) は、32 のワークアイテムにより *cl::sycl::range<1>* オブジェクトで設定されます。カーネル関数は、*cl::sycl::program::get_kernel* 関数から指定されます。カーネルは、カーネル定義に使用された関数名で選択されます。

```
cgh.parallel_for(r, p.get_kernel("sinf_test"));
```

カーネルは、32 のワークアイテムに sin 関数を実行します。入力角度が 0 から 2pi ラジアンの、32 の正弦を含む出力配列が生成されます。

プログラムの最後で、配列から出力された出力正弦値が浮動小数点形式と 16 進浮動小数点形式で表示されます。

```
for(auto& elem : kernelData)
    std::cout << std::defaultfloat << elem << " " << std::hexfloat << elem <<
std::endl;
```

違いを調べる

純粋な SYCL* サンプルで異なる点は次の 3 つです。

1. このバージョンは、ホストの *M_PI* 倍精度 pi マクロを使用して、単精度浮動小数点にダウンキャストしています。*M_PI_F* は pi の値を表すために使用できる OpenCL* C 1.2 マクロです。*M_PI_F* および *M_PI* はすべてのプラットフォームに移植できるとは限らないことに注意してください。開発者によっては、カーネルとホスト側のプログラムに pi の値を直接提供しようとすることがあります。初期の標準規格を採用する開発者には、C++20 の *std::numbers::pi* も一貫性を高めるのに役立つでしょう。

   ```
   const float M_PI_F = static_cast<const float>(M_PI);
   ```

2. SYCL* のみプログラムでは、*cl::sycl::handler::set_args* 関数を使用してバッファーをカーネル・パラメーターにバインドする必要がないため、*cgh.set_args(b_accessor);* が削除されています。
3. カーネルは *cl::sycl::program::build_with_source* 関数を介してビルドされていません。この SYCL* バージョンは、*cl::sycl::handler::parallel_for* 関数内で定義されたラムダ関数カーネルをパラメーターとして使用しています。正弦演算は実行されます。出力値は、アクセサーを介してメモリーに書き込まれます。

   ```
   cgh.parallel_for(r, [=](nd_item<1> item) {
       b_accessor[item.get_global_id(0)] =
   sin(item.get_global_id(0)*2*M_PI_F/item.get_global_range()[0]);
   });
   ```

新規開発者向けのヒント

その他の相互運用性

この記事では、相互運用性の 1 つのケースとして SYCL* ベースの OpenCL* C カーネルの取り込みについて説明しました。通常、プロダクション・アプリケーションでは、さらに 2 つの相互運用性ユースケースがあります。

SYCL* オブジェクトは、基本となる相互運用可能な OpenCL* API オブジェクトを出力できます。既存の SYCL* プログラムを拡張して OpenCL* API も使用する場合は、これらの関数を検討してください。多くの SYCL* オブジェクトには、SYCL* オブジェクトで使用される OpenCL* オブジェクトを派生する *get()* メソッドが用意されています。

- cl::sycl::device
- cl::sycl::event
- cl::sycl::kernel
- cl::sycl::program
- cl::sycl::queue
- cl::sycl::context
- cl::sycl::platform

SYCL* オブジェクトは、コンストラクターのパラメーターとして提供される相互運用可能な OpenCL* API オブジェクトから構築することもできます。SYCL* ランタイム機能を既存の OpenCL* ソースベースに追加する場合は、これらのコンストラクターを検討してください。以下のオブジェクトはすべて、SYCL* コンストラクターで使用できます。以下のオブジェクトが相互運用可能な OpenCL* API 呼び出しから作成されている場合、SYCL* コンストラクターを使用して対応する SYCL* オブジェクトを作成できます。

- cl_device_id
- cl_event
- cl_kernel
- cl_program
- cl_command_queue
- cl_context
- cl_platform_id
- **cl_mem***

SYCL 1.2.1 仕様の時点で、*cl::sycl::image* および *cl::sycl::buffer* オブジェクトには、相互運用可能な既存の *cl_mem* オブジェクトに基づくコンストラクターがあります。*cl::sycl::image* および *cl::sycl::buffer* オブジェクトには、基本となる *cl_mem* オブジェクトにアクセスする *get()* メソッドがありません。

基本となる OpenCL* オブジェクトと OpenCL* API 呼び出しを SYCL* プログラムで直接使用する場合は、OpenCL* ICD ローダー・ライブラリー (-lOpenCL または OpenCL.lib) をプログラムにリンクすることを忘れないでください。この記事の最初のサンプルのように、SYCL* で直接取り込む場合には OpenCL* API 呼び出しも OpenCL* データ型も必要ないため、'dpcpp' コンパイラー・ドライバーを使用するときに OpenCL* ICD ローダー・ライブラリーをリンクする必要はありません。

