oneAPI for AMD* GPU 2024.2.0 ガイド

この記事は、Codeplay 社の許可を得て iSUS (IA Software User Society) が作成した 2024 年 7 月 22 日時点の『oneAPI for AMD* GPUs 2024.2.0』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。

以前のバージョンのガイド: **2023.0.0** | **2023.1.0** | **2023.2.1** | **2024.0.0**
『oneAPI for NVIDIA* GPU ガイド』はこちら

oneAPI for AMD* GPU は、開発者が DPC++/SYCL* を利用して oneAPI アプリケーションを作成し、それらを AMD* GPU 上で実行できるようにするインテル® oneAPI ツールキット向けのプラグインです。

このプラグインは、HIP バックエンドを DPC++ 環境に追加します。このドキュメントでは、「oneAPI for AMD* GPU」と「DPC++ HIP プラグイン」が同じ意味で使われています。

oneAPI の詳細については、インテル® oneAPI の概要 (英語) を参照してください。

oneAPI for AMD* GPU の使用を開始するには、「導入ガイド」を参照してください。

導入ガイド

- oneAPI for AMD* GPU のインストール
- DPC++ を使用して AMD* GPU をターゲットにする
- DPC++ のリソース
- SYCL* のリソース
- SYCL* アプリケーションのデバッグ
- MPI ガイド

パフォーマンス・ガイド

- はじめに
- プログラミング・モデル
- 最適化の目的
- パフォーマンス解析
- AMD* GPU 上のパフォーマンス
- AMD* パフォーマンス・ツール
- 一般的な最適化

サポート

- 機能
- 更新履歴
- トラブルシューティング
- ライセンス (英語)

導入ガイド

oneAPI for AMD* GPU のインストール

このガイドには、DPC++ と DPC++ HIP プラグインバージョン 2024.2.0 を使用して、AMD* GPU で SYCL* アプリケーションを実行する方法を説明します。

DPC++ に関連する一般的な情報は、「DPC++ のリソース」の節を参照してください。

サポートされるプラットフォーム

このリリースは、次のプラットフォームで検証されています (Linux* カーネルのアップストリーム AMD* GPU ドライバーを使用)。

GPU ハードウェア	アーキテクチャー	オペレーティング・システム	HIP	Linux* カーネルバージョン
AMD Radeon* Pro W6800	gfx1030	Ubuntu* 22.04.2 LTS	6.1.0	6.5.0-1020-oem
AMD Radeon* Pro W6800	gfx1030	Ubuntu* 22.04.2 LTS	6.0.2	6.5.0-1020-oem
AMD Radeon* Pro W6800	gfx1030	Ubuntu* 22.04.2 LTS	5.7.1	6.5.0-1020-oem
AMD Radeon* Pro W6800	gfx1030	Ubuntu* 22.04.2 LTS	5.4.3	6.5.0-1020-oem
AMD Instinct* MI210	gfx90a	Ubuntu* 22.04.2 LTS	6.0.2	6.2.0-35-generic

理論的には、このリリースは ROCm* と互換性のある (英語) すべてのデバイスで動作します。

- このリリースは HIP 5.x または HIP 6.x で動作するはずですが、HIP 5.4.3、5.7.1、6.0.2、および 6.1.0 でのみテストされています。Codeplay は、他の HIP リリースでは正しい動作を保証することはできません。
    - テスト済みの各バージョンは、既存の HIP インストールと共存できます。ROCm* インストール・ガイドの「マルチバージョンの ROCm* のインストール」(英語) セクションを参照してください。
- このリリースは、ROCm* がサポートする各種 AMD* GPU で動作するはずですが、Codeplay はテストしていないプラットフォームでの正しい動作を保証することはできません。また、ROCm* が正式にサポートしていない AMD* GPU でも動作する可能性があります。
- ROCm* Linux* で正式にサポートされている AMD* GPU は、こちら (英語) で確認できます。
- このパッケージは Ubuntu* 22.04 でのみテストされていますが、一般的な Linux* システムにインストールできます。
- oneAPI for AMD* GPU パッケージは現在 Linux* のみをサポートしています。

要件

1. C++ 開発ツールインストールします。

   oneAPI アプリケーションをビルドして実行するには、C++ 開発ツールの `cmake`、`gcc`、`g++`、`make` および `pkg-config` をインストールする必要があります。

   次のコンソールコマンドは、一般的な Linux* ディストリビューションに上記のツールをインストールします。

   Ubuntu*

   ```
   $ sudo apt update
   $ sudo apt -y install cmake pkg-config build-essential
   ```

   Red Hat* と Fedora*

   ```
   $ sudo yum update
   $ sudo yum -y install cmake pkgconfig
   $ sudo yum groupinstall "Development Tools"
   ```

   SUSE*

   ```
   $ sudo zypper update
   $ sudo zypper --non-interactive install cmake pkg-config
   $ sudo zypper --non-interactive install pattern devel_C_C++
   ```

   次のコマンドで、ツールがインストールされていることを確認します。

   ```
   $ which cmake pkg-config make gcc g++
   ```

   次のような出力が得られるはずです。

   ```
   /usr/bin/cmake
   /usr/bin/pkg-config
   /usr/bin/make
   /usr/bin/gcc
   /usr/bin/g++
   ```

2. DPC++/C++ コンパイラーを含むインテル® oneAPI ツールキット 2024.2.0 をインストールします。
   - インテル® oneAPI ベース・ツールキットは、多くの利用環境に適用できます。
   - oneAPI for AMD* GPU をインストールするには、インテル® oneAPI ツールキットのバージョン 2024.2.0 が必要です。これよりも古いバージョンにはインストールできません。
3. AMD* GPU 向けの GPU ドライバーと ROCm* ソフトウェア・スタックをインストールします。
   - 例えば、ROCm* 5.4.3 の場合、「インストール・スクリプトを使用したインストール」(英語) の手順に従ってください。
   - `--usecase="dkms,graphics,opencl,hip,hiplibsdk` 引数を指定して `amdgpu-install` インストーラーを起動し、必要となるすべてのコンポーネントを確実にインストールすることを推奨します。

インストール

1. 以下のいずれかの方法で最新の oneAPI for AMD GPU インストーラーをダウンロードします。

    1. ウェブサイト (英語) から直接ダウンロード
    2. cURL または WGET でダウンロード API を使用 (英語) (アカウントが必要です)

2. ダウンロードした自己展開型インストーラーを実行します。

```
$ sh oneapi-for-amd-gpus-2024.2.0-rocm-5.4.3-linux.sh
```

    - インストーラーは、デフォルトの場所にあるインテル® oneAPI ツールキット 2024.2.0 のインストールを検索します。インテル® oneAPI ツールキットが独自の場所にインストールされている場合、`--install-dir /path/to/intel/oneapi` でパスを指定します。
    - インテル® oneAPI ツールキットが home ディレクトリー外にある場合、`sudo` を使用してコマンドを実行する必要があります。

環境を設定

1. 実行中のセッションで oneAPI 環境を設定するには、インテルが提供する `setvars.sh` スクリプトを source します。

    システム全体へのインストールの場合:

```
$ . /opt/intel/oneapi/setvars.sh --include-intel-llvm
```

    プライベート・インストールの場合 (デフォルトの場所):

```
$ . ~/intel/oneapi/setvars.sh --include-intel-llvm
```

    - `clang++` などの LLVM ツールにパスを追加するには、`--include-intel-llvm` オプションを使用します。
    - ターミナルを開くたびにこのスクリプトを実行する必要があります。セッションごとに設定を自動化する方法については、「CLI 開発向けの環境変数を設定する」 (英語) など、関連するインテル® oneAPI ツールキットのドキュメントを参照してください。

2. HIP ライブラリーとツールが環境内にあることを確認します。
    - `rocminfo` を実行します。実行時の表示に明らかなエラーが認められなければ、環境は正しく設定されています。
    - 問題があれば、環境変数を手動で設定します。

```
$ export PATH=/PATH_TO_ROCM_ROOT/bin:$PATH
$ export LD_LIBRARY_PATH=/PATH_TO_ROCM_ROOT/lib:$LD_LIBRARY_PATH
```

      ROCm* は通常 `/opt/rocm-x.x.x/` にインストールされます。

インストールの確認

DPC++ HIP プラグインのインストールを確認するには、DPC++ の `sycl-ls` ツールを使用して、SYCL* で利用可能な AMD* GPU があることを確認します。AMD* GPU が利用できる場合、`sycl-ls` の出力に次のような情報が表示されます。

```
[hip:gpu][hip:0] AMD HIP BACKEND, AMD Radeon PRO W6800 gfx1030 [HIP 60140.9]
```

- 上記のように利用可能な AMD* GPU が表示されていれば、DPC++ HIP プラグインが適切にインストールされ、設定されていることが確認できます。
- インストールや設定に問題がある場合、「トラブルシューティング」の「sycl-ls の出力でデバイスが見つからない場合」を確認してください。
- 利用可能なハードウェアとインストールされている DPC++ プラグインに応じて、OpenCL* デバイス、インテル® GPU、または NVIDIA* GPU など、ほかのデバイスもリストされることがあります。

サンプル・アプリケーションを実行

1. 次の C++/SYCL* コードで構成される `simple-sycl-app.cpp` ファイルを作成します。

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

int main() {
  // カーネルコード内で使用する 4 つの int バッファーを作成
  sycl::buffer<int, 1> Buffer{4};

  // SYCL* キューを作成
  sycl::queue Queue{};

  // カーネルのインデックス空間サイズ
  sycl::range<1> NumOfWorkItems{Buffer.size()};

  // キューへコマンドグループ（ワーク）を送信
  Queue.submit([&](sycl::handler &cgh) {

  // デバイス上のバッファーへの書き込み専用アクセサーを作成
  auto Accessor = Buffer.get_access<sycl::access::mode::write>(cgh);

  // カーネルを実行
  cgh.parallel_for<class FillBuffer>(
    NumOfWorkItems, [=](sycl::id<1> WIid) {
      // インデックスでバッファーを埋めます
      Accessor[WIid] = static_cast<int>(WIid.get(0));
    });
  });

  // ホスト上のバッファーへの読み取り専用アクセサーを作成。
  // キューのワークが完了するのを待機する暗黙のバリア
  auto HostAccessor = Buffer.get_host_access();

  // 結果をチェック
  bool MismatchFound{false};
```

```
  for (size_t I{0}; I < Buffer.size(); ++I) {
    if (HostAccessor[I] != I) {
      std::cout << "The result is incorrect for element: " << I
                << " , expected: " << I << " , got: " << HostAccessor[I]
                << std::endl;
      MismatchFound = true;
    }
  }

  if (!MismatchFound) {
    std::cout << "The results are correct!" << std::endl;
  }

  return MismatchFound;
}
```

2. アプリケーションをコンパイルします。

```
$ icpx -fsycl -fsycl-targets=amdgcn-amd-amdhsa \
-Xsycl-target-backend --offload-arch=<ARCH> \
simple-sycl-app.cpp -o simple-sycl-app
```

ARCH には GPU のアーキテクチャー (例えば `gfx1030`) を指定します。次のコマンドで確認できます。

```
$ rocminfo | grep 'Name: *gfx.*'
```

出力に GPU アーキテクチャーが表示されます。例えば、次のようになります。

```
Name: gfx1030
```

3. アプリケーションを実行します。

```
$ ONEAPI_DEVICE_SELECTOR="hip:*" SYCL_PI_TRACE=1 ./simple-sycl-app
```

次のような出力が得られます。

```
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_hip.so
[ PluginVersion: 15.51.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded:
libpi_unified_runtime.so [ PluginVersion: 15.51.1 ]
SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
SYCL_PI_TRACE[all]: Selected device: -> final score = 1500
SYCL_PI_TRACE[all]:   platform: AMD HIP BACKEND
SYCL_PI_TRACE[all]:   device: AMD Radeon PRO W6800
The results are correct!
```

これで、oneAPI for AMD* GPU の環境設定が確認でき、oneAPI アプリケーションの開発を開始できます。

以降では、AMD* GPU で oneAPI アプリケーションをコンパイルして実行するための一般的な情報を説明します。

DPC++ を使用して AMD* GPU をターゲットにする

AMD* GPU 向けのコンパイル

AMD* GPU 対応の SYCL* アプリケーションをコンパイルするには、DPC++ に含まれる `clang++` コンパイラーを使用します。

例:

