さあはじめよう データ並列 C++ の動作の仕組み

2021 年 1 月

IA Software User Society (iSUS)
編集長　すがわら　きよふみ

内容

- データ並列 C++ プログラムが動作する仕組み
  - コンパイルの手順
  - レガシーコンパイル
  - JIT コンパイル
  - AOT コンパイル
  - ファットバイナリー
- 課題と解決方法

DPC++ プログラムの例

単一ソース

- ホストコードとヘテロジニアス･アクセラレーター･カーネルを同じソースファイルに混在して記述

使い慣れた C++

- ライブラリー構造は、次の機能を提供:

コンストラクト	目的
queue	ワークを投入
buffer	データ管理
parallel_for	並列処理

ホストコード

アクセラレーター･コード

ホストコード

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R };

  queue{}.submit([&](handler& h) {
    auto out =
      A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0];
    });
  });
  auto result =
    A.get_access<access::mode::read>();
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "¥n";

  return 0;
}
```

コンパイルの手順

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R };

  queue{}.submit([&](handler& h) {
    auto out =
      A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0];
    });
  });
  auto result =
    A.get_access<access::mode::read>();
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "¥n";

  return 0;
}
```

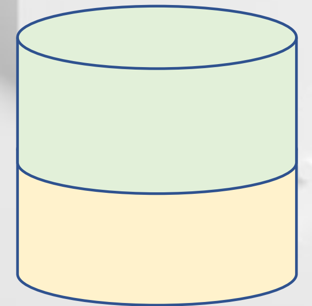
```
$ dpcpp sample.cpp -o sample
```

ホストコード

デバイスコード

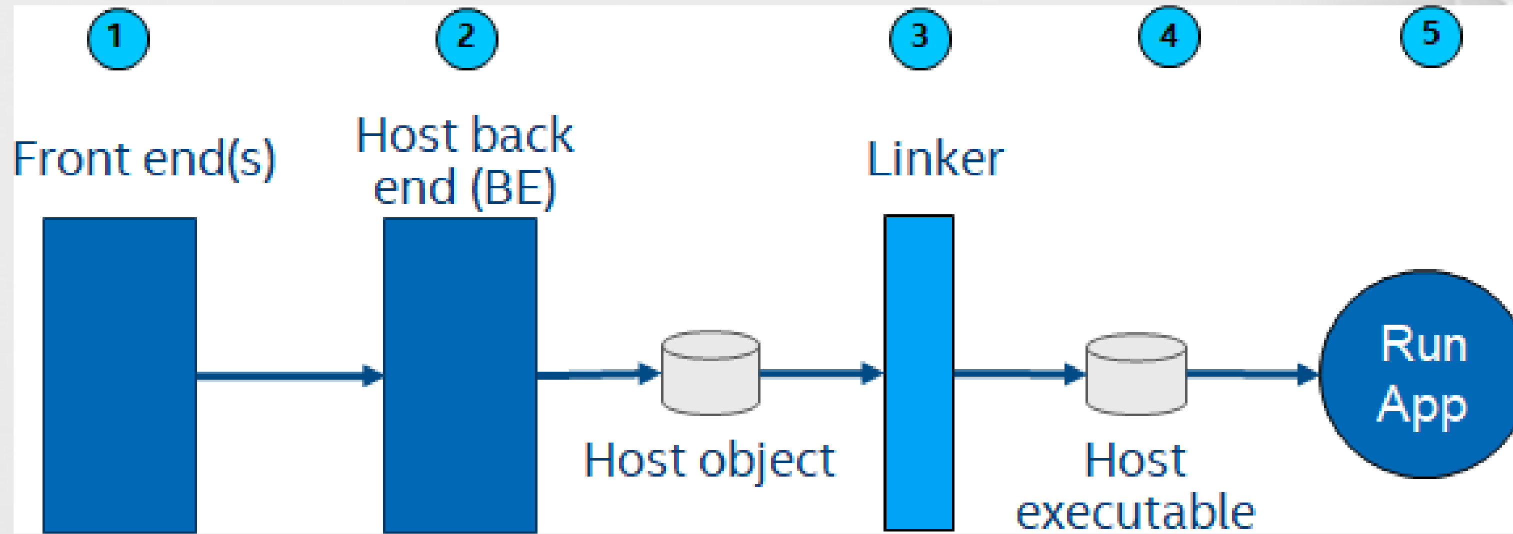
ホストコード

FAT バイナリー
.exe または a.out



JIT コンパイル
AOT コンパイル

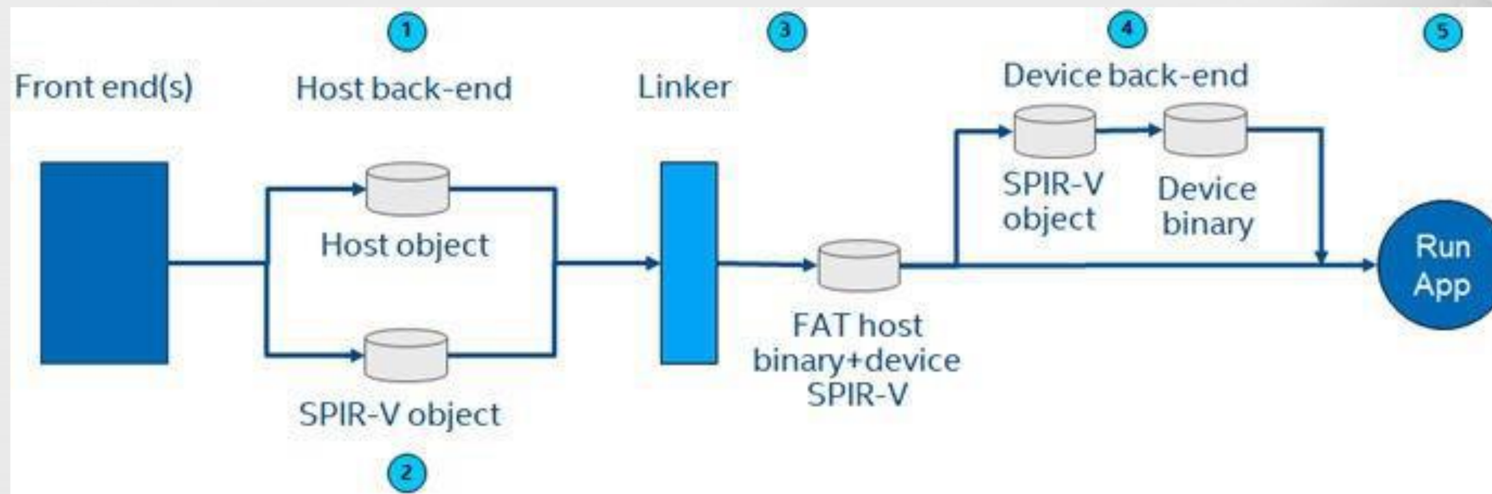
レガシーコンパイル



1. フロントエンドは、ソースを中間表現 (IR) に変換し、バックエンドへ渡す
2. バックエンドは、IR をオブジェクト・コードに変換してオブジェクト・ファイルに出力
3. オブジェクトはリンカーに渡され
4. リンカーはオブジェクトとライブラリーをリンクして実行形式ファイルを生成