相互運用可能なプログラムのビルド

DPC++/SYCL* および OpenCL* の相互運用可能なプログラムをビルドする場合は、次のソースから OpenCL* ヘッダーとライブラリーを入手することを推奨します。

- Khronos* Github* ポータルから
  - ヘッダー (英語)
  - ICD ローダー・ライブラリー (英語) リポジトリーからビルド
- インテル® oneAPI ベース・ツールキットの DPC++/SYCL* ディストリビューション
- システム・パッケージ・マネージャーから (例: apt install ocl-icd-libopencl1)

インテル® SDK for OpenCL* Applications およびインテル® System Studio に含まれているツールセットは**推奨しません**。

エラー処理

cl::sycl::queue には、次のような非同期例外ハンドラーの使用を許可するコンストラクターがあります。

```
auto async_exception_handler = [] (cl::sycl::exception_list exceptions) {
    for (std::exception_ptr const &e : exceptions) {
        try {
            std::rethrow_exception(e);
        }
        catch (cl::sycl::exception const &e) {
            std::cout << "Async Exception: " << e.what() << std::endl;
            std::terminate();
        }
    }
 };
queue q(gpu_selector(), async_exception_handler);
```

ハンドラーは、例外リストを反復処理して、コマンドキューの実行中に発生した非同期エラーをレポートします。

基本的な OpenCL* エラーは SYCL* 例外オブジェクトによりレポートされます。SYCL* 例外オブジェクトの動作は、OpenCL* API 呼び出しにより一般に書き込まれる *cl_err* データ型を使用してエラーを管理する労力をカプセル化します。OpenCL* から SYCL* 開発に参加する開発者は、例外処理を行うことを強く推奨します。例外ハンドラーを使用すると、エラーの場所と種類の特定が実用的になります。

例外処理のドキュメントは、SYCL* 1.2.1 仕様 (英語) 関数リファレンスを参照してください。

- *void cl::sycl::queue::wait_and_throw()*
- *void cl::sycl::queue::throw_asynchronous()*

ベースキット・サンプル・リポジトリー (英語) も参照してください。リポジトリーの vector-add および sepia フィルターのサンプルは、同期例外の try-catch 領域を示します。*cl::sycl::queue::wait_and_throw* 関数の try-catch および使用法を確認します。wait_and_throw() 関数は、ユーザー定義のハンドラーでキャッチされる未処理の非同期例外を示します。

このサンプルでは、*q.wait_and_throw()* と try-catch ブロックを使用しています。バッファー定義と *cl::sycl::queue::submit* は次のようにカプセル化できます。

```
try {
    buffer<float, 1> b(kernelData.data(), range<1>(szKernelData));
    q.submit([&](handler& cgh) {
        auto b_accessor = b.get_access<access::mode::read_write>(cgh);
        cgh.parallel_for(range<1>(szKernelData), [=](nd_item<1> item) {
            b_accessor[item.get_global_id(0)] =
sin(item.get_global_id(0)*2*M_PI_F/item.get_global_range()[0]);
        });
    });
    q.wait_and_throw();
} catch ( cl::sycl::exception const &e ) {
    std::cerr << "Sync exception: " << e.what() << std::endl;
    std::terminate();
}
```

注: 例外フィードバックをテストする簡単な例は、パラメーターとして *cl::sycl::handler* オブジェクトを省略した場合です。*cl::sycl::queue::submit(…)* 領域の *b.get_access<access::mode::read_write>(**cgh**)* から 'cgh' を削除してみてください。アプリケーションを再コンパイルして再実行し、例外を確認します。

コンパイル時エラーと実行時エラーの両方の特定のエラーレポート文字列を、ベータ版 DPC++ の一部として検討しています。必要に応じて、Khronos* の *CL/cl.h* ヘッダーで定義されている OpenCL* API エラーを参照してください。

精度と結果の検証

プラットフォームによって、同じ浮動小数点演算の結果が異なることがあります。含まれているトレーニング・サンプルで gpu_selector() を cpu_selector に変更すると、異なるハードウェアがターゲットにされて、違いが生じることがあります。

精度の問題に特に影響を受けるアプリケーションは、OpenCL* および SYCL* 仕様の精度の説明を確認する必要があります。ULP (最小桁の単位) のドキュメントを検索して、仕様で許容されているエラーを確認してください。

システムページのサイズまたは 2 の累乗にアライメントされたメモリー割り当てを使用することで、アプリケーションで一貫した動作を確実にできます。アライメントにより、ランタイムまたはドライバーレイヤーのメモリーの再編成や再パックを最小限に抑えることができます。C++17 のアライメントされたメモリー割り当てのリファレンスを次に示します。

- void* operator new (std::size_t count, std::align_val_t al)
- void* std::aligned_alloc(std::size_t alignment, std::size_t size)

C の等価の機能を次に示します。

- void *aligned_alloc(size_t alignment, size_t size); (C11)
- C オプション (英語) (GNU* コンパイラーで利用可能)
- _aligned_malloc (Microsoft* Visual Studio* で利用可能)

