新製品 oneAPI Construction Kit の概要

この記事は、Codeplay developer Blogs で公開されている「Introducing the oneAPI Construction Kit」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 は、単一のコードベースで GPU や FPGA など複数のアクセラレーター・アーキテクチャーに対応できるオープンなクロスアーキテクチャーのプログラミング・モデルです。SYCL* をベースにしており、DPC++/C++ コンパイラーを使用して数学や AI などのライブラリー群を実装します。DPC++ は LLVM ベースのコンパイラー・プロジェクトで、SYCL* 言語のコンパイラーとランタイムサポートを実装しています。

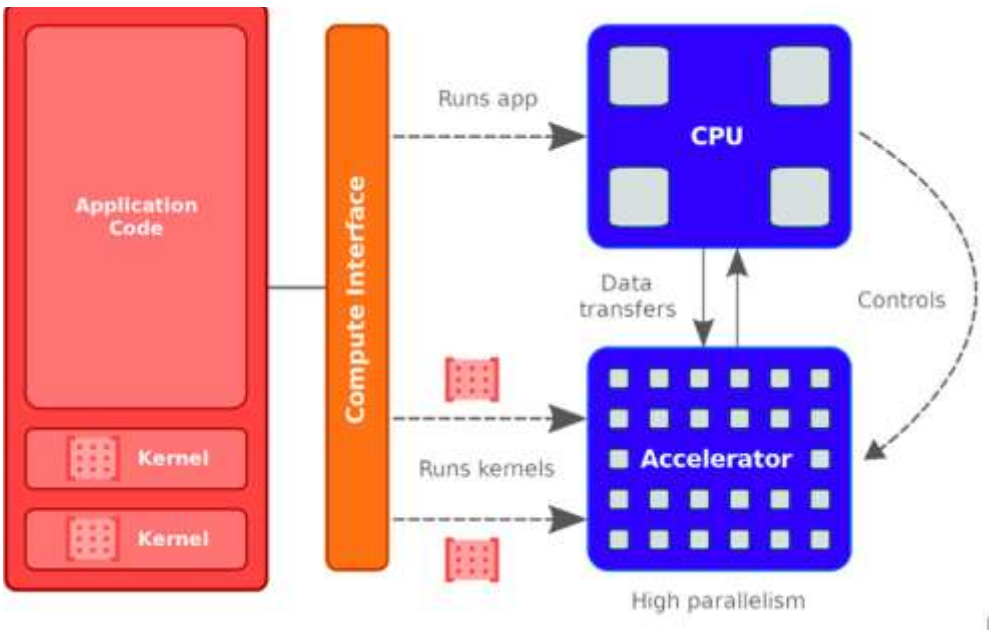
oneAPI Construction Kit により、oneAPI コンポーネント、特に DPC++ コンパイラーを新しいアクセラレーター・プロセッサー・アーキテクチャーで利用できるようになります。

この記事では、oneAPI Construction Kit を使用して、RISC-V* ベースのリファレンス・アクセラレーター・アーキテクチャー向けの SYCL* コードを DPC++ でコンパイルする方法を実証し、コンピューターの CPU と RISC-V* シミュレーターで SYCL* プログラムを動作させます。

皆さんがご自身で試すことができるように、ここでは新しいデバイス向けに更新する必要があるすべてのコードをカバーするのではなく、主要な構成部分に注目します。使用するアクセラレーター・プロセッサーに応じて、設計に特化したいくつかの変更を加える必要があります。これについてはドキュメントに記載されており、今後のブログで取り上げる予定です。

