oneAPI for NVIDIA* GPU 2023.0.0 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 1 月 19 日時点の『oneAPI for NVIDIA® GPUs 2023.0.0』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。

oneAPI for NVIDIA* 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 NVIDIA* 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、CUDA* 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 NVIDIA* GPU」と「DPC++ CUDA* プラグイン」は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 NVIDIA* GPU の使用を開始するには、「導入ガイド」を参照ください。

導入ガイド

- 導入ガイド
- SYCL* アプリケーションのデバッグ

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

このガイドでは、DPC++ と DPC++ CUDA* プラグインを使用して、NVIDIA* 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。

oneAPI for NVIDIA* GPU のインストール

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	CUDA*	GPU ドライバー
NVIDIA* GeForce RTX* 2060	Turing* - sm_75	Ubuntu* 20.04.5 LTS	11.7	515.65.01
NVIDIA* A100 PCIe* 40GB	Ampere - sm_80	Ubuntu* 20.04.4 LTS	11.7	515.86.01

- このリリースは各種 NVIDIA* GPU と CUDA* 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* 20.04 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。
- このリリースの oneAPI for NVIDIA* GPU プラグインは、Windows* 上の oneAPI では使用できませんが、Windows* 向けのパッケージは今後リリースする予定です。
- プラグインは、システムにインストールされている CUDA* のバージョンに依存します。CUDA* が macOS* をサポートしなくなったため、oneAPI for NVIDIA* GPU パッケージは macOS* では利用できません。

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 2023.0.0 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 NVIDIA* 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 2023.0.0 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. 「Linux* 向けの NVIDIA* CUDA* インストール・ガイド」(英語) の手順に従って、NVIDIA* GPU ドライバーと CUDA* ソフトウェア・スタックをインストールします。

インストール

1. oneAPI for NVIDIA* 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-nvidia-gpus-2023.0.0-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. CUDA* 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。
   - NVIDIA-smi 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_CUDA_ROOT/bin:$PATH
$ export LD_LIBRARY_PATH=/PATH_TO_CUDA_ROOT/lib:$LD_LIBRARY_PATH
```

インストールの確認

DPC++ CUDA* 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 NVIDIA* GPU があることを確認します。NVIDIA* 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_cuda:gpu:0] NVIDIA CUDA BACKEND, TITAN RTX 0.0 [CUDA 11.0]
```

- 上記のように利用可能な NVIDIA* GPU が表示されていれば、DPC++ CUDA* 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 AMD* 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。

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

インストールされている CODA* のバージョンによっては、次のような警告が表示されることがありますが、これは無視してもかまいません。

```
$ clang++: warning: CUDA version is newer than the latest supported version
11.5 [-Wunknown-cuda-version]
```

3. アプリケーションを実行します。

```
$ SYCL_DEVICE_FILTER=cuda 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_cuda.so
[ PluginVersion: 11.15.1 ]
SYCL_PI_TRACE[all]: Selected device: -> final score = 1500
SYCL_PI_TRACE[all]:   platform: NVIDIA CUDA BACKEND
SYCL_PI_TRACE[all]:   device: NVIDIA GeForce RTX 2060
The results are correct!
```

これで、oneAPI for NVIDIA* 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、NVIDIA* 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 NVIDIA* GPU をターゲットにする

NVIDIA* GPU 向けのコンパイル

NVIDIA* 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:

```
$ clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda sycl-app.cpp -o sycl-app
```

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=nvptx64-nvidia-cuda`: NVIDIA* GPU をターゲットとして、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。

また、次のフラグを使用して、特定の NVIDIA* アーキテクチャー向けの SYCL* カーネルをビルドすることができます。

- `-Xsycl-target-backend=nvptx64-nvidia-cuda --cuda-gpu-arch=sm_80`

デフォルトではカーネルは sm_50 用にビルドされ、多様なアーキテクチャーで動作しますが、新しい CUDA* 機能の利用は制限され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。

複数ターゲット向けのコンパイル

NVIDIA* 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、NVIDIA* GPU、AMD* 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。

```
clang++ -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
```