```
$ icpx -fsycl -fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=gfx1030 -o sycl-app sycl-app.cpp
```

次のフラグが必要です。

- `-fsycl`: C++ ソースファイルを SYCL* モードでコンパイルするようにコンパイラーに指示します。このフラグは暗黙的に C++ 17 を有効にし、SYCL* ランタイム・ライブラリーを自動でリンクします。
- `-fsycl-targets=amdgcn-amd-amdhsa`: AMD* GPU をターゲットに SYCL* カーネルをビルドすることをコンパイラーに指示します。
- `-Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=gfx1030`: `gfx1030` AMD* GPU をターゲットに SYCL* カーネルをビルドすることをコンパイラーに指示します。

AMD* GPU をターゲットにする場合、GPU の特定のアーキテクチャーを指定する必要があることに注意してください。

利用できる SYCL* コンパイルフラグの詳細は、『DPC++ コンパイラー・ユーザーズ・マニュアル』(英語) を参照してください。すべての DPC++ コンパイラー・オプションの詳細は、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』の「コンパイラー・オプション」(英語) を参照してください。

icpx コンパイラーを使用する

`icpx` コンパイラーは、デフォルトで `-O2` と `-ffast-math` オプションを有効にするため、通常の `clang++` ドライバーよりも積極的な最適化を行います。多くの場合、これによりパフォーマンスは向上しますが、一部のアプリケーションでは問題が生じる可能性があります。その場合、`-fno-fast-math` を使用して `-ffast-math` を無効にして、`-O2` 以外の `-O` オプションを指定することで最適化レベルを変更できます。`$releasedir/compiler/latest/linux/bin-llvm/clang++` にある `clang++` ドライバーを直接起動することで、通常の `clang++` の最適化レベルを適用できます。

複数ターゲット向けのコンパイル

AMD* GPU をターゲットにするだけではなく、一度のコンパイルで複数のハードウェア・ターゲットで実行できる SYCL* アプリケーションを生成できます。次の例は、AMD* GPU、NVIDIA* GPU、および SPIR* をサポートする任意のデバイス (インテル® GPU など) で実行できるコードを含む単一のバイナリーを生成する方法を示しています。

```
$ icpx -fsycl -fsycl-targets=amdgcn-amd-amdhsa,nvptx64-nvidia-cuda,spir64 \
        -Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=gfx1030 \
        -Xsycl-target-backend=nvptx64-nvidia-cuda --offload-arch=sm_80 \
        -o sycl-app sycl-app.cpp
```

AMD* GPU で SYCL* アプリケーションを実行

AMD ターゲットの SYCL* アプリケーションをコンパイルしたら、ランタイムが SYCL* デバイスとして AMD* GPU を選択しているか確認する必要があります。

通常、デフォルトのデバイスセレクターを使用するだけで、利用可能な AMD* GPU の 1 つが選択されます。しかし、場合によっては、SYCL* アプリケーションを変更して、GPU セレクターやカスタムセレクターなど、より正確な SYCL* デバイスセレクターを設定することもあります。

環境変数 `ONEAPI_DEVICE_SELECTOR` を設定して、利用可能なデバイスセットを限定することで SYCL* デバイスセレクターを支援できます。例えば、DPC++ HIP プラグインでサポートされるデバイスのみを許可するには、次のように設定します。

```
$ export ONEAPI_DEVICE_SELECTOR="hip:*"
```

この環境変数の詳細については、インテル® oneAPI DPC++ コンパイラーのドキュメントで「環境変数」参照してください。

DPC++ のリソース

- インテル® DPC++ の概要 (英語)
- DPC++ 導入ガイド
- DPC++ コンパイラー・ユーザーズ・マニュアル (英語)
- DPC++ コンパイラーとランタイムのアーキテクチャー設計
- DPC++ 環境変数

SYCL* のリソース

- SYCL* 2020 仕様
- SYCL* アカデミー学習教材 (英語)
- Codingame インタラクティブ SYCL* チュートリアル (英語)
- IWOCL SYCL* トーク (英語)
- 無料の DPC++ 電子書籍 (英語)
- SYCL* の最新ニュース、学習教材、プロジェクトの紹介 (英語)

SYCL* アプリケーションのデバッグ

この節では、さまざまなデバイスで SYCL* アプリケーションをデバッグするための情報、ヒント、およびポインターについて説明します。

SYCL* アプリケーションのホストコードは、単純に C++ アプリケーションとしてデバッグできますが、カーネルデバッグのサポートやツールは、ターゲットデバイスによって異なる可能性があります。

注意: SYCL* アプリケーションに汎用性がある場合、実際のターゲットデバイスではなく、インテルの OpenCL* CPU デバイスなど、豊富なデバッグサポートとツールを備えたデバイスでデバッグしたほうが有用なことがあります。

インテルの OpenCL* CPU デバイスでのデバッグ

インテルの OpenCL* CPU デバイスを使用した DPC++ アプリケーションのデバッグについては、『インテル® oneAPI プログラミング・ガイド』の「DPC++ と OpenMP* オフロードプロセスのデバッグ」の節を参照してください。

NVIDIA* GPU 上での DPC++ を使用したコードのデバッグ

DPC++ と NVIDIA* GPU では、`cuda-gdb` を使用して、DPC++ `cuda` バックエンド向けにコンパイルされたカーネルをデバッグできます。インテル® oneAPI ベース・ツールキットに含まれる `oneapi-gdb` ツールは、インテル® プロセッサーのデバッグにのみ対応しているため、NVIDIA* GPU では使用できません。`cuda-gdb` ツールは、NVIDIA からダウンロードされる CUDA* ツールキットの一部であるため、このデバッガーを使用するには、CUDA* がインストールされていることを確認する必要があります。

`cuda-gdb` の詳しい使用法については、`cuda-gdb` のドキュメント (英語) を参照してください。

`cuda-gdb` デバッガーを使用するコードをコンパイルする場合、次のフラグを使用してデバッグシンボルを有効にする必要があります。

```
$ icpx -G -O0 …
```

注意: この記事で見られるような予期しないセグメンテーション・エラーや通信エラーが発生した場合、問題を軽減するには、`cuda-gdb` を起動するときに次の環境変数を設定することを推奨します。

```
CUDBG_USE_LEGACY_DEBUGGER=1
```

コマンドラインからのデバッグ

コマンドラインからデバッガーを起動するには、実行可能ファイルで次のコマンドを使用します。

```
cuda-gdb ./myexecutable
```

デバッガーを使用するためのコマンドとドキュメントは、NVIDIA のウェブサイト (英語) を参照してください。

注意: デバッグセッションを開始すると、`cuda-gdb` は oneAPI Python* スクリプトに関するセキュリティーの問題を警告する場合があります。この警告を削除するには、`/home/.config/gdb/cuda-gdb` にファイル `cuda-gdbinit` を作成し、インテル® oneAPI ベース・ツールキットの gdb Python* スクリプトへのパスを安全なパスリストに追加する行を挿入します。

```
add-auto-load-safe-path /path/to/libsycl.so.6.1.0-gdb.py
```

例えば、`/opt/intel/oneapi/compiler/2023.1.0/linux/lib/libsycl.so.6.1.0-gdb.py` のようになります。

含めるコマンドの例を以下に示します。

```
add-auto-load-safe-path
/opt/intel/oneapi/compiler/2023.1.0/linux/lib/libsycl.so.6.1.0-gdb.py
```

cuda-gdb を使用した VSCode によるデバッグ

注意: VSCode を SYCL* で動作させる設定方法については、次のガイドを参照してください。

VSCode でデバッガーを使用するには、起動時に `cuda-gdb` コマンドを使用するように VSCode デバッガー構成ファイル `launch.json` を変更する必要があります。また、必要なデバッグシンボルが生成されるように、前述のコンパイルフラグ (`-G` および `-O0`) が VSCode プロジェクトのコンパイル構成ファイル `task.json` で `icpx` コンパイラーへのパラメーターとして含まれていることを確認する必要があります。

VSCode がデバッグセッションを起動するように構成されている場合は、Shift + Ctrl + P キーを押してコマンドを表示し、「`Debug: Select Debug Session`」と入力して、リストから必要なデバッグセッションの種類を選択し、Enter キーを押します。

VSCode を使用して例にアタッチしてデバッグセッションを開始

すでに実行中のプログラムにデバッガーをアタッチすることもできます。デバッグビューの VSCode ターミナルパネルを使用して、コマンドライン・プロンプトに次のように入力します。

```
CUDBG_USE_LEGACY_DEBUGGER=1 ./<TheDPC++Executable> &
```

Shift + Ctrl + P キーを押して、[Debug: Start a new session] を選択し、[C/C++: Intel icpx build attached cuda-gdb debug CUDA target] を選択します。VSCode は実行中のプロセスの一覧を表示します。実行可能ファイルの名前で識別されるプロセスを選択します。

VSCode は、デバッグコンソールから `cuda-gdb` コマンドの直接入力をサポートしています。`cuda-gdb` コマンドを入力するには、次に示すように、`-exec` プレフィクスを追加する必要があります。デバッガーコマンドを実行すると、VSCode GUI とコードパネルがそれに応じて更新されます。

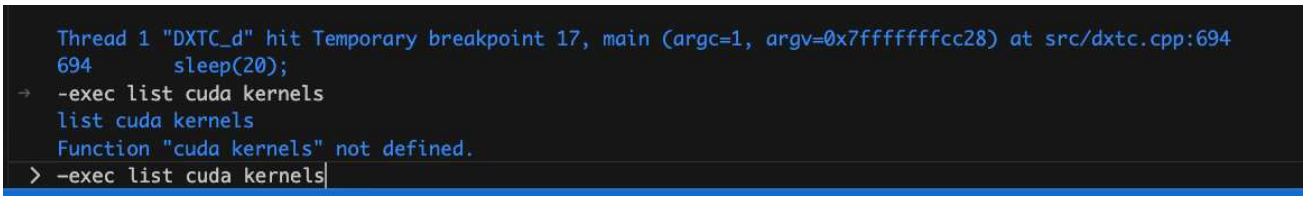

図 1: VSCode デバッグコンソールに `cuda-gdb` コマンドを直接入力

カーネルコードのデバッグ

カーネル `cgh.parallel_for()` にステップインすることは可能ですが、カーネルコードに到達するには数回ステップアウトする必要があります。

`parallel_for` ステートメントによってカーネル内またはコードにステップインする場合、次の `cuda-gdb` コマンドを使用して、どのカーネルまたはスレッド (カーネルはスレッド) とそのカーネル内のどの行が現在デバッグされているかを判断したり、カーネルがキューに入れられたばかりのプログラム内のポイントを見つけたりできます。

```
info threads
thread
info cuda kernels
```

ホストスレッドがデバッグフォーカスから外れたと判断され、次に示すようにカーネルがインスタンス化されると、デバッグフォーカスをカーネルに移動できます。

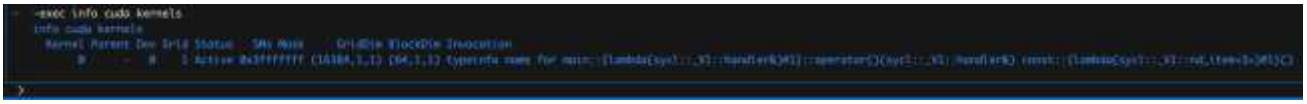

図 2: DPC++ カーネルはキューに投入されているがデバッグフォーカスから外れている

フォーカスを切り替えるには、`cuda-gdb` コマンドを使用します。

```
cuda kernel <id number of the kernel>
```

リストされているカーネルの横に星マークが表示され、カーネルにデバッグフォーカスがあることを示します。

フォーカスを切り替えると、デバッガーは残りの実行中のカーネル (別のスレッド) にあるブレークポイントで停止しますが、デバッガーは SYCL* ライブラリー内の別の場所で停止する可能性があります。この場合、`finish` コマンドを使用してスタックフレームを上に移動し、デバッガーを目的のカーネルコードに戻します。

デバッガーがキューの `wait()` ポイントでホストスレッドに一瞬戻りますが、ステップ実行しているカーネルコードに戻るまで待ちます。

インテル `oneapi-gdb` デバッガーでは、`gdb` の `set scheduler-locking on` または `step` コマンドを使用してデバッガーをロックし、実行中の他のカーネルにランダムにジャンプするのを防ぐことができます。ただし、`cuda-gdb` デバッガーには同等のコマンドがありません。

このような場合、デバッガーが別のカーネルに切り替わったときに、`cuda kernel id` コマンドで目的のカーネルに戻すことができます。デバッガーを使用していて、異なるカーネル (サブグループ) からの複数のバリアに遭遇すると、デバッガーが応答しなくなることがあります。状態を回復するには、デバッガーを強制終了して再起動します。

プロセスを強制終了する方法

デバッグセッションがハングアップしたり、何らかの問題が発生した場合、プログラムのプロセスはまだ実行中である可能性があり、新しいデバッグセッションを開始する前に停止する必要があります。プロセスの強制終了は、デバッガー自体から `kill` コマンドを使用して行うこともできます。

Linux* の `ps ux -u` <ユーザー名> コマンドを使用して、実行中のプロセスを一覧表示できます。一覧から、プログラムによって識別されるプロセスを見つけます。コマンドラインで、`kill -9` <プロセス `ID` 番号> と入力します。

複数種類のバックエンドを使用

DPC++ が複数のバックエンド (OpenCL* や CUDA* など) をサポートするように設定されているシステムを使用している場合、フィルターを使用してデバッグセッションを適切なバックエンドに関連付ける必要があることがあります。これには、デバッグセッションを開始するときに `SYCL_DEVICE_FILTER` 環境変数を使用します。ターミナル・コマンドラインから、次のコマンドを実行します。