オフロードと oneAPI Construction Kit の背景

次の図は、ホストデバイスとアクセラレーターの連携を示しています。



CPU コアは、ユーザー・アプリケーション・コードの大部分を実行し、アクセラレーターが行う作業を管理します。アクセラレーター・コアは、「カーネル」と呼ばれるプログラムの小さな領域を実行します。アクセラレーター・コアは、CPU から実行するカーネルがオフロードされるまでアイドル状態です。CPU は、アクセラレーターがカーネルの実行を完了するのを待つ間、ほかの作業を行うこともできます。この記事で使用する Refsi G1 の例では、CPU は x86 で、RISC-V* コアはシミュレーター・ライブラリーでシミュレートされます。

必要条件

ここでは、最新の llvm バイナリーリリースを使用できるように Ubuntu* 22 を使用していますが、基本的な手順は、通常のサポート・プラットフォームである Ubuntu* 20 でも動作するはずです。

まず、新しいディレクトリーを作成し、そこに oneAPI Construction Kit をクローンします。

```
$ mkdir oneapiconstructionkit
$ cd oneapiconstructionkit
$ git clone https://github.com/codeplaysoftware/oneapi-construction-kit.git
```

このコマンドラインは、GitHub* アカウントが設定済みで、パブリック ssh キーが追加済みであることを想定しています。

Docker* は以下のコマンドでビルドできます。

```
$ cp oneapi-construction-kit/examples/technical_blogs/refsi_simple_blog1/* .
$ sudo docker build -t refsi_blog .
```

Docker* のセットアップでは、すべてが /home/demo 以下に配置されると想定し、環境変数 $BLOG_TOP_LEVEL をこの値に設定します (ほかの変数については以降で説明します)。

これで、Docker* コンテナを実行できるようになりました。

```
$ sudo docker run -it --rm refsi_blog bash
```

Docker* には、以下のパッケージがプリインストールされています。

```
$ sudo apt update
$ sudo apt install -y build-essential git cmake libtinfo-dev python3
$ sudo apt install -y ninja-build doxygen python3-pip
$ sudo apt install -y wget spirv-tools libzstd-dev libtinfo5
$ sudo pip install lit virtualenv cmakelint cookiecutter
```

リファレンス実装では、LLVM コンパイラー・プロジェクトで RISC-V* ターゲットのサポートを開発しました。ほとんどのカスタムターゲットでは、LLVM をビルドする必要がありますが、この記事では LLVM GitHub* サイトからビルド済みバージョンをダウンロードします。

```
$ wget https://github.com/llvm/llvm-project/releases/download/llvmorg-
16.0.4/clang+llvm-16.0.4-x86_64-linux-gnu-ubuntu-22.04.tar.xz
$ tar xf clang+llvm-16.0.4-x86_64-linux-gnu-ubuntu-22.04.tar.xz
```

次に、RISC-V* ターゲット向けに SYCL* コードをコンパイルするため、インテル® oneAPI DPC++/C++ コンパイラーとランタイムをダウンロードします。https://github.com/intel/llvm/releases (英語) からプレリリース版を取得できます。ここでは、2023 年 5 月 18 日にリリースされた「DPC++ daily 2023-05-18」を使用します。

```
$ wget https://github.com/intel/llvm/releases/download/sycl-
nightly%2F20230518/dpcpp-compiler.tar.gz
$ tar xf dpcpp-compiler.tar.gz
```

