さあはじめよう
データ並列 C++ プログラミング

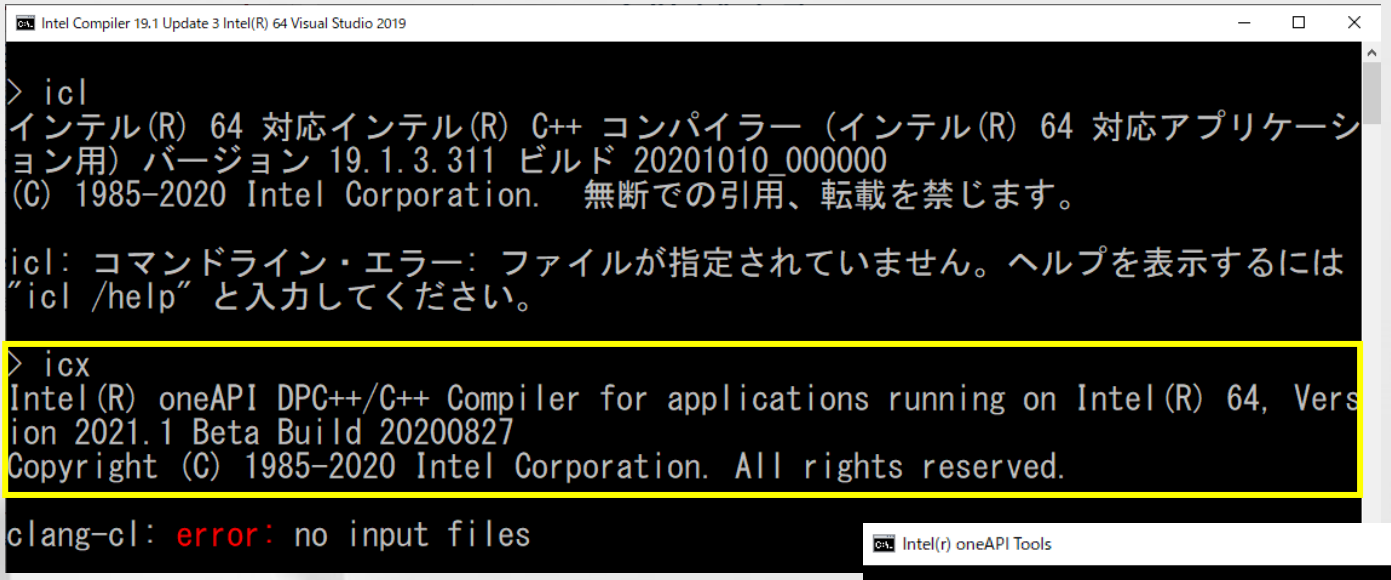
2021 年 2 月

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

内容

- コンパイラーの機能比較
- データ並列 C++ の概念
- データ並列 C++ の対象ユーザー
- C/C++ でのベクトル加算
- SYCL* でのベクトル加算
  - SYCL* キュー
  - SYCL* バッファー
  - デバイス実行の制御
  - 計算カーネルとバッファー
- SYCL* 2020 の統合共有メモリー
- 実行例