iSUS

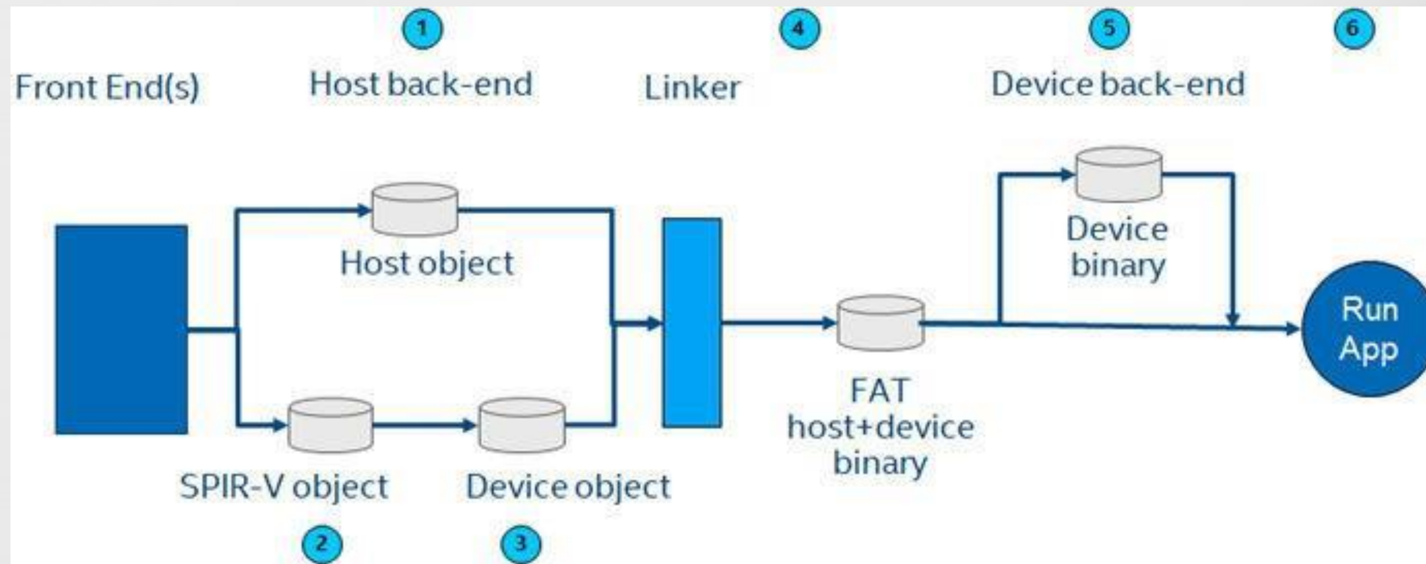
ジャストインタイム (JIT) コンパイル



1. ホストコードは、バックエンドでオブジェクト・コードに変換される
2. デバイスコードは SPIR-V 形式に変換される
3. リンカーは、ホスト・オブジェクト・コードとデバイス SPIR-V を組み合わせた、(SPIR-V が埋め込まれた) ホスト実行コードを含むファットバイナリーを生成

4. 実行されると次のように処理される
   a. ホスト上のデバイスランタイムは、デバイスの SPIR-V をデバイスのバイナリーに変換
   b. 変換されたデバイスバイナリーはデバイスへロードされる

事前 (AOT) コンパイル



1. ホストコードは、バックエンドでオブジェクト·コードに変換される
2. デバイスコードは SPIR-V 形式 (SPIR-V.obj) に変換される
3. デバイスの SPIR-V は、ユーザーが指定したデバイス向けのデバイス·コード·オブジェクトに変換される
4. リンカーは、ホスト·オブジェクト·コードとデバイス·オブジェクト·コードを組み合わせた、デバイスバイナリーが埋め込まれたホスト実行コードを含むファットバイナリーを生成
5. 実行時に、デバイスバイナリーはデバイスへロードされる

JIT vs. AOT

JIT コンパイル

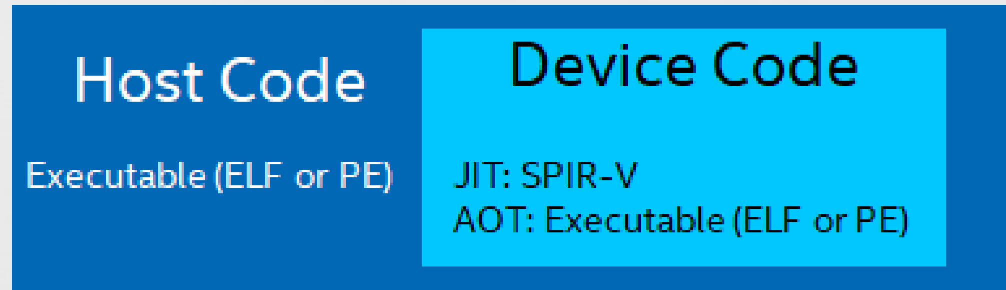
アプリケーションが起動されると、ランタイムは利用可能なデバイスを判別してそのデバイス固有のコードを生成します。これにより、AOT よりも、アプリケーションの実行環境とパフォーマンスの柔軟性が高まります。しかし、アプリケーションの実行時にコンパイル (JIT) が行われるため、アプリケーションの起動時間が増加する可能性があります。大量のデバイスコードを持つ大規模なアプリケーションでは、パフォーマンスへの影響が顕著に表れることがあります

