SYCL* の基本

C++ with SYCL* プログラミング

ヘテロジニアス・コンピューティングに C++ with SYCL* を活用する方法の紹介

目的

- SYCL* プログラムをコンパイルする
- 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
- 統合共有メモリー (USM) モデルとバッファー・メモリー・モデルについて説明する
- アクセラレーター・デバイスに計算をオフロードする完全な SYCL* プログラムを記述する
- SYCL* プログラミング機能を使用して低レベルのハードウェア機能にアクセスする

C++ with SYCL*

- 異なるベンダーのヘテロジニアス・ハードウェアに対応したプログラミングが可能
- ホストコードとカーネルコードを含むシングルソースで CPU、GPU、FPGA、その他のアクセラレーター・デバイスへオフロード
- オープン・スタンダードの C++ と Khronos* SYCL* ベース

SYCL* の oneAPI 実装

SYCL* の oneAPI 実装

= C++ と SYCL* 標準と拡張

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

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


int main() {
  sycl::queue q;
  int *data = sycl::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";
  sycl::free(data, q);
  return 0;
}
```

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

SYCL* プログラムをコンパイルする

インテルの CPU と GPU 向けにコンパイルするには

- インテル® oneAPI DPC++/C++ コンパイラーを含むインテル® oneAPI ベース・ツールキットをインストール
- 環境変数を設定して、icpx コンパイラーで -fsycl オプションを指定して、以下のように 1 つまたは複数の C++/SYCL* ソースファイルをコンパイル

```
source /opt/intel/oneapi/setvars.sh

icpx -fsycl test.cpp
```

その他の GPU 向けにコンパイルするには

- オープンソースの LLVM コンパイラーをビルド: github.com/intel/llvm (英語)

SYCL* のクラス

計算をデバイスにオフロードするために必要な、
重要な SYCL* クラスの説明

キュー

- sycl::queue はデバイスにワークを投入するためのメカニズム
- 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 1 つのデバイスにマップ

```
sycl::queue q;

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

デバイス

- device クラスはアクセラレーターの能力を表す
- デバイスクラスには複数のデバイスが作成される SYCL* 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

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

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 (コンパイラーが最適なものを選択):	`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_v);`
	`queue(gpu_selector_v);`
	`queue(ext::intel::fpga_selector_v);`
	`queue(accelerator_selector_v);`

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 {`
	`  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

```
sycl::queue q;
q.submit([&](sycl::handler& h) {
  h.parallel_for(N, [=](auto i) {
    c[i] = a[i] + b[i];
  });
});
```

SYCL* 言語とランタイム

- SYCL* 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++){
    c[i] = a[i] + b[i];
});
```

parallel_for でアクセラレーターへオフロード

```
q.parallel_for(1024, [=](auto i){
    c[i] =  a[i] + b[i];
});
```

基本並列カーネル

デバイスに実行をオフロードするカーネルの**最も簡単な**記述方法。ハードウェア・リソースへの実行のマップは**制御できず**、コンパイラー実装によって行われる

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
});
```

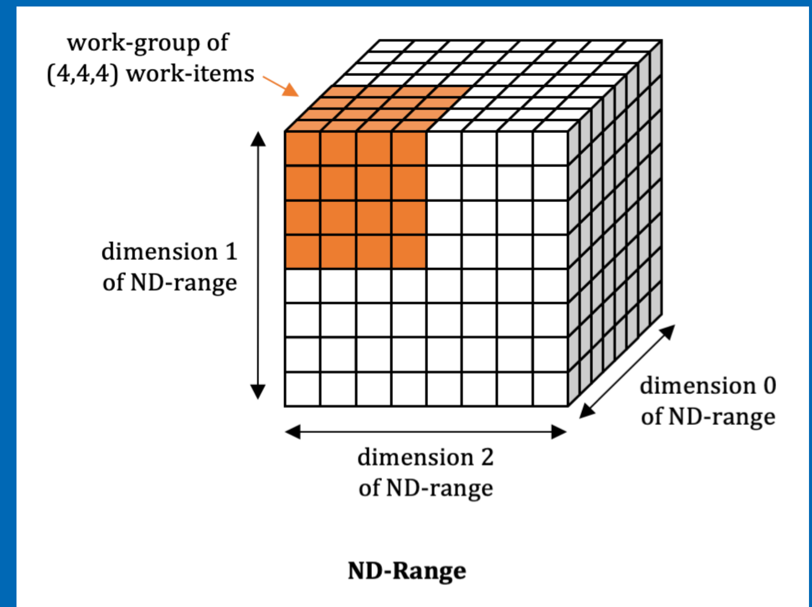
```
h.parallel_for(range<1>(1024), [=](item<1> item){
          auto idx = item.get_id();
          auto R = item.get_range();
          // デバイスで実行するコード
});
```

