ベータ版 oneAPI for AMD* GPU 2023.1.0 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 2023 年 4 月 4 日時点の『oneAPI for AMD* GPUs (beta) 2023.1.0』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。

バージョン 2023.0.0 のガイドはこちら。
『oneAPI for NVIDIA* GPU 2023.1.0 ガイド』はこちら。

ベータ版 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。

注意: これはベータ品質のソフトウェアであり、主要機能のほとんどが含まれていますが、まだ完全ではなく、既知および未確認のバグがあることに留意してください。サポートされる機能の詳細については、「機能」を参照してください。

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、「ベータ版 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。

ベータ版 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

- ベータ版 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

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
- NVIDIA* GPU 上のパフォーマンス
- 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
- 使用許諾契約書 (英語)

導入ガイド

ベータ版 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。

これはベータ品質のソフトウェアであり、主要機能のほとんどが含まれていますが、まだ完全ではなく、既知および未確認のバグがあることに留意してください。サポートされる機能の詳細については、「機能」を参照してください。

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。

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	GPU ドライバー
AMD Radeon* Pro W6800	gfx1030	Ubuntu* 22.04.2 LTS	5.4.1	6.1.0-1006-oem

- このリリースは HIP 5.x バージョンで動作するはずですが、HIP 5.4.1 でのみテストされています。CodePlay は、HIP 5.x よりも古いバージョンでは正常な動作を保証いたしかねます。
    - HIP 5.4.1 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「複数バージョンの ROCm* インストールでインストーラー・スクリプトを使用する」(英語) の説明を参照してください。
- このリリースは各種 AMD* GPU と HIP バージョンで動作するはずですが、CodePlay は評価されていないプラットフォームでの正常な動作を保証するものではありません。
- 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。
- プラグインは、システムにインストールされている HIP のバージョンに依存します。HIP は Windows* と macOS* をサポートしていないため、これらのオペレーティング・システムでは ベータ版 oneAPI for AMD* GPU パッケージは利用できません。

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 2023.1.0 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 2023.1.0 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* 5.4.1 の場合、『ROCm* インストール・ガイド v5.4.1』(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. ベータ版 oneAPI for AMD* GPU のインストーラー (英語) をダウンロードします。
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-2023.1.0-rocm-5.4.1-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 2023.0.0 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。

```
[ext_oneapi_hip:gpu:0] AMD HIP BACKEND, AMD Radeon PRO W6800 0.0 [HIP 40421.43]
```

- 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<sycl::cl_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] = (sycl::cl_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
const auto HostAccessor = Buffer.get_access<sycl::access::mode::read>();

