oneAPI DPC++ 導入ガイド

このドキュメントは、インテル社の GitHub* で公開されている「Getting Started with oneAPI DPC++」の日本語参考訳です。

DPC++ コンパイラーは、CPU と GPU や FPGA などさまざまなコンピューティング・アクセラレーターの両方のコードを使用する C++ と SYCL* ソースファイルをコンパイルします。

目次

要件

- `git` - ダウンロード(英語)
- `cmake` バージョン 3.14 以降 - ダウンロード(英語)
- `python` - ダウンロード(英語)
- `ninja` - ダウンロード(英語)

- C++ コンパイラー
  - Linux*:GCC バージョン 7.1.0 以降(libstdc++ を含む) - ダウンロード(英語)
  - Windows*:Visual Studio* バージョン 15.7 プレビュー 4 以降 - ダウンロード(英語)

DPC++ ワークスペースの作成

このドキュメントを通して、DPCPP_HOME は、DPC++ ワークスペースとして作成されたローカル・ディレクトリーへのパスを指します。同じ名前の環境変数を使用すると分かりやすいかもしれません。

Linux*:

```
export DPCPP_HOME=~/sycl_workspace
mkdir $DPCPP_HOME
cd $DPCPP_HOME

git clone https://github.com/intel/llvm -b sycl
```

Windows*(64 ビット):

次のいずれかの方法でコマンドプロンプトを開きます。

- [スタート] メニューをクリックして、[Visual Studio* XXXX] > [**x64** Native Tools Command Prompt for VS XXXX] を検索します。XXXX はインストールされている Visual Studio* のバージョンです。
- Windows キー + R を押して、"cmd" を入力して [OK] をクリックし、"C:¥Program Files (x86)¥Microsoft Visual Studio¥2017¥Community¥VC¥Auxiliary¥Build¥vcvarsall.bat" x64 を実行します。

```
set DPCPP_HOME=%USERPROFILE%¥sycl_workspace
mkdir %DPCPP_HOME%
cd %DPCPP_HOME%

git clone --config core.autocrlf=false https://github.com/intel/llvm -b sycl
```

DPC++ ツールチェーンのビルド

最も簡単な方法は、buildbot configure(英語)と compile(英語)スクリプトを使用する方法です。

CMake を手動で直接設定する場合、変数の最新のリファレンスはこれらのファイルにあります。

Linux*:

```
python $DPCPP_HOME/llvm/buildbot/configure.py
python $DPCPP_HOME/llvm/buildbot/compile.py
```

Windows*(64 ビット):

```
python %DPCPP_HOME%¥llvm¥buildbot¥configure.py
python %DPCPP_HOME%¥llvm¥buildbot¥compile.py
```

`configure.py` で次のフラグを利用できます(利用可能なフラグのリストは `--help` を指定してスクリプトを起動すると表示されます)。

- `--system-ocl` -> Cmake 経由で OpenCL* ヘッダーとライブラリーをダウンロードせずに、システムにあるものを使用します
- `--no-werror` -> llvm をコンパイルする際に警告をエラーとして扱いません
- `--cuda` -> cuda* バックエンドを使用します(NVIDIA CUDA*(英語)を参照)
- `--shared-libs` -> 共有ライブラリーをビルドします
- `-t` -> ビルドタイプ(debug または release)
- `-o` -> ビルド・ディレクトリーへのパス
- `--cmake-gen` -> ビルドシステムのタイプを設定(例:`--cmake-gen "Unix Makefiles"`)

注:フラグに関するデータは `configure.py` と `compile.py` スクリプト間で共有されません。`-o` フラグを使用してデフォルト・ディレクトリー以外にビルドを配置する場合、`compile.py` オプションでこのフラグと同じパスを指定する必要があります。これにより、例えば複数の異なるビルドを構成して、必要なビルドだけを行うことができます。

libc++ ライブラリーを使用して DPC++ ツールチェーンをビルド

DPC++ のランタイムをビルドして、libstdc++ の代わりに libc++ ライブラリーとリンクする実験的なサポートもあります。有効にするには、次の CMake オプションを使用する必要があります。

