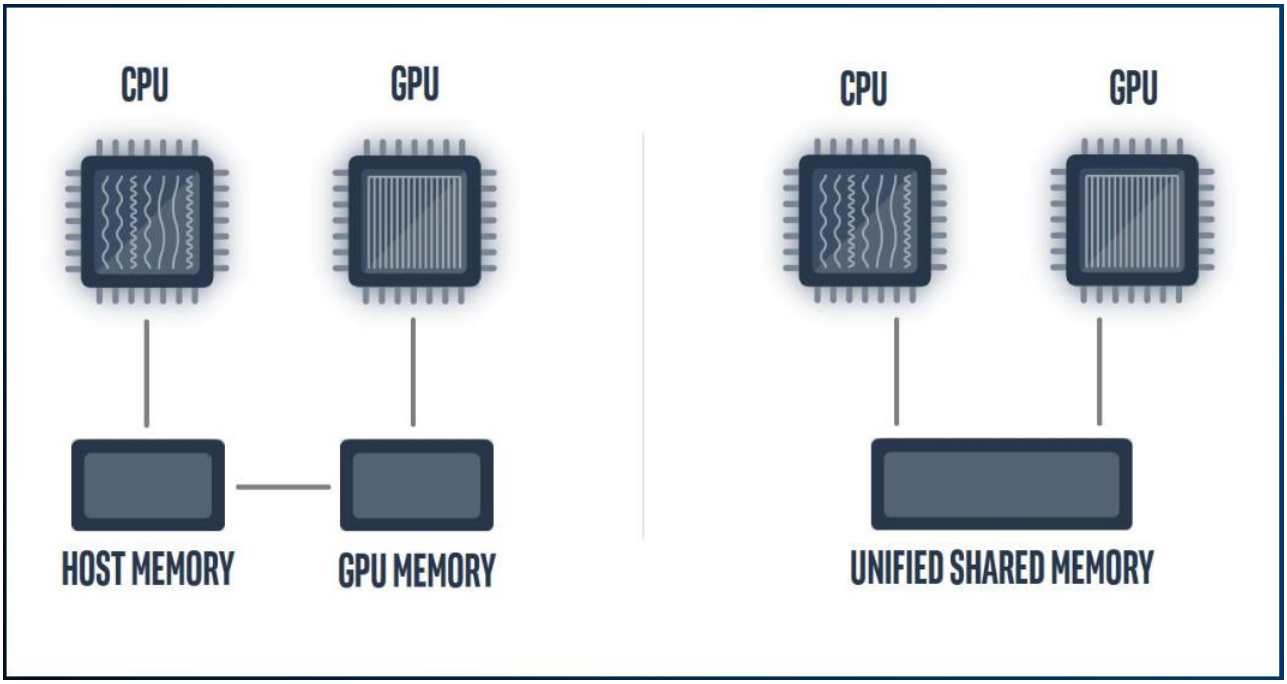
データ並列 C++ USM サンプルコード

この記事は、インテル® デベロッパー・ゾーンで公開されている「DPC++ USM Code Sample Walk-Through」の日本語参考訳です。

統合共有メモリー(USM)を使用したデータ並列 C++(DPC++)サンプルコード

この記事では、データ並列 C++(DPC++)プログラミング言語の基本機能を紹介します。ここでは、バッファーに代わって統合共有メモリー(USM)を使用して、ホストとデバイスのメモリーの管理とアクセスを行います。このプログラムは、並列計算パターンと DPC++ を使用して、2 次元複素平面の各点が集合内に存在するかどうかを計算します。マンデルブロ・サンプルを使って USM を検証します。



マンデルブロ・サンプルは、GitHub*(英語)からダウンロードできます。

統合共有メモリーの概要

- USM は DPC++ の言語機能です。
- 統合仮想アドレス空間のハードウェア・サポートが必要です(ホストとデバイス間で一貫したポインター値を可能にします)。
- すべてのメモリーはホストによって次のいずれかのタイプで割り当てられます。
    - ホスト: ホスト上に配置され、ホストまたはデバイスからアクセス可能
    - デバイス: デバイス上に配置され、デバイスからのみアクセス可能
    - 共有: ホストまたはデバイス上に配置され(コンパイラーが管理)、ホストまたはデバイスからアクセス可能