ND-Range カーネル

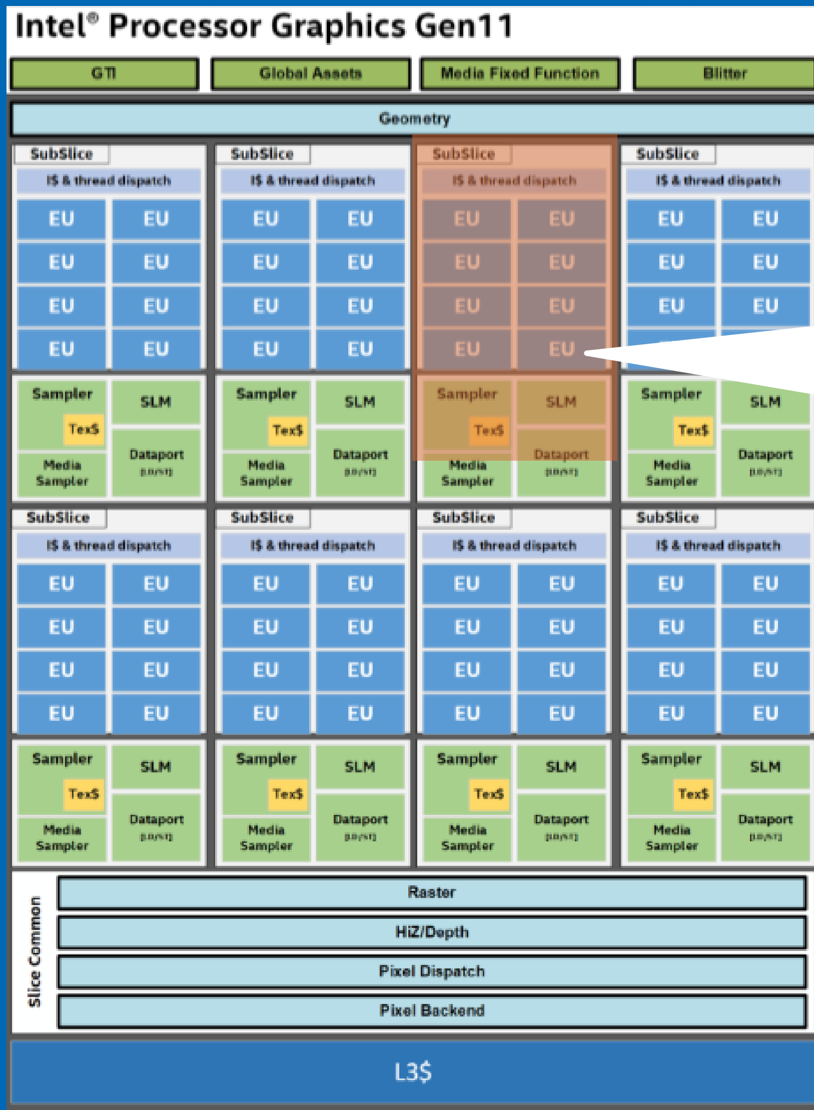
基本並列カーネルは for ループを容易に並列化できるが、ハードウェア·レベルでパフォーマンスを最適化できない

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

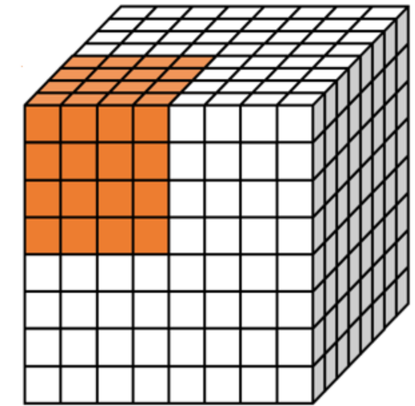
- 反復空間全体をワークグループと呼ばれる小さなグループに分割して、ワークグループ内のワークアイテムをハードウェア上の単一の計算ユニットにスケジュールする
- カーネル実行をワークグループにグループ化することで、リソースの使用を制御し、ワークをバランス良く分散



ND-