NVIDIA* GPU でアプリケーションを実行

NVIDIA* ターゲット向けに 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 NVIDIA* 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 NVIDIA* 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++ CUDA* 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=cuda
```

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」(英語) 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 (英語)

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。

CUDA* デバッガーのサポート

CUDA* ツールキットには、CUDA* アプリケーションの NVIDIA* GPU カーネルのデバッグをサポートする `cuda-gdb` デバッガーが付属しています。`cuda-gdb` は、DPC++ `ext_oneapi_cuda` バックエンド用にコンパイルされたカーネルのデバッグにも使用できます。現在、NVIDIA* `nvcc` コンパイラーでコンパイルされたカーネルと比較して、DPC++ でコンパイルされたカーネルのデバッグに `cuda-gdb` が利用される場合、予測される動作に違いは報告されていません。`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。

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	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)	部分的 [1]

機能	サポート
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	はい
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	はい
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
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
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	はい

非コア機能

機能	サポート
image	いいえ (SYCL_PI_CUDA_ENABLE_IMAGE_SUPPORT の一部)
fp16 データタイプ	はい
fp64 データタイプ	はい

機能	サポート
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	はい
アトミックメモリーの順序付け	relaxed、acquire、release、acq_rel
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	sub_group、work_group、device、system
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	はい
バイナリー形式	NVPTX と SASS
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	はい
キューのプロファイル	はい
mem_advise	read_mostly、優先場所、アクセス元
バックエンド仕様	WIP[3]
アプリケーション・バックエンドの相互運用	部分的 [3]
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)	はい
reqd_work_group_size	部分的 [4]
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 [5]
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)	はい
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	一部分

機能	サポート
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	部分的 [6]
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	はい
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	はい
ext_oneapi_bfloat16	はい
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 (1、2)	https://github.com/KhronosGroup/SYCL-Docs/pull/197 (英語) を参照
4	https://github.com/intel/llvm/issues/6103 (英語) を参照
5	https://github.com/intel/llvm/pull/6289 (英語) に追加 (未テスト)

6	拡張仕様 https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_matrix.asciidoc (英語)

更新履歴

2023.0.0

oneAPI for NVIDIA* GPU の最初のリリースです。

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

- CUDA* バックエンドのサポート

SYCL* コンパイラー

- sycl::half タイプのサポート
- ストレージタイプで動作する `bf16` ビルトインのサポート
- リレーショナル、ジオメトリー、共通、および数学カテゴリーからの SYCL* ビルトインのサポート
- sub_group 拡張のサポート
- グループ・アルゴリズムのサポート
- `group_ballot` 組込み関数のサポート
- スコープとメモリー順序によるアトミックのサポート
- 同時実行を改善する各キューでの複数ストリームのサポート
- `sycl::queue::mem_advise` のサポート
- CUDA* libclc での `--ffast-math` のサポート
- デバイスでの `assert` をサポート
- CUDA* libclc での float/double 変換、比較交換のアトミック操作をサポート
- CXX 標準ライブラリー関数に対応
- デフォルト ctor の sycl::event のネイティブイベントは COMPLETE 状態

SYCL* ライブラリー

- `fma`、`fmin`、`fmax` および `fmax` に `bf16` ビルトインを追加
- `sycl::aspect::fp16` をサポート
- `tanh` (float/half) と `exp2` (half) のネイティブ定義を追加
- `sycl::get_native(sycl::buffer)` をサポート
- `mem_advise` リセットと同時メモリーチェック管理を実装
- `bfloat16` のサポートを含む、`joint_matrix` での要素ごとの操作をサポート
- 統合共有メモリー (USM) をサポート

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。

icpx でビットコード出力が無効になる

現在のバージョンの `icpx` には、NVIDIA* または AMD* ターゲット向けにコンパイルする際に既知の問題があり、次のエラーが出力される場合があります。

```
LLVM ERROR: Bitcode output disabled because proprietary optimizations have been
performed.
```

これは次のリリースで修正される予定ですが、それまでは NVIDIA* または AMD* GPU をターゲットとする場合、`icpx` 実行ファイルの代わりに、このガイドで説明されている `clang++` 実行ファイルを直接使用してください。

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」(英語) 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: "cannot find libdevice (libdevice が見つかりません)"

CUDA* SDK がデフォルトの位置にインストールされていないと、`clang++` が SDK を検出できず、コンパイル中に次のようなエラーが発生することがあります。

```
clang-16: error: cannot find libdevice for sm_50; provide path to different CUDA
installation via '--cuda-path', or pass '-nocudalib' to build without linking
with libdevice
```

この問題を解決するには、`--cuda-path` オプションで CUDA* SDK のインストール・パスを指定します。

コンパイラーのエラー: "needs target feature (ターゲットの機能が必要です)"

DPC++ ランタイムで使用される一部の nvptx ビルトインは、コンパイルに最小限の計算機能を要求します。プログラムが使用するビルトインに対し十分な計算機能をターゲットにしていない場合 (コンパイル引数 `-Xsycl-target-backend --cuda-gpu-arch=sm_xx` を使用)、次のエラーが報告されます。

```
error: '__builtin_name' needs target feature (sm_70|sm_72|..),...
```

このようなエラーを回避するには、十分な計算機能を持つデバイスをターゲットにしてコンパイルしていることを確認してください。サポートされている計算機能を持つデバイスをコンパイラーに指定してもこのようなエラーが発生する場合、`-Xsycl-target-backend` に 32 ビット・トリプル `nvptx-NVIDIA-cuda` を渡していると考えられます。`nvptx-NVIDIA-cuda` トリプルは、ターゲット機能のビルトインをコンパイルできず、DPC++ では公式にサポートされていません。64 ビット・トリプル `nvptx64-NVIDIA-cuda` は、最近の NVIDIA* デバイスをすべてサポートするため、こちらを使用することを推奨します。

コンパイラーの警告: "CUDA version is newer than the latest supported version (CUDA* のバージョンがサポートされるバージョンよりも新しいです)"

このリリースでは、使用する CUDA* のバージョンによっては、コンパイラーが次のような警告を出力することがあります。

```
clang++: warning: CUDA version is newer than the latest supported version 11.5 [-
Wunknown-cuda-version]
```

通常、この警告は無視してもかまいません。DPC++ は、最新の CUDA* でサポートされる機能を使用しないかもしれませんが、大部分のシナリオでは問題なく動作するはずです。

Linking Error with nvvm-reflect-ftz (nvvm-reflect-ftz とのリンクエラー)

現在、DPC++ のバグにより、CUDA* プラグインをターゲットにすると FTZ を有効にできません。また、FTZ は `-ffast-math` や `-Ofast` によって有効になる場合がありますが、この場合次のようなエラーが発生します。

```
error: linking module flags 'nvvm-reflect-ftz': IDs have conflicting override
values in '/tmp/lit-tmp-045kklh1/clang++-b050f4/libsycl-crt-a42c81.cubin' and
'llvm-link'
```

この場合、FTZ を無効にして問題を解決する必要があります。`-ffast-math` や `-Ofast` を使用する場合、`-Xclang -fdenormal-fp-math=ieee` を追加することで対応できます。

さらに、FTZ が `-fcuda-flush-denormals-to-zero`、`-fdenormal-fp-math=preserve-sign` または `-fdenormal-fp-math=positive-zero` などのフラグで有効にされていないことを確認してください。

oneAPI for NVIDIA* 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。