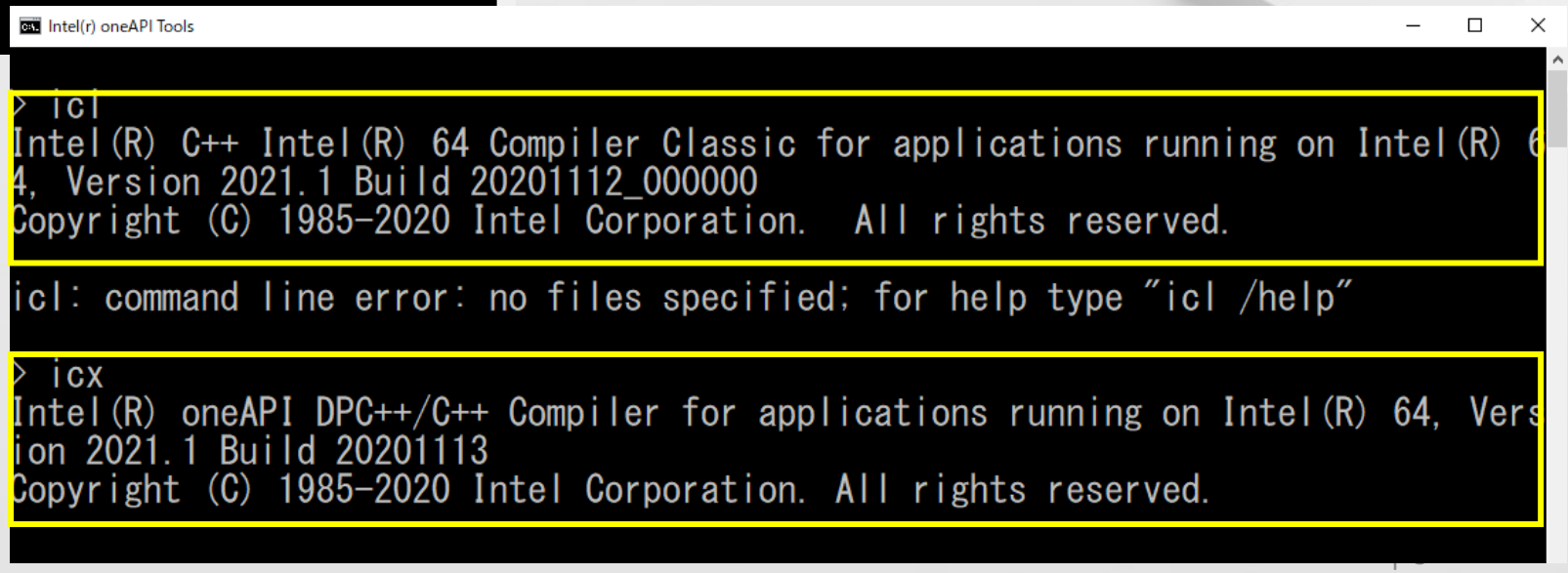
インテル® Parallel Studio XE のコンパイラーと比較



インテル® Parallel Studio XE 2020 に含まれる C++ コンパイラー・パッケージにも DPC++/C++ コンパイラーのベータ版が含まれていますが、こちらは使用しないでください

インテル® oneAPI ツールキットでは、クラッシック・コンパイラーと新しいコンパイラーが提供されます

oneAPI ツールへの移行をお願いします



新旧コンパイラーの機能比較

icl / icc / icpc / ifort: クラッシック・コンパイラーのドライバー:

- OpenMP* 4.5 と 5.0 の一部をサポート (オフロードはサポートしません)
- icl / icc / icpc / ifort は、日本語メッセージの出力をサポートします
- 記述されるソースコードの文字コード形式には柔軟性があります

dpcpp/ icx / icpx / ifx: 新しいコンパイラーのドライバー:

- icx / icpx / ifx は OpenMP* 5.0 のオフロード機能をサポート
- dpcpp は OpenMP* をサポートしません
- icx / icpx / ifx /dpcpp は、日本語メッセージの出力をサポートしません
- UTF-8 形式のソースコードのみを受け入れます
- icx / icpx / ifx は、クラッシック・コンパイラーのすべてのコンパイルオプションをサポートするわけではありませんが、クラッシック・コンパイラーと同じ感覚で利用いただけます

データ並列 C++ (DPC++) を使ってみよう

データ並列 C++ の対象ユーザー

- このセッションは、C++ と並列処理をある程度理解しているプログラマー向けです。C++ と並列処理に関する資料はすでに多く公開されていますが、SYCL* に関する情報は少なく、DPC++ についてはさらに少ないため、ここではこれらに注目します

DPC++ の対象でないユーザー

- C++ を導入していない、または Fortran ユーザー
- オープンな業界標準を使用して Fortran、C、または C++11 よりも前の C++ 標準で CPU や GPU をプログラムする場合は、OpenMP* を使用することをお勧めします
- C++ を使用しない SYCL*/DPC++ に代わるもう 1 つの選択肢は OpenCL* です。OpenCL* は SYCL* よりも冗長ですが、C プログラマーであれば、構文効率よりも明示的な制御を選択するかもしれません