// 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。

   ```
   $ SYCL_DEVICE_FILTER=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: 11.15.1 ]
   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。

サポートされない機能

`icpx` コンパイラーは、次の機能をサポートしていません。

- CXX stdlib のサポート
- SYCL* グループ・アルゴリズム: `broadcast`、`joint_exclusive_scan`、`joint_inclusive_scan`、`exclusive_scan_over_group`、`inclusive_scan_over_group`
- デバイスコードの分割
- SYCL* リダクション

AMD* GPU バックエンド

これらの機能は、C++ の `clang++` ドライバーによってサポートされます。アプリケーションでこれらの機能を使用する場合、`icpx` の代わりに DPC++ `clang++` ドライバーを使用してください。`clang++` ドライバーは `$releaseDir/compiler/latest/linux/bin-llvm/clang++` にあります。

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 `SYCL_DEVICE_FILTER` 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 SYCL_DEVICE_FILTER=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。

注意: この環境変数は、今後のリリースで廃止される予定です。

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。

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。

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。

GPU 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。

NVIDIA* GPU をターゲットにする固有の最適化については、「NVIDIA* GPU 上のパフォーマンス」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 6) の 3.7.2 節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 6) の 3.7.2.1 節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 6) の 3.7.2.2 節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。

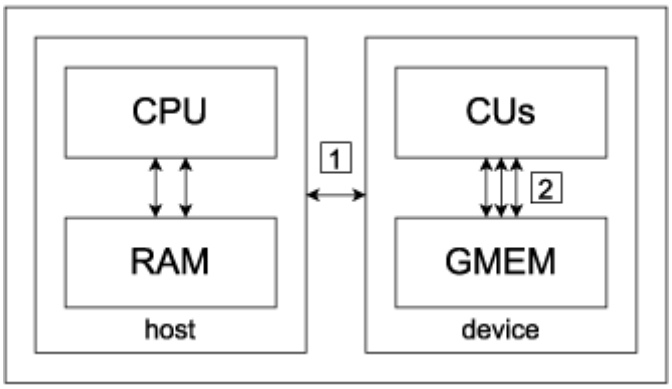

図 1

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。

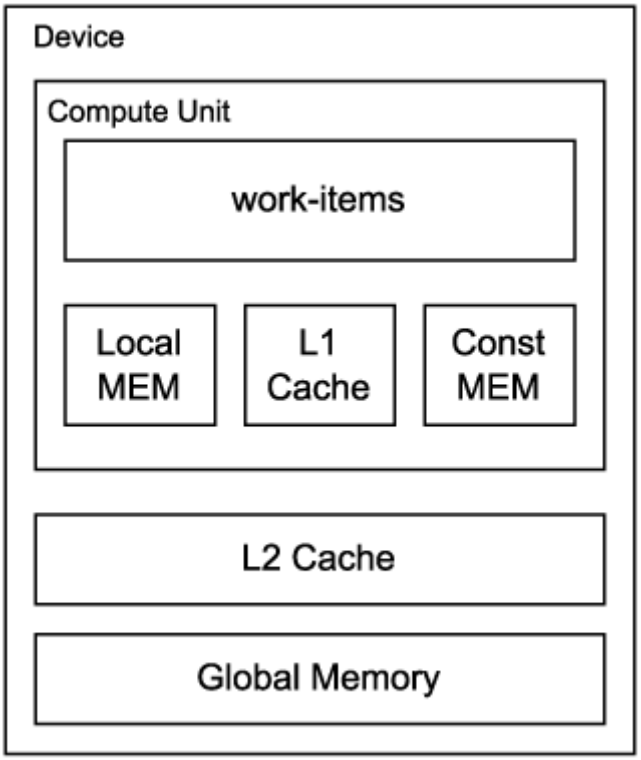

図 2

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。

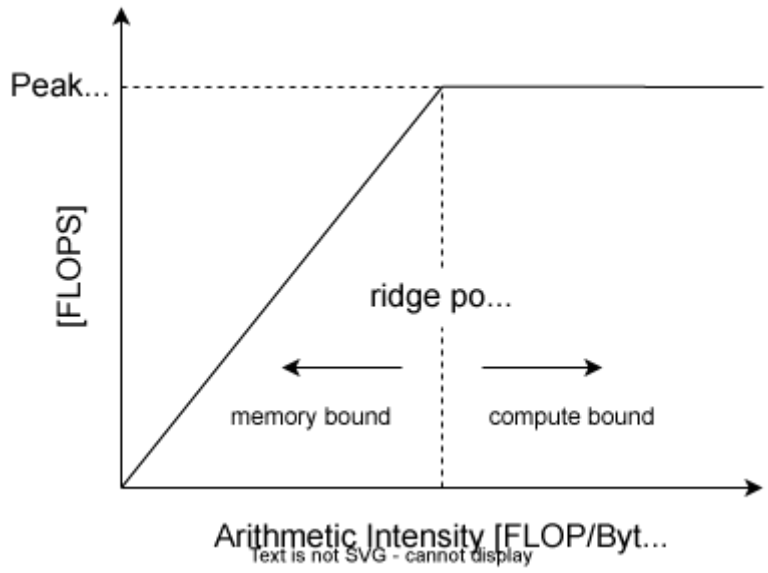

図 3

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。