Linux*:

```
-DSYCL_USE_LIBCXX=ON ¥
-DSYCL_LIBCXX_INCLUDE_PATH=<path to libc++ headers> ¥
-DSYCL_LIBCXX_LIBRARY_PATH=<path to libc++ and libc++abi libraries>
```

configure スクリプトを使用して有効にすることもできます。

```
python %DPCPP_HOME%¥llvm¥buildbot¥configure.py --use-libcxx ¥
--libcxx-include <path to libc++ headers> ¥
--libcxx-library <path to libc++ and libc++ abi libraries>
python %DPCPP_HOME%¥llvm¥buildbot¥compile.py
```

NVIDIA CUDA* をサポートする DPC++ ツールチェーンをビルド

CUDA* デバイス向けの実験的なサポートがあります。

CUDA* デバイスのサポートを有効にする方法は、Linux* DPC++ ツールチェーンの手順に順じますが、`configure.py` に `--cuda` を追加する必要があります。

このフラグを有効にするには、システムに CUDA* 10.2 (英語) がインストールされている必要があります。『Linux* 向けの NVIDIA CUDA* インストールガイド』(英語) を参照してください。

現時点でテストされている組み合わせは、Titan RTX* GPU (SM 71) を使用する CUDA* 10.2 と Ubuntu* 18.04 だけですが、SM 50 以降と互換性のあるすべての GPU で動作すると思われます。NVIDIA CUDA* バックエンドのデフォルト SM は 5.0 です。これよりも小さな値を指定できますが、一部の機能がサポートされない可能性があります。

Doxygen ドキュメントのビルド

Doxygen ドキュメントのビルドは、製品自体のビルドに似ています。最初に次のツールをインストールする必要があります。

- doxygen
- graphviz

次に、CMake 設定コマンドに次のオプションを追加します。

```
-DLLVM_ENABLE_DOXYGEN=ON
```

CMake キャッシュが生成されたら、`doxygen-sycl` ターゲットを使用してドキュメントを作成します。ドキュメントは、`$DPCPP_HOME/llvm/build/tools/sycl/doc/html` ディレクトリーに配置されます。

展開

TODO: ビルドされた DPC++ ツールチェーンを展開する方法の説明を追加予定。

DPC++ ツールチェーンを使用する

低レベルランタイムのインストール

OpenCL* デバイスで DPC++ アプリケーションを実行するには、OpenCL* 実装がシステムに存在する必要があります。

レベルゼロデバイスで DPC++ アプリケーションを実行するには、レベルゼロ実装がシステムに存在する必要があります。レベルゼロ仕様のリンクは、以降のセッション「関連情報」にあります。

インテル® GPU またはインテル® CPU デバイスで DPC++ アプリケーションを実行するのに必要な GPU 向けのレベルゼロ RT と OpenCL* RT、CPU 向けの OpenCL* RT、FPGA エミュレーション RT、および TBB ランタイムは、依存関係設定ファイル(英語)からダウンロードして、次の手順に従ってインストールできます。同じバージョンの PR テストで使用されます。

Linux*:

1. アーカイブを展開します。例えば、アーカイブ `oclcpuexp_<cpu_version>.tar.gz` および `fpgaemu_<fpga_version>.tar.gz` の場合、次のコマンドを実行します。

```
# OpenCL* FPGA エミュレーション RT を展開します
mkdir -p /opt/intel/oclfpgaemu_<fpga_version>
cd /opt/intel/oclfpgaemu_<fpga_version>
tar zxvf fpgaemu_<fpga_version>.tar.gz
# OpenCL* CPU RT を展開します
mkdir -p /opt/intel/oclcpuexp_<cpu_version>
cd /opt/intel/oclcpuexp_<cpu_version>
tar -zxvf oclcpu_rt_<cpu_version>.tar.gz
```

2. 新しいランタイムを示す ICD ファイルを作成します(root 権限が必要です)。

```
# OpenCL* FPGA エミュレーション RT
echo /opt/intel/oclfpgaemu_<fpga_version>/x64/libintelocl_emu.so >
  /etc/OpenCL/vendors/intel_fpgaemu.icd
# OpenCL* CPU RT
echo /opt/intel/oclcpuexp_<cpu_version>/x64/libintelocl.so >
 /etc/OpenCL/vendors/intel_expcpu.icd
```

3. 依存関係設定ファイル(英語)のリンクを使用して TBB ライブラリーを展開もしくはビルドします。例えば、アーカイブ oneapi-tbb-<tbb_version>-lin.tgz の場合、次の操作を行います。

```
mkdir -p /opt/intel
cd /opt/intel
tar -zxvf oneapi-tbb*lin.tgz
```

