ヤコビ反復法を CUDA* から SYCL* へ移行する

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「Migrating the Jacobi Iterative Method from CUDA* to SYCL*」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

公開日: 2022 年 5 月 4 日

はじめに

この記事では、CUDA* で記述された線形代数ヤコビ反復法を、ヘテロジニアス・プログラミング言語である SYCL* に移行する方法を紹介します。

ヤコビ反復法

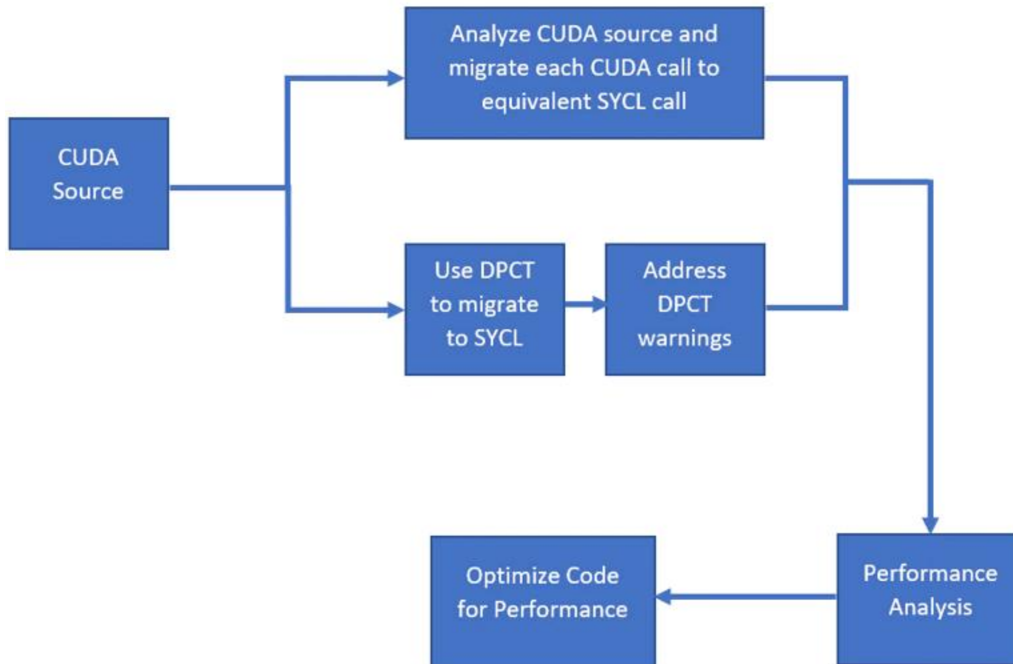
ヤコビ反復法は、数値線形代数における対角優位な連立一次方程式 Ax=b の近似数値解を求めるために用いられます。このアルゴリズムは、x の初期推定値から開始して、収束するまで繰り返し更新します。ヤコビ法は、行列 A が対角優位であれば収束が保証されます。

CUDA* から SYCL* への移行アプローチ

この記事では、CUDA* から SYCL* へ移行する 2 つのアプローチを説明します。

- 最初のアプローチは、インテル® DPC++ 互換性ツール (インテル® DPCT) を使用して CUDA* ソースを SYCL* ソースへ自動的に移行します。このツールはコードの 80 ～ 90% を移行し、残りのコードについては警告を生成するため、手動で SYCL* に移行する必要があります。インテル® DPC++ 互換性ツールが生成した警告を調べ、インテル® DPC++ 互換性ツールで移行されなかったコードを移行する方法を学びます。このアプローチは、CUDA* ソースから SYCL* への移行にかかる時間を短縮し、特に大規模なコードベースに対して有効であることが実証されています。
- 2 つ目のアプローチは、CUDA* ソースを解析し、すべての CUDA* 固有の呼び出しを同等の SYCL* 呼び出しに手動で置き換えて移行します。このアプローチは、CUDA* 開発者が SYCL* プログラミングを理解するのに役立ちます。移行後、インテル® VTune™ プロファイラーとインテル® Advisor のルーフライン機能を使用してパフォーマンスを解析し、パフォーマンスのボトルネックを把握します。その後、パフォーマンスを向上するコードの最適化について検討します。詳細は、SYCL* 2020 仕様を参照してください。

次のフローチャートは、CUDA* から SYCL* への移行に使用されるアプローチを示しています。



インテル® DPC++ 互換性ツールを使用した移行

インテル® DPC++ 互換性ツールとその使用法

インテル® DPC++ 互換性ツールは、インテル® oneAPI ベース・ツールキットのコンポーネントで、CUDA* で記述されたプログラムを DPC++ プログラムに移行する開発者を支援します。

インテル® DPC++ 互換性ツールはコードの大部分を自動的に移行しますが、完全な移行には手動での作業が必要です。このツールは、手作業が必要な場所と方法を示す警告を出力します。これらの警告には「DPCT10XX」形式の ID が割り当てられ、デベロッパー・ガイドおよびリファレンス (英語) で参照できます。このガイドには、すべての警告とその説明および修正方法の提案の一覧が含まれています。

インテル DPC++ 互換性ツールを使用した CUDA* から SYCL* への移行

インテル® DPC++ 互換性ツールは、CUDA* コードを SYCL* に移行する際に有用であり、元のコードからの識別子を保持したまま、プログラマーが可読可能なコードを生成します。また、このツールは、標準的な CUDA* インデックス計算を検出し、SYCL* に変換します。このサンプルの目的は、インテル® DPC++ 互換性ツールを使用して CUDA* から SYCL* への移行プロセスを実行し、異なる GPU および CPU デバイスで移行された SYCL* コードによってもたらされる移植性を実証することです。このツールは、ビルドプロセスをインターセプトして、CUDA* コードを対応する SYCL* コードに置き換えます。

CUDA* ソースは、主に CUDA* 表現を同等の SYCL* 表現に置き換え、カーネル呼び出しをラムダ式による parallel_for の SYCL* キューへの送信に変換することで SYCL* に移行されます。

インテル® DPC++ 互換性ツールによって移行されたヤコビ反復法のコードは、sycl_dpct_output (英語) で参照できます。

必要な CUDA* バージョンとツールは、インテル® DPC++ 互換性ツールのシステム要件 (英語) を参照してください。

以下の手順に従って、CUDA* ヤコビ反復法サンプルを SYCL* へ移行します。

1. システムに NVIDIA* CUDA* SDK が (デフォルトのパスに) インストールされていることと、インテル® oneAPI ベース・ツールキットのインテル® DPC++ 互換性ツールがインストールされていることを確認します。
2. 環境変数を設定します。setvars.sh スクリプトは、oneAPI インストール・フォルダーのルート (通常は `/opt/intel/oneapi/`) にインストールされます。

