oneAPI の基本

SYCL* のプログラム構造

SYCL* プログラムの構造、重要な SYCL* クラス、および SYCL* のバッファー・メモリー・モデルの紹介

intel®

目的

- SYCL* の基本クラスについて説明する
- デバイス選択を使用してカーネル・ワークロードをオフロードする
- どのような場合に基本並列カーネルと ND-Range カーネルを使用するか判断する
- バッファー・メモリー・モデルを使用してホストとデバイス間のデータを同期するさまざまな方法を理解する
- 計算をアクセラレーター・デバイスへオフロードする完全な SYCL* プログラムを記述する

SYCL* の oneAPI 実装

SYCL* の oneAPI 実装 (DPC++)

= C++ と SYCL* 標準と拡張

最新の C++ ベース

- C++ の生産性と使い慣れた構造

標準ベースのクロスアーキテクチャー言語

- SYCL* を組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

SYCL* 標準を拡張

生産性の強化

- シンプルなものはシンプルに表現
- 冗長性とプログラマーの負担を軽減

パフォーマンスの向上

- プログラムの実行をプログラマーが制御
- ハードウェア固有の機能を利用

SYCL* 標準への統合を目指して急ピッチで進められている取り組み

- LLVM へのアップストリームを目的としたオープンソース実装
- 拡張は SYCL* コアまたは Khronos* 拡張への統合を目指している

完全な SYCL* プログラム

シングルソース

- ホストコードとヘテロジニアス・アクセラレーター・カーネルを同一ソースファイルに混在させることが可能

使い慣れた C++

- ライブラリー構造により機能を追加

構造	目的
queue	ワークターゲット
malloc_shared	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・デバイス・コード

ホストコード

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
  queue q;
  int *data = malloc_shared<int>(N, q);
  q.parallel_for(N, [=](auto i) {
      data[i] = i;
  }).wait();
  for (int i=0; i<N; i++) std::cout << data[i] << "¥n";
  free(data, q);
  return 0;
}
```

SYCL* のクラス

デバイス

- device クラスは oneAPI システムのアクセラレーターの能力を表す
- device クラスには複数のデバイスが作成される DPC++ プログラムで役立つ、デバイスに関する情報を照会するメンバー関数が含まれる
- get_info 関数はデバイスに関する情報を提供
  - デバイスの名前、ベンダー、バージョン
  - ローカルおよびグローバル・ワークアイテム ID
  - ビルトインタイプの幅、クロック周波数、キャッシュの幅とサイズ、オンライン/オフライン

```
queue q;
device my_device = q.get_device();
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```

デバイスセレクター

- device_selector クラスは、ユーザーが提供するヒューリスティックに基づいて、カーネルを実行する特定のデバイスをランタイムに選択できるようにする
- 以下のコード例は標準のデバイスセレクター (default_selector、cpu_selector、gpu_selector…) と派生 device_selector の使用方法を示す

```
default_selector selector;
// host_selector selector;
// cpu_selector selector;
// gpu_selector selector;
queue q(selector);
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
```

キュー

- キューは SYCL* ランタイムによって実行されるコマンドグループを送信
- キューはデバイスにワークを投入するためのメカニズム
- 1 つのキューは 1 つのデバイスにマップされ、複数のキューを同じデバイスにマップ可能

```
queue q;

q.submit([&](handler& h) {
    // コマンド・グループ・コード
});
```

デバイスカーネルの実行場所の選択

ワークはキューに送信される

- 各キューは 1 つのデバイス (特定の GPU や FPGA など) にのみ関連付けられる
- 以下が可能:
  - 必要に応じて、キューをどのデバイスに関連付けるか決定できる
  - ヘテロジニアス・システムではワークをディスパッチするため任意の数のキューを持つことができる

あらゆるデバイスをターゲットとするキューを作成	queue();

あらかじめ設定されたクラスのデバイスをターゲットとするキューを作成	queue(cpu_selector{});
	queue(gpu_selector{});
	queue(intel::fpga_selector{});
	queue(accelerator_selector{});
	queue(host_selector{});
	常に利用可能