データ並列 C++
標準ベースのクロスアーキテクチャー言語

さまざまな CPU とアクセラレーターに妥協のない並列プログラミングの生産性とパフォーマンスを提供する言語

- 同じデータ並列プログラミング・モデルですべての SVMS アーキテクチャーをサポートする機能と抽象化を提供

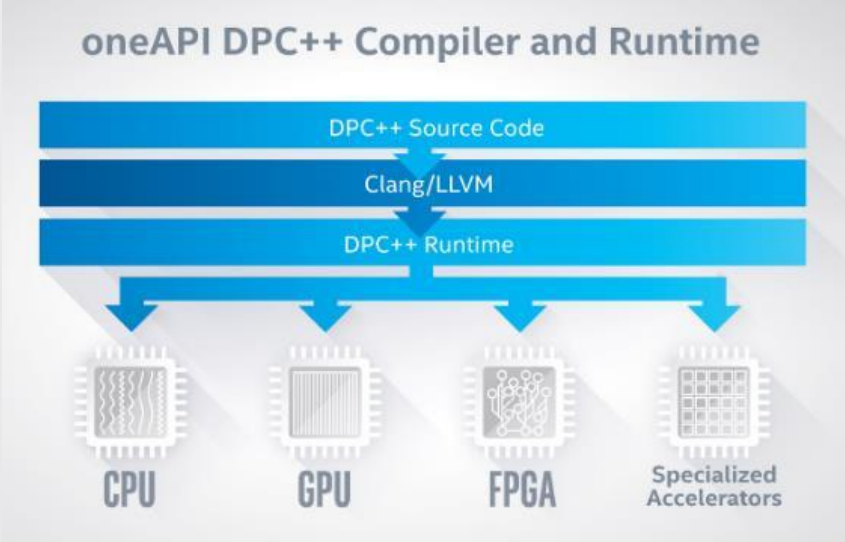
C++ ベース

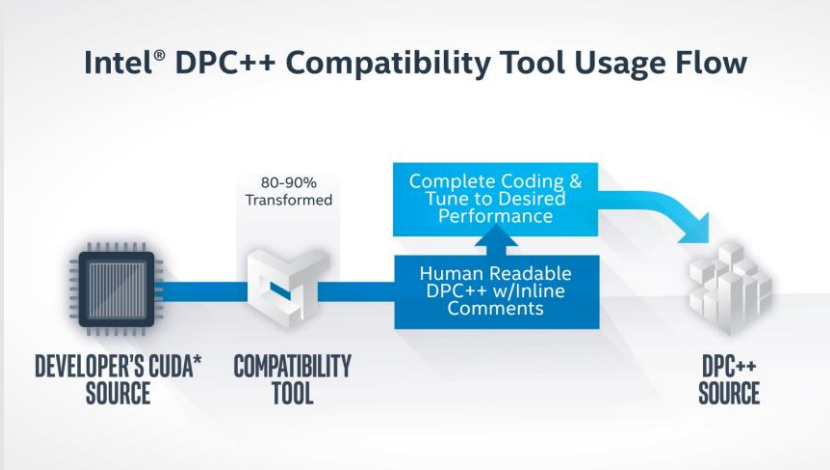
- 使い慣れた C/C++ 構文を使用できるため C++ の生産性が得られる
- Khronos Group の SYCL* を組み込むことでデータ並列処理とヘテロジニアス・プログラミングをサポート

コミュニティー・プロジェクトを通して推進される言語拡張

- データ並列プログラミングを容易にする拡張
- 継続的な進化のためのオープンな共同開発

インテルのアーキテクチャーとコンパイラーの長年の経験に基づいて構築





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;
}
```

C/C++ でのベクトル加算

- ここでは、SAXPY (Single-precision A times X plus Y — 単精度の A × X + Y) を実装します。C または C++ では次のように実装できます

```
for (size_t i = 0; i < length; ++i) {
      Z[i] += A * X[i] + Y[i];
}
```

- これを C++ で記述する方法は多数あります。例えば、レンジを使用すると SYCL* 2020 のようなコードになります。ここでの目的は、C++ で記述可能なすべてのループを示すことではないため、上記のコードを使用します

インクルード・ファイル

まず最初に、すべての移行はここから始まります !!

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

SYCL* でのベクトル加算

- 以下は、前述のループを SYCL* で記述したコードです。さまざまな要素が含まれているため、1 つずつ説明します

```
h.parallel_for<class nstream>(sycl::range<1>[length],[=](sycl::id<1> it) {
      const int i = it[0];
      Z[i] += A * X[i] + Y[i];
});
```

1. parallel_for は並列 for ループです。ループ本体はラムダで表現されています。[..]{..} 形式のコードがラムダです
2. ループ・イテレーターは、sycl::range と sycl::id で表現されています。この例では、<1> であることから、どちらも 1 次元です
3. parallel_for の <class nstream> テンプレート引数でカーネルに名前を付けます
4. h.parallel_for の h については、後述します

SYCL* キュー

- デバイス上で計算するには常にワークキューを作成する必要があります