   ```
   ./opt/intel/oneapi/setvars.sh
   ```

3. ヤコビ反復法の CUDA* 実装を、JacobiCUDA_サンプル (英語) から取得します。
4. CUDA* ソースフォルダーに移動し、`intercept-build` ツールでコンパイル・データベースを生成します。これにより、すべてのコンパイラーの呼び出しを含む JSON ファイルが作成され、入力ファイルの名前とコンパイラー・オプションが格納されます。

   ```
   intercept-build make
   ```

5. インテル® DPC++ 互換性ツールを使用してコードを移行します。移行フォルダー `dpct_output` に結果が格納されます。

移行とデバッグを容易にするインテル® DPC++ 互換性ツールのオプション	
--keep-original-code	生成された SYCL* ファイルのコメント内にオリジナルの CUDA* コードを保持します。オリジナルの CUDA* コードと生成された SYCL* コードを簡単に比較できます。
--comments	生成されたコードの説明をコメントとして挿入します。
---always-use-async-handler	常に非同期例外ハンドラーで cl::sycl::queue を作成します。

   ```
   dpct -p compile_commands.json
   ```

6. 移行後のコードを確認し、インテル® DPC++ 互換性ツールによって出力された警告の詳細を診断リファレンス (英語) で参照して対処してください。
7. DPCPP コンパイラーを使用するように makefile を変更し、CUDA* 固有のコンパイラー・フラグを削除します。

詳細は、「インテル® DPC++ 互換性ツールのベストプラクティス」を参照してください。

移行されなかった SYCL* コードの実装

インテル® DPC++ 互換性ツールでコードを移行すると、移行されなかったコードは警告で識別できます。これらの警告には ID が割り当てられており、デベロッパー・ガイドおよびリファレンスを参照して手動で回避策を適用することで解決できます。

インテル® DPC++ 互換性ツールで完全移行されたヤコビ反復法コードは、sycl_dpct_migrated (英語) にあります。

移行で生成された警告と手動の回避策

- DPCT1025: The SYCL queue is created ignoring the flag and priority options. (SYCL* キューは、フラグと優先度オプションを無視して作成されます。)

```
cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);
```

DPCPP で CUDA* ストリームに相当するのはキューであるため、インテル® DPC++ 互換性ツールはフラグと優先度オプションを無視して SYCL* キューを作成します。

```
sycl::queue *stream1;
```

- DPCT1065: Consider replacing `sycl::nd_item::barrier()` with `sycl::nd_item::barrier(sycl::access::fence_space::local_space)` for better performance if there is no access to global memory. (グローバルメモリーへのアクセスがない場合、パフォーマンスを向上するため `sycl::nd_item::barrier()` を `sycl::nd_item::barrier(sycl::access::fence_space::local_space)` に置き換えることを検討してください。)

```
cg::sync(cta);
```

カーネル内部では、インテル® DPC++ 互換性ツールはグローバルメモリーへのアクセスがない場合、パフォーマンスを向上するため `barrier()` を置き換えることを提案します。この場合、ユーザーはメモリーアクセスを確認して、変更する必要があります。

- DPCT1007: Migration of this CUDA API is not supported by the DPC++ Compatibility Tool. (この CUDA* API の移行は、インテル® DPC++ 互換ツールではサポートされていません。)

```
cg::thread_block_tile<32> tile32 = cg::tiled_partition<32>(cta);
atomicAdd(sum, temp_sum);
```

多くの CUDA* デバイス・プロパティーは、SYCL* に相当するものがないか、わずかに異なるか、または現在サポートされていません。多くの場合、これは不正確な値を取得する原因となります。このため、ユーザーはデバイスへの情報クエリーを手動で確認し、修正する必要があります。

以下は、上記のコードに対する回避策です。

```
sub_group tile_sg = item_ct1.get_sub_group();
int tile_sg_size = tile_sg.get_local_range().get(0);
atomic_ref<double, memory_order::relaxed, memory_scope::device,
      access::address_space::global_space> at_sum { *sum };

at_sum.fetch_add(temp_sum);
```

- DPCT1039: The generated code assumes that "sum" points to the global memory address space. If it points to a local memory address space, replace "`dpct::atomic_fetch_add`" with (生成されたコードでは、「sum」がグローバル・メモリー・アドレス空間を指していると仮定しています。ローカル・メモリー・アドレス空間を指している場合は、「dpct::atomic_fetch_add」を次のように置き換えてください。)

```
"dpct::atomic_fetch_add<double, sycl::access::address_space::local_space>"
     atomicAdd(sum, temp_sum);
```

メモリーアクセスとパフォーマンスを向上するため、アクセスアドレス空間をグローバル空間かローカル空間に指定する必要があります。

```
sycl::atomic<int>
at_sum(sycl::make_ptr<int,sycl::access::address_space::global_space>((int*)
sum));

sycl::atomic_fetch_add<int>(at_sum, temp_sum);
```

- DPCT1049: The work-group size passed to the SYCL kernel may exceed the limit. To get the device limit, query `info::device::max_work_group_size`. Adjust the work-group size if needed. (SYCL* カーネルに渡されるワークグループ・サイズは、この制限を超える場合があります。デバイスの制限値を取得するには、info::device::max_work_group_size を照会してください。必要に応じて、ワークグループ・サイズを調整します。)

インテル® DPC++ 互換性ツールは、`info::device::max_work_group_size` をクエリーしてデバイスの制限を取得し、それに応じてワークグループ・サイズを調整することを推奨しています。

```
cgh.parallel_for(nd_range<3>(nblocks * nthreads, nthreads),
     [=](nd_item<3>item_ct1) [[intel::reqd_sub_group_size(ROWS_PER_CTA)]] {
         JacobiMethod(A, b, conv_threshold, x_new, x, d_sum, item_ct1,
x_shared_acc_ct1.get_pointer(), b_shared_acc_ct1.get_pointer());
        });
```

- DPCT1083: The size of local memory in the migrated code may be different from the original code. Check that the allocated memory size in the migrated code is correct. (移行後のコードのローカル・メモリー・サイズが、移行前のコードと異なる可能性があります。移行後のコードの割り当てメモリーサイズが正しいことを確認してください。)