インクルード・ヘッダー

ほかの一般的なライブラリーとともに、マンデルブロ・コードはデータの可視化に Sean's Toolbox（STB）を使用します。STB ライブラリーは、画像ファイルの読み書きを可能にします。

```
#include <complex>
#include <exception>
#include <iomanip>
#include <iostream>

// stb/*.h ファイルは dev-utilities の include フォルダーにあります。
// 例: $ONEAPI_ROOT/dev-utilities/<version>/include/stb/*.h

#define STB_IMAGE_IMPLEMENTATION
#include "stb/stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb/stb_image_write.h"
```

マンデルブロ・コードは、dpc_common が提供する機能も利用しています。

```
// dpc_common.hpp は dev-utilities の include フォルダーにあります。
// 例: $ONEAPI_ROOT/dev-utilities/<version>/include/dpc_common.hpp

#include "dpc_common.hpp"
```

ドライバー関数: main.cpp

ドライバー関数 main.cpp には、マンデルブロ集合の実行と評価を行うコードが含まれています。

キューの作成

キューは、デフォルトのセレクターを使用して main で作成されます。デフォルトのセレクターは、最初に GPU でカーネルコードの起動を試みて、互換デバイスが見つからない場合はホスト/CPU にフォールバックします。キューは、カーネルコードの非同期例外処理を可能にする dpc_common 例外ハンドラーを利用します。

```
// デフォルトのデバイス上にキューを作成します。SYCL_DEVICE_TYPE 環境変数を
// (CPU|GPU|FPGA|HOST)に設定してデバイスを変更します。

queue q(default_selector{}, dpc_common::exception_handler);
```

ShowDevice()

ShowDevice() 関数は、選択したデバイスに関する重要な情報を表示します。

```
void ShowDevice(queue &q) {

  // プラットフォームとデバイスの情報を出力

  auto device = q.get_device();
  auto p_name = device.get_platform().get_info<info::platform::name>();
  cout << std::setw(20) << "Platform Name: " << p_name << "\n";
  auto p_version = device.get_platform().get_info<info::platform::version>();
  cout << std::setw(20) << "Platform Version: " << p_version << "\n";
```

```
  auto d_name = device.get_info<info::device::name>();
  cout << std::setw(20) << "Device Name: " << d_name << "\n";
  auto max_work_group = device.get_info<info::device::max_work_group_size>();
  cout << std::setw(20) << "Max Work Group: " << max_work_group << "\n";
  auto max_compute_units =
device.get_info<info::device::max_compute_units>();
  cout << std::setw(20) << "Max Compute Units: " << max_compute_units <<
"\n\n";
}
```

Execute()

Execute() 関数は、MandelParallelUsm オブジェクトを初期化して、マンデルブロ集合を評価し、結果を出力します。

```
void Execute(queue &q) {

  // マンデルブロ計算のシリアルバージョンと並列バージョンを実行

#ifdef MANDELBROT_USM
  cout << "Parallel Mandelbrot set using USM.\n";
  MandelParallelUsm m_par(row_size, col_size, max_iterations, &q);
#else
  cout << "Parallel Mandelbrot set using buffers.\n";
  MandelParallel m_par(row_size, col_size, max_iterations);
#endif

  MandelSerial m_ser(row_size, col_size, max_iterations);

  // コードを一度実行して JIT をトリガー

  m_par.Evaluate(q);

  // 並列バージョンを実行して経過時間を収集

  dpc_common::TimeInterval t_par;
  for (int i = 0; i < repetitions; ++i) m_par.Evaluate(q);
  double parallel_time = t_par.Elapsed();

  // 結果を出力

  m_par.Print();
  m_par.WriteImage();

  // シリアルバージョンを実行して経過時間を収集

  dpc_common::TimeInterval t_ser;
  m_ser.Evaluate();
  double serial_time = t_ser.Elapsed();

  // 結果を出力

  cout << std::setw(20) << "Serial time: " << serial_time << "s\n";
  cout << std::setw(20) << "Parallel time: " << (parallel_time / repetitions)
       << "s\n";

  // 確認

  m_par.Verify(m_ser);
}
```