```
sycl::queue q(sycl::defaut_selector{});
```

- デフォルトでは、利用可能な場合は GPU が選択され、そうでない場合は CPU が選択されます。次のコードを使用して、特定のデバイスタイプのキューを作成できます

```
sycl::queue q(sycl::host_selector{});   CPU で実行 (ランタイムなし)
sycl::queue q(sycl::cpu_selector{});    CPU で実行 (ランタイムあり)
sycl::queue q(sycl::gpu_selector{});    GPU で実行
sycl::queue q(sycl::acceletator_selector{}); FPGA などアクセラレーターで実行
```

バッファーを使用した SYCL* のデータ管理

- SYCL* の標準的なデータ管理ではバッファーを使用します。SYCL* バッファーは不透明なコンテナーです
- アプリケーションによっては後述する USM 拡張によって提供されるポインターが必要になります

```
// T は float などのデータ型です。
std::vector<T> h_X(length, xval);
sycl::buffer<T,1> d_X{ h_X.data(), sycl::range<1>(h_X.size()) };
```

SYCL* バッファーのデストラクター—が呼び出されるまで、ユーザーは SYCL* 以外のメカニズムを介してデータにアクセスすることはできません

デバイス実行の制御

- デバイスコードはホストとは異なるコンパイラーやコード生成メカニズムを必要とするため、デバイスコード領域を明確に識別する必要があります
- submit メソッドを使用して、デバイスキュー q にワークを追加します。このメソッドは、カーネルを実行するための不透明なハンドラーを、このケースでは parallel_for を介して返します

```
q.submit([&](sycl::handler& h) {
    …
    h.parallel_for<class nstream> (sycl::range<1>{length},[=](sycl::id<1> i){
        …
    });
});
q.wait();
```

SYCL* アクセサー: 計算カーネルとバッファー

SYCL* ではプログラマーが copy() メソッドなどを使用してデータを明示的に移動できますが、アクセサーメソッドではコンパイラーとランタイムが適切なタイミングでデータを移動できるようにデータフロー・グラフを生成するため明示的な処理は必要ありません

```
q.submit([&](sycl::handler& h) {
   auto X = d_X.template get_access<sycl::access::mode::read>(h);
   auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
   auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

   h.parallel_for<class nstream> (sycl::range<1>{length},[=](sycl::id<1> i){
      …
   });
});
q.wait();
```

SYCL* SAXPY プログラムのまとめ

```
std::vector<float> h_X(length,xval);
   std::vector<float> h_Y(length,yval);
   std::vector<float> h_Z(length,zval);

   try {

       sycl::queue q(sycl::default_selector{});

       const float A(aval);

       sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
       sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
       sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

       q.submit([&](sycl::handler& h) {

           auto X = d_X.template get_access<sycl::access::mode::read>(h);
           auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
           auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

           h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
               const int i = it[0];
               Z[i] += A * X[i] + Y[i];
           });
         });
         q.wait();
   }
   catch (sycl::exception & e) {
       std::cout << e.what() << std::endl;
       return 1;
   }
```

SYCL キュー

SYCL バッファー

SYCL キューへサブミット

SYCL アクセサー

SYCL カーネル

SYCL* 2020 の統合共有メモリー (USM) - 1

- ここで q 引数は、割り当てられたデータが (永続的または一時的に) 配置されているデバイスのキューです

```
// 共有割り当て （ホストとデバイス間で移行可能）
auto d_X = sycl::malloc_shared<floft>(length, q);

// デバイス割り当て （移行不可）
auto d_X = sycl::malloc_device<float>(length, q);

// 解放 （すべての割り当てタイプで動作）
sycl::free(d_X, q);
```

USM は、2 つの重要なモデルをサポートします:
- ホストとデバイス間の自動データ移動をサポートします
- デバイス割り当てとの間の明示的なデータ移動をサポートします

SYCL* 2020 の統合共有メモリー (USM) - 2

- デバイス割り当てを使用する場合、データを明示的に (例えば、SYCL* の memcpy メソッドを使用して) 移動する必要があります。memcpy メソッドは、std::memcpy と同様に動作します (例えば、デスティネーションは左にあります)