4. OpenCL* RT フォルダー内の TBB ライブラリーからファイルをコピーするか、TBB ライブラリーへのシンボリック・リンクを作成します。

```
# OpenCL* FPGA エミュレーション RT
ln -s /opt/intel/oneapi-tbb-<tbb_version>/lib/intel64/gcc4.8/libtbb.so
  /opt/intel/oclfpgaemu_<fpga_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so
  /opt/intel/oclfpgaemu_<fpga_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbb.so.12
  /opt/intel/oclfpgaemu_<fpga_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so.2
  /opt/intel/oclfpgaemu_<fpga_version>/x64
# OpenCL* CPU RT
ln -s /opt/intel/oneapi-tbb-<tbb_version>/lib/intel64/gcc4.8/libtbb.so
  /opt/intel/oclcpuexp_<cpu_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so
  /opt/intel/oclcpuexp_<cpu_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbb.so.12
  /opt/intel/oclcpuexp_<cpu_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so.2
          /opt/intel/oclcpuexp_<cpu_version>/x64
```

5. ライブラリーのパスを設定します(root 権限が必要です)。

```
echo /opt/intel/oclfpgaemu_<fpga_version>/x64 >
  /etc/ld.so.conf.d/libintelopenclexp.conf
echo /opt/intel/oclcpuexp_<cpu_version>/x64 >>
  /etc/ld.so.conf.d/libintelopenclexp.conf
ldconfig -f /etc/ld.so.conf.d/libintelopenclexp.conf
```

Windows*(64 ビット):

1. インテル® GPU 向けの OpenCL* ランタイムが必要な場合、最初に更新またはインストールします。インテル® CPU 向けの OpenCL* ランタイムをインストールする**前に**、インテル® GPU 向けの OpenCL* ランタイムをインストールしてください。これは、インテル® GPU の OpenCL* ランタイムがいくつかの重要なファイルや設定を置き換えてインテル® CPU 用の OpenCL* ランタイムが正しく機能しなくなる可能性があるためです。
2. 依存関係設定ファイル(英語)内のリンクを使用して、インテル® CPU やインテル® FPGA エミュレーション用の OpenCL* ランタイムのアーカイブを展開します。例えば、`c:¥oclcpu_rt_<cpu_version>` に展開します。
3. TBB ランタイムのアーカイブを展開するか、依存関係設定ファイル(英語)のリンクを使用してソースからビルドします。例えば、`c:¥oneapi-tbb-<tbb_version>` に展開します。
4. 管理者権限でコマンドプロンプトを実行します。これには [スタート] ボタンを右クリックし、リストから [コマンドプロンプト(管理者)] を選択して、[はい] をクリックします。
5. 開いたウィンドウで、展開したファイルのディレクトリーから `install.bat` を実行して、システムにランタイムをインストールし、環境変数を設定します。展開されたファイルが `c:¥oclcpu_rt_<cpu_version>¥` フォルダーにある場合、次のコマンドを入力します。

```
# OpenCL* FPGA エミュレーション RT をインストールします
# 問いに [N] を入力して、以前の OCL_ICD_FILENAMES 設定をクリーンアップします
c:¥oclfpga_rt_<fpga_version>¥install.bat c:¥oneapi-tbb-
<tbb_version>¥redist¥intel64¥vc14
# OpenCL* CPU RT をインストールします
# 問いに [Y] を入力して、FPGA RT と CPU RT を相互に設定します
c:¥oclcpu_rt_<cpu_version>¥install.bat c:¥oneapi-tbb-
<tbb_version>¥redist¥intel64¥vc14
```

事前(AOT)コンパイルの前提条件を取得

事前コンパイル(英語)では、`PATH` に事前コンパイル用のコンパイラーのパスが設定されている必要があります。デバイスごとに AOT コンパイラーがあります。

- `GPU`:レベルゼロおよび OpenCL* ランタイムがサポートされます。
- `CPU`:OpenCL* ランタイムがサポートされます。
- `Accelerator`(FPGA または FPGA エミュレーション):OpenCL* ランタイムがサポートされます。