AOT コンパイル

JIT とは異なり、AOT では実行ファイルが作成される前にデバイスバイナリーに変換されます。AOT では、コンパイル時にターゲットデバイスを決定する必要があるため、JIT よりも柔軟性は低くなります。しかし、実行ファイルの起動時間は JIT よりも短くなります

ファット (FAT) バイナリー

ファットバイナリーは、JIT と AOT コンパイルから生成されるデバイスコードが埋め込まれたホストバイナリーです。デバイスコード自体は、コンパイル手順によって異なります



- ホストコードは、ELF (Linux*) または PE (Windows*) 形式の実行ファイルです
- デバイスコードは、JIT では SPIR-V、AOT ではデバイスバイナリー (実行可能) です
- 実行ファイルは次のいずれかの形式です:
  - CPU: ELF (Linux*)、PE (Windows*)
  - GPU: ELF (Windows*、Linux*)
  - FPGA: ELF (Linux*)、PE (Windows*)

いくつかの疑問

1. コンパイル時にターゲットを固定できないか
2. 実行時にターゲットを自由に指定できないか
3. 複数のデバイスで同時に実行できないか
4. デバイスの実行で問題が発生したことを検出して、利用可能なデバイスにフォールバックできないか
5. OpenMP* と DPC++ は混在できるか

答えは、すべて可能、です

iSUS

コンパイル時にターゲットを指定

DPC++ は、CPU / GPU ターゲットのオンラインおよびオフラインのコンパイルモードをサポートしています。オンラインコンパイルは、ほかのすべてのターゲットと同様です

オフライン・コンパイル・オプション (AOT)
CPU:

```
-fsycl-targets=spir64_x86_64-unknown-linux-sycldevice
-Xsycl-target-backend=spir64_x86_64-unknown-linux-sycldevice “-march=avx512”
```

GPU:

```
-fsycl-targets=spir64_gen-unknown-linux-sycldevice
-Xsycl-target-backend=spir64_gen-unknown-linux-sycldevice "-device skl"
```

実行時にターゲットを自由に指定できないか

デバイス間で複数のカーネルをターゲットにするアプリケーションに複数の独立したカーネルで並列処理可能なスコープがある場合、ターゲットデバイスごとに異なるキューを使用します

これは、データ間に依存関係がなければ同時に実行することもできます

または、SYCL_DEVICE_TYPE=[CPU | GPU | HOST | ACC] .. を設定します

```
sycl::host_selector device1;
sycl::queue cpu(device1);
sycl::gpu_selector device2;
sycl::queue gpu(device2);

if(select == 1){
    std::cout << "CPU" << std::endl;
    cpu.submit([&](handler& h) {
     ….
    });
} else {
    std::cout << "GPU" << std::endl;
    gpu.submit([&](handler& h) {
     …
    });

}
```

iSUS

エラー時のフォールバック

```
cl::sycl::device my_dev;
try {
   my_dev = cl::sycl::device(cl::sycl::gpu_selector());
} catch (...) {
   std::cout << "警告: GPU デバイスでエラーが検出されました。CPU で処理を続行します¥n";
}
```

- try / catch 例外処理を利用して、デバイスのエラーを認識して、利用可能なデバイスで処理を続行するプログラム構造を作成できます
- 非同期例外をキャッチすることもできます

iSUS

例外処理の例

```
auto exception_handler = [&](cl::sycl::exception_list exceptions){
    for(std::exception_ptr const& e: exceptions){
        try {
            std::rethrow_exception(e);
        } catch (cl::sycl::exception const& e){
            std::cout << "Failed" << std::endl;
            std::terminate();
        }
    }
};

try {
    queue{}.submit([&](handler& h)
    auto out = A.get_access<access
      h.parallel_for(R, [=](id<1>
        out[idx] = idx[0];
      });
    });
} catch (cl::sycl::exception const&
    std::cout << "failed to run on
    exp = -1;
}
```