```
const size_t bytes = length * sizeof(float);
q.memcpy(d_Z, h_Z.data(), bytes); // d_Z <- h_Z
q.memcpy(h_Z.data(), d_Z, bytes); // h_Z <- d_Z
q.wait();
```

- USM を使用すると、アクセサーは不要となり、前述のカーネルコードを次のように簡潔にできます

```
q.submit([&](sycl::handler& h) {
   h.parallel_for<class saxpy> (sycl::range<1>{length},[=](sycl::id<1> i){
      d_Z[i] += A * d_X[i] + d_Y[i];
   });
});
```

SYCL* 2020 の簡潔な構文

- 最後に、ここで紹介した各プログラムでは不透明なハンドラー h は必要ありません。以下は、SYCL* 2020 暫定仕様で追加された等価な実装です

```
h.parallel_for(sycl::range<1>{length}, [=] sycl::id<1> i){
       d_Z[i] += A * d_X[i] + d_Y[i];
});
```

さらに、SYCL* 2020 暫定仕様では、ラムダ名はオプションです。この 2 つの変更により、SYCL* カーネルはこのチュートリアルの冒頭にあるオリジナルの C++ ループと同じ長さになります

簡素化されたコード

```
void Compute(TYPE aval) {
    cl::sycl::DEVICE device;
    cl::sycl::queue q(device);

    const float A(aval);
    cl::sycl::range<1> matrix_range(MAX);

    printf("Start to calc¥n");

    auto Xd = sycl::malloc_device<TYPE>(MAX, q);
    auto Yd = sycl::malloc_device<TYPE>(MAX, q);
    auto Zd = sycl::malloc_device<TYPE>(MAX, q);

    const size_t bytes = MAX * sizeof(TYPE);

    q.memcpy(Xd, X, bytes);
    q.memcpy(Yd, Y, bytes);
    q.memcpy(Zd, Z, bytes);

    q.parallel_for(cl::sycl::range<1>{MAX},[=](cl::sycl::id<1> i){
        Zd[i] += A * Xd[i] + Yd[i];
    });

    q.wait();
    q.memcpy(Z, Zd, bytes);
}
```

デバイス上にメモリーを割り当て

ホストメモリーからデバイスへコピー

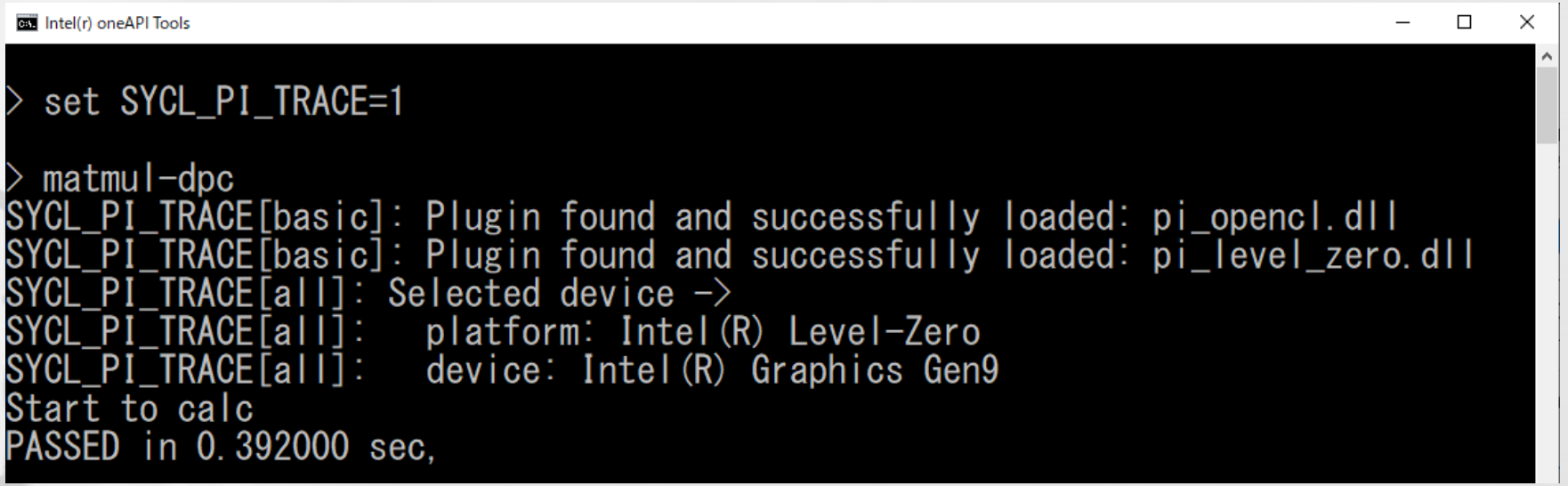
カーネル

デバイスから結果をコピー

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
```