一部の型は、移行後のコードと移行前のコードではサイズが異なります (例: `sycl::float3` と `float3`)。そのため、ローカルメモリーの割り当てサイズは、移行後のコードで検証する必要があります。

- ブロックサイズをマクロで指定し、`sycl::range` の生成に使用した場合、展開された値をマクロに戻す必要があります。このような場合、ツールは元のマクロをコメントとして残します。

CUDA* ソースの解析

ヤコビ反復法の CUDA* 実装は、JacobiCUDA_サンプル (英語) にあります。

ヤコビ反復法の実装の CUDA* ソースは、次のフォルダーにあります。

- **main.cpp**—ホストコード
  - CUDA* ストリームの設定
  - GPU 上でのメモリー割り当て
  - CPU 上でのデータの初期化
  - 計算のため GPU メモリーにデータをコピー
  - GPU での計算の開始
  - 結果の検証と出力
- **jacobi.cu** — GPU 上で実行するヤコビ反復法計算のカーネルコード
  - カーネルの定義
  - 共有ローカルメモリーの割り当て
  - スレッドブロックを分割する協調グループ
  - ワープ・プリミティブの使用
  - タイルの合計値を加算するアトミック操作の使用

CUDA* コードは main.cpp と jacobi.cu の 2 つのファイルで構成されます。main.cpp には、ヤコビ法のメモリー割り当て、初期化、カーネル起動、メモリーコピー、SDK タイマーを使った実行時間計算などの CPU 実装が含まれます。

jacobi.cu には、カーネル、ヤコビ法、最終誤差が含まれます。ヤコビ法では、メモリーアクセスを高速化するためベクトルを共有メモリーにロードし、スレッドブロックをタイルに分割します。そして、分割したタイルごとに、ワープレベルのプリミティブを用いて入力データのリダクションを行います。これらの中間結果は、アトミックな加算によって最終的な総和変数に合計されます。これは、不要なブロックレベルの同期を回避することで、実装の高速化にもつながります。

このサンプル・アプリケーションは、ストリーム・キャプチャー、アトミック、共有メモリー、協調グループなどの主要な概念を用いて、CUDA* ヤコビ反復法を実証しています。

CUDA* から SYCL* への移行

このセクションでは、CUDA* コードを分析し、関連する SYCL* の機能を特定して、CUDA* コードを SYCL* に移行します。CUDA* と SYCL* の基本概念は似ていますが、CUDA* コードを SYCL* コードに移行するには、それぞれの言語の命名法を理解する必要があります。詳細は、SYCL* 2020 仕様を参照してください。

main.cpp と jacobi.cu の CUDA* コードは、SYCL* バージョンの main.cpp と jacobi.cpp に移行されます。

CUDA* ヘッダーと SYCL* ヘッダー

CUDA* 実装では、パブリックホスト関数、CUDA* **ランタイム** API の組込み型定義、CUDA 言語拡張とデバイス組込み関数の関数オーバーレイを定義する cuda_runtime.h ヘッダーが使用されています。

```
#include <cuda_runtime.h>
```

SYCL* 実装では、この単一のヘッダーを使用して、API インターフェイスと API で定義された依存型のすべてをインクルードします。

```
#include <CL/sycl.hpp>
```

CUDA* ストリームと SYCL* キュー

CUDA* ストリームは、ホストコードによって発行された順序によりデバイス上で実行される一連の操作です。ホストは CUDA* 操作 (例えば、カーネル起動、メモリーコピー) をストリーム内に配置し、直ちに続行します。その後、デバイスはリソースに空きがあるときにストリームから作業をスケジュールします。同じストリーム内の操作は、先入れ先出し (FIFO) 順に実行されます。一方、異なるストリームのコマンドは、アウトオブオーダーで、あるいは同時に実行されます。

SYCL* は、ホストプログラムと 1 つのデバイスを接続するキューを備えています。プログラムは、キューを介してデバイスにタスクを送信し、キューの完了を監視できます。CUDA* ストリームと同様に、SYCL* キューは非同期で実行するコマンドグループを送信します。しかし、より上位のプログラミング・モデルである SYCL* では、データ転送操作は、キューに送信されたカーネルの依存関係から暗黙的に推定されます。

CUDA* 実装では、最初に新しい非同期ストリームを作成します。フラグ引数がストリームの動作を決定します。`cudaStreamNonBlocking` は、作成されたストリームで実行される作業がストリーム 0 (NULL ストリーム) の作業と同時に実行される可能性があり、作成されたストリームがストリーム 0 との暗黙の同期を実行しないように指定します。CUDA* ストリームは、同時実行モデルを実装するため非同期の `memset` と `memcpy` を実行し、呼び出し後すぐにホストスレッドに戻るように `stream` を指定してカーネルを起動します。CUDA* ストリームは以下のように設定されています (main.cpp)。

```
cudaStream_t stream1;
checkCudaErrors(cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking));
```

SYCL* では、CUDA* ストリームと同様の方法でキューを使用します。キューは、非同期に実行するコマンドグループを送信します。SYCL* ランタイムは、異なるデバイスの複数のキューにまたがって、異なるコマンドグループ (カーネル + 依存関係) の実行順序を自動的に処理します。次のように、`in_order` キュー・プロパティーを指定して SYCL* キューを設定し、`default_selector()` で使用する SYCL* デバイスを決定します。デフォルトのセレクターは、最初に利用可能な SYCL* デバイスを選択します。キューが必要とするシステムリソースは、キューがスコープ外になると自動的に解放されます。in_order キューは、`memcpy` 操作が完了した後にのみカーネル計算が開始されるようにし、カーネル実行のオーバーラップが発生しないようにします。

```
sycl::queue q{sycl::default_selector(),sycl::property::queue::in_order()};
```

詳細は、SYCL* キュー (英語) を参照してください。

GPU デバイス上のメモリー割り当て—cudaMalloc と sycl::malloc_device

データを GPU メモリーにコピーして、GPU での計算に利用するため、まず GPU デバイスにメモリーを割り当てる必要があります。cudaMalloc 関数 (英語) はホストまたはデバイスから呼び出すことができ、ホスト用の malloc と同様にデバイス上のメモリーを割り当てます。cudaMalloc (英語) で割り当てたメモリーは、cudaFree (英語) で解放する必要があります。