特定のデバイスをターゲットとするキューを作成 (カスタム条件)	class custom_selector : public device_selector {
	int operator()(…… **// 任意の条件**
	…
	queue(custom_selector{});

カーネル

- kernel クラスは、コマンドグループをインスタンス化する際に、デバイス上でコードを実行するためのメソッドとデータをカプセル化
- kernel オブジェクトはユーザーによって明示的に構築されない
- kernel オブジェクトは parallel_for などのカーネル・ディスパッチ関数の呼び出し時に構築される

```
q.submit([&](handler& h) {
  h.parallel_for(range<1>(N), [=](id<1> i) {
    A[i] = B[i] + C[i]);
  });
});
```

DPC++ 言語とランタイム

- DPC++ 言語とランタイムは、C++ クラス、テンプレート、ライブラリーのセットで構成される
- アプリケーション・スコープとコマンド・グループ・スコープ:
  - ホストで実行するコード
  - アプリケーションおよびコマンド・グループ・スコープで C++ の全機能を利用可能
- カーネルスコープ:
  - デバイスで実行するコード
  - カーネルスコープでは利用可能な C++ 機能が制限される

並列カーネル

- 並列カーネルは操作の複数のインスタンスを並列に実行できるようにする
- 各反復が完全に独立しており、任意の順序で実行できる基本的な for ループの並列実行をオフロードするのに便利
- 並列カーネルは parallel_for 関数で表現される

CPU アプリケーションの **for** ループ

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

parallel_for でアクセラレーターへオフロード

```
h.parallel_for(range<1>(1024), [=](id<1> i){
    A[i] =  B[i] + C[i];
});
```

基本並列カーネル

基本並列カーネルの機能は、range、id、および item クラスを介して利用可能

- range クラスは並列実行の反復空間を表す
- id クラスは並列に実行するカーネルの個々のインスタンスにインデックスを付与する
- item クラスはカーネル関数の個々のインスタンスを表し、実行範囲のプロパティーを照会する追加の関数を利用できるようにする

```
h.parallel_for(range<1>(1024), [=](id<1> idx){
            // デバイスで実行するコード
});
```

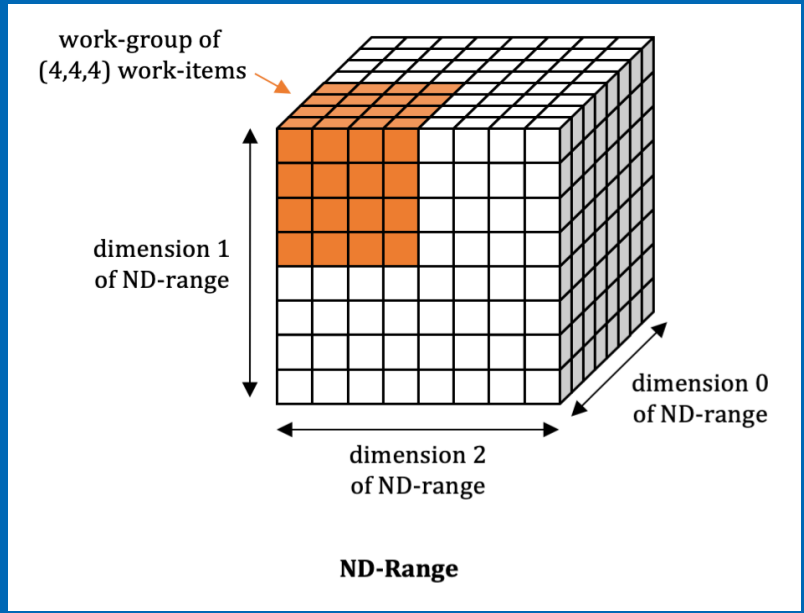
```
h.parallel_for(range<1>(1024), [=](item<1> item){
            auto idx = item.get_id();
            auto R = item.get_range();
            // デバイスで実行するコード
});
```