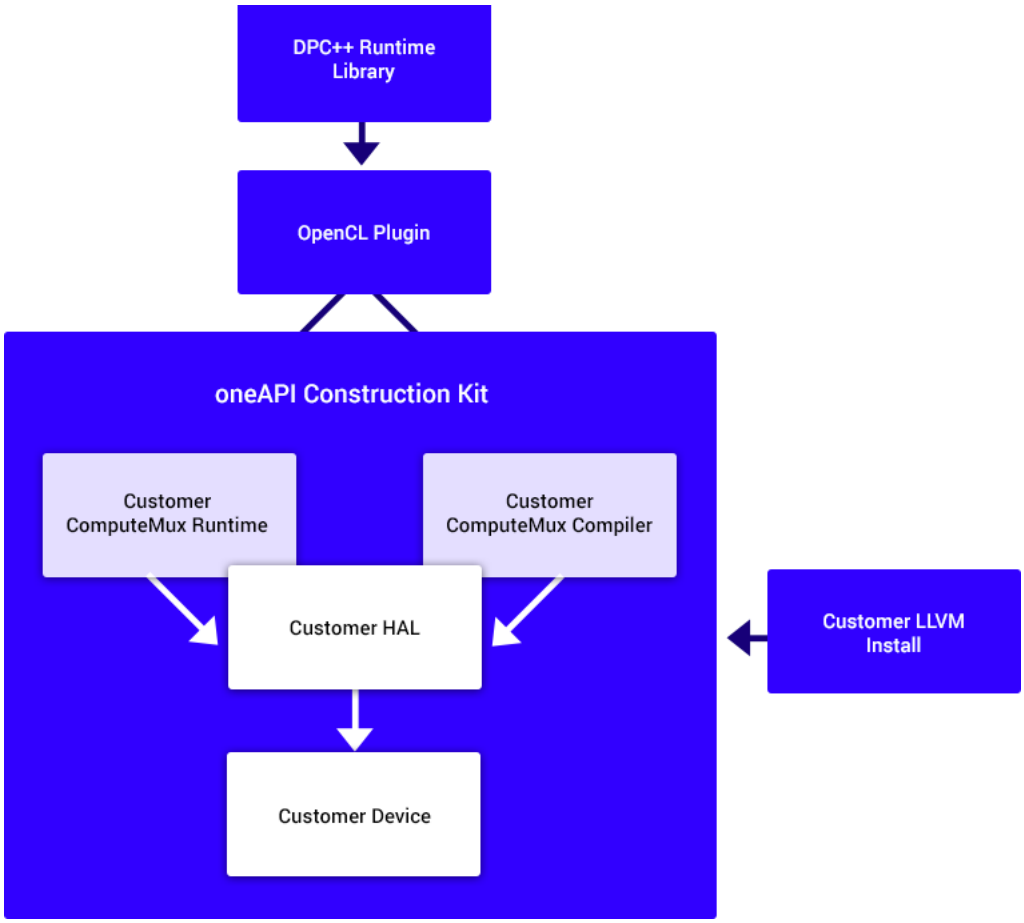
さらに、oneAPI GitHub* リポジトリーから oneAPI-Samples を clone し、これらのサンプルの 1 つをコンパイルして実行してみましょう。

```
$ git clone https://github.com/oneapi-src/oneAPI-samples.git
```

カスタムターゲット

このセクションでは、カスタムターゲットとは何かを説明し、リファレンス RISC-V* デバイスである Refsi G1 (2 つのコアを持つ RISC-V* デバイス) で動作するターゲットを作成します。

次の図は、oneAPI Construction Kit を利用して新しいデバイスを追加し、DPC++ SYCL* コンパイラーを利用できるようにする方法を示しています。



DPC++ ランタイムでは、OpenCL* をプラグインとして使用することができます。oneAPI Construction Kit はこのインターフェイスをサポートし、ランタイムとコンパイラーを提供します。コンパイラー・インターフェイスは、カスタム LLVM インストールを使用します (この例では、RISC-V* をサポートする標準的なものを使用します)。

カスタムターゲットは、3 つの重要な部分から構成されます。

- ランタイムコード (この例では ComputeMux Runtime)
- コンパイラー・コード (この例では ComputeMux Compiler)
- オプションの HAL (ハードウェア抽象化レイヤー) - 簡単に使い始めることができるように、デバイスの静的情報と簡素化されたランタイム・インターフェイスを提供します。

ランタイムコードは、ホストデバイス (つまり、筆者が使用しているマシン、多くの場合は CPU) 上で動作し、ターゲットデバイスであるアクセラレーター (この例では RISC-V* シミュレーター) とのインターフェイスを提供します。コンパイラーは、メモリーの割り当てや読み書き、コマンドのキューイング、デバイス上でのカーネルの実行などの処理を行います。コンパイラーは通常、複数回の LLVM パスにより、オリジナルのカーネルをデバイス上で実行するのに必要なインターフェイスと一致するコードに変換します。