```
SYCL_DEVICE_FILTER=cuda cuda-gdb ./myapp
```

ROCm* デバッガーのサポート

ROCm* SDK には `rocgdb` デバッガーが付属しており、HIP アプリケーションの AMD* GPU 上のカーネルをデバッグできます。

ただし、DPC++ では現在、AMD* GPU ターゲットの SYCL* カーネルに対し、適切なデバッグ情報を生成することができません。そのため、`rocgdb` を使用して SYCL* カーネルをデバッグすると、次のようなエラーが表示されれることがあります。

```
Thread 5 "dbg" hit Breakpoint 1, with lanes [0-63],
main::{lambda(sycl::_V1::handler&)#1}::operator()(sycl::_V1::handler&)
const::{lambda(sycl::_V1::id<1>)#1}::operator()(sycl::_V1::id<1>) const
(/long_pathname_so_that_rpms_can_package_the_debug_info/src/rocm-
gdb/gdb/dwarf2/frame.c:1032: internal-error: Unknown CFA rule.
```

デバッグ情報を生成せずにアプリケーションをビルドしても、デバッガーは役立ちます。例えば、カーネルが無効なメモリーアドバイスなどのエラーをスローする場合、rocgdb を使用してプログラムを実行することができます。エラー発生時にブレークして、`disas` コマンドを使用してエラーを引き起こした場所のカーネル・アセンブリー行を確認できます。

MPI ガイド

MPI と SYCL* は、選択したバックエンドが両方の実装をサポートしている場合、シームレスに連携できます。プログラマーは、SYCL* と GPU 対応 MPI を組み合わせて、さまざまなバックエンドにわたって 100% 移植可能なコードを作成できます。実際、CPU バックエンドのメモリーを含むあらゆる種類の SYCL* デバイス割り当てメモリーをサポートする MPI を包括するには、より一般的な用語である「デバイス対応」MPI を使用するほうが適切です。

MPI + SYCL* コードを使用する場合、いくつかのバックエンド固有の考慮事項があります。このガイドでは、DPC++ の `cuda` または `hip` バックエンドに関するこのような問題について詳しく説明します。インテル® GPU で GPU 対応 MPI を使用する (`level_zero` バックエンドを使用する) 場合の具体的な問題については、適切なインテルのドキュメントを参照してください。

GPU 対応 MPI を DPC++ の `cuda` または `hip` バックエンドで使用するのは簡単で、ネイティブ `cuda` (`nvcc`) または `hip` (`hipcc`) コンパイラーを使用する方法と似ています。コンパイル方法は、ネイティブ・コンパイラーで使用されるものとほぼ同じです。特定のコンピューティング・クラスターに特化したネイティブ・コンパイラーで GPU 対応 MPI を使用する方法を詳しく説明したドキュメントを参照している場合、命令を `icpx` に移行するのは、ネイティブ・コンパイラー呼び出しを `icpx` に置き換える (および関連する (アーキテクチャー指定子など) コンパイラー・フラグをそれに応じて調整する) のと同じくらい簡単です。次のセクションでは、より完全な詳細を説明します。

必要要件

ここでは、`hip` バックエンドをサポートするインテル® oneAPI DPC++ コンパイラーが正常にインストール済みであることを前提としています。このバックエンドをサポートする Codeplay oneAPI プラグインのインストール方法については、「oneAPI for AMD* GPU のインストール」を参照してください。また、ROCm* 対応の MPI 実装のビルドも必要です。インテル® oneAPI ツールキットには、CUDA*/ROCm* 非対応のインテル® MPI 実装が付属していることがあります。その場合は、ROCm* 対応を有効にしてソースからビルドされた OpenMPI または MPICH、または適切なデバイス・アクセラレーションを備えた CRAY* MPICH モジュールなど、別の実装を使用する必要があります。

このガイドで使用されているコード例は、SYCL-samples リポジトリー (英語) リポジトリーで入手できます。

Codeplay oneAPI プラグインで MPI を使用する

`send_recv_buff.cpp` (英語) および `send_recv_usm.cpp` (英語) サンプルコードは、バッファーまたは USM を使用してデバイス対応 MPI を DPC++ で使用する例を示す入門サンプルです。デバイス対応 MPI でバッファーを使用するには、`host_task` 内で MPI 呼び出しを行う必要があります。詳細については、`send_recv_buff.cpp` (英語) サンプルを参照してください。SYCL* USM を MPI で使用するには、常にメインスレッドから MPI 関数を直接呼び出す必要があります。`host_tack` 内から SYCL* USM を取得する MPI 関数の呼び出しの動作は、現在未定義です。さらに、`scatter_reduce_gather.cpp` (英語) サンプルでは、MPI を SYCL* 2020 のリダクションおよび `parallel_for` と併用して、最適化された簡略な複数ランクでのリダクションを実現する方法を示しています。

MPI ラッパーを使用してコンパイルする

サンプルをコンパイルするには、コンパイラー・ラッパー (`mpicxx` など) が DPC++ コンパイラー (`icpx`) を起動できるようにする必要があります。この方法でビルドする方法については、MPI 実装のドキュメントを参照してください。実装によっては、コマンドライン引数または環境変数 (例: `MPICH_CXX`、`OMPI_CXX`) を使用して、再構築せずに既定のコンパイラーを変更することもできます。

最初に、パスにラッパーが設定されていることを確認します。

```
$ export PATH=/path/to/your-mpi-install/bin:$PATH
```

次に、サンプルをコンパイルします。

```
$ mpicxx -fsycl -fsycl-targets=TARGET send_recv_usm.cpp -o ./res
```

`hip` バックエンドで `TARGET` を正しく指定する方法の詳細については、「導入ガイド」を参照してください。

サンプルを実行するには 2 つのランクが必要です。次のコマンドで MPI を使用してアプリケーションを実行します。

```
$ mpirun -n 2 ./res
```

`-n 2` は、2 つのランクを使用してアプリケーションを実行することを指定します。サンプルに変更を加えるか、各ランクで特別な環境変数を設定しない限り、これは 1 つの GPU を使用して 2 つのランクが実行されることに注意してください。

Cray* モジュールを使用してコンパイル

一部のクラスターでは、DPC++ と直接リンクできる Cray* MPICH モジュールが利用可能です。これを行うには、通常、ハードウェア固有の Cray* モジュールもロードされていることを確認する必要があります。次に例を示します。

```
$ module load craype-accel-amd-gfx90a
```

その後、`icpx` で直接コンパイルできます。適切な Cray* MPICH ライブラリーをインクルード/リンクする必要があります。リンクするライブラリーは使用するクラスターによって異なるため、適切なドキュメントを参照する必要がありますが、次のようなものになる場合があります。

```
$ icpx -fsycl -fsycl-targets=TARGET send_recv_usm.cpp -o res \
-I/opt/cray/pe/mpich/8.1.25/ofi/cray/10.0/include/ \
-L/opt/cray/pe/mpich/8.1.25/ofi/cray/10.0/lib -lmpi -o res
```

次の環境変数の設定が必要な場合もあります。

```
MPICH_GPU_SUPPORT_ENABLED=1
```

その後、特定のクラスターに応じて異なる標準のジョブ送信手順によりプログラムを実行できるようになります。

特定のデバイスに MPI ランクを割り当て

単一ノード内のデバイス対応 MPI では、ユーザーは各ランクが特定のデバイスを使用するかどうかを制御する機能が必要になります。これを実現する 1 つの方法は、プログラム内で特定のランクを特定のデバイスにマップすることです。

現在、DPC++ の `cuda` バックエンドでは、各 CUDA* デバイスに独自のプラットフォームを持ちます。`sycl::platform::get_platforms()` から返されるプラットフォームのベクトルは、CUDA* デバイス ID で順序付けされ、利用可能な最小 ID が最初に配置されます。これは、デバイス 0 と 1 が利用できる場合、次の方法でキューを初期化することで、2 ランクの MPI プログラムのランク 0 と 1 にこれらのデバイスを割り当てることを意味します。

```
std::vector<sycl::device> Devs;
for (const auto &plt : sycl::platform::get_platforms()) {
  if (plt.get_backend() == sycl::backend::hip) {}
    Devs=plt.get_devices();
    break;
  }
}
sycl::queue q{Devs[rank]};
```

AMD* GPU のみが利用可能である場合、よりシンプルな方法でリストを作成できます。