NVIDIA* GPU の場合、NVIDIA* Nsight* Compute (英語) は占有計算機を提供しており、理論上の占有率がどのように計算されるか判断するのに役立ちます。非推奨ですが、NVIDIA の spreadsheet (英語) でも同様の機能を提供することを示しています。

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るているかを開発者に示すことから重要です。

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。

NVIDIA* GPU 上のパフォーマンス

アーキテクチャー

NVIDIA* には、長年にわたるさまざまな GPU アーキテクチャーがあります。ここでは、計算能力が 7.0 以上の GPU のみを対象にします。つまり、Volta*、Turing*、および Ampere* (さらに最近発表された Hopper* アーキテクチャー) を対象にします。

注意: ここでは、CUDA* と SYCL* 用語の簡単な対比を示しています。詳しい説明については、ComputeCpp の『SYCL* for CUDA* Developer』(英語) を参照してください。

NVIDIA* GPU の基本計算ユニットは、ストリーミング・マルチプロセッサーまたは SM と呼ばれます。SM は、32 個の work-item で構成される sub-group を実行します。NVIDIA では sub-group を ワープ (*warp*) と呼んでいます。work-item を実行するエンティティーはスレッド (*thread*) と呼ばれます。

work-group は、スレッドブロック (*thread block*) または協調スレッドアレイ (CTA: *cooperative thread array*) と呼ばれます。これには通常の規則が適用されます。CTA は同じ SM で同時に実行されることが保証され、SYCL* ローカルメモリー (CUDA* では共有メモリー) などのローカルリソースを利用でき、work-item 間で同期できます。

以下は、NVIDIA*/CUDA* と SYCL* の用語の対比表です。

NVIDIA*/CUDA*	SYCL*
ストリーミング・マルチプロセッサー (SM)	計算ユニット (CU)
ワープ	sub-group
スレッド	work-item
スレッドブロック	work-group
協調スレッドアレイ (CTA)	work-group

NVIDIA*/CUDA*	SYCL*
共有メモリー	ローカルメモリー
グローバルメモリー	デバイスメモリー
ローカルメモリー	プライベート・メモリー

work-group サイズは、sub-group サイズの 32 の倍数である必要があります。適切な work-group サイズは、占有率を最大化するように選択され、カーネルによって使用されるリソースに依存します。これは 1024 を越えることはありません。

NVIDIA* GPU では、デバイスクエリー `sycl::info::device::sub_group_sizes` が、値 32 の単一要素を持つベクトルを返します。

work-group サイズを選択したら、グローバルサイズを選択します。グローバルサイズを計算ユニット数の倍数にすると、負荷分散に役立つことがあります。デバイスクエリー `get_info<info::device::max_compute_units>` は、計算ユニットの数を返します。

次に、それぞれの work-item (カーネル・インスタンス) が問題の複数の項目で動作するようにカーネルコードを記述します。起動パラメーターは、問題サイズにかかわりなく、ハードウェアのレイアウトに基づいて調整できます。

例えば、次の SYCL* コードは、ハードウェア・レイアウトに基づいて起動パラメーターを決定し、カーネル内のループで問題サイズを調整します。CUDA* では、このタイプの内部ループはグリッドストライド (*grid-stride*) と呼ばれます。