新しいカスタムターゲットの作成を支援するスクリプトが用意されています。このスクリプトは、基本機能を備えた汎用テンプレート・コードを生成し、開発者はニーズに合わせてこれをカスタマイズできます。create_target.py と呼ばれるこのスクリプトは、トップレベルの scripts フォルダーにあります。このスクリプトは、オプションの HAL コンポーネントの使用を想定しています。新しいターゲットを作成するには、以下のような詳細を含む json ファイルを提供する必要があります。

```
"target_name": "refsi_tutorial",
"llvm_name": "RISCV",
"llvm_cpu": "\"generic-rv64\"",
"llvm_features": "'+m,+f,+a,+d,+c,+v,+zbc,+zvl128b'",
```

これは、ランタイム (ComputeMux Runtime) とコンパイラー (ComputeMux Compute) の両方でカスタムターゲットの作成に必要なファイルを作成するため、入力として使用されます。例えば、llvm_features オプションは、RISC-V* バックエンドのさまざまな機能を有効にします。

スクリプトを実行するには、oneAPI Construction Kit のトップ・レベル・ディレクトリーから以下を呼び出します。

```
$ ./scripts/create_target.py $PWD scripts/new_target_templates/refsi_g1.json --
external-dir $BLOG_TOP_LEVEL/refsi_blog
```

mux と compiler という 2 つのディレクトリーが作成され、$BLOG_TOP_LEVEL/refsi_blog 以下に CMakeLists.txt が作成されます。mux ディレクトリーはすべてのランタイムコードに関係し、compiler ディレクトリーはすべてのコンパイラー関連コードに関係します。

この例では Refsi G1 を使用しているので、oneAPI Construction Kit のサンプル HAL から直接 HAL を使用できます。HAL には、メモリーの割り当てや読み書き、カーネルの実行など、低レベルのアクセス方法が含まれています。今後の記事で、独自の HAL を作成する方法を紹介する予定です。mux とコンパイラーが生成したコードはどちらも HAL と対話しますが、mux はより動的な方法で対話するのに対し、コンパイラーは静的な情報にアクセスします。

この記事の残りの部分では、以下の環境変数が設定されていると想定します。

ONEAPI_CON_KIT_DIR	oneAPI Construction Kit のベース
DPCPP_DIR	DPC++ コンパイラーのトップレベルへのパス
ONEAPI_SAMPLES_DIR	oneAPI-samples へのパス
LD_LIBRARY_PATH	DPC++ インストールの $DPCPP_DIR/lib
LLVM_INSTALL_DIR	LLVM インストールへのパス

新しいターゲット向けにツールキットをビルドするには、$BLOG_TOP_LEVEL/refsi_blog に以下の CMake 行がなければなりません。

```
cmake -Bbuild \
 -DCA_EXTERNAL_ONEAPI_CON_KIT_DIR=$ONEAPI_CON_KIT_DIR \
 -DCA_EXTERNAL_REFSI_G1_HAL_DIR=$ONEAPI_CON_KIT_DIR/examples/refsi/hal_refsi \
 -DCA_MUX_TARGETS_TO_ENABLE="refsi_g1" \
 -DCA_REFSI_G1_ENABLED=ON \
 -DCA_ENABLE_API=cl \
 -DCA_LLVM_INSTALL_DIR=$LLVM_INSTALL_DIR \
 -DCA_CL_ENABLE_OFFLINE_KERNEL_TESTS=OFF \
 -GNinja
```