CUDA* では、GPU 上のメモリー割り当ては `cudaMalloc` 関数を使用して次のように行われます。

```
checkCudaErrors(cudaMalloc(&d_b, sizeof(double) * N_ROWS));
checkCudaErrors(cudaMalloc(&d_A, sizeof(float) * N_ROWS * N_ROWS));
```

SYCL* では、アクセラレーター・デバイス上のメモリー割り当ては、以下のように `sycl::malloc_device` 関数を用いて行われます。

```
d_b     = sycl::malloc_device<double>(N_ROWS, q);
d_A     = sycl::malloc_device<float>(N_ROWS * N_ROWS, q);
```

`sycl::malloc_device` は成功すると、指定されたデバイスに新しく割り当てられたメモリーへのポインターを返します。このメモリーはホスト上ではアクセスできません。sycl::malloc_device (英語) で割り当てられたメモリーは、メモリリークを回避するため sycl::free (英語) で解放されなければなりません。

統合共有メモリー (USM) の概念とメモリー割り当ての詳細は、SYCL* USM (英語) を参照してください。

ホストから GPU メモリーへのメモリーコピー

GPU 上にメモリーが割り当てられたら、ホストからデバイスにメモリーをコピーして、デバイスでデータを計算できるようにする必要があります。

CUDA* では、以下のように cudaMemsetAsync を使って、ホストから GPU にメモリーをコピーします。メモリーはホストへ非同期にコピーされるため、ホストは転送をストリームに配置し、呼び出しはすぐにリターンする可能性があります。オプションで、非ゼロの stream 引数を渡すことにより、操作をストリームに関連付けることができます。stream が非ゼロの場合、操作はほかのストリームの操作とオーバーラップする可能性があります。

```
checkCudaErrors(cudaMemsetAsync(d_x, 0, sizeof(double) * N_ROWS, stream1));
checkCudaErrors(cudaMemsetAsync(d_x_new, 0, sizeof(double) * N_ROWS, stream1));
checkCudaErrors(cudaMemcpyAsync(d_A, A, sizeof(float) * N_ROWS * N_ROWS,
cudaMemcpyHostToDevice, stream1));
checkCudaErrors(cudaMemcpyAsync(d_b, b, sizeof(double) * N_ROWS,
cudaMemcpyHostToDevice, stream1));
```

CUDA* ストリームは cudaStreamSynchronize で同期され、ストリーム内のすべての発行済み CUDA* 呼び出しが完了するまでホストはブロックされます。

```
checkCudaErrors(cudaStreamSynchronize(stream));
```

SYCL* では、`memcpy` を使用して、ホストからデバイスメモリーへメモリーをコピーします。メモリーを初期化するには、`memset` を使って、次のようにデータでベクトルを初期化できます。

```
q.memset(d_x,     0, sizeof(double) * N_ROWS);
q.memset(d_x_new, 0, sizeof(double) * N_ROWS);
q.memcpy(d_A, A, sizeof(float)  * N_ROWS * N_ROWS);
q.memcpy(d_b, b, sizeof(double) * N_ROWS);
```

最初の引数は、値を持つメモリー・アドレス・ポインターです。これは USM 割り当てでなければなりません。SYCL* の memcpy はポインターのソースからデスティネーションにデータをコピーします。コピー元とコピー先には、ホストまたは USM ポインターを指定できます。

メモリーは非同期にコピーされるため、メモリーを使用する前に、次のように同期してコピーの完了を確認する必要があります。

```
q.wait();
```

`wait()` は、キューに送信されたすべてのコマンドグループの実行が終了するまで、呼び出し側のスレッドの実行をブロックします。

SYCL* の `memcpy` と非同期コピー、およびデータの同期の詳細は、SYCL* キュー (英語) と memcpy (英語)、および wait (英語) を参照してください。

これで、ホスト側の CUDA* コード (main.cpp) の SYCL* への移行は完了です。

main.cpp の CUDA* ホストコードは、main.cpp (英語) にあります。

main.cpp ホストコードの SYCL* コードは、main.cpp (英語) にあります。

次のセクションでは、CUDA* カーネルコード (jacobi.cu) を SYCL* へ移行します。

GPU への計算のオフロード

CUDA* カーネルコードは jacobi.cu にあります。計算は、ヤコビ法と最終誤差の 2 つのカーネルで行われます。これらの計算はデバイスにオフロードされます。ヤコビ法と最終誤差の計算では、共有メモリー、協調グループ、リダクションを使用します。ベクトルは共有メモリーにロードされ、ブロックへのメモリーアクセスを高速かつ頻繁に行うことができます。協調グループは、作業グループをさらにサブグループに分割する際に使用されます。前出の計算はサブグループ内で行われるため、ブロックバリアは不要であり、リダクション・アルゴリズムの粒度が小さいため、各スレッドを効率良く実行させたり、作業を効果的に分散させたりするのに適しています。リダクションは、`sync()` を使用してグリッド全体ではなく、異なるスレッドブロックで実行されるため、同期ブロックの回避により、実装はかなり高速です。shift_group_left は、サブグループ内の計算に使用される SYCL* プリミティブで、すべてのスレッド値を加算して、結果を最初のスレッドに渡します。そして、すべてのサブグループの合計は、アトミックな加算によって計算されます。

最終誤差は、CPU 計算と GPU 計算の誤差の合計を計算し、出力を検証するために使用します。x の絶対値から 1 を引いた値をワープサムに加算し (各スレッドの値を加算)、すべてのワープサムの値をブロックサムに加算します。最終誤差は `g_sum` に格納されます。

CUDA* では、スレッドのグループをスレッドブロック、または単にブロックと呼びます。これは、SYCL* のワークグループの概念と同等です。ブロックとワークグループはどちらも階層の同じレベルにアクセスし、同様の同期操作を利用できます。

CUDA* ストリームは、ホストコードから送信される一連の CUDA* 操作です。これらの操作は、送信順に非同期で実行されます。CUDA* ストリームを指定しない場合、デフォルトの CUDA* ストリームが作成され、すべての操作はデフォルトのストリームに送信されます。

CUDA* ストリームと同様に、SYCL* キューは非同期で実行するコマンドグループを送信します。ただし、SYCL* のデータ転送操作は、キューに送信されたカーネルの依存関係から暗黙的に推定されます。

CUDA* では、カーネルは次のパラメーターで起動します。nblocks はグリッドの次元とサイズ、nthreads は各ブロックの次元とサイズ、第 3 パラメーターは**ブロックごと**に動的に割り当てられる共有メモリーのバイト数、stream は関連ストリーム (デフォルトは 0) を指定するオプションのパラメーターです。

```
JacobiMethod<<<nblocks, nthreads, 0, stream>>> (A, b, conv_threshold, x, x_new,
d_sum);
```