```
int N = some_big_number;
int wgsize = 256;
int ncus = dev.get_info<info::device::max_compute_units>();
int nglobal = 32 * ncus;
cgh.parallel_for(nd_range<1>(nglobal * wgsize, wgsize),
  [=](nd_item<1> item)
  {
    int global_size = item.get_global_range()[0];
    for (int i = item.get_global_id(0); i < N; i += global_size)
      y[i] = a * x[i] + y[i];
  });
```

SM

NVIDIA* SM は、SM サブ・パーティション (英語) と呼ばれる 4 つの処理ブロックに分割されます。それぞれのワープは、存続期間全体で単一サブ・パーティションに存在します。ワープが停止すると、ハードウェア・ワープ・スケジューラーは準備ができている別のワープにスイッチできます。これは、いつでも実行できる十分な数のワープが必要であることを意味します。以降で、ハードウェア・メトリックを使用して、ワープ・スケジューラーの効率とストール時間を評価する方法を紹介します。

メモリー

メモリーにはいくつかの種類があります。グローバルメモリー (SYCL* デバイスメモリー) はデバイス上にありますが、一部のグローバルアドレスは、ホストまたは別のデバイス上にマップされたメモリー (SYCL* USM を利用) を参照する場合があります。すべてのグローバル・メモリー・アクセスは、CU ごとに L1 キャッシュと共有 GPU L2 キャッシュを経由します。

SYCL* プライベート・メモリー (NVIDIA* ローカルメモリー) は、特定の work-item からのみアクセスできますが、デバイスメモリーにマップされているため、デバイスメモリーと比較してパフォーマンス上の優位性はありません。work-item ごとに同じアドレスにマップされます。これは、work-item スタック、レジスタースピル、およびその他の work-item のローカルデータに使用されます。

SYCL* ローカルメモリー (NVIDIA* 共有メモリー) は work-group で使用できます。デバイスメモリーよりも帯域幅が広く、レイテンシーが短縮されます。SYCL* ローカルメモリーには 32 のバンクがあります。連続する 32 ビット・ワードは、異なるバンクに割り当てられます。

SYCL* ローカルメモリー内の 32 バイト float 配列にアクセスするストライド 1 のループを考えてみてください。最初の sub-group では、work-item 0 は配列要素 0 にアクセスし、work-item 1 は配列要素 1 にアクセスします。つなり、sub-group 全体として配列要素 0:31 にアクセスするため、バンク競合は起こりません。

一方、ループのストライドが 32 である場合、work-item は要素 32、64、96、... にアクセスし、sub-group は 32 要素離れた 32 の位置にアクセスするため、結果として 32 ウェイのバンク競合が発生します。これは、SYCL* ローカルメモリーの帯域幅を 1/32 に減少させます。

Volta のプレゼンテーション (英語) では、スライド 54 から 72 で共有メモリーバンクの競合について分かりやすく説明しています。

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。

異なる work-item は通常、デバイスメモリー内の異なる位置にアクセスします。同じロード命令内の特定の sub-group でアクセスされるアドレスが、キャッシュラインの同じセットである場合、メモリーシステムは最小数のデバイスメモリーへのアクセスを発行します。これはメモリー結合と呼ばれます。この要件は、隣接するメモリー位置にアクセスする work-item によって容易に満たすことができます。データ構造を 32 バイト境界に配置することで、パフォーマンスをさらに向上できます。間接アクセス/大きなストライドによって、明らかにこれをが困難であることがあります。

最近の NVIDIA* GPU では、洗練されたキャッシュとメモリーシステムを備えているため、デバイスのメモリーアクセスが結合される可能性は高くなります。結合の測定に有効なハードウェア・メトリックがあります。

キャッシュ

すべての GPU ユニットは L2 キャッシュを共有します。L2 キャッシュは物理アドレスでアクセスされます。これには、データ圧縮とグローバルアトミック (浮動小数点加算など) の機能も含まれます。

各 SM には、複数の機能に使用される L1 キャッシュがあります。L1 のスループットは、パフォーマンスを制限する要因になることがあります。

占有率

NVIDIA* では、同時にアクティブな CU 数を、利用可能な最大 CU 数で割ったものを占有率として定義しています。明らかに、占有率が上がると GPU 利用率が上がり、パフォーマンスが向上することが期待されます。理論上の占有率は、ハードウェアの制限、カーネルで使用されるレジスター数、およびカーネルで使用される共有メモリーの量によって決定されます。

理論上の占有率は、work-group サイズと特定のカーネルを起動するパラメーターを決定するガイドラインとなります。NVIDIA* プロファイリング・ツールは、起動されるカーネルごとの実際の占有率と理論上の占有率を示します。実行時に work-group サイズを変更して、結果をベンチマークして最適な work-group サイズを選択できます。しかし、コードを実行することなく理論的な占有率の推測を利用することもできます。

NVIDIA は、オンラインの spreadsheet (https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls) を提供しています。公式には非推奨ですが、NVIDIA* Nsight* Compute の占有率セクションと同じ機能を提供し、理論上の占有率がどのように決定されるかを理解するのに役立ちます。代わりに NVIDIA* Nsight* Compute (英語) を使用することが推奨されています。

パフォーマンス・ツール

一般に、NVIDIA* が CUDA* 向けに提供するすべてのパフォーマンス・ツールは、DPC++ CUDA* プラグインを使用して SYCL* アプリケーションをシームレスに処理します。

この節では、それらのパフォーマンス・ツールの一部を紹介します。

NVIDIA Nsight* Systems (nsys)

NVIDIA Nsight* Systems (英語) は、コマンドライン・ツール *nsys* と GUI *nsys-ui* を含んでいます。さらに、NVTX (英語) トレース・ライブラリーをアプリケーションで使用して、nsys 解析の対象領域を制限し、アプリケーション固有のイベントを nsys 解析に追加することができます。

基本的な使い方