ここで関連する CMake 変数は以下のとおりです。

CA_EXTERNAL_ONEAPI_CON_KIT_DIR	主要な oneAPI Construction Kit へのパス
CA_EXTERNAL_REFSI_G1_HAL_DIR	デバイスに関連する HAL へのパス (この例では、RefSi に適した HAL が oneAPI Construction Kit に含まれています)
CA_MUX_TARGETS_TO_ENABLE	有効にするターゲット – この例では refsi_g1 のみ
CA_REFSI_G1_ENABLED	このターゲットのビルドを有効にします
CA_LLVM_INSTALL_DIR	llvm インストールへのパス
CA_ENABLE_API	有効にする API - この例では OpenCL* のみ

これで、動作する Refsi ターゲットをビルドできます。ninja を呼び出してすべてをビルドします。

```
$ cd build
$ ninja
```

そして、簡単なテストを実行します。OpenCL* テストスイートである UnitCL から 1 つのテストを実行し、OpenCL* ターゲットが動作していることを確認します。これを行うには、OpenCL* ICD Loader に OpenCL* ライブラリーの検索場所を伝える環境変数 (OCL_ICD_FILENAMES) を追加する必要があります。

```
$ OCL_ICD_FILENAMES=$BLOG_TOP_LEVEL/refsi_blog/build/oneAPIConstructionKit/lib/li
bCL.so ./OneAPIConstructionKit/bin/UnitCL --
gtest_filter=Execution/Execution.Task_01_02_Add/OpenCLC
```

これは簡単な add カーネルを実行し、以下を表示します。

```
 [ RUN ] Execution/Execution.Task_01_02_Add/OpenCLC
 [ OK ] Execution/Execution.Task_01_02_Add/OpenCLC (43 ms)
 [----------] 1 test from Execution/Execution (43 ms total)
 [----------] Global test environment tear-down
 [==========] 1 test from 1 test suite ran. (46 ms total)
 [  PASSED  ] 1 test.
```

このテストでは、Refsi のデバイスメモリーに割り当てられた 2 つの入力バッファーを受け取り、デバイス上でカーネルを並列に実行して 2 つのバッファーの格納値を加算し、結果を出力バッファーに書き出します。

次に、SYCL* プログラムをビルドして、デバイス上で使用します。ここでは、2 つのバッファーに格納されている整数値を加算する、簡単な SYCL* の例を使用します。以下に主なコードを示します。

```
 q.submit([&](handler &h) {
   // Create an accessor for each buffer with access permission: read, write or
   // read/write. The accessor is a mean to access the memory in the buffer.
   accessor a(a_buf, h, read_only);
   accessor b(b_buf, h, read_only);

   // The sum_accessor is used to store (with write permission) the sum data.
   accessor sum(sum_buf, h, write_only, no_init);

   // Use parallel_for to run vector addition in parallel on device. This
   // executes the kernel.
   //    1st parameter is the number of work items.
   //    2nd parameter is the kernel, a lambda that specifies what to do per
   //    work item. The parameter of the lambda is the work item id.
   // SYCL supports unnamed lambda kernel by default.
   h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
 });
```

これは、Refsi G1 デバイスで並列に実行される小さな sum カーネルをキューにサブミットします。最も重要な部分は { sum[i] = a[i] + b[i]; } で、これは基本カーネルがデバイス上で並列に実行されることを示します。

まず、ダウンロードした DPC++ から clang++ を使って SYCL* 実行ファイルをビルドします。ソースコードは、先ほどクローンした oneAPI-Samples レポジトリーにあり、以下のようにビルドします。

```
$ DPCPP_DIR /bin/clang++ -fsycl
ONEAPI_SAMPLES_DIR/DirectProgramming/C++SYCL/DenseLinearAlgebra/vector-
add/src/vector-add-buffers.cpp -o simple_add
```