```
std::vector<sycl::device> Devs =
      sycl::device::get_devices(sycl::info::device_type::gpu);
```

MPI ランクを一意の GPU にマッピングする別の方法は、MPI プロセスごとに HIP_VISIBLE_DEVICES 環境変数 (英語) を単一の値にすることです。一般的には、`HIP_VISIBLE_DEVICES` をローカル MPI ランク ID と同じ値に設定します。ローカルランク ID を取得する方法は、MPI 実装のドキュメントを参照してください。

SLURM システムを利用する場合、GPU アフィニティー・オプション `--gpu-bind` を使用して、`HIP_VISIBLE_DEVICES` と同様の効果を実現できます。ただし、`--gpu-bind` オプションは計算クラスター固有であるため、クラスターのドキュメントを確認してください。詳細については、SLURM のドキュメント (英語) 参照してください。

パフォーマンス・ガイド

はじめに

このガイドは、SYCL* プログラミング・モデルと一般的な GPU におけるパフォーマンスの紹介から始まります。次に、GPU でのパフォーマンス解析の基本と、そこで使用される一般的なツールを紹介します。最後に、ベンダー固有の GPU と利用可能なツールについて紹介します。また、無料の書籍『Data Parallel C++』(英語) を一読されることを推奨します。第 15 章では、SYCL* と DPC++ に関連した GPU 上でのパフォーマンスについて説明されています。

NVIDIA* GPU と AMD* GPU の両方に適用される一般的な SYCL* 最適化については、「一般的な最適化」を参照してください。

AMD* GPU をターゲットにする固有の最適化については、「AMD* GPU 上のパフォーマンス」を参照してください。

インテル® GPU 固有のパフォーマンス最適化については、対応するインテル® GPU 固有のパフォーマンス・ガイドを参照してください。

プログラミング・モデル

グラフィックス処理ユニットは、超並列アーキテクチャーにより、CPU よりも 1 秒あたり多くの浮動小数点演算を実行でき、メモリー帯域幅も高くなっています。これらの機能は、コードの開発時点で GPU アーキテクチャーを使用することを選択した場合にのみ活用できます。

ここでは、GPU における大規模並列処理を表現するプログラミング・モデルが基本となります。SYCL* は OpenCL* や CUDA* と同様のプログラミング・モデルを採用しており、カーネル (GPU によって実行される関数) は work-item によって実行される操作で表現されます。

SYCL* 仕様 (Rev 8) の 3.7.2 節 (英語) では次のように定義されています。

> カーネルが実行のため送信されると、インデックス空間が定義されます。カーネルボディーのインスタンスは、インデックス空間の各ポイントで実行されます。カーネル・インスタンスは work-item (ワーク項目) と呼ばれ、グローバル id を提供するインデックス空間内のポイントで識別されます。それぞれの work-item は同じコードを実行しますが、コードと操作されるデータの実行パスは、work-item のグローバル id を使用して計算を特殊化することで異なります。

SYCL* では、2 つの異なるカーネル実行モデルを利用できます。

SYCL* 仕様 (Rev 8) の 3.7.2.1 節 (英語) では次のように記述されています。

> `range<N>` (`N` は 1、2 または 3) で定義される N 次元のインデックス空間でカーネルを呼び出す単純な実行モデルをサポートします。この場合、カーネルの work-item は独立して実行されます。各 work-item は、タイプ `item<N>` の値によって識別されます。タイプ `item<N>` は、タイ

プ `id<N>` の work-item 識別子と、カーネルを実行する work-item の数を示す `range<N>` をカプセル化します。

SYCL* 仕様 (Rev 8) の 3.7.2.2 節 (英語) では次のように記述されています。

> work-item を wark-group に編成できる ND-range の実行モデルは インデックス空間よりも粗い粒度の分解を提供します。それぞれの work-group には、work-item で使用できるインデックス空間と同じ次元の work-group id が割り当てられます。work-item には、それぞれ work-group 内で一意のローカル id が割り当てられるため、単一 work-item は、グローバル id、またはローカル id と work-group id の組み合わせで識別できます。特定の work-group 内の work-item は、単一の計算ユニットの処理ユニットで同時に実行されます。SYCL* で使用される work-group は、ND-range と呼ばれます。ND-range は、N 次元のインデックス空間であり、N は 1、2 または 3 です。SYCL* では、ND-range は `nd_range<N>` クラスを介して表現されます。`nd_range<N>` は、グローバルレンジとローカルレンジで構成され、それぞれ `range<N>` タイプの値で表現されます。さらに、タイプ `id<N>` 値で表現されるグローバルオフセットが存在することもありますが、これは SYCL* 2020 では非推奨です。タイプ `nd_range<N>` と `id<N>` は、それぞれ N 要素の整数配列です。`nd_range<N>` で定義される反復回数は、ND-range のグローバルオフセットで開始される N 次元のインデックス空間であり、サイズはグローバルレンジで、ローカル・レンジ・サイズの work-group に分割されます。ND-range の各 work-item は、タイプ `nd_range<N>` の値によって識別されます。タイプ `nd_range<N>` は、グローバル id、ローカル id、および work-group id をすべて `id<N>` (`id<N>` タイプの反復空間オフセットですが、SYCL* 2020 では非推奨) としてカプセル化し、グローバルとローカルレンジを同期して work-group を有効にします。work-group には、work-item のグローバル id と同様の方法で id が割り当てられます。work-item には work-group とゼロからその次元の work-group サイズから 1 を引いた範囲のコンポーネントを保持するローカル id が割り当てられます。つまり、work-group id と work-group 内のローカル id の組み合わせで work-item が一意に定義されます。

work-item は、次の OpenCL* メモリーモデルに従って 3 つの異なるメモリー領域にアクセスできます。

- **グローバルメモリー:** すべての work-group のすべての work-item 間で共有されます。
- **ローカルメモリー:** 同一 work-group のすべての work-item 間で共有されます。
- **プライベート・メモリー:** 各 work-item でプライベートです。

アーキテクチャー

SYCL* 仕様では、独立して動作する 1 つ以上の計算ユニット (CU) で構成されるデバイスを考慮することで、OpenCL* 1.2 の仕様に従います。NVIDIA では CU を ストリーミング・マルチプロセッサー (*streaming multiprocessor*) と呼び、AMD では単純に計算ユニット (*compute unit*) と呼んでいます。それぞれの CU は、1 つ以上の処理エレメント (PE) とローカルメモリーで構成されます。work-group は単一の CU で実行されますが、work-item は 1 つ以上の PE で実行されることがあります。一般に、CU は SIMD 形式で work-item の小さなセット (*sub-group* として定義) を実行します。sub-group は NVIDIA では ワープ (warp) 、AMD ではウェーブフロント (wavefront) と呼ばれます。sub-group サイズは NVIDIA 向けには 32 で、AMD 向けには通常 64 (一部のアーキテクチャー向けには 32) です。

計算

カーネルを構成する work-group は、CU 全体にスケジュールされます。この時点で、それぞれの CU は処理エレメントで 1 つ以上の *sub-group* を実行します。計算ユニットには、算術演算を実行する整数論理ユニットや浮動小数点ユニット、メモリー操作を行うロード/ストアユニット、超越関数 (正弦、余弦、逆数、平方根など) を実行する特別なユニット、AI で役立つ行列操作など、さまざまな種類の処理エレメントが含まれます。処理エレメントが操作を完了するのに要する時間 (クロックサイクルで測定) は、レイテンシーと呼ばれます。レイテンシーは操作の種類によって異なります。例えば、グローバル・メモリー・トランザクションのレイテンシーは、レジスター呼び出しに比べ桁違いに大きく、これは各種算術演算でも同じことが当てはまります。

スループットは、実行された操作の数と、それらの完了に要する時間の比率です。この比率は、命令のレイテンシーを減らすか、同時に実行する命令数を増やすことで高めることができます。これまで、CPU はクロック周波数を上げて命令レイテンシーを最小化することでスループットを向上させてきました。一方、GPU はレイテンシーを隠匿することでスループットを向上させます。これにより、CU は sub-group 間で「コンテキスト」(レジスター、命令カウンターなど) をわずかな労力で変更できます。そのため、操作に多くのクロックサイクルを要する場合、CU は「コンテキスト」を変更し、別の *sub-group* の操作を実行することでそれらを隠匿できます。アーキテクチャーによって、同時に実行できる *sub-group* の最大数は異なります。実際に実行中の *sub-group* と実行中の *sub-group* の最大数の比率は「占有率」として定義されます。次の節で詳しく説明します。

GPU における work-item の同時実行は、複数レベルで実現されます。

1. 同一 *sub-group* 内の異なる *work-item* は SIMD 形式で同期実行されます。つまり、同じ操作が異なるデータを実行します。
2. 前述したように、CU はレイテンシーを隠匿するため、同一または異なる *work-group* から複数の *sub-group* を同時に実行します。
3. GPU を構成する CU は、異なる *work-group* に属する、異なる *sub-group* を同時に実行します。

これらの並列実行の機能は、起動されたカーネルが GPU 全体をビジー状態にする十分な大きさの *work-item* を持っている場合にフル活用されます。

メモリー

次の図は、ディスクリート GPU を搭載したシステムにおける一般的な接続方法を示しています。[1] ホストとデバイスを接続し、[2] CU をグローバルメモリーに接続します。例えば、NVIDIA* GA100 GPU の目安となる帯域幅は次のようになります。[1] PCIe* x16 4.0 では 31GB/秒、および [2] HBM2 では 1555GB/秒。

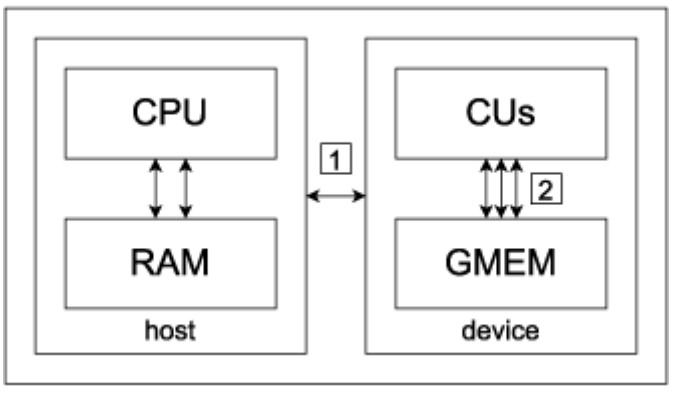

図 1

CPU と GPU 間の接続 [1] が大きなボトルネックになる可能性があります。そのため、ホストとデバイス間のデータ転送を慎重に検討し、GPU 上のデータの局所性を可能な限り維持することが重要です。ただし、カーネルの実行とオーバーラップすることで、PCIe* メモリーのトランザクションで生じるレイテンシーを隠匿することができます。

GPU の主要な特徴として、CU とグローバルメモリー間の高い帯域幅があります [3]。これは、それらを接続するメモリー・コントローラーの数と幅によるものです。例えば、NVIDIA* GA100 GPU には、12 個の 512 ビットの HBM メモリー・コントローラーがあります。これにより、クロックサイクルごとに大量のデータを転送できます。NVIDIA* GA100 GPU では、クロックごとに 6144 ビットです。ただし、この高帯域幅のメモリーを十分に活用するには、メモリーアクセスを結合する必要があります。つまり、work-item はキャッシュに最適な方法でメモリーアクセスする必要があります。

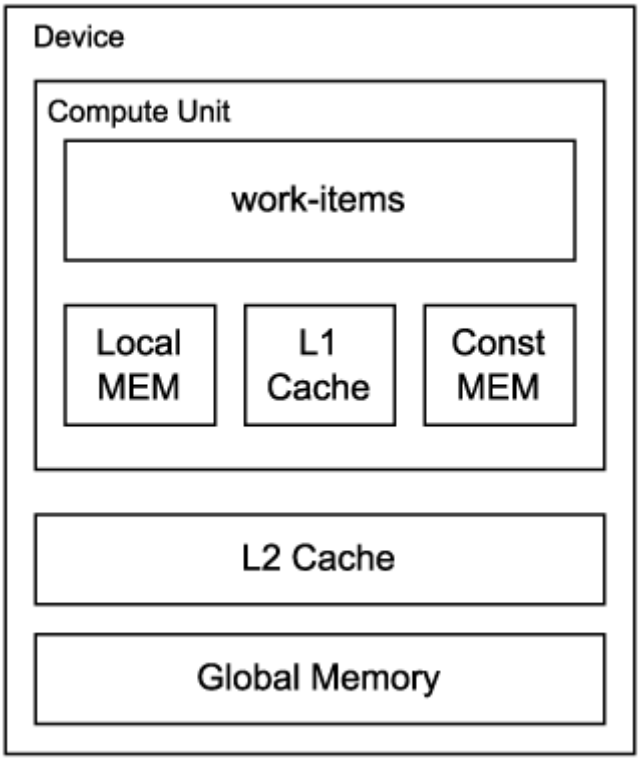

図 2

work-item とグローバルメモリー間にはいくつかのメモリー階層があります。以下に、それらをアクセス・レイテンシーが低い順に示します。

- **レジスター**は、ワークメモリーとして使用される work-item からは透過なデータを維持します。
- **コンスタント・メモリー**は、CU が使用する読み取り専用メモリーです。
- **ローカルメモリー**は CU ごとにあり、同一 work-group 内の work-item 間で共有されます。ローカルメモリーは、グローバルメモリーよりも高速であり、再利用されるグローバルメモリーのデータをキャッシュするために使用されます。
- グローバルメモリー (DDR または HBM) と CU を接続するメモリーシステムを構成する **L1** および **L2** キャッシュ。

最適化の目的

優先度

GPU コードのパフォーマンスに影響する主な要因を重要度の高い順に示します。

- **合成されていない**グローバル・メモリー・アクセス。キャッシュが完全に活用されると、メモリーアクセスは結合され、高い帯域幅を維持できます。結合の方法はアーキテクチャーによって異なりますが、一般に、同じ sub-group 内の work-item が連続したメモリー位置をアクセスすることで実現されます。
- ローカルメモリーの**バンク競合**。ローカルメモリーは複数のバンクに分割されており、異なる work-item から同時にアクセスできます。異なる work-item が同じメモリーバンクにアクセスすると、バンク競合が発生してトランザクションはシリアル化されます。
- if 文などの条件式やループの反復回数は work-item によって異なるため、同じ *sub-group* に属する work-item が異なる命令を実行することで**発散**が発生します。近年のアーキテクチャーではこの事象が緩和され、パフォーマンスのペナルティーが軽減されています。

計算の種類が異なれば最適化の優先順位も変わってきます。例えば、メモリー・トランザクションに対し算術演算が少ないメモリー依存のタスクを考えてみます。この場合、GPU を十分に活用するには、メモリーアクセスを結合することが重要です。一方、メモリー・トランザクションに対し算術演算が多い計算依存タスクがあります。この場合、スレッドの発散を回避することが有用な場合があります。算術演算数とリード/ライトデータのバイト数の比率は、**演算強度**として定義されます。

`I = (浮動小数点操作数) / (リード/ライトデータのバイト数) [FLOP/バイト]`

ルーフライン・モデルを利用して、カーネルの演算強度をハードウェア特性に関連付けることで、カーネルがメモリー依存であるか計算依存であるか確認できます。**ルーフライン・モデル**は 2 次元座標として表示され、x 軸には演算強度が、y 軸には浮動小数点演算のスループット (FLOPS: 1 秒あたりの浮動小数点演算) が示されます。

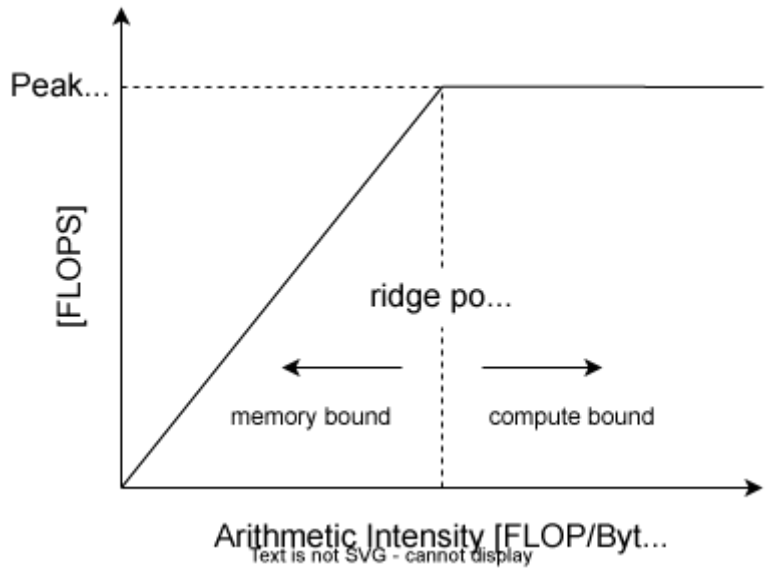

図 3

実際の**ルーフライン**を構成する最初のセグメントは、`y = x * B` を示します。ここで、`B` はグローバル・メモリー・システムの帯域幅です。次に水平線 (`y = Pmax`) は、FMA など特定の操作の最大浮動小数点スループット (`Pmax`) に依存します。セグメントが遭遇するポイントは**リッジポイント**と呼ばれます。

カーネルのパフォーマンスは、**ルーフライン**にプロットされたポイント (点) で示されます。x 軸はカーネルの演算強度を示し、y 軸は計測されたカーネルの FLOPS を示します。このポイントがリッジポイントの左にある場合、そのカーネルは**メモリー依存**であり、右にある場合は**計算依存**です。

占有率

カーネルのパフォーマンスを評価するには、その**占有率**を考慮します。占有率は、次のように定義される計算ユニットの式で求められます。