SYCL* では、single_task、parallel_for、parallel_for_work_group などのカーネル・コンストラクトは、それぞれ関数オブジェクトまたはラムダ関数を引数の 1 つとして受け取ります。関数オブジェクトまたはラムダ関数内のコードは、デバイス上で実行されます。

```
q1.submit([&](handler &cgh) {
cgh.parallel_for(nd_range<3>(nblocks * nthreads, nthreads), [=](nd_item<3>
item_ct1) {
          JacobiMethod(A, b, conv_threshold, x, x_new, d_sum, item_ct1,
x_shared_acc_ct1.get_pointer(), b_shared_acc_ct1.get_pointer(), stream_ct1);
          });
});
```

キューをセットアップした後、コマンドグループで parallel_for を使用してカーネルを送信します。この関数は、複数のワークアイテムに対してカーネルを並列実行します。nd_range は、それぞれがカーネル関数を実行するワークアイテムの 1 次元、2 次元、3 次元のグリッドを指定し、ワークグループ単位でまとめて実行します。nd_range は、グローバル・ワークサイズ (ワークアイテムの全範囲を指定) とローカル・ワークサイズ (各ワークグループの範囲を指定) の 2 つの 1 次元、2 次元、3 次元の範囲で構成されます。

nd_item は sycl::nd_range の中のポイントの場所を示します。通常、nd_item は parallel_for の中でカーネル関数に渡されます。ワークグループとグローバル空間におけるワークアイテムの ID に加えて、nd_item にはインデックス空間を定義する sycl::nd_range も含まれます。

CUDA* スレッドブロックと SYCL* ワークグループ

CUDA* では、協調グループは、スレッドのグループを定義、分割、同期するデバイスコード API を提供します。パフォーマンスと設計の柔軟性を向上するため、スレッドブロックよりも小さなスレッドのグループを定義して同期しなければならないことはよくあります。

thread_block のインスタンスは、CUDA* スレッドブロック内のスレッドグループへのハンドルで、次のように初期化します。

```
cg::thread_block cta = cg::this_thread_block();
```

その行を実行するすべてのスレッドは、変数ブロックの独自のインスタンスを持ちます。CUDA* 組込み変数 blockIdx の値が同じスレッドは、同じスレッド・ブロック・グループに属します。

SYCL* では、カーネルの 1 回の実行は、**ワークグループ**と**ワークアイテム**で構成されます。各ワークグループには同じ数のワークアイテムが含まれ、一意の**ワークグループ ID** によって識別されます。また、ワークグループ内では、ワークアイテムは**ローカル ID** で識別でき、ローカル ID とワークグループ ID の組み合わせがグローバル ID に相当します。

```
auto cta = item_ct1.get_group();
```

SYCL* の get_group は、与えられた次元の nd_range 全体におけるワークグループの位置を表すグループ ID の構成要素を返します。

共有ローカル・メモリー・アクセス

CUDA* では、共有メモリーはオンチップであり、ローカルメモリーやグローバルメモリーよりはるかに高速です。共有メモリーのレイテンシーは、キャッシュされていないグローバルメモリーのレイテンシーの約 1/100 です。スレッドは、同じスレッドブロック内のほかのスレッドによってグローバルメモリーからロードされた共有メモリー内のデータにアクセスできます。メモリーアクセスは、スレッド同期によって制御され、競合状態を回避できます (__syncthreads)。

```
__shared__ double x_shared[N_ROWS];
__shared__ double b_shared[ROWS_PER_CTA + 1];
```

SYCL* では、共有ローカルメモリー (SLM) はワークグループごとにオンチップであり、グローバルメモリーよりもはるかに帯域幅が高く、レイテンシーは低くなります。ワークグループ内のすべてのワークアイテムからアクセスできるため、ワークグループのサイズによっては、数百のワークアイテム間のデータ共有や通信に対応できます。ワークグループ内のワークアイテムは、グローバルメモリーから SLM に明示的にデータをロードできます。このデータは、ワークグループの存続期間中は SLM に保持され、高速にアクセスすることが可能です。ワークグループが終了する前に、ワークアイテムは SLM 内のデータを明示的にグローバルメモリーに書き戻すことができます。ワークグループの実行が終了すると、SLM 内のデータは無効になります。

```
accessor<double, 1, access_mode::read_write, access::target::local>
x_shared_acc_ct1(range<1>(N_ROWS), cgh);
accessor<double, 1, access_mode::read_write, access::target::local>
b_shared_acc_ct1(range<1>(ROWS_PER_CTA + 1), cgh);
```

CUDA* スレッドブロック同期と SYCL* バリア同期

同期は、同じリソースを共有するスレッドの状態を同期します。

CUDA* では、同期はすべてのスレッドグループでサポートされています。グループの同期を行うには、集合的な `sync()` メソッドを呼び出すか、`cooperative_groups::sync()` 関数を呼び出します。これらはグループ内のすべてのスレッド間でバリア同期を実行します。

```
cg::sync(cta);
```

SYCL* では、メモリーの状態を同期させるため、`item::barrier(access::fence_space)` 操作を使用します。これは、ワークグループ内の各ワークアイテムがバリア呼び出しに到達することを確認するものです。つまり、コード内のある時点でワークグループが同期されていることを保証します。

```
item_ct1.barrier();
```

`item::barrier` は指定された空間でメモリーフェンスを発生させます。
`access::fence_space::local_space`、`::global_space`、または `::global_and_local` のいずれかを指定します。フェンスは、指定された空間の状態がワークグループ内のすべてのワークアイテムで一貫していることを保証します。

CUDA* 協調グループと SYCL* サブグループ

CUDA* 協調グループと SYCL* サブグループは、カーネルがスレッドのグループを動的に構成し、スレッドが協調してデータを共有して集合操作を実行できるように、プログラミング・モデルを拡張することを目的としています。

CUDA* では、協調グループは、既存のグループを分割して (英語) 新しいグループを作成する柔軟性を備えています。これにより、より細かい粒度での協調と同期が可能になります。`cg::tiled_partition()` 関数は、スレッドブロックを複数の「タイル」に分割します。

```
cg::thread_block_tile<32> tile32 = cg::tiled_partition<32>(cta);
```