マンデルブロ USM の使用法

MandleParameter クラス

MandelParameter 構造体には、マンデルブロ集合の計算に必要なすべての機能が含まれています。

データ型: ComplexF

MandelParameter は、複素浮動小数点数を表すデータ型 ComplexF を定義します。

```
typedef std::complex<float> ComplexF;
```

Point()

Point() 関数は、複素点 c を引数として受け取り、それがマンデルブロ集合に属するかどうか判定します。この関数は、再帰関数 $z_{n+1} = (zn)^2 + c$ の場合（ここで、$z_0 = 0$）、パラメーター z が制限されたままの反復回数（任意の max_iterations まで）をチェックします。そして、反復回数を返します。

```
int Point(const ComplexF &c) const {
  int count = 0;
  ComplexF z = 0;

  for (int i = 0; i < max_iterations_; ++i) {
    auto r = z.real();
    auto im = z.imag();

  // 発散したらループを終了

  if (((r * r) + (im * im)) >= 4.0f) {
    break;
  }

  // z = z * z + c;

  z = complex_square(z) + c;
    count++;
  }

  return count;
}
```

ScaleRow()/ScaleCol()

スケール関数は、行 / 列インデックスを複素平面内の座標に変換します。これは、配列インデックスを対応する複素座標に変換するために必要です。この関数の使用法は、後続の「MandelParallelUsm クラス」で確認できます。

```
// 0..row_count から -1.5..0.5 へスケール

float ScaleRow(int i) const { return -1.5f + (i * (2.0f / row_count_)); }

// 0..col_count から -1..1 へスケール

float ScaleCol(int i) const { return -1.0f + (i * (2.0f / col_count_)); }
```

Mandle クラス

Mandel クラスは、MandelParallelUsm が継承する親クラスです。後続の「その他の関数」で説明する、データの可視化を出力するメンバー関数が含まれています。

メンバー変数

- MandelParameters p_: MandelParameters オブジェクト
- int *data_: 出力データを格納するメモリーへのポインター

MandleParallelUsm クラス

このクラスは Mandel クラスから派生したもので、USM を使用してマンデルブロ計算をオフロードするすべてのデバイスコードを処理します。

デバイスの初期化: コンストラクター

MandelParallelUSM コンストラクターは、最初に Mandel コンストラクターを呼び出し、引数の値を対応するメンバー変数に代入します。次に、キュー・オブジェクトのアドレスをメンバー変数 q に渡し、後でデバイスコードを起動するために使用できるようにします。最後に、Alloc() 仮想メンバー関数を呼び出します。

```
MandelParallelUsm(int row_count, int col_count, int max_iterations, queue *q)
    : Mandel(row_count, col_count, max_iterations) {
  this->q = q;
  Alloc();
}
```

USM の初期化: Alloc()

Alloc() 仮想メンバー関数は、USM を有効にするため MandelParallelUsm クラスでオーバーライドされます。この仮想メンバー関数は、malloc_shared() を呼び出し、メモリーブロックのアドレスを作成して返します。作成されたメモリーブロックは、ホストとデバイスで共有されます。

```
virtual void Alloc() {
  MandelParameters p = GetParameters();
  data_ = malloc_shared<int>(p.row_count() * p.col_count(), *q);
}
```

カーネルの起動: Evaluate()

Evaluate() メンバー関数は、カーネルコードを起動してマンデルブロ集合を計算します。

parallel_for() 内部では、ワークアイテム id（インデックス）が行と列の座標にマップされており、ScaleRow()/ScaleCol() 関数を使用して複素平面内の点を構築するために使用されます。MandelParameters Point() 関数は、複素点がマンデルブロ集合に属するかどうかを判定するために呼び出され、判定結果は共有メモリーの対応する位置に書き込まれます。

```
void Evaluate(queue &q) {

  // 画像を反復処理して各ポイントがマンデルブロ集合に属するかどうかチェック

  MandelParameters p = GetParameters();

  const int rows = p.row_count();
  const int cols = p.col_count();
  auto ldata = data_;

  // 画像を反復処理して各ポイントの mandel を計算

  auto e = q.parallel_for(range(rows * cols), [=](id<1> index) {
    int i = index / cols;
    int j = index % cols;
    auto c = MandelParameters::ComplexF(p.ScaleRow(i), p.ScaleCol(j));
    ldata[index] = p.Point(c);
  });

  // デバイス上の非同期計算の完了を待機

  e.wait();
}
```