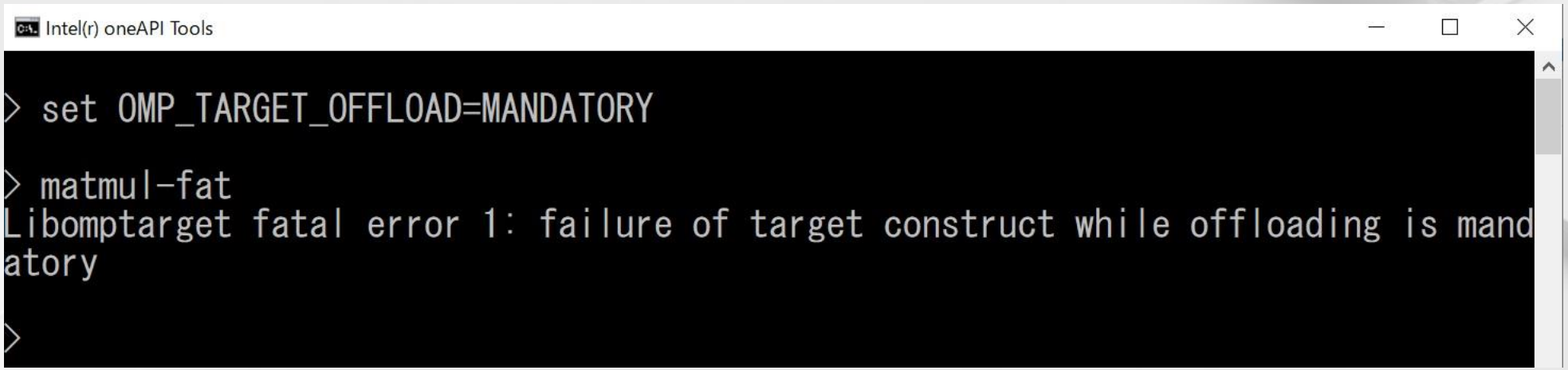


その他の環境変数:
SYCL_BE=PI_OPENCL
SYCL_BE=PI_LEVEL0

ユーティリティー:
sycl-ls –verbose

オフロードプログラムを実行する注意点

- DPC++ や OpenMP* target 構造を使用して、内蔵グラフィックスにオフロードを行う場合、ランタイムに対応する最新のデバイスドライバーを使用してください



これは、OpenMP* を使用した例ですが、対応しないグラフィックス・ドライバーでエラーとなる場合があります。DPC++ バイナリーでは、SYCL_PI_TRACE で詳細を確認できます

コンパイルと実行例


Intel(r) oneAPI Tools
> dpcpp matmul.cpp /DDPCPP /Zi -o matmul-dpc
> 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 0.376000 sec,
> dpcpp matmul.cpp /Zi -o matmul
> matmul
Start to calc
PASSED in 0.000000 sec,
>

まとめ

- このチュートリアルでは、CPU 上でシーケンシャルに実行する 3 行のコードから開始して、最終的に CPU、GPU、FPGA、およびその他のデバイスで並列に実行する 3 行のコードを得ることができました

- すべてのものが SAXPY のように単純ではありませんが、少なくとも SYCL* は簡単なものを難しくすることなく、新たな学習を必要とせず、多くの最新の C++ 機能や「parallel for」などの普遍的な概念をベースにしていることをご理解いただけたかと思います

関連資料

- Khronos SYCL* 1.2.1 仕様
  https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
- oneAPI DPC++ ドキュメント
  https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html
- DPC++ 言語拡張
  https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions

- インテル® oneAPI プログラミング・ガイド
  https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html
- インテル® oneAPI プログラミング・ガイド日本語参考訳
  (購入者限定コンテンツ)
  https://www.xlsoft.com/jp/products/intel/download/oneapi/contents.html
- 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