分割を実行する各スレッドは、1 つの 32 スレッドグループへのハンドルを (tile32 で) 取得します。

SYCL* では、サブグループは、低レベルのハードウェアにマップするワークグループの分割を可能にし、追加のスケジューリング保証を提供します。サブグループは SYCL* 実行モデルの拡張であり、`work_group` と `work_item` の間の階層です。SYCL* 実装では、サブグループを低レベルのハードウェア機能にマップすることがよくあります。例えば、ベクトル命令をサポートするハードウェア上では、サブグループ内のワークアイテムは一般に SIMD で実行されます。

```
sub_group tile_sg = item_ct1.get_sub_group();
```

デバイスがサポートするサブグループ・サイズのセットはデバイス固有であり、個々のカーネルはコンパイル時に特定のサブグループ・サイズを要求することができます。このサブグループ・サイズはコンパイル時定数です。最高のパフォーマンスを得るには、最適なサブグループ・サイズをハードウェア上の計算ユニットのサイズに合わせる必要があります。サブグループの最適なサイズが設定されていない場合、コンパイラーは選択を試みます。

```
[[intel::reqd_sub_group_size(SIZE)]]
```

CUDA* ワープ・プリミティブと SYCL* グループ・アルゴリズム

プリミティブは、ワープレベルのプログラミングを安全かつ効果的に行うために導入されました。CUDA* はスレッドのグループを SIMT (Single Instruction, Multiple Thread) 方式で実行します。ワープ実行を活用することで、ハイパフォーマンスが得られます。

CUDA* では、`thread_block_tile::shfl_down()` はワープレベルのリダクションを簡素化し、共有メモリーの必要性を排除するために使用されます。

```
for (int offset = tile32.size() / 2; offset > 0; offset /= 2) {
    rowThreadSum += tile32.shfl_down(rowThreadSum, offset);
    }
```

各反復はアクティブなスレッドの数を半分にし、各スレッドはその部分和をブロックの最初のスレッドに追加します。

CUDA* の `shfl_down` に相当する SYCL* の `shift_group_left` は、グループ内のワークアイテムが持つ値を直接グループ内の別のワークアイテムに、一定数のワークアイテムを左にシフトして移動させます。

```
for (int offset = tile_sg.get_local_range().get(0) / 2; offset > 0; offset /= 2)
{
    rowThreadSum += shift_group_left(tile_sg,rowThreadSum, offset);
    }
```

CUDA* アトミックと SYCL* アトミック

アトミック操作は、ほかのスレッドからの干渉を受けずに実行される操作です。多くの場合、アトミック操作はマルチスレッド・アプリケーションで問題になる競合状態を防ぐために使用されます。

CUDA* では、`atomicAdd()` はグローバルメモリーまたは共有メモリー内のあるアドレスでワードを読み取り、それに数値を加算して、結果を同じアドレスに書き戻します。操作が完了するまで、ほかのスレッドはこのアドレスにアクセスできません。アトミック関数はメモリーフェンスとして機能せず、メモリー操作の同期や順序の制約を意味しません。

```
if (tile32.thread_rank() == 0) {
    atomicAdd(&b_shared[i % (ROWS_PER_CTA + 1)], -rowThreadSum);
    }
```

SYCL* では、`atomic_ref` がこれに相当し、float、double データ型をサポートします。サポートされる順序のセットはデバイスに固有ですが、すべてのデバイスは少なくとも `memory_order::relaxed` をサポートすることが保証されています。

```
if (tile32_sg.get_local_id()[0] == 0) {
    atomic_ref<double, memory_order::relaxed, memory_scope::device,
access::address_space::local_space> at_h_sum { b_shared[i % (ROWS_PER_CTA + 1)]};
      at_h_sum.fetch_add(-rowThreadSum);
    }
```

テンプレート・パラメーター空間は、`access::address_space::generic_space`、`access::address_space::global_space`、または `access::address_space::local_space` が許可されています。

これで、カーネル側の CUDA* コード (jacobi.cu) の SYCL* への移行は完了です。

jacobi.cu の CUDA* カーネルコードは、jacobi.cu (英語) にあります。

jacobi.cpp の SYCL* カーネルコードは、jacobi.cpp (英語) にあります。

これで、CUDA* から SYCL* への移行はすべて終了しました。NVIDIA* GPU でしか実行できない CUDA* ソースではなく、適切な SYCL* コンパイラーを使用してどの GPU でもコンパイルできるソースファイル main.cpp と jacobi.cpp になりました。

この SYCL* ソースがあれば、インテル® oneAPI DPC++ コンパイラー (英語) でコンパイルして、インテルの GPU または CPU でヤコビ反復法を実行できます。また、オープンソースの LLVM コンパイラー (英語) や hipSYCL コンパイラー (英語) を使用して、NVIDIA GPU/AMD GPU 上で実行できるようにコンパイルすることも可能です。

SYCL* は、ベンダー固有のハードウェアにロックされることなく、異なるベンダーの CPU や GPU にソースコードを移行できるようにします。

パフォーマンス解析ツール

インテル® VTune™ プロファイラー

インテル® VTune™ プロファイラーは、シリアルおよびマルチスレッド・アプリケーションのパフォーマンス解析ツールです。アルゴリズムの選択を解析し、アプリケーションが利用可能なハードウェア・リソースの恩恵を受けられる場所と方法を特定するのに役立ちます。データコレクターは、OS タイマーを使用してアプリケーションをプロファイルし、プロセスに割り込み、サンプリング間隔 10ms ですべてのアクティブな命令アドレスのサンプルを収集し、各サンプルの呼び出しシーケンス (スタック) をキャプチャーします。デフォルトでは、コレクターはシステム全体のパフォーマンス・データを収集せず、アプリケーションのみに注目します。詳細は、「インテル® VTune™ プロファイラー導入ガイド」(英語) を参照してください。

プロファイル・データを収集するには、コマンドラインで次のスクリプトを実行します。

```
#!/bin/bash

source /opt/intel/oneapi/setvars.sh

#Vtune GPU Hotspot script

bin="jacobiSYCL"

prj_dir="vtune_data"

echo $bin

rm -r ${prj_dir}

echo "Vtune Collect hotspots"

vtune -collect gpu-hotspots -result-dir ${prj_dir} $(pwd)/${bin}

echo "Vtune Summary Report"

vtune -report summary -result-dir ${prj_dir} -format html -report-output
$(pwd)/vtune_${bin}.html
```