ND-Range カーネル

基本並列カーネルは for ループを容易に並列化できるが、ハードウェア・レベルでパフォーマンスを最適化できない

ND-Range カーネルはローカルメモリーへのアクセスを提供し、実行をハードウェア上の計算ユニットへマップすることで低レベルのパフォーマンス・チューニングを可能にする並列化手法

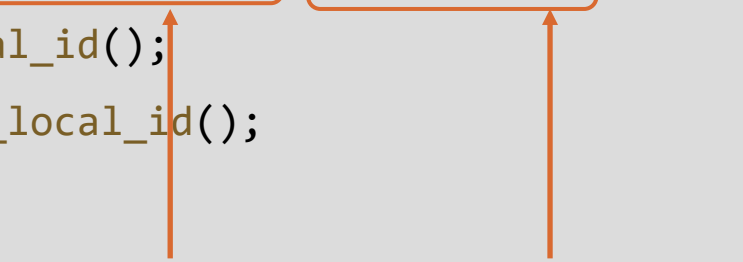
- 反復空間全体をワークグループと呼ばれる小さなグループに分割して、ワークグループ内のワークアイテムをハードウェア上の単一の計算ユニットにスケジュールする
- カーネル実行をワークグループにグループ化することで、リソースの使用を制御し、ワークをバランス良く分散



ND-Range カーネル

nd_range カーネルの機能は nd_range クラスと nd_item クラスを介して利用可能

```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)), [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // デバイスで実行するコード
});
```



- nd_range クラスはグローバル実行範囲と各ワークグループのローカル実行範囲でグループ化された実行範囲を表す
- nd_item クラスはカーネル関数の個々のインスタンスを表し、ワークグループの範囲とインデックスを照会できる

メモリーモデル

SYCL* プログラムでは統合共有メモリー (USM) と呼ばれるポインターベースのメモリーモデル、またはバッファー-アクセサー・メモリー・モデルを使用できる

- 統合共有メモリー: ホストとデバイスのデータにアクセスするポインターベースのメモリーモデル
- バッファー・メモリー・モデル: SYCL* カーネルが使用できる 1、2、3 次元の共有配列を定義し、アクセサークラスを使用してアクセスする必要がある

統合共有メモリー

統合共有メモリーではホスト側とデバイス側のメモリーに同一のポインター参照でアクセス可能

統合共有メモリーの設定

ホストで初期化

デバイスで変更

ホストで出力

```
queue q;
auto data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.parallel_for(N, [=](auto i){
        data[i] += 1;
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

バッファー・メモリー・モデル

バッファー: SYCL* アプリケーションのデータをカプセル化する

- デバイスとホストの両方にまたがる

アクセサー: バッファーデータにアクセスする仕組み

- SYCL* グラフでデータ依存関係を作成し、カーネル実行を順序付ける

```
queue q;
std::vector<int> v(N, 10);
{
  buffer buf(v);
  q.submit([&](handler& h) {
    accessor a(buf, h , write_only);
    h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
}
for (int i = 0; i < N; i++) std::cout << v[i] << " ";
```

SYCL* コードの構造

- SYCL* プログラムには CL/sycl.hpp をインクルードする必要がある
- sycl 名前空間への参照を簡潔に入力できるように namespace ステートメントを使用することを推奨