GPU

- Linux*

  GPU AOT コンパイラーの `ocloc` を取得するには、2 つの方法があります。

  - (Ubuntu*) intel/compute-runtime releases (英語) から `intel-ocloc_***.deb` パッケージをダウンロードしてインストールします。このパッケージは、システムにインストールされているレベルゼロ/OpenCL* GPU ランタイムと同じバージョンである必要があります。
  - (他のディストリビューション) `ocloc` は、汎用 GPU 向けのインテル® ソフトウェア・パッケージ (英語) に含まれます。

- Windows*

  - GPU AOT コンパイラー `ocloc` は、インテル® oneAPI ベース・ツールキット (英語) のインテル® oneAPI DPC++/C++ コンパイラー・コンポーネントに含まれます。
    `ocloc` バイナリーへのパスが `PATH` 環境変数に設定されていることを確認してください。
    - `<oneAPI インストール場所>/compiler/<バージョン>/windows/lib/ocloc`

CPU

- CPU AOT コンパイラー `opencl-aot` はデフォルトで使用できます。詳細は opencl-aot documentation (英語) をご覧ください。

アクセラレーター

- アクセラレーター AOT コンパイラー `aoc` は、インテル® oneAPI ベース・ツールキット (英語) のインテル® oneAPI DPC++/C++ コンパイラー・コンポーネントに含まれます。
  これらのバイナリーへのパスが `PATH` 環境変数に設定されていることを確認してください。
  - `aoc` は `<oneAPI installation location>/compiler/<version>/<OS>/lib/oclfpga/bin`
  - `aocl-ioc64` は `<oneAPI installation location>/compiler/<version>/<OS>/bin`

DPC++ ツールチェーンのテスト

ツリー内 LIT テストを実行

ビルドされた DPC++ ツールチェーンが正しく動作していることを確認するには、次のコマンドを実行します。

Linux*:

```
python $DPCPP_HOME/llvm/buildbot/check.py
```

Windows*(64 ビット):

```
python %DPCPP_HOME%¥llvm¥buildbot¥check.py
```

OpenCL* GPU/CPU ランタイムが利用できない場合、それらのテストはスキップされます。

CUDA* サポートがビルドされている場合、CUDA* デバイスが利用可能である時にのみテストが行われます。

DPC++ E2E テストスイートの実行

README(英語)の手順に従ってテストをビルドして実行します

Khronos SYCL* 適合性テストスイートの実行(オプション)

Khronos SYCL* 適合性テスト(CTS)は、Khronos SYCL* 仕様への実装の適合性を検証することを目的としています。DPC++ コンパイラーは、相当数のテストに合格することが期待されていますが、改善が続けられています。

README(英語)ファイルの Khronos SYCL* CTS の指示に従って、テストのソース、およびテストをビルドおよび実行する手順を取得します。

DPC++ ツールチェーンをビルドするには、`SYCL_IMPLEMENTATION=Intel_SYCL` と `Intel_SYCL_ROOT=<SYCL*` インストールへのパス> の CMake 環境変数を設定します。

Linux*:

```
cmake -DIntel_SYCL_ROOT=$DPCPP_HOME/deploy -DSYCL_IMPLEMENTATION=Intel_SYCL ...
```

Windows*(64 ビット):

```
cmake -DIntel_SYCL_ROOT=%DPCPP_HOME%¥deploy -DSYCL_IMPLEMENTATION=Intel_SYCL ...
```

簡単な DPC++ アプリケーションを実行

簡単な DPC++ または SYCL* プログラムは、次のように構成されます。

1. ヘッダーセクション
2. データのバッファー割り当て
3. SYCL* キューの作成
4. SYCL* キューにカーネルを含むコマンドグループを送信
5. キューのワークが完了するのを待機
6. バッファーアクセサーを使用してデバイスで結果を取得してデータを検証
7. 終了

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 <CL/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
  cl::sycl::buffer<cl::sycl::cl_int, 1> Buffer(4);

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

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

  // キューへコマンドグループ (ワーク) を送信
  Queue.submit([&](cl::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<cl::sycl::access::mode::write>(cgh);
    // カーネルを実行
    cgh.parallel_for<class FillBuffer>(
        NumOfWorkItems, [=](cl::sycl::id<1> WIid) {
          // インデックスでバッファーを埋める
          Accessor[WIid] = (cl::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ー
  const auto HostAccessor = Buffer.get_access<cl::sycl::access::mode::read>();

  // 結果をチェック
  bool MismatchFound = false;
  for (size_t I = 0; I < Buffer.get_count(); ++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;
}
```

simple-sycl-app をビルドするには、bin と lib へのパスを PATH に設定します。

Linux*:

```
export PATH=$DPCPP_HOME/llvm/build/bin:$PATH
export LD_LIBRARY_PATH=$DPCPP_HOME/llvm/build/lib:$LD_LIBRARY_PATH
```

Windows*(64 ビット):

```
set PATH=%DPCPP_HOME%¥llvm¥build¥bin;%PATH%
set LIB=%DPCPP_HOME%¥llvm¥build¥lib;%LIB%
```

次に以下のコマンドを実行します。

```
clang++ -fsycl simple-sycl-app.cpp -o simple-sycl-app.exe
```

CUDA* 向けのビルドを行う場合、次のように CUDA* ターゲットトリプルを使用します。

```
clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice ¥
  simple-sycl-app.cpp -o simple-sycl-app-cuda.exe
```

GPU、CPU、またはアクセラレーター・デバイス向けに `simple-sycl-app` の事前コンパイルを行うには、ターゲット・アーキテクチャーを指定します。

`-fsycl-targets=spir64_gen-unknown-unknown-sycldevice`(GPU の場合)
`-fsycl-targets=spir64_x86_64-unknown-unknown-sycldevice`(CPU の場合)
`-fsycl-targets=spir64_fpga-unknown-unknown-sycldevice`(アクセラレーターの場合)

複数のターゲット・アーキテクチャーがサポートされます。

例えば、次のコマンドは事前コンパイルモードで、GPU および CPU デバイス向けに `simple-sycl-app` をビルドします。

```
clang++ -fsycl -fsycl-targets=spir64_gen-unknown-unknown-sycldevice,spir64_x86_
64-unknown-unknown-sycldevice simple-sycl-app.cpp -o simple-sycl-app-aot.exe
```

また、ユーザーは `-Xsycl-target-backend` オプションを使用して、AOT コンパイラーの特定のオプションを DPC++ コンパイラーに渡すことができます。詳細は、デバイス・コード・フォーマット(英語)を参照してください。利用可能なオプションを見るには次のコマンドを実行します。

GPU 向け `ocloc compile --help`
CPU 向け `opencl-aot --help`
アクセラレーター向け `aoc -help -sycl`

`simple-sycl-app.exe` アプリケーションは、実行する SYCL* デバイスを指定しないため、SYCL* ランタイムが `default_selector` ロジックを使用して、システムまたは SYCL* ホストデバイスで利用可能なアクセラレーターの 1 つを選択します。この場合、`default_selector` の動作は `SYCL_BE` 環境変数で変更できます。`PI_CUDA` を設定すると、CUDA* バックエンドが強制的に選択されます(利用可能であれば)。`PI_OPENCL` は、OpenCL* バックエンドを使用することを強制します。

```
SYCL_BE=PI_CUDA ./simple-sycl-app-cuda.exe
```

デフォルトは、利用可能であれば OpenCL* バックエンドを使用します。OpenCL* または CUDA* デバイスが利用できない場合、SYCL* ホストデバイスが使用されます。SYCL* ホストデバイスは、低レベルの API を使用することなく、ホスト内で SYCL* アプリケーションを直接実行します。

注: clang が cmake オプション `SYCL_BUILD_PI_CUDA=ON` でビルドされている場合、`-fsycl-targets` オプションで `nvptx64-nvidia-cuda-sycldevice` が使用できます

Linux* & Windows*(64 ビット):

```
./simple-sycl-app.exe
The results are correct!
```

注: 現在、アプリケーションが CUDA* ターゲットでビルドされる場合、実行時に `SYCL_BE` 環境変数で CUDA* ターゲットを選択する必要があります。