直接実行でき、Refsi G1 アクセラレーターにオフロードできる実行ファイルが生成されます。

これを実行するには、環境変数 (ONEAPI_DEVICE_SELECTOR) を追加して、DPC++ に OpenCL* アクセラレーター・プラグインを使用して OpenCL* アクセラレーターを選択するよう指示する必要があります。まず、sycl-ls を使用して、SYCL* がアクセラレーターを検出できることを確認します。このコマンドは、DPC++ コンパイラーが検出した利用可能なデバイスをリストします。

```
$ OCL_ICD_FILENAMES=$BLOG_TOP_LEVEL/refsi_blog/build/OneAPIConstructionKit/lib/li
bCL.so DPCPP_DIR/bin/sycl-ls
```

Refsi G1 アクセラレーターは、以下のように表示されます。

```
[opencl:acc:0] ComputeAorta, RefSi G1 RV64 OpenCL 3.0 ComputeAorta 2.0.0 LLVM
15.0.0 [2.0]
OCL_ICD_FILENAMES==$BLOG_TOP_LEVEL/refsi_blog/build/OneAPIConstructionKit/lib/lib
CL.so
ONEAPI_DEVICE_SELECTOR=opencl:0  ./simple_add
```

実行結果は、以下のようになります。

```
Running on device: RefSi G1 RV64
Vector size: 10000
[0]: 0 + 0 = 0
[1]: 1 + 1 = 2
[2]: 2 + 2 = 4
...
[9999]: 9999 + 9999 = 19998
```

シミュレーター上で実際に動作していることを示すため、CA_HAL_DEBUG=1 を渡したときのデバッグ出力を追加しました。

```
$ CA_HAL_DEBUG=1
OCL_ICD_FILENAMES=$BLOG_TOP_LEVEL/refsi_blog/build/OneAPIConstructionKit/lib/libC
L.so ONEAPI_DEVICE_SELECTOR=opencl:0 ./simple_add
```

実行される HAL 機能が表示されます。

```
refsi_hal_device::mem_alloc(size=16, align=128) -> 0x9ff0ff80
refsi_hal_device::mem_write(dst=0x9ff0ff80, size=16)
```

メモリー割り当て、デバイスメモリーへの書き込み、カーネルの実行など、デバイスとのインターフェイスを素早く起動して実行するため記述する必要があるものを示しています。

さらに、SPIKE_SIM_DEBUG=1 を使用して、デバッグモードで実行される RISC-V* 命令を表示することもできます (続行するには Return キーまたは R キーを複数回押します)。

```
$ SPIKE_SIM_DEBUG=1
OCL_ICD_FILENAMES=$BLOG_TOP_LEVEL/refsi_blog/build/OneAPIConstructionKit/lib/libC
L.so ONEAPI_DEVICE_SELECTOR=opencl:0 ./simple_add
```

以下のような出力が表示されます。

```
core   0: 0x0000000000001000 (0x00000093) li      ra, 0
core   1: 0x0000000000001000 (0x00000093) li      ra, 0
core   0: 0x0000000000001004 (0x00000113) li      sp, 0
core   1: 0x0000000000001004 (0x00000113) li      sp, 0
```

まとめ

このブログでは、リファレンス・シミュレーター Refsi G1 上で oneAPI SYCL* コンパイラーを使用して OpenCL* と SYCL* を実行することができる oneAPI 対応ドライバーをゼロから作成しました。スクリプトを使ってランタイムとコンパイラーのインターフェイスを作成し、ドライバーに接続する方法を紹介しました。また、ドライバーへの主要インターフェイスを素早く立ち上げるために使用する、シンプルな HAL インターフェイスについても触れました。

今後のブログでは、独自の HAL を構築する方法と、生成されたコードがどのように動作するかについて、詳しく説明する予定です。

oneAPI Construction Kit は、オープンソースであり、こちらから (英語) 無料で入手できます。Codeplay は、セットアップと使用方法のガイド (英語) も提供しています。

oneAPI Construction Kit の詳細については、Codeplay CEO 兼共同創設者である Andrew Richards によるプレスリリース (英語) を参照してください。