```
#include <CL/sycl.hpp>
using namespace sycl;
```

SYCL* コードの構造

```
void dpcpp_code(int* a, int* b, int* c) {
  // DPC++ デバイスキューを設定
  queue q;
  // 入力と出力ベクトル用のバッファーを設定
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  // コマンドグループ関数オブジェクトをキューに送信
  q.submit([&](handler &h){
    // グローバルメモリーに割り当てられたバッファーへのデバイスアクセサーを作成
    accessor A(buf_a, h, read_only);
    accessor B(buf_b, h, read_only);
    accessor C(buf_c, h, write_only);
    // デバイスカーネル本体をラムダ関数として指定
    h.parallel_for(range<1>(N), [=](auto i){
      C[i] = A[i] + B[i];
    });
  });
}
```

カーネル呼び出しは並列に実行

カーネルは範囲の各要素に対して呼び出される

カーネル呼び出しは呼び出し ID にアクセス可能

ステップ 1: デバイスキューの作成
(開発者はデバイスセレクターでデバイスを指定するか、デフォルトのセレクターを使用)

ステップ 2: バッファーの作成
(ホストとデバイスそれぞれのメモリー用)

ステップ 3: (非同期) 実行コマンドの送信

ステップ 4: デバイス上のバッファーデータにアクセスするバッファーアクセサーの作成

ステップ 5: 実行するカーネル (ラムダ関数) の送信

ステップ 6: カーネルの記述

完了!
結果は `buf_c` バッファーを破棄する際にベクトル `c`にコピーバックされる

カスタム・デバイス・セレクター

以下はデバイス・セレクター・ヒューリスティックを使用した派生 device_selector の例で、返される整数値が CPU やほかのアクセレーターよりも高いため GPU が優先される

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class my_device_selector : public device_selector {
public:
  int operator()(const device& dev) const override {
    int rating = 0;
    if (dev.is_gpu() & (dev.get_info<info::device::name>().find("Intel") != std::string::npos))
      rating = 3;
    else if (dev.is_gpu()) rating = 2;
    else if (dev.is_cpu()) rating = 1;
    return rating;
  };
};
int main() {
  my_device_selector selector;
  queue q(selector);
  std::cout << "Device:  " << q.get_device().get_info<info::device::name>() << std::endl;
  return 0;
}
```

非同期実行

SYCL* アプリケーションは 2 つの部分で構成される

1. ホストコード
2. カーネル実行のグラフ

同期操作を除き、これらは独立して実行される

- ホストコードはワークを送信してグラフを構築 (自分で計算することも可能)
- カーネル実行とデータ移動のグラフは SYCL* ランタイムによって管理され、ホストコードから非同期実行される

非同期実行

ホスト

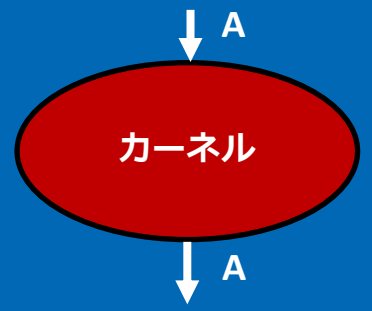
ホストコードの実行

カーネルをグラフにエンキューして続行

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
  std::vector<int> data(N);
  {
    buffer A(data);
    queue q;
    q.submit([&](handler& h) {
      accessor out(A, h, write_only);
      h.parallel_for(N, [=](auto i) {
        out[i] = i;
      });
    });
  }
  for (int i=0; i<N; ++i) std::cout << data[i];
}
```

グラフ

グラフはホストプログラムと非同期に実行



カーネル間の暗黙の依存関係

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue q;

  q.submit([&](handler& h) {
    accessor out(A, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor out(A, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor out(B, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor in(A, h, read_only);
    accessor inout(B, h);
    h.parallel_for(R, [=](id<1> i) {
      inout[i] *= in[i]; }); });
}
```