-fsycl-targets=spir64_gen-unknown-linux-sycldevice
でコードを生成しインテル® Xeon® プロセッサー上で実行

Intel(r) oneAPI Tools

```
> sample-fat
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: pi_opencl.dll
SYCL_PI_TRACE[all]: Selected device ->
SYCL_PI_TRACE[all]:   platform: Intel(R) OpenCL
SYCL_PI_TRACE[all]:   device: Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
> sample-gpu
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: pi_opencl.dll
SYCL_PI_TRACE[all]: Selected device ->
SYCL_PI_TRACE[all]:   platform: Intel(R) OpenCL
SYCL_PI_TRACE[all]:   device: Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz
failed to run on device
```

OpenMP* と DPC++ .. いくつかの制限はあります

```
int main() {
std::array<float, 1024u> Vec; float Pi;
#pragma omp parallel sections
  {
  #pragma omp section
     iota(Vec.data(), Vec.size());   // SYCL*
  #pragma omp section
     Pi = computePi(8192u);          // OpenMP*
  }
std::cout << "Pi = " << Pi << std::endl;
retuen 0:
}
```

- OpenMP* ディレクティブは、デバイスで実行される DPC++/SYCL* カーネル内では使用できません
- 同様に、DPC++/SYCL* コードは、OpenMP* target 領域内では使用できません

```
icpx -fsycl -fiopenmp -fopenmp-targets=spir64
```

OpenMP* ユーザーへの注意

This patch release fixes an issue causing ICX OpenMP offload to hang with the latest Level 0 driver. This patch is also recommended for DPCPP to work with the latest Level 0 driver.

Intel(r) oneAPI Tools

```
> icx
Intel(R) oneAPI DPC++/C++ Compiler for applications running on Intel(R) 64, Version 2021.1.2 Build 20201214
Copyright (C) 1985-2020 Intel Corporation. All rights reserved.
```

iSUS

実行環境の確認

コードがどこで実行されてるかトレースするため、SYCL_PI_TRACE 環境変数が用意されています:

SYCL_PI_TRACE=1 // コードが実行されているデバイス情報
SYCL_PI_TRACE=-1 // 詳細情報を表示

Intel(r) oneAPI Tools

```
> set SYCL_PI_TRACE=1

> matmul-dpc
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: pi_opencl.dll
SYCL_PI_TRACE[basic]: Plugin found and successfully loaded: pi_level_zero.dll
SYCL_PI_TRACE[all]: Selected device ->
SYCL_PI_TRACE[all]:   platform: Intel(R) Level-Zero
SYCL_PI_TRACE[all]:   device: Intel(R) Graphics Gen9
Start to calc
PASSED in 0.392000 sec.
```

その他の環境変数:
SYCL_BE=PI_OPENCL
SYCL_BE=PI_LEVEL0

ユーティリティー:
sycl-ls –verbose

iSUS

まとめ

- oneAPI は、この複雑なコード生成の作業を開発者から透過にし、CPU/GPU/FPGA/他のアクセラレーターを利用するプログラムの開発を容易にします
- 開発に必要なすべてのソフトウェア開発ツールは、すでに用意されています
- さあ、はじめましょう

関連資料

https://www.isus.jp/oneapi/

- GPU 構成を調査してパフォーマンスを向上する方法
- DPC++ 言語を使用した MPI プログラムのコンパイルと実行
- 革新的なインテル® oneAPI HPC ツールキットを使用して並列安定ソートのパフォーマンスを最適化する方法
- データ並列 C++ を開始する
- OpenCL* 向けの設計を DPC++ へ移行
- 複数アーキテクチャーにおける DPC++ のデータ管理

- インテル® oneAPI プログラミング・ガイド日本語参考訳 (ベータ版)
  https://www.isus.jp/wp-content/uploads/pdf/oneAPIProgrammingGuide_JA.pdf

iSUS

ソフトウェア·セミナー