リダクション型アルゴリズムや浮動小数点データを広いベクトルにパックする実装では、浮動小数点オペランドの並べ替えが行われる可能性があります。並べ替えが行われると、同じプラットフォームで同じアプリケーションを実行した場合でも、実行結果が異なることがあります。次のことを検討してください。

- 計算の制約を緩和する可能性があるコンパイル・オプションを排除し、より厳密なコンパイル・オプションを選択します。
    - この設定は、ホスト・アプリケーションとカーネルプログラムで別々に適用できます。*clBuildProgram* 関数 (1.2、2.0、2.1) を使用してカーネルをビルドする場合の一般的なビルドオプションは、ドキュメント (英語) を参照してください。カーネルビルドの *-cl-fast-relaxed-math* は、パフォーマンスのトレードオフを考慮したオプションの 1 つです。
- 大幅に異なる指数を含む依存データを操作する前に、同様の指数浮動小数点値を含む操作の並べ替えを強制します。この方法は、一部のアプリケーションでは実行が困難な場合があります。
- 整数演算はプログラムの目的に適しているか?
- アルゴリズムによっては、単精度の代わりに拡張倍精度を使用するとエラーの伝播や分散の影響を軽減できる場合もありますが、通常、大きなデータ型のオペランドの使用には、パフォーマンスのトレードオフが伴います。

比較に役立つように、アプリケーションは計算値の 16 進浮動小数点形式を出力します。出力から、複数のプラットフォームで取得した符号、仮数、指数を簡単に確認して比較できます。浮動小数点オブジェクトをフォーマットして出力すると、異なる浮動小数点値の出力が同じになる場合があります。エラー分析を行う場合は、論理比較や 16 進浮動小数点表現で値をチェックしてください。

デバッグとパフォーマンス測定

OpenCL* アプリケーションの分析とデバッグ用のインターセプト・レイヤー (英語) (clIntercept) を使用して、開発者は OpenCL* アプリケーションの OpenCL* ランタイムの動作を観察できます。もちろん、SYCL* アプリケーションも対象にできます。

- デバッグとパフォーマンス分析のために OpenCL* 呼び出しをインターセプトおよび変更できます。
- clIntercept には、OpenCL* プログラムに関するメタ情報を出力するシンプルな環境変数コントロールの大規模なライブラリーがあります。
- 多くのユーザーは、自身のプログラムを手動で計測する代わりに clIntercept を使用しています。

ブルートフォース (総当り) デバッグ

フォールバックとして、多くの開発者は *printf(...)* 関数を使用して OpenCL* C カーネル内でデバッグを行います。SYCL* にも同様の動作を提供できるストリーム・オブジェクトがあります。カーネルからのストリーム操作による実行結果は通常は望ましくないため、本番のプログラムではこれらの不要なストリームを排除してください。

```
q.submit([&](cl::sycl::handler &h){
//In case stdout debug is needed see cl::sycl::stream object
    cl::sycl::stream os(1024,256, h);
    h.single_task([=] {
        os << "Pure SYCL" << cl::sycl::endl;
    });
});
```

参考資料

- インテル® oneAPI: データ並列 C++ と SYCL* テクノロジー
  - インテル® oneAPI ベース・ツールキット (英語)
  - Github* リポジトリーの DPC++ テストサンプル (英語) (*kernel-and-program* は相互運用性テストを含む)
    - OpenCL* カーネル取り込みの DPC++/SYCL* テストサンプル (英語)
  - DPC++ 互換性ツールのリリースノート (英語)
  - SYCL* 1.2.1 仕様ドキュメント (英語)
    - セクション: 4.8.2 で OpenCL* の関係を説明
  - SYCL* 1.2.1 リファレンス・カード (英語)
- OpenCL* テクノロジー
  - OpenCL* 1.2 C API リファレンス (英語)
    - リファレンス・カード (英語)
  - OpenCL* 2.1 C API リファレンス (英語) (OpenCL* C 2.0 を含む)
    - リファレンス・カード (英語)
  - OpenCL* C 2.0 仕様 (英語)
  - OpenCL* C++ API ホスト・ランタイム・ラッパー・マニュアル (英語)
  - OpenCL* ヘッダー (英語)
  - OpenCL* ICD ローダー・ライブラリー (英語)
  - OpenCL* エラーコード (英語)
  - インテル® プロセッサー向け OpenCL* ランタイム (英語)
    - SYCL* 対応 OpenCL* アプリケーション向け実験的インテル® CPU ランタイム (英語) (Prerequisites セクションを参照)
    - OpenCL* ドライバー向けインテル® グラフィックス計算ランタイム (英語)
  - Ioc64、インテル® SDK for OpenCL* Applications の一部 (インテル® System Studio 2020: OpenCL* ツール)
  - clIntercept (英語)

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。