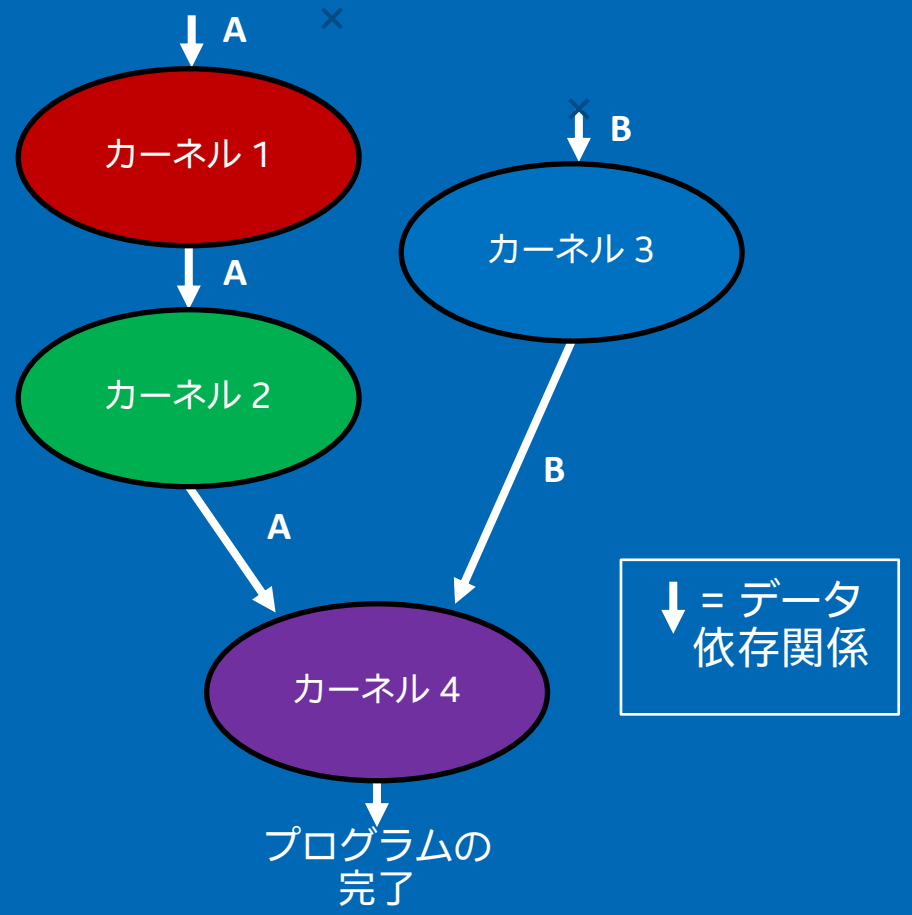
カーネル 1

カーネル 2

カーネル 3

カーネル 4

データと制御の依存関係を自動的に解決!



ホストアクセサー

- ホストバッファーのアクセスターゲットを使用するアクセサー
- コマンド・グループ・スコープ外で作成される
- アクセスするデータはホスト上で利用可能
- ホスト・アクセサー・オブジェクトを構築して、データをホストに戻して同期するために使用される

同期 – ホストアクセサー

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N = 16;

int main() {
  std::vector<double> v(N, 10);
  queue q;

  buffer buf(v);
  q.submit([&](handler& h) {
    accessor a(buf, h)
    h.parallel_for(N, [=](auto i) {
      a[i] -= 2;
    });
  });

  host_accessor b(buf, read_only);
  for (int i = 0; i < N; i++)
    std::cout << b[i] << "¥n";
  return 0;
}
```

バッファーがベクトルに格納されたデータの所有権を持つ

ホストアクセサーの作成はブロッキング呼び出しであり、いずれかのキューの同じバッファーを変更するすべてのエンキューされたカーネルが実行を完了し、このホストアクセサーを介してデータがホストで利用可能になった後にのみリターンする

同期 – バッファーの破棄

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N=16;

void dpcpp_code(std::vector<double> &v, queue &q){
  buffer buf(v);
  q.submit([&](handler& h) {
    accessor a(buf, h);
    h.parallel_for(N, [=](auto i) {
      a[i] -= 2;
    });
  });
}

int main() {
  std::vector<double> v(N, 10);
  queue q;
  dpcpp_code(v,q);
  for (int i = 0; i < N; i++)
      std::cout << v[i] << "¥n";
  return 0;
}
```

バッファーの作成は別の関数スコープ内で行われる

実行がこの関数スコープ外になると、バッファー・デストラクターが呼び出され、データの所有権が破棄され、データがホストメモリーにコピーバックされる


intel