```
占有率 = アクティブな sub-group 数 / アクティブな sub-group の最大数
```

アクティブな sub-group は、CU で実際に実行される sub-group です。アクティブな sub-group の最大数は、計算ユニットのアーキテクチャーによって異なります。例えば、NVIDIA* GA100 CU アーキテクチャーでは 64 です。

占有率を高めるにはアクティブな sub-group 数を最大化する必要があります。ただし、計算ユニットのアーキテクチャーによって制約は異なります。

- **work-group あたりの work-item の最大数**
- **CU で実行される work-group の最大数:** work-group サイズが小さすぎると、CU はアクティブな sub-group の最大数を実行することができません。
- **レジスター数の制限:** カーネルコードが複雑になるとレジスターの使用量が増加します。コードを簡素にすることでレジスターの使用量を軽減できます。これは、コードを複数のカーネルに分割することで実現することもできます。
- **ローカルメモリー量の制限:** work-group がローカルメモリーを消費しすぎると、同時に実行できる work-group 数が減少します。

work-group が使用するレジスターやローカルメモリーが多いと、占有率が制限される可能性があります。ユーザーは、work-group のサイズを変更することで占有率を改善できます。このサイズは、sub-group サイズの倍数で、アクティブな sub-group の最大数の除数である必要があります。

例えば、NVIDIA* GA100 GPU では、各 work-item は最大 32 個のレジスターを使用して完全な占有を実現できます。

```
r_max = (CU ごとのレジスター数) / (アクティブな sub-group の最大数) * (sub-group サイズ) = 32
```

work-item が 32 未満のレジスターを使用する場合、CU で同時に実行できる work-group の最大数 `wg_max` とすると、ローカルメモリーにも同じことが当てはまります。

```
wg_max = (アクティブな sub-group の最大数) * (sub-group サイズ) / (実際の work-group サイズ)
```

各 work-group が最大 `48Kb / wg_max` のローカルメモリーを割り当てる場合、完全な占有率が得られます。

実効占有率は重要ですが、パフォーマンスにおける最重要のメトリックではありません。命令レベルの並列処理が十分にあり、同じ *sub-group* に属する独立した命令の同時実行が可能であれば、低い占有率でもレイテンシーを十分に隠匿できます。これについては、こちら (英語) をご覧ください。

さらに、GPU のすべての CU を利用するためカーネルで起動される work-item の最小数は、少なくとも次の `wi_min` でなければなりません。

```
wi_min = (アクティブな sub-group の最大数) * (sub-group サイズ) * (CU 数) = 262144
```

これらのパラメーターはすべて、特定ベンダーの各アーキテクチャーのドキュメントに記載されていますが、以下の表にいくつかの一般的な GPU アーキテクチャーの数値を示します。

一般的な GPU アーキテクチャーのリファレンス占有率

アーキテクチャー	アクティブな sub-group の最大数	work-item の最大数	work-group の最大数	レジスター数	ローカルメモリー (バイト)
NVIDIA* S.M. 7.0	64	2048	32	65536	65536
NVIDIA* S.M. 7.5	32	1024	16	65536	65536
NVIDIA* S.M. 8.0	64	2048	32	65536	65536
AMD* GFX9xx	40[1]	1024	16	29184 (?)	65536

[1] この図は、AMD アーキテクチャー全般に適用されます。work-group が 1 つの sub-group (例えば 64 work-item) のみである場合、CU あたりの work-group の最大数は 40 です。

パフォーマンス解析

パフォーマンス解析と最適化は繰り返し作業です。開発者は、ツールを使用してアプリケーションのパフォーマンスを測定しボトルネックを特定して、それらを改善しながら、この手順を繰り返します。それぞれの反復作業で、以前は見つからなかったボトルネックが明らかになることがあります。

ある時点で、アプリケーションの制限要因となる部分で、可能な限り高いパフォーマンスを特定することが重要です。これは、光速またはルーフラインと呼ばれることもあり、アプリケーションの理論上のピーク・パフォーマンスを予測したり、そのパフォーマンスにどれだけ近づいているかを判断するのに役立ちます。

以降の節では、解析ツールと制限要因について詳しく説明します。

解析の方法論

パフォーマンス解析に使用されるツールはプロファイラーとも呼ばれます。プロファイルという用語はいろいろな意味で使用されます。ここでは、パフォーマンス解析に使用されるツールの総称という意味で使用します。特定のパフォーマンス・ツールの説明では、より具体的な意味で使用されることがあります。

パフォーマンス解析は、大きく分けてトレースとサンプリングに分類されます。トレースは、アプリケーションの実行中に 1 つ以上のイベントが発生するたびに記録します。サンプリングは、実行中のアプリケーションの状態を定期的に調査して、その状態を記録します。頻繁に発生するイベントでは、トレースで大量のデータが蓄積される可能性があります。サンプリングでは、サンプリング間隔を調整することでデータ量を制御できます。間隔を長くするとデータ量は減りますが、短い間隔の動作を記録できないことがあります。どちらも改良すべき点はありますが、トレースまたはサンプリングのいずれかを実行中のデータ軽減と組み合わせることができます。

どの解析ツールでも、考慮すべき 2 つのことがあります。

- **オーバーヘッド:** ツールが通常のプログラムの実行時間をどれくらい増加させるかを表わします。パフォーマンス・ツールは、オーバーヘッドを最小限に抑えることが求められます。ただし、オーバーヘッドの増加を十分に理解している場合は、データを解釈する際にこれを補正できます。例えば、あるツールはコードの GPU 実行領域では正確な結果を提供し、CPU 実行領域では実行時間が長くなります。
- **データ量:** 生成されるファイルの大きさを示します。データ量が多いと、オーバーヘッドも増加します。また、大きな出力データセットは管理が困難で、特に出力データセットを表示するためリモートマシンに移動する場合、後処理ツールの応答性にも問題があります。

システムレベルの解析

システムレベルの解析では、同一ノードまたは異なるノード上のプロセス間の相互作用、および CPU と GPU 間の相互作用を調査します。

複雑なワークロードにおける CPU と GPU 間の相互作用を解析するのは困難なことがあります。ベンダーは、このような解析を支援するためトレースツールを提供することがあります。それらは、メモリー割り当て、メモリー転送、カーネルの起動、同期など、GPU 間の API 呼び出しのタイムスタンプと期間を記録します。これらのツールには、シリアル化や過度のアイドル時間などのボトルネックを視覚的に特定するタイムライン表示が含まれます。

状況に応じて、OS のカーネルトレース (Linux* ftrace など) を使用して、それをアプリケーションの実行に関連付けると便利です。これには、root 権限が必要になります。パフォーマンスの問題に関連するカーネルのアクティビティーが理解できない場合は、循環バッファーを利用するすべての OS アクティビティーを記録し、パフォーマンスの問題が検出されたときにアプリケーションの制御下でバッファーをダンプすると便利です (例えば、タイムステップが平均時間や予測時間よりも大幅に長くかかる場合)。循環バッファーによる手法は、トレース・データ・ストリーム全体を記録する際にコストが高い場合に有効です。

分散アプリケーションのスケーリング (通常メッセージ・パッシング・インターフェイス (英語) を使用) は特筆に値します。一般に使用されるスケーリングには 2 つの定義があります。**強力なスケーリング**は、問題のサイズを一定に保ち、MPI ランクの数が増加するのにしたがって経過時間を測定します。**弱いスケーリング**は、MPI ランクの数に比例して問題サイズを大きくします。

強力なスケーリングはより困難な問題です。多くの場合、すべての MPI ランクをビジーに保つのに十分なワークがありません。MPI プロファイル・ツールを利用して、異なる数のランクでスイープを実行し、MPI プロファイルを比較することが有用です。

特に大規模なスケーリングでは、そのほかの MPI の問題がしばしば発生します。リダクション操作は `log(N)` に反比例します。さらに、小さなリダクション操作 (スカラー値への `MPI_Allreduce` など) は、OS によるノイズの影響を受ける可能性があります。ネットワークが混雑する可能性があるため、大規模な共有クラスターではポイントツーポイント操作でも影響を受ける可能性があります。強力なスケーリングでは、メッセージサイズは通常、ランク数が多いほど小さくなるため、MPI レイテンシーがさらに重要になります。

開発者は、アプリケーションの動作が大規模なノードと小規模なノードで実行される際の違いを予測する必要があります。通常のように MPI プロファイル・ツールを使用すると、動作の違いを理解するのに役立ちます。オーバーヘッドの低いツールは、大規模なケースでは特に重要です。

カーネルレベルの解析

カーネルレベルの解析では、GPU カーネルの実行に費やされた時間と、個々の GPU カーネルのパフォーマンスに注目します。

前の節で説明したツールは、通常、起動パラメーター、起動回数、カーネルで消費された時間など、カーネル実行ごとのサマリーを示します。アプリケーションの合計時間は、CPU の経過時間と GPU の経過時間の合計として見積もられることが多く、GPU での経過時間はカーネルの実行時間の合計として概算されます。これにより、GPU カーネルの実行時間を改善することで、全体でどれだけ改善されるかが分かります。実行がオーバーラップしていたり、データの転送時間が長い場合は、常に正確であるとは限りませんが、経験則としては適切です。

カーネルのパフォーマンスを詳しく解析するには以下が必要です。

- カーネルのソースコードを調査
- カーネル向けにコンパイラーが生成するアセンブリー言語の調査
- カーネル実行中のハードウェア・パフォーマンス・メトリックの収集

アセンブリー言語を生成する方法は、コンパイラーと GPU によって異なります。詳細については以降で説明します。

ここからは、GPU (多くの場合 CPU にも該当) で利用可能なメトリックと、それらを解釈してパフォーマンスを改善する作業で導入できる一般的な手法について説明します。異なる GPU 向けの詳細については、このドキュメントの後半で説明します。

重要な GPU メトリック

レートメトリック

アプリケーションが GPU を使用するのは、利用可能な計算リソースを増やすためです。通常、計算スループットは、単位時間あたりに処理される演算数で示されます。例えば、倍精度浮動小数点演算の数/秒、32 ビット整数演算の数/秒などです。特定の GPU では、これらのピーク値が公開されています。

多くの場合、アプリケーションのピーク・パフォーマンスは、非計算リソース (特にメインメモリーやスクラッチパッド・メモリーなどさまざまなメモリー領域) へのアクセスによって制限されます。ここにもピーク値があります。例えば、メインメモリーの帯域幅は、単位時間あたりのバイト数で表現されます。

従来のルーフラインのようなモデルでは、ほかのリソースによる制限 (一般的なものはメインメモリーの帯域幅) を考慮して、達成可能な計算パフォーマンスを定量化しようとします。アプリケーションが計算以外のメトリックでピーク・パフォーマンスに達している場合、ピーク計算パフォーマンスを達成することはできません。これにより、開発者は、アプリケーションで達成可能なピーク・パフォーマンスに関する情報を得ることができます。

利用率メトリック

特定のリソースや機能ユニットがどれだけビジーであるかを知るのは有用です。この利用率メトリックは、レートメトリックとは異なります。リソースの利用率が高くても、ピーク・パフォーマンスにほど遠い場合があります。1 つの例として、メモリー・アクセス・パターンが不均一なカーネルが挙げられます。この場合、メモリー帯域幅がピークから離れていても、メモリーユニットの利用率は非常に高くなることがあります。利用率メトリックは、ルーフライン・モデルでは明らかにならないボトルネックを理解するのに役立ちます。

通常、利用率メトリックはメモリーユニットと計算ユニットで利用できます。また、キャッシュやローカルメモリーなど、各種マイクロアーキテクチャー・ブロックでも利用できることがあります。

発散

前述のように、GPU は複数の work-item を SIMD (単一命令複数データ) 方式で同時に実行する複数の計算ユニット (CU) で構成されています。

開発者は、単一の work-item に対して実行する操作を記述します。コンパイラーは、このコードを複数の work-item を同時に処理する命令に変換します。各 GPU には、sub-group サイズと呼ばれる、同時に実行される work-item の最小数がネイティブに設定されています。

発散 (Divergence) は、異なる work-item が異なるパスをたどることで発生します。多くの work-item が特定の命令で実行される場合、コンパイラーは可能なすべてのパスの組み合わせを考慮して命令を生成する必要があります。特定の命令で非アクティブな work-item は無効になります。これにより SIMD レーンの一部しか利用されないため、利用率は低下します。

GPU は発散を測定するメトリックを提供し (通常、sub-group ごとにアクティブな work-item)、ネイティブの sub-group サイズと比較できます。

占有率

GPU の占有率については前述しましたが、これは簡単に言うと、特定のカーネルで実際にアクティブな sub-group の数と、アクティブな sub-group の理論上の最大数との比率です。占有率は、カーネルが利用可能な最大の並列性をどれくらい活用できているかを開発者に示すため重要です。

一部の GPU には、実際の占有率を測定するハードウェア機能が備わっています。理論上の占有率は、コンパイルされたカーネルとハードウェアのプロパティーから計算できます。

起動パラメーター

カーネルは、グローバルレンジとローカルレンジで起動されます。後者は work-group のサイズです。work-group のサイズは、sub-group サイズの倍数である必要があります。そのため、グローバル問題サイズを切り上げたり、グローバル問題サイズ外の work-item を処理しないようカーネルにコードを追加する必要があります。

占有率を改善するため、特定の GPU ハードウェアの work-group サイズに制約が課される場合があります。CU 数など、特定の GPU ハードウェアと何らかの関連性のあるグローバル問題サイズを選択することも有益な場合があります。グローバルとローカルの問題サイズは自然なサイズに合わせる必要がなく、ハードウェアに適合するように選択できます。

すべての GPU は、カーネルの起動ごとに実際の起動パラメーターを確認するメカニズムを提供しています。これには、グローバルおよびローカル問題サイズ、レジスター数、およびローカル・メモリー・サイズなどのカーネル・プロパティーが含まれます。

AMD* GPU 上のパフォーマンス

このセクションでは、SYCL* プログラミング・モデルのコンテキスト内で AMD* GPU アーキテクチャーと関連するパフォーマンスの考慮事項を説明します。AMD* アーキテクチャー固有のパフォーマンスに関する最新の考慮事項については、AMD* のドキュメント (英語) を参照ください。

AMD* GPU アーキテクチャー

AMD には、ROCm* でサポートされる (英語) GCN、RDNA*、CDNA* の 3 つのアーキテクチャー・タイプがあります。RDNA* はグラフィックとゲーム用に設計されており、CDNA* はデータセンター環境での計算パフォーマンスを考慮して設計されています。古い GCN アーキテクチャーは、グラフィックと計算固有のカードの両方で利用されていました。

AMD CDNA* のホワイトペーパー (英語) では CDNA アーキテクチャーが解説されていますが、さらに詳しい情報は、CDNA1 ISA (英語) と CDNA2 ISA (英語) のドキュメントを参照してください。

AMD CDNA* GPU の基本計算ユニットは、**計算ユニット** (CU) と呼ばれます。GCN および CDNA* アーキテクチャーでは、CU は 64 個の work-item で構成される sub-group を実行します。RDNA* は sub-group サイズ 32 に最適化されていますが、サイズ 64 もサポートしています。ただし、oneAPI リリースの RDNA* GPU では、現在 sub-group サイズ 32 のみがサポートされます。AMD では、sub-group を**ウェーブフロント** (wavefront) または**ウェーブ** (wave) と呼んでいます。

AMD では OpenCL*/SYCL* で使用される work-group という用語も使用しています。work-group は、同一 CU 上で同時実行されることが保証され、SYCL* ローカルメモリーなどのローカルリソースを利用でき、work-item 間で同期できる sub-group (またはウェーブフロント) の集合です。

work-group のサイズは、常に sub-group サイズの倍数である必要があります。最適な work-group サイズは、特定のカーネルが使用するリソースに応じて、通常、占有率を最大化するように選択されます。work-item は 1024 を超えることはありません。

メモリーとキャッシュ

何種類かのメモリーが利用できます。すべてのグローバル (デバイス) 読み取りメモリーアクセスは、CU ごとの L1 キャッシュとデバイス全体の共有 L2 キャッシュを経由します。ストア操作はライト・コンバイン・キャッシュを経由し、次にアトミック操作を実行できる L2 キャッシュを経由します。

各 CU には、SYCL* ローカルメモリーに直接マッピングされる専用のローカルメモリー (Local Data Share、略称 LDS) があります。CU が持つローカルメモリーの量はアーキテクチャーによって異なります。ローカルメモリーは work-group で使用でき、グローバルメモリーよりも高い帯域幅と低いレイテンシーを備えます。ローカルメモリーには、32 のバンクがあり、異なるバンクに同時にアクセスすることで最高のパフォーマンスが得られます。同じバンクにアクセスするとバンク競合が発生し、パフォーマンスが低下します。バンク競合を測定するハードウェア・メトリックがあります。AMD* GPU でバンク競合を測定できる rocProf アプリケーションの詳細は、「AMD* パフォーマンス・ツール」を参照してください。

通常、異なる work-item はグローバルメモリー内の異なる場所にアクセスします。同じロード命令で特定の sub-group がアクセスするアドレスが同一キャッシュラインにある場合、メモリーシステムは最小限のグローバルアクセスを行います。これは、**メモリー結合** (memory coalescing) と呼ばれ、ライト・コンバイン・キャッシュによって支援され、隣接するメモリーにアクセスする work-item を隣接するスレッドに割り当てることで実現できます。大量のデータ構造を 64 バイト境界に配置すると、さらにパフォーマンスを向上できます。ただし、間接アクセスや大きなストライドによって、これが困難な場合もあります。

占有率

GPU はハードウェア・キューを使用して多数の sub-group を利用可能な状態に保ち、ストールした実行中の sub-group を別の sub-group と交換できるようにして、ハードウェアをビジー状態に保ちます。ハードウェア・キュー内の sub-group は、アクティブ sub-group と呼ばれます。アクティブ sub-group の理論上の最大数は、work-group サイズ、カーネルのレジスターとローカルメモリーの使用量、およびハードウェア・キューのサイズによって制限されます。占有率は、アクティブ sub-group の実際の数と理論上の最大数の比率として定義されます。ベンダー固有の詳細については、AMD のドキュメントを参照してください。

ただし、占有率が高いことが必ずしもパフォーマンスが高いことを意味するわけではありません。ほかにも多くの要因が関係する可能性があります。

AMD* パフォーマンス・ツール

ROCm* プロファイル・インフラストラクチャーは、rocProfiler と rocTracer の 2 つの API で構成されます。rocProfiler は NVIDIA* ツールの Nsight* Compute に、rocTracer は Nsight* Systems に似ています。どちらのツールも API に基づいており、アプリケーションに組み込んだり、rocProf (英語) コマンドライン・ツールを使用して (`LD_PRELOAD` などで) プリロードできます。このドキュメントでは、これらのツールの概要を説明します。詳細については、documentation (英語) と rocTracer (英語) のドキュメントを参照してください。

トレース

次のコマンドラインは、アプリケーションを実行して、カーネルの呼び出し、時間、デバイスメモリー転送などの情報をトレースします。