```
SYCL_BE=PI_CUDA ./simple-sycl-app-cuda.exe
```

注: DPC++/SYCL* 開発者は、次のセクション「特定の GPU 向けにプログラムを記述する」のように、デバイスセレクター(例: `cl::sycl::cpu_selector`、`cl::sycl::gpu_selector`、インテル FPGA セレクター(英語))を使用して実行する SYCL* デバイスを指定できます。

特定の GPU 向けにプログラムを記述

OpenCL* デバイスを指定するため、SYCL* は `cl::sycl::device_selector` クラスを提供します。これにより、ランタイムが最適なデバイスを選択する方法を定義できます。

SYCL* `cl::sycl::device_selector` のメソッド `cl::sycl::device_selector::operator()` は、SYCL* デバイスへの参照を取得して、整数値のスコアを返す抽象化メンバー関数です。この抽象化メンバー関数を派生クラスに実装して、SYCL* デバイスを選択するロジックを提供できます。SYCL* ランタイムは、最も高いスコアを返したデバイスを選択します。このオブジェクトは、`cl::sycl::queue` と `cl::sycl::device` コンストラクターに渡すことができます。

次の例は、`cl::sycl::device_selector` を使用して、インテル® GPU デバイスにバインドされたデバイスとキュー・オブジェクトを作成する方法を示します。

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

int main() {
  class NEOGPUDeviceSelector : public cl::sycl::device_selector {
  public:
    int operator()(const cl::sycl::device &Device) const override {
      using namespace cl::sycl::info;

      const std::string DeviceName = Device.get_info<device::name>();
      const std::string DeviceVendor = Device.get_info<device::vendor>();

      return Device.is_gpu() && (DeviceName.find("HD Graphics NEO") !=
std::string::npos);
```

```
    }
  };

  NEOGPUDeviceSelector Selector;
  try {
    cl::sycl::queue Queue(Selector);
    cl::sycl::device Device(Selector);
  } catch (cl::sycl::invalid_parameter_error &E) {
    std::cout << E.what() << std::endl;
  }
}
```

次のデバイスセレクターは、NVIDIA デバイスのみを選択し、デバイスが検出されないと実行されません。

```
class CUDASelector : public cl::sycl::device_selector {
  public:
    int operator()(const cl::sycl::device &Device) const override {
      using namespace cl::sycl::info;
      const std::string DriverVersion = Device.get_info<device::driver_version>();

      if (Device.is_gpu() && (DriverVersion.find("CUDA") != std::string::npos))
{
        std::cout << " CUDA device found " << std::endl;
        return 1;
      };
      return -1;
    }
};
```

CUDA* プラットフォームで DPC++ ツールチェーンを使用する

CUDA* プラットフォームでの DPC++ ツールチェーンのサポートは、まだ実験段階です。現在、DPC++ ツールチェーンは、アプリケーションを DPC++ ランタイムとリンクするため、システムに最新の OpenCL* 実装があることに依存しています。アプリケーションが CUDA* バックエンドのみを使用する場合、OpenCL* 実装は実行時に使用されませんが、インストールされている必要があります。

CUDA* SDK が提供する OpenCL* 実装は OpenCL* 1.2 です。これは、DPC++ ランタイムとリンクするには古すぎるため、いくつかのシンボルが不足しています。

次のセクションの手順に従って、低レベルの CPU ランタイムをインストールすることを推奨します。

低レベルの CPU ランタイムをインストールする代わりに、必要なすべてのシンボルを含む Khronos ICD loader (英語)をビルドしてインストールできます。

C++ 標準

- DPC++ ランタイムとヘッダーには C++14 以降が必要です。
- DPC++ コンパイラーは、デフォルトで C++17 アプリとしてアプリケーションをビルドします。

既知の問題と制限事項

- 同一カーネルが異なる変換ユニットで使用されると、DPC++ デバイス・コンパイラーはコンパイルに失敗します。
- SYCL* ホストデバイスのサポートは完全ではありません。
- SYCL* 2020 のサポートは作業中です。
- 32 ビットのホスト/ターゲットはサポートされません。
- DPC++ は、アウトオブオーダー・キューをサポートする OpenCL* 低レベルランタイムでのみ機能します。
- Windows* では、DPC++ アプリケーションを `/MTd` フラグを使用してリンクするとクラッシュする問題が報告されています。

CUDA* バックエンドの制限

- バックエンドは Linux* のみでサポートされます。
- テストされている組み合わせは、Titan RTX* GPU (SM 71)を使用する CUDA* 10.2 と Ubuntu* 18.04 だけですが、SM 50 以降と互換性のあるすべての GPU で動作すると思われます
- NVIDIA OpenCL* ヘッダーは、このプロジェクトに必要な OpenCL* ヘッダーを競合し、一部のプラットフォームではコンパイルの問題を引き起こす可能性があります。

関連情報

- DPC++ 仕様: https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html (英語)
- SYCL* 1.2.1 仕様: www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf (英語)
- oneAPI ゼロレベル仕様: https://spec.oneapi.com/versions/latest/oneL0/index.html (英語)