```
nsys profile <command> <arguments>
```

このコマンドは、指定するコマンドを与える引数で実行し、デフォルト設定でプロファイルを行います。出力は、NVIDIA* 固有の形式でレポートファイルに書き込まれます。

nsys はシステム全体を監視することに注意してください。コマンドは何であっても構いませんが、nsys は単にコマンドを起動して監視を開始し、コマンドが終了するまで監視を続行します。

```
nsys stats <report file>
```

このコマンドは、レポートファイルから sqlite データベースを作成し、データベースでいくつかのレポートを作成します。レポートには、API の使用状況、GPU に関連するメモリーコピー、および GPU カーネルのタイミングなどが含まれます。結果はデフォルトでコンソールに出力されます。レポートを csv 形式のファイルに書き込むオプションもあります。

nsys-ui コマンドを使用して、レポートファイルを GUI で調査することもできます。これには、ローカルまたは VNC のようなリモート X ビューアーを介して接続されたディスプレイが必要です。レポートファイルは、ラップトップなどのローカルマシンに移動して、そこで GUI を実行できます。レポートファイルは非常に大きくなる可能性があり、大量のメモリーを必要とする場合があります。

アプリケーションで NVTX アノテーションを戦略的に使用し、コマンドライン引数を追加することで、レポートサイズを縮小し、解析ワークフローを円滑にすることができます。

レポートは JSON 形式でエクスポートすることもできます。これには、sqlite データベースと同じデータが含まれていますが、SQL を熟知していない開発者にも解析が容易です。

NVTX アノテーション

NVTX は、NVIDIA* が提供するインストルメント API で、パフォーマンス・ツールで利用されます。この API は、`nvToolsExt.h` をインクルードすることで利用できます。ライブラリーのリンクなどは必要ありません。

NVTX は、大規模なアプリケーションで、ほかの NVIDIA* ツールのデータ収集をアプリケーションの特定の部分に制限するのに役立ちます。

NVTX は、デバイスコードではなく、ホストコードの測定にのみ利用できることに注意してください。

NVTX の使い方と機能の詳細については、NVIDIA* のドキュメント (英語) を参照してください。

NVIDIA* Nsight* Compute (ncu)

前の節では、NVIDIA Nsight* Systems (nsys) について説明しました。NVIDIA* Nsight* Compute (ncu) は、GPU ハードウェアのパフォーマンスに注目する支援ツールです。ncu を使用すると、NVIDIA* GPU で利用可能なハードウェア・カウンターにアクセスできるだけでなく、特定のカーネルが GPU をどの程度活用しているかを理解するのに役立つ事前定義された解析タイプ (セクションと呼ばれます) を知ることができます。

ncu には GUI 版がありますが、ここでは nsys の説明のようにコマンドラインといくつかの処理スクリプトを利用します。ncu CLI については、https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html (英語) で説明されています。

ncu オーバーヘッド

ncu は、アプリケーションの実行時間を大幅に増加させる可能性があります。いくつかの原因が考えられます。

- 指定されたメトリックを収集するため、場合によってはカーネルを複数回実行する必要があります。
- 一部のメトリックでは、オーバーヘッドが高いカーネルのバイナリー・インストルメンテーションが必要になります。
- 通常、カーネルの実行はシリアル化されるため、同時実行性が低下します。

詳細については、「カーネル・プロファイル・ガイド」(英語) を参照してください。

速度が低下すると、通常、収集プロセスを制限する必要があります。NVIDIA* では、これに対処するいくつかの方法を提案しています。

- nsys の節で説明したように、NVTX を使用してアプリケーションをインストルメントし、ncu にコマンドライン・オプション `--nvtx` および `--nvtx-include` または `--nvtx-exclude` を指定して、特定の NVTX レンジを含めるか除外するかを指示します。ドメイン内のレンジを指定する構文は、nsys とは逆であることに注意してください。`range@domain` ではなく `domain@range` となります。ncu の構文は非常に豊富ですが複雑です。「NVTX フィルター処理」(英語) のセクションを参照してください。
- `-k kernelname` を指定して、単一カーネルのデータのみを収集します。カーネル名には正規表現を利用できます。

  注意: C++ ユーザーは、`--kernel-name-base=mangled` を指定してマングルされた名前を使用することもできます。