上記のスクリプト「vtune_report.sh」ファイルがアプリケーション・バイナリーと同じ場所にあることを確認し、バイナリー名が異なる場合はスクリプト内のバイナリー名に必要な変更を加えてからスクリプトを実行すると、インテル® VTune™ プロファイラーはデータを収集して HTML レポートを生成します。

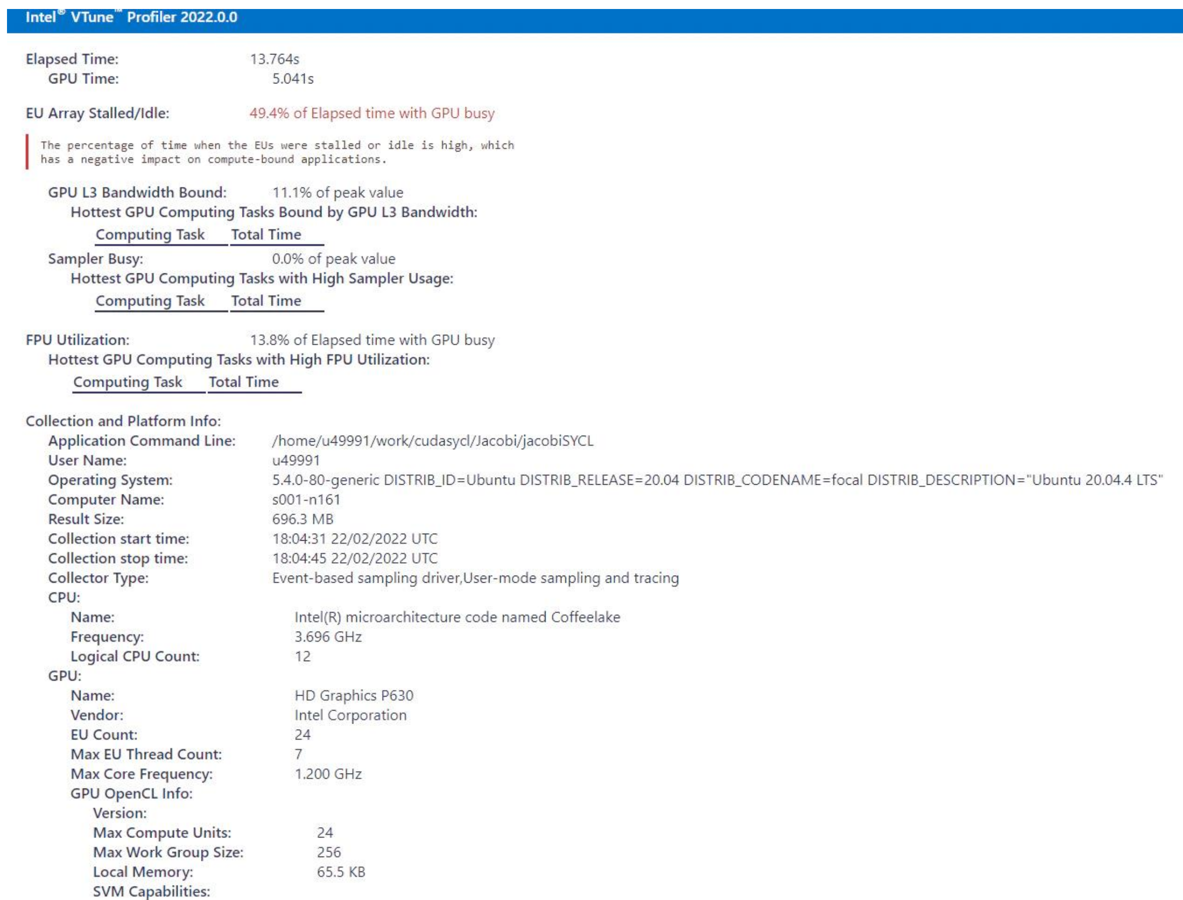


図 1: インテル® VTune™ プロファイラーのメトリック

図 1 は、インテル® VTune™ プロファイラーのスクリーンショットで、移行後のヤコビ反復法の SYCL* コードの総経過時間を示しています。総経過時間は 13.764 秒で、このうち GPU 時間は 5.041 秒です。レポートには GPU アイドル時間も示されており、実行期間中、GPU コアの帯域幅の 50% しか利用されていないため、改善の余地があります。

インテル® Advisor のルーフライン

ルーフライン・グラフは、メモリー帯域幅や計算ピークなどのハードウェア制限に関連するアプリケーション・パフォーマンスを視覚的に表現したものです。ルーフラインには、サーベイ解析と Flops とトリップカウント解析の両方からのデータが必要です。これらの解析は別々に実行するか、ショートカット・コマンドを使用して順番に実行できます。詳細は、「インテル® Advisor 導入ガイド」(英語) を参照してください。

プロファイル・データを収集するには、コマンドラインで次のスクリプトを実行します。

```
#!/bin/bash

source /opt/intel/oneapi/setvars.sh

#Advisor Roofline script

bin="jacobiSYCL"

prj_dir="./roofline_data"

echo $bin

rm -r ${prj_dir}

advisor --collect=survey --project-dir=${prj_dir} --profile-gpu -- ./${bin} -q

advisor --collect=tripcounts --project-dir=${prj_dir} --flop --profile-gpu -
- ./${bin} -q

advisor --report=roofline --gpu --project-dir=${prj_dir} --report-
output=./roofline_gpu_${bin}.html -q
```

上記のスクリプト「roofline_report.sh」ファイルがアプリケーション・バイナリーと同じ場所にあることを確認し、バイナリー名が異なる場合はスクリプト内のバイナリー名に必要な変更を加えてからスクリプトを実行すると、インテル® Advisor はルーフライン・データを収集して HTML レポートを生成します。