共有メモリーの解放: デストラクター

デストラクターは、Free() メンバー関数を呼び出して共有メモリーを解放し、プログラムでメモリーリークが発生しないようにします。

```
virtual void Free() { free(data_, *q); }
```

その他の関数

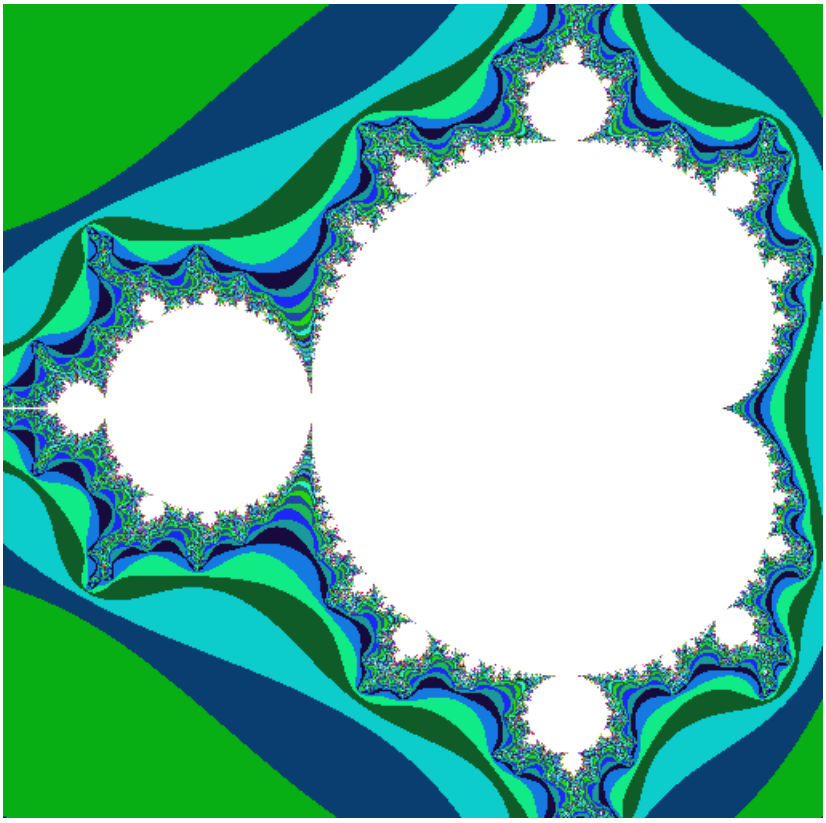
マンデルブロ集合の基本的な可視化の生成

Mandel クラスには、データを可視化するメンバー関数も含まれています。WriteImage() は、各ピクセルが複素平面上の点を表し, その明度が Point() によって計算された繰り返しの深さを表す PNG 画像を生成します。

```
void WriteImage() {
  constexpr int channel_num{3};
  int row_count = p_.row_count();
  int col_count = p_.col_count();

  uint8_t *pixels = new uint8_t[col_count * row_count * channel_num];

  int index = 0;

  for (int j = 0; j < row_count; ++j) {
    for (int i = 0; i < col_count; ++i) {
      float normalized = (1.0 * data_[i * col_count + j]) / max_iterations;
      int color = int(normalized * 0xFFFFFF);  // 16M color.

      int r = (color >> 16) & 0xFF;
      int g = (color >> 8) & 0xFF;
      int b = color & 0xFF;

      pixels[index++] = r;
      pixels[index++] = g;
```

```
      pixels[index++] = b;
    }
  }

  stbi_write_png("mandelbrot.png", row_count, col_count, channel_num, pixels,
                 col_count * channel_num);

  delete[] pixels;
}
```

データ出力の画像例:



Mandel クラスの Print() メンバー関数は、stdout に出力される可視化と同様のものを生成します。

まとめ

ここでは、馴染みのある C / C++ パターンを使用して、マンデルブロを例にホストとデバイスメモリー内のデータを管理する方法を紹介しました。

製品とパフォーマンス情報

[1] 実際の性能は利用法、構成、その他の要因によって異なります。
詳細は、www.Intel.com/PerformanceIndex（英語）を参照してください。