- `--launch-count` と `--launch-skip` を使用して、特定数のカーネルの起動を収集します。

上記は組み合わせて使用できます。

セクション

ncu には、セクションと呼ばれる事前定義された多数のメトリックのセットが用意されています。各セクションは、特定のパフォーマンスに関する疑問を解決するのを支援します (例: アプリケーションはメモリー依存であるか、など)。

セクションは、`ncu --list-sections` で一覧を表示できます。ncu バージョン 2022.1.1.0 の 出力を以下に示します。

識別子	表記名
ComputeWorkloadAnalysis	Compute Workload Analysis
InstructionStats	Instruction Statistics
LaunchStats	Launch Statistics
MemoryWorkloadAnalysis	Memory Workload Analysis
MemoryWorkloadAnalysis_Chart	Memory Workload Analysis Chart
MemoryWorkloadAnalysis_Deprecated	(Deprecated) Memory Workload Analysis
MemoryWorkloadAnalysis_Tables	Memory Workload Analysis Tables
Nvlink	NVLink
Nvlink_Tables	NVLink Tables
Nvlink_Topology	NVLink Topology
Occupancy	Occupancy
SchedulerStats	Scheduler Statistics

識別子	表記名
SourceCounters	Source Counters
SpeedOfLight	GPU Speed Of Light Throughput
SpeedOfLight_HierarchicalDoubleRooflineChart	GPU Speed Of Light Hierarchical Roofline Chart (Double Precision)
SpeedOfLight_HierarchicalHalfRooflineChart	GPU Speed Of Light Hierarchical Roofline Chart (Half Precision)
SpeedOfLight_HierarchicalSingleRooflineChart	GPU Speed Of Light Hierarchical Roofline Chart (Single Precision)
SpeedOfLight_HierarchicalTensorRooflineChart	GPU Speed Of Light Hierarchical Roofline Chart (Tensor Core)
SpeedOfLight_RooflineChart	GPU Speed Of Light Roofline Chart
WarpStateStats	Warp State Statistics

メトリック

セクションの代わりに、`--metrics` を使用して特定のメトリックを収集するよう ncu に指示できます。利用可能なメトリックは、`--query-metrics` で照会できます。

出力

この節の例では、解析が容易な出力を生成するため、`--csv` オプションを使用しています。`--log-file` オプションは、この出力と他の出力をともに `stdout` の代わりにファイルへ送ります。

プロファイル・レポートは、ncu GUI によって使用されます。これは、`--export` オプションで保存できます。その後、ファイルを別のマシンに移動して ncu-gui を使用し、ローカルで表示できます。

分断された名前は、`--print-kernel-base=mangled` で選択できます。

カーネル・アセンブリーの抽出

状況によっては、特定のカーネルのパフォーマンスを理解するため、アセンブリーを調べることが有効な場合があります。NVIDIA* の場合は、コンパイラーがカーネル向けに生成した PTX を調査します。

NVIDIA* GPU 用にビルドされた SYCL* アプリケーションから PTX を抽出するには、環境変数 `SYCL_DUMP_IMAGES` に `1` を設定してアプリケーションを実行します。これにより、現在の作業ディレクトリーに `sycl_nvptx641.bin` のような名前のファイルが生成されます。これらのファイルは CUDA* **fat バイナリー**であり、NVIDIA* ターゲットに依存しない仮想アセンブリー言語である PTX と、単一または複数のターゲットのマシンコードである SASS が含まれます。

次のように CUDA* ツール `cuobjdump` (英語) を使用して、FAT バイナリーから PTX と SASS の両方を抽出できます。