```
rocprof --timestamp on --hip-trace -o <outname>.csv <program> <program arguments>
```

コマンドラインはトレース付きで実行され、結果は `outname` で始まるか、`-o` を省略した場合は `results` という文字列で始まる複数のファイルに保存されます。`outname` には、`trace-results/myprog.csv` など、既存のパス名を指定できます。この場合、生成されるすべてのファイルは、`myprog` で始まります。

結果ファイルには、3 つの .csv があります (実際のファイルには上記のようにプリフィクスが付きます)。

- `copy_stats.csv` は、デバイスとの間のメモリー転送を要約します。
- `hip_stats.csv` は、HIP API 呼び出しを要約します。
- `stats.csv` は、カーネルの起動を要約します。

Chrome* トレースビューア (Chrome* ブラウザーで "chrome://tracing/" にアクセス) への入力として `.json` ファイルと sqlite3 データベースの `.db` ファイルもあります。

コピーとカーネル `.csv` ファイルには、コピーファイルの転送バイト数、カーネルファイルのグローバルサイズと work-group サイズなど、いくつかの情報が不足しているため、`.db` ファイルを参照してください。

時間計算

アプリケーションの実行時間は、ホスト時間、データ転送時間、GPU カーネル実行時間で構成されています。これらの操作の一部は重複することがあります。rocTracer は、ランタイム API 呼び出しをインターセプトし、非同期アクティビティーを追跡する API ライブラリーです。rocTracer で収集されたデータには、開始時間スタンプと終了時間スタンプが含まれるため、重複が発生するタイミングを確認できます。これは、Chrome* トレースビューアの使い方の一例です。

プロファイル

プロファイルは、ハードウェア・イベントを使用して、カーネルのさまざまな動作を測定します。イベントから、カーネルがハードウェアをどの程度効率良く使用しているかを示すメトリックを参照できます。これは、コードの改善を示す可能性があります。詳細については、rocProfiler のドキュメント (英語) を参照してください。

一般的な最適化

ここでは、DPC++ を使用する際の一般的なパフォーマンスの問題や落とし穴、そしてその対処方法について説明します。

インデックスの入れ替え

SYCL* 仕様の 4.9.1 節では、次のことが規定されています。

> 整数から多次元 id やレンジを構成する場合、多次元空間の線形化において右端の要素が最も速く変化するように要素を記述します。

そのため、インテル® DPC++ コンパイラーでは、右端の次元が CUDA* または HIP の x 次元にマップされ、右から 2 つ目の次元が CUDA* や HIP の y 次元にマップされます。以下に例を示します。

```
cgh.parallel_for(sycl::nd_range{sycl::range(WG_X * WI_X),
sycl::range(WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<2>{sycl::range<2>(WG_Y * WI_Y, WG_X * WI_X),
sycl::range<2>(WI_Y, WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<3>{sycl::range<3>(WG_Z * WI_Z, WG_Y * WI_Y, WG_X
* WI_X), sycl::range<3>(WI_Z, WI_Y, WI_X)}, ...)
```

`WG_X` と `WI_X` は、x 次元の **work-group 数**と **work-group ごとの work-item 数** (CUDA* では、名前付きの**グリッドサイズ**と**ブロックあたりのスレッド数**) であり、`_Y` と `_Z` は y 次元と z 次元のものになります。

次の場合、2 次元または 3 次元のカーネルの `parallel_for` 実行では特に重要であることに注意してください。

- ローカルまたはグローバルメモリー内の (1-d) 配列には、手動で線形化されたアクセスがあります。結合されていないグローバル・メモリー・アクセスまたはローカルメモリー内のバンク競合によるパフォーマンスの問題を回避するため、これを考慮する必要があります。線形化の詳細については、SYCL* 仕様の 3.11 節の多次元オブジェクトと線形化を参照してください。
- 次のエラー (または同等のエラー) が発生します。

  ```
  Number of work-groups exceed limit for dimension 1 : 379957 > 65535
  ```

  これは、CUDA* など一部のプラットフォームでは、x 次元が y および z 次元よりも多くの work-group をサポートするためです。

  ```
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  ```

このトピックの詳細については、こちら (英語) を参照してください。

インライン展開

DPC++ は、さまざまなデバイスでパフォーマンスとコンパイル時間のバランスを考慮して、自動的に関数をインライン展開するかどうかを選択します。しかし、プログラマーは、特定の関数に `always_inline` 属性を追加するなどして、インライン展開を強制することもできます。

```
__attribute__((always_inline)) void function(...) {
  ...
}

...

q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(..., [=](...) {
      function(...);
    });
}
```

関数を手動でインライン展開する場合は、十分注意してください。関数を手動でインライン展開すると、一部の NVIDIA* デバイスではパフォーマンスが向上する可能性がありますが、他の NVIDIA* デバイスではパフォーマンスが低下する可能性があることに注意してください。今後のリリースでは、コンパイラーの最適化ヒューリスティックを改善していく予定です。

高速数学ビルトイン

SYCL* 数学ビルトインは、同等の OpenCL* 1.2 数学ビルトインの精度要件と一致するように定義されていますが、一部のアプリケーションでは必要以上に精度が高くなり、パフォーマンスが低下する可能性があります。

これに対処するため、SYCL* 仕様では、数学関数のサブセットのネイティブバージョン (4.17.5 節の「数学関数」に完全なリストがあります) が提供されています。これには、精度とパフォーマンスのトレードオフがあります。これらは、ネイティブ名前空間で定義されています。例えば、`sycl::cos()` のネイティブバージョンは、`sycl::native::cos()` です。

一般に、精度が問題にならない場合、ネイティブバリアントを使用すると大幅に改善できる可能性がありますが、すべてのバックエンドがすべてのビルトインに対し緩和された精度を使用するわけではないことに注意してください。

注意: `-ffast-math` コンパイルオプションは、標準の `sycl::` 数学関数を、対応する `sycl::native::` 関数に入れ替えます (利用可能であれば)。指定された数学関数のネイティブバージョンが存在しない場合、`-ffast-math` フラグは影響しません。

`icpx` コンパイラーでは `-ffast-math` がデフォルトです。`icpx` で `-ffast-math` を無効にするには、`-fno-fast-math` を使用します。

ループアンロール

コンパイラーは一部のループアンロールを自動的に行いますが、次のように `unrolling` プラグマを使用して、デバイスコード内の計算集約型ループのアンロールを手動でコンパイラーに指示することが有益な場合もあります。

```
#pragma unroll <unroll factor>
for( ... ) {
 ...
}
```

関数を手動でインライン展開する場合は、十分注意してください。関数を手動でインライン展開すると、一部の NVIDIA* デバイスではパフォーマンスが向上する可能性がありますが、他の NVIDIA* デバイスではパフォーマンスが低下する可能性があることに注意してください。今後のリリースでは、コンパイラーの最適化ヒューリスティックを改善していく予定です。

インデックスのダウンキャスト

SYCL* では、`nd_range`、`range`、およびその他のインデックス・タイプは、値タイプとして `size_t` を使用します。これは、`threadIdx.{x|y|z}` が `int` である CUDA* や HIP などのバックエンド API とは対照的です。カーネル内のレジスタースペースを節約するには、インデックス計算関数から返される `size_t` から、バックエンド API がインデックス作成に使用するタイプ (通常は `int`) にインデックス・タイプをダウンキャストすると効果的です。

```
auto bigIdx = item.get_id(0); // size_t
int intIdx = static_cast<int>(item.get_id(0));
```

これは、レジスター・プレッシャーが高いカーネルでは特に有益です。

get_linear_id や get_global_linear_id などのメンバー関数は、SYCL* 実装では、異なる size_t インデックスが次のような方法で結合されます。

```
size_t item<2>::get_linear_id() {
  return get_id(0) * get_range(1) + get_id(1);
}
```

SYCL* は、32 ビット・タイプの使用時に発生するオーバーフローを回避するため、このような計算に size_t を使用するように指定されています。演算は get_range および get_id によって返される size_t タイプで行われるため、このメンバー関数の結果を int または他のタイプにキャストしても、関数の実装内で size_t の使用が排除されるわけではありません。したがって、プログラマーがこれらの計算に 32 ビット・タイプ (またはそれ以下) のみを使用する場合は、線形インデックス計算を抽象化する呼び出しを省略する必要があります。次のような unsafe_get_linear_id の実装は、インデックス計算から 64 ビット・サイズのタイプをすべて削除します。

```
inline int unsafe_get_linear_id(item<2> it) {
  return static_cast<int>(it.get_id(0)) * static_cast<int>(it.get_range(1))
              + static_cast<int>(it.get_id(1));
}
```

これにより、インデックス計算用の SYCL* コードのデバイスコードのフットプリントがネイティブ CUDA* または HIP コードと同じになります。

上記の例は、プログラマーが整数オーバーフローが発生しないようにする責任を負うため、安全ではないと考えられます。

エイリアス解析

エイリアス解析では、2 つのメモリー参照が互いにエイリアスでないことが証明できます。これにより最適化が有効になることがあります。デフォルトでは、コンパイラーはエイリアス解析によって証明されない限り、メモリー参照はエイリアスであると想定する必要があります。ただし、デバイスコード内のメモリー参照がエイリアスではないことをコンパイラーに明示的に通知することもできます。これは、バッファー/アクセサーと USM モデルのそれぞれのキーワードを使用することで実現できます。

前者は、oneapi 拡張の no_alias プロパティーをアクセサーに追加することができます。

```
q.submit([&](sycl::handler &cgh) {
  sycl::accessor acc{...,
sycl::ext::oneapi::accessor_property_list{sycl::ext::oneapi::no_alias}};
  ...
});
```

後者の場合、__restrict__ 修飾子をポインターに追加できます。

__restrict__ は C++ では非標準であり、SYCL* 実装全体で一貫性がない可能性があることに注意してください。dpc++ では、restrict 修飾されたデバイス関数 (SYCL* カーネルから呼び出される関数) パラメーターのみが考慮されます。

例:

```
void function(int *__restrict__ ptr) {
    ...
}

...
int *ptr = sycl::malloc_device<int>(..., q);
...
q.submit([&](sycl::handler &cgh) {
      cgh.parallel_for(..., [=](...) {
        function(ptr);
      });
});
```

より強制的なアプローチは、`[[intel::kernel_args_restrict]]` 属性をカーネルに追加することです。これは、各 USM ポインター間、またはそのモデルがカーネル内で使用される場合はバッファーアクセサー間のすべてのエイリアス依存関係を無視するようにコンパイラーに指示します。

例 (バッファー/アクセサーモデル):

```
q.submit([&](handler& cgh) {
      accessor in_accessor(in_buf, cgh, read_only);
      accessor out_accessor(out_buf, cgh, write_only);
        cgh.single_task<NoAliases>([=]() [[intel::kernel_args_restrict]] {
          for (int i = 0; i < N; i++)
              out_accessor[i] = in_accessor[i];
        });
});
```

テクスチャー・キャッシュの使用

CUDA* プラットフォームでは、少なくともカーネルの存続期間中は一定であるデータをテクスチャー・キャッシュにキャッシュできます。

これは、`sycl::ext::oneapi::experimental::cuda::ldg` 関数を使用して実現できます。この関数は、デバイスメモリーへのポインターを受け取り、L1/tex キャッシュからロードして、アドレスに格納された値を返します。以下に例を示します。

```
float some_value = ldg(&some_data_in_device_memory[some_index]);
```

警告: この関数でロードされたデータがカーネル内に書き込まれることをコンパイラーが検出した場合でも、プログラムはコンパイルできますが、テクスチャー・キャッシュは使用されないことに注意してください。

テクスチャー・キャッシュを使用することでパフォーマンス向上に影響する要因は数多くあります。そのため、最大の高速化を達成するのは困難な場合があります。実際、多くのユースケースではメリットがほとんどないか、全くありません。ただし、パフォーマンスが低下する可能性は低く、低下した場合でも規模は小さく、`ldg` は最小限のコード変更で使用できるため、カーネルのパフォーマンスを素早く向上する素晴らしい方法となるかもしれません。

`ldg` は、HIP AMD を含むほかのすべてのプラットフォームでも移植可能であることに注意してください。しかし、CUDA* は現在 `ldg` により特殊なキャッシングが可能な唯一のプラットフォームです。HIP AMD バックエンドは、`ldg` を使用するかどうかにかかわらず、常にすべてのレジスターデータを L1 キャッシュと L2 キャッシュにロードします。

テクスチャー・キャッシュの詳細については、こちらのブログ (英語) を参照してください。`ldg` 関数の詳細は、対応する拡張機能のドキュメント (英語) をご覧ください。

サポート

機能

コア機能

機能	サポート
コンテキスト内の複数デバイス	いいえ
サブグループ (sub-group)	はい
グループ関数/アルゴリズム	はい
整数関数	はい
数学関数 (スカラー)	はい
数学関数 (ベクトル)	はい
数学関数 (marray)	いいえ
共通関数	はい
ジオメトリー関数	はい
リレーショナル関数	はい
atomic ref	はい
オペレーティング・システム	Linux*
バッファーの再解釈	はい
stream	いいえ
デバイスイベント	はい
グループの非同期コピー	はい
プラットフォームの get info	はい
カーネルの get info	はい
`sycl::nan` と `sycl::isnan`	はい
デバイスセレクター	いいえ
階層的並列化	はい
ホストタスク	はい
インオーダー・キュー	はい
リダクション	はい (half を除く)
キューのショートカット	はい
vec	はい
marray	はい
errc	はい
匿名カーネルラムダ	はい
機能を評価するマクロ	はい

機能	サポート
sycl::span	はい
sycl::dynamic_extent	いいえ[1]
sycl::bit_cast	はい
aspect_selector	いいえ
カーネルバンドル	いいえ
特殊化定数	エミュレーション

非コア機能

機能	サポート
image	いいえ
fp16 データタイプ	はい
fp64 データタイプ	はい
prefetch	はい
USM	ホスト、デバイス、共有
USM アトミックホスト割り当て	はい
USM アトミック共有割り当て	問題あり [^shared-usm-status]
USM システムに割り当て	はい
SYCL_EXTERNAL	はい
アトミックメモリーの順序付け	relaxed
アトミック・フェンス・メモリーの順序付け	いいえ
アトミック・メモリー・スコープ	work_group
アトミック・フェンス・メモリーのスコープ	いいえ
64 ビット・アトミック	いいえ
バイナリー形式	AMDGCN
デバイスのパーティション化	いいえ
ホストデバッグ可能デバイス	いいえ
オンラインコンパイル	いいえ
オンラインリンカー	いいえ
キューのプロファイル	はい
mem_advise	いいえ
バックエンド仕様	いいえ
アプリケーション・バックエンドの相互運用	いいえ
カーネル・バックエンドの相互運用	いいえ
ホストタスク (ハンドルと相互運用)	いいえ
reqd_work_group_size	いいえ

機能	サポート
キャッシュビルド結果	いいえ
ビルドログ	いいえ
ビルトインカーネル関数	なし

拡張機能

機能	サポート
uniform	いいえ
USM アドレス空間 (デバイス、ホスト)	部分的[2]
固定ホストメモリーの使用	はい
サブグループ・マスク (+ グループ投票)	いいえ
静的ローカルメモリー使用量照会	いいえ
sRGB イメージ	いいえ
デフォルト・プラットフォーム・コンテキスト	はい
メモリーチャネル	いいえ
最大ワークグループ照会	部分的
結合行列	実験的 (gfx90a のみ)
すべて制限 (restrict all)	いいえ
プロパティー・リスト (property list)	いいえ
カーネル・プロパティー (kernel properties)	いいえ
SIMD 呼び出し	いいえ
低レベルデバイス情報	いいえ
カーネルキャッシュ設定	いいえ
FPGA lsu	いいえ
FPGA reg	いいえ
データ・フロー・パイプ	いいえ
キューに投入されたバリア	いいえ
フィルターセレクター	はい
グループソート	はい
フリー関数の照会	いいえ
明示的な SIMD	いいえ
discard_queue_events	はい
device_if	いいえ
device_global	いいえ
C と C++ 標準ライブラリーのサポート	いいえ
カーネルでの assert	はい

機能	サポート
buffer_location	いいえ
accessor_property_list (+ no_offset、no_alias)	はい
グループ・ローカル・メモリー	はい
printf	いいえ
ext_oneapi_bfloat16	いいえ
拡張デバイス情報	いいえ
sycl_ext_oneapi_cuda_tex_cache_read	はい[3]
sycl_ext_oneapi_native_math	はい
sycl_ext_oneapi_bfloat16_math_functions	いいえ
sycl_ext_oneapi_cuda_async_barrier	いいえ
sycl_ext_oneapi_bindless_images	いいえ
sycl_ext_oneapi_graph	実験的
sycl_ext_oneapi_non_uniform_groups	いいえ
sycl_ext_oneapi_peer_access	はい

[1] numeric_limits<size_t>::max() の使用

[2] 一部のシステムの SYCL* ビルトインとアトミックの USM には、まだいくつかの既知の問題があります。

[3] 技術的にサポートされますが、sycl_ext_oneapi_cuda_tex_cache_read は NVIDIA* GPU でのみ有効です。

更新履歴

2024.2.0

- AMD プラグインがベータ版ではなくなりました。

改良点

- ROCm* 5.7、6.0、6.1 をサポートしました (ROCm* 4.5 のサポートが終了しました)。
- `sycl_ext_oneapi_graph` 拡張機能 (ROCm* 5.5+) を実験的にサポートしました [897b2707]。
- メモリーアドバイスをサポートしました [a669374b]。
- `sycl_ext_oneapi_device_global` 拡張機能をサポートしました [d377464c]。
- `sycl_ext_oneapi_peer_access` 拡張機能をサポートしました [unified-runtime f39d41f7]。

バグフィックス

- 半浮動小数点型のシャッフルを修正しました [b13a3c4]。
- 共有 USM 割り当てを使用する際の数学ビルトインを修正しました [9e4768c]。
- ローカル作業サイズ推測を修正および改善しました [unified-runtime 43f0963]。
- SYCL* ターゲットのショートカット構文に不足しているアーキテクチャーを追加しました [c1ce1594]。
- 新しい ROCm* バージョンでの USM 2D コピーを修正しました [unified-runtime 532dac51]。

2024.1.0

改良点

- 半浮動小数点型をサポートしました [89875490]。
- SYCL* サブグループ演算の実装を完了しました [289aeaef、3f3df772]。
- `sycl_ext_oneapi_matrix` を介して CDNA2 アーキテクチャー (gfx90a) の AMD* Matrix コアサポートを追加しました [31481cea]。
- `sycl_ext_oneapi_device_architecture` をサポートしました [1ad69e59]。
- `ext_oneapi_queue_priority` をサポートしました [0c33fea5]。
- エラーコードの戻り値の関連性を改善しました [66a24f7b、b7a43a42]。
- 不足している `make_` 相互運用性エントリーポイントをいくつか実装しました [5e9d07b1]。
- プライマリー HIP コンテキストの使用に切り替えました [d1c92cb9]。

バグフィックス

- `-mllvm -enable-global-offset=false` フラグの使用を修正しました [00cf4c29]。
- カーネル起動前の余分なプリフェッチ呼び出しを削除しました [unified-runtime 841a2870]。

- 一部のシステムでの `atomic_sub` バグを解決しました [c8a6f0dd]。
- イベント・プロファイルの競合状態を修正しました [e8ffd021]。

非推奨

- コンテキストの相互運用性が非推奨となり、代わりにプライマリー・コンテキストを使用する必要があります [e213fe2f]。

2024.0.2

- 変更なし。

2024.0.1

- 変更なし。

2024.0.0

- 変更なし。

2023.2.0

改良点

SYCL* コンパイラー

- gfx9+ HIP atomic が追加されました [b13561c9]。
- basic HIP atomic が追加されました [c3c5e923]。
- AMD HIP プラットフォームで `__CUDA_ARCH__` マクロが無効化されました [8a7cf2b2]。

SYCL* ライブラリー

- AMD バックエンドで `sycl_ext_oneapi_memcpy2d` をサポートしました –・ (英語) [9008a5d2]。
- PCI デバイス ID と UUID のサポートが追加されました [e09ff588]。
- `SYCL_PI_HIP_MAX_LOCAL_MEM_SIZE` 環境変数がサポートされました [92f6d688]。
- `-fsycl-targets` で `amd-gpu-gfx1034` を指定できるようになりました [5e86a41d]。

バグフィックス

- 無効な work-group サイズに関するエラーを `PI_ERROR_INVALID_WORK_GROUP_SIZE` に置き換えるようになりました [2357af0a]。
- `sycl::ctz` 関数からの間違った結果に対応しました [5a9f601e]。
- イベントが意図したとおりに待機しない原因となる問題に対処しました [1b225447]、[ce7c594f]。

2023.1.0

改良点

SYCL* コンパイラー

- `-fsycl-targets` の引数として AMD* アーキテクチャー (`amd_gpu_gfx1032` など) を指定できるようになりました [e5de913f]。

SYCL* ライブラリー

- デバイス拡張に `cl_khr_fp64` が追加されました [cd832bff]。
- HIP* バックエンドのゼロ・レンジ・カーネルをサポートしました [a3958865]。

バグフィックス

- ガードが正しく構築されない問題を修正しました [ce7c594f]。
- 相互運用ヘッダーとデバイスの特殊化が追加されました [998fd91e]。

2023.0.0

oneAPI for AMD* GPU の最初の**ベータ**リリースです。

このリリースは、intel/llvm repository at commit 0f579ba (英語) から作成されました。

新機能

- HIP バックエンドのベータサポート

SYCL* コンパイラー

- デバイスでの `assert` をサポート
- ローカル・メモリー・アクセサーのサポート
- グループ集合関数のサポート
- `sycl::ext::oneapi::sub_group::get_local_id` のサポート

SYCL* ライブラリー

- `atomic64` デバイス機能の照会をサポート
- SYCL* キューごとに複数の HIP ストリームをサポート
- 相互運用のサポート
- `sycl::queue::submit_barrier` のサポート

トラブルシューティング

この節では、トラブルシューティングのヒントと一般的な問題の解決方法について説明します。ここで説明する方法で問題が解決しない場合は、Codeplay のコミュニティー・サポート・ウェブサイト (英語) からサポートリクエストをお送りください。完全なサポートは保証できませんが、できる限り支援させていただきます。サポートリクエストを送信する前に、ソフトウェアが最新の安定したバージョンであることを確認してください。

問題、パフォーマンス、機能要望は、oneAPI DPC++ コンパイラーのオープンソース・リポジトリー (英語) からも報告できます。

sycl-ls の出力にデバイスが表示されない

`sycl-ls` がシステム上の期待されるデバイスを報告しない場合:

1. システムに互換性のあるバージョンの CUDA* または ROCm* ツールキット (それぞれ CUDA* と HIP プラグイン向け)、および互換性のあるドライバーがインストールされていることを確認してください。
2. `nvidia-smi` または `rocm-smi` がデバイスを正しく認識できることを確認します。
3. プラグインが正しくロードされていることを確認します。これは、環境変数 `SYCL_PI_TRACE` に `1` を設定して、`sycl-ls` を再度実行することで分かります。

   例:

   ```
   $ SYCL_PI_TRACE=1 sycl-ls
   ```

   次のような出力が得られるはずです。

   ```
   SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_opencl.so
   [ PluginVersion: 11.15.1 ]
   SYCL_PI_TRACE[basic]: Plugin found and successfully loaded:
   libpi_level_zero.so [ PluginVersion: 11.15.1 ]
   SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_cuda.so
   [ PluginVersion: 11.15.1 ]
   [cuda:gpu][cuda:0] NVIDIA CUDA BACKEND, NVIDIA A100-PCIE-40GB 8.0 [CUDA
   12.5]
   ```

   インストールしたプラグインが `sycl-ls` の出力に表示されない場合、`SYCL_PI_TRACE` に `-1` を設定して再度実行することで、詳細なエラー情報を取得できます。

   ```
   $ SYCL_PI_TRACE=-1 sycl-ls
   ```

   大量の出力が得られますが、次のようなエラーが表示されているか確認してください。

   ```
   SYCL_PI_TRACE[-1]:
   dlopen(/opt/intel/oneapi/compiler/2024.2.0/linux/lib/libpi_hip.so) failed
   with <libamdhip64.so.4: cannot open shared object file: No such file or
   directory>
   SYCL_PI_TRACE[all]: Check if plugin is present.Failed to load plugin:
   libpi_hip.so
   ```

   - CUDA* プラグインには、CUDA* SDK で提供される `libcuda.so` と `libcupti.so` が必要です。

- HIP プラグインには、ROCm* の `libamdhip64.so` が必要です。

CUDA* または ROCm* のインストールと、環境が適切に設定されていることを確認してください。また、`LD_LIBRARY_PATH` が上記のライブラリーを検出できる場所を指しているか確認してください。

4. `ONEAPI_DEVICE_SELECTOR` または `SYCL_DEVICE_ALLOWLIST` などのデバイスフィルター環境変数が設定されていないことを確認します (`ONEAPI_DEVICE_SELECTOR` が設定されていると、`sycl-ls` は警告を表示します)。
5. 権限を確認します。POSIX* では、アクセラレーター・デバイスへのアクセスは、通常、適切なグループのメンバーであることを条件としています。例えば、Ubuntu* Linux* の場合、GPU へのアクセスには `video` グループと `render` グループのメンバーである必要がありますが、これは設定によって異なります。

不正バイナリーエラーの扱い

不適切なプラットフォーム

よくある間違いは、SYCL* プログラムに互換性のあるバイナリーがないプラットフォームを使用して SYCL* プログラムを実行することです。例えば、SYCL* プログラムは SPIR-V* バックエンド用にコンパイルされた後、HIP デバイス上で実行される可能性があります。この場合、`PI_ERROR_INVALID_BINARY` エラーコードがスローされます。この場合、次の点を確認してください。

1. プログラムが適切なプラットフォーム向けにコンパイルされるよう、`-fsycl-targets` にターゲット・プラットフォームが指定されていることを確認します。
2. プログラムが、実行可能ファイルがコンパイルされたプラットフォームと互換性のある SYCL* プラットフォームまたはデバイスセレクターを使用していることを確認します。環境変数 `SYCL_PI_TRACE=1` を設定すると、選択されたデバイスに関連するトレース情報を表示できます。

適切なプラットフォームと不適切なデバイス

CUDA* または HIP をターゲットにする SYCL* アプリケーションを実行すると、特定の状況でアプリケーションが失敗し、無効なバイナリーであることを示すエラーが報告されることがあります。例えば、CUDA* の場合は `CUDA_ERROR_NO_BINARY_FOR_GPU` がレポートされる場合があります。

これは、選択された SYCL* デバイスに適切でないアーキテクチャーのバイナリーが送信されたことを意味します。この場合、次の点を確認してください。

1. アプリケーションが、利用するハードウェアのアーキテクチャーと一致するようにビルドされていることを確認してください。
   - CUDA* 向けのフラグ:
     `-Xsycl-target-backend=nvptx64-nvidia-cuda --cuda-gpu-arch=<arch>`
   - HIP 向けのフラグ:
     `-Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=<arch>`
2. 実行時に適切な SYCL* デバイス (ビルドされたアプリケーションのアーキテクチャーに一致するもの) が選択されていることを確認します。環境変数 `SYCL_PI_TRACE=1` を設定すると、選択されたデバイスに関連するトレース情報を表示できます。以下に例を示します。

```
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_opencl.so
[ PluginVersion: 11.16.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded:
libpi_level_zero.so [ PluginVersion: 11.16.1 ]
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: libpi_cuda.so
[ PluginVersion: 11.16.1 ]
SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
SYCL_PI_TRACE[all]: Requested device_type: info::device_type::automatic
SYCL_PI_TRACE[all]: Selected device: -> final score = 1500
SYCL_PI_TRACE[all]: platform: NVIDIA CUDA BACKEND
SYCL_PI_TRACE[all]: device: NVIDIA GeForce GTX 1050 Ti
```

3. 誤ったデバイスが選択されている場合、環境変数 `ONEAPI_DEVICE_SELECTOR` を使用して SYCL* デバイスセレクターが選択するデバイスを変更できます。インテル® oneAPI DPC++/C++ コンパイラーのドキュメントにある「環境変数」の節を参照してください。

外部参照関数「…」を解決できません/外部シンボル「…」が未定義です

これにはいくつかの原因が考えられます。

1. 現在 DPC++ では `std::complex` はサポートされていません。代わりに `sycl::complex` を使用してください。
2. DPC++ の AMD* GPU バックエンドのカーネルコードでは、`<cmath>` で宣言された C++ 標準ライブラリーの一部の数学機能 (`std::cos`、`logf`、`sinf` など) がサポートされていません。代わりに、同等の `sycl` 名前空間バージョンを使用してください。

   詳細は、「oneAPI for AMD* GPU のインストール」を参照してください。

プラットフォーム/アーキテクチャー間で移植されたコードの sub-group サイズの問題

カーネル属性 `reqd_sub_group_size` を使用して特定の sub-group サイズを設定し、その後、異なるプラットフォームに移植するか、元のアーキテクチャーとは異なるアーキテクチャーで実行されるコードについて考えてみます。このような場合、要求される sub-group サイズがプラットフォーム/アーキテクチャーでサポートされないと、実行時に次のようなエラーがスローされます。

```
Sub-group size x is not supported on the device
```

CUDA* プラットフォームでは、単一の sub-group サイズのみがサポートされるため、次の警告が出力されます。

```
CUDA requires sub_group size 32
```

そして、ランタイムは要求された sub-group サイズに代わって sub-group サイズ 32 を適用します。`reqd_sub_group_size` カーネル属性は、複数の sub-group サイズをサポートするプラットフォーム/アーキテクチャー向けに設計されています。一部の SYCL* コードは、異なる sub-group サイズ間では移植できないことに注意してください。例えば、sub-group 集合の `reduce_over_group` の結果は、sub-group サイズに依存します。異なる sub-group サイズを使用するプラットフォーム/アーキテクチャー間で移植できるコードを作成する場合、次のいずれかを考慮する必要があります。

- 結果が sub-group サイズに依存しないよう、移植可能な方法でコードを記述します。
- コードの sub-group サイズに依存する部分については、sub-group サイズの違いを考慮して、プラットフォーム/アーキテクチャーごとに異なるバージョンを用意します。