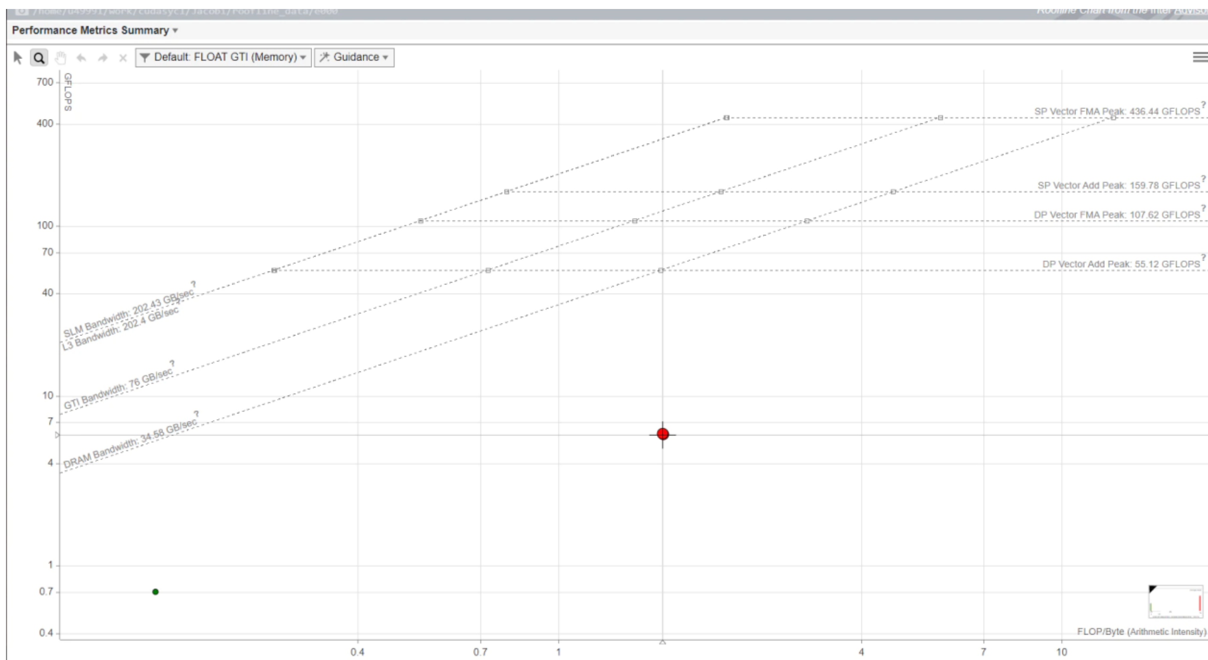


図 2: GPU ルーフライン・グラフ

GPU ルーフライン・グラフは、アプリケーション・パフォーマンスをメモリーと計算の観点から示しています。X 軸は演算強度、Y 軸は計算パフォーマンスを表しています。レポートから、DRAM を使用した場合の最大帯域幅は 34.58GB/秒、GTI 帯域幅は 76GB/秒、L3 帯域幅は 202.4GB/秒、SLM 帯域幅は 202.43GB/秒であり、さらに単精度と倍精度のベクトル FMA とベクトル加算の最大計算パフォーマンスが分かります。

グラフ中の各ドットは、アプリケーションのループや関数を表しています。ドットの位置はループや関数のパフォーマンスを表し、最適化と演算強度に影響されます。ドットのサイズと色は、アプリケーションの総時間のうち、そのループや関数が占める割合を示しています。大きな赤いドットは最も多くの時間を費やしているため、最適化の最良の候補です。小さな緑のドットは実行時間が短いため、最適化の労力が無駄になるかもしれません。

パフォーマンスを向上する SYCL* コードの最適化

リダクション操作の最適化

shift_group_left は、グループ内のワークアイテムが持つ値を直接グループ内の別のワークアイテムに、一定数のワークアイテムを左にシフトして移動させます。

```
for (int offset = tile_sg.get_local_range().get(0) / 2; offset > 0; offset /= 2)
{
    rowThreadSum += shift_group_left(tile_sg, rowThreadSum, offset);
    }
```

上記のコード例は、最適化前のヤコビ反復法のコードです。ここでは、パフォーマンスを向上するため shift_group_left を reduce_over_group に置き換えています。

reduce_over_group は、グループ内のワークアイテムが保持する値を 1 つにまとめて、配列要素の一般化された和を内部的に実装します。ワークグループはグループのサイズに等しい数 (グループ内のすべてのワークアイテム) の値をレデュースし、各ワークアイテムは 1 つの値を提供します。

```
rowThreadSum = reduce_over_group(tile_sg, rowThreadSum, sycl::plus<double>());
```

上記のコード例は、ヤコビ反復法の SYCL* 最適化コードです。reduce_over_group API を使用することで、インテル® GPU 上のアプリケーションの実行時間が約 35% 短縮されています。

アトミック操作の最適化

fetch_add は、atomic_ref で参照されるオブジェクトの値にオペランドをアトミックに加算し、結果を参照オブジェクトの値に代入します。ここでは、アトミックな fetch_add を使用して、すべてのサブグループの値を temp_sum 変数に合計しています。

```
if (tile_sg.get_local_id()[0] == 0) {
     atomic_ref<double, memory_order::relaxed, memory_scope::device,
access::address_space::global_space> at_sum{*sum};
      at_sum.fetch_add(temp_sum);
    }
```

上記のコード例は、最適化前のヤコビ反復法のコードです。最適化された実装では、fetch_add が削除され、temp_sum の値を直接グローバル変数 at_sum に加算しています。

```
if (tile_sg.get_local_id()[0] == 0) {
     atomic_ref<double, memory_order::relaxed, memory_scope::device,
                 access::address_space::global_space>
     at_sum{*sum};
     at_sum += temp_sum;
    }
```

`fetch_add` を削除することで、インテル® GPU 上でアプリケーションの実行時間が約 5% 短縮されました。

上記のコードは、移行後のヤコビ反復法の SYCL* 最適化コードです。

移行後のヤコビ反復法の SYCL* 最適化コードは、sycl_migrated_optimized (英語) にあります。

ソースコードのリンク

CUDA* ソース	GitHub* リンク (英語)
SYCL* ソース—手動による移行 (1 対 1 のマッピング)	GitHub* リンク (英語)
SYCL* ソース—手動による移行 (最適化後)	GitHub* リンク (英語)
SYCL* ソース—インテル® DPCT の出力 (移行されなかったコードを含む)	GitHub* リンク (英語)
SYCL* ソース—インテル® DPCT の出力 (移行されなかったコードを実装後)	GitHub* リンク (英語)

参考資料

- SYCL* 2020 仕様 (リビジョン 4)
- インテル® DPC++ 互換性ツール (インテル® DPCT) デベロッパー・ガイドおよびリファレンス (英語)
- 『Data Parallel C++』、James Reinders ほか著 (英語)
- oneAPI GPU 最適化ガイド
- CUDA* ツールキット・ドキュメント (英語)

製品および性能に関する情報

[1] 性能は、使用状況、構成、その他の要因によって異なります。詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。