```
# Extract PTX
cuobjdump --dump-ptx sycl_nvptx641.bin

# Extract SASS
cuobjdump -sass sycl_nvptx641.bin
```

NVIDIA* Nsight* Compute ツールの GUI バ—ションでも逆アセンブリーを表示できます。

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), sycl::range(WI_X)}, ...)

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

cgh.parallel_for(sycl::nd_range<3>{sycl::range<3>(WG_Z, WG_Y, WG_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

場合によっては、DPC++ がインライン展開に対し保守的になり、パフォーマンスが低下することもあります。

これが発生する一般的なケースは、カーネルラムダがカーネル実装を含む大きな関数を単純に呼び出すような SYCL* アプリケーションの場合です。

この問題を回避するには、特定の関数に `always_inline` 属性を追加して、インライン展開を強制します。以下に例を示します。

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

...

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

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精度が高すぎて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。

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( ... ) {
 ...
}
```

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];
        });
});
```

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)	部分的 [1]
グループ関数/アルゴリズム	部分的 [1]
整数関数	はい
数学関数 (スカラー)	はい
数学関数 (ベクトル)	部分的 [1]
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	いいえ
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	部分的 [1]
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 [2]
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	いいえ

非コア機能

機能	サポート
image	いいえ
fp16 データタイプ	いいえ
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	いいえ
USM アトミック共有割り当て	いいえ
USM システムに割り当て	はい
SYCL_EXTERNAL	いいえ
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 [3]
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	一部分
結合行列	いいえ
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	はい
明示的な SIMD	いいえ
discard_queue_events	部分的 [1]
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	いいえ

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

1 (1、2、3) 一部のテストで失敗

2 numeric_limits<size_t>::max() の使用

3 https://github.com/intel/llvm/pull/6289 (英語) に追加 (未テスト)

更新履歴

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バージョンであることを確認してください。

問題は、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 ]
   [ext_oneapi_cuda:gpu:0] NVIDIA CUDA BACKEND, NVIDIA A100-PCIE-40GB 0.0
   [CUDA 11.7]
   ```

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/2023.0.0/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. `SYCL_DEVICE_FILTER` 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 (SYCL_DEVICE_FILTER が設定されていると、`sycl-ls` は警告を表示します)。

不正バイナリーエラーの扱い

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いる場合、環境変数 `SYCL_DEVICE_FILTER` 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。

   注意: `SYCL_DEVICE_FILTER` 環境変数は、今後のリリースで廃止される予定です。

コンパイラーのエラー:「Do not know how to split the result of this operator! (このオペレーターの結果を分割する方法が不明です)」

`icpx` コンパイラーは、`-fsycl-device-code-split=per_kernel` オプションをサポートしていません。デバイスコードの分割を行う場合は、DPC++ `clang++` コンパイラー・ドライバーを使用してください。

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

コードの実行中にハングアップする

次のグループ・アルゴリズムを `icpx` コンパイラーでコンパイルすると、コマンドの実行中に CUDA* GPU または AMD* GPU がバックエンドでハングアップします。

- `broadcast`
- `joint_exclusive_scan`
- `joint_inclusive_scan`

グループ・アルゴリズムを使用する場合は、DPC++ `clang++` コンパイラー・ドライバーを使用する必要があります。

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

グループ・アルゴリズムの結果が不正である

`icpx` コンパイラー・ドライバーは、CUDA* GPU または AMD* GPU バックエンドで次のグループ・アルゴリズムに対し不正な結果を返します。

- `exclusive_scan_over_group`
- `inclusive_scan_over_group`

CUDA* GPU または AMD GPU バックエンドで上記のアルゴリズムを使用する場合、DPC++ `clang++` コンパイラー・ドライバーを使用してください。

詳細は、「ベータ版 oneAPI for AMD* GPU のインストール」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. `icpx` コンパイラーは、デバイスコード内で `std::cos`、`logf`、`sinf` などの CXX stdlib 関数呼び出しをサポートしていません。カーネル内で CXX stdlib 関数を使用する場合、`clang++` コンパイラー・ドライバーを使用してください。

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

oneAPI for AMD* GPU 使用許諾契約書

重要 - ソフトウェアを複製、インストール、または使用する前に使用許諾契約書 (英語) をお読みになり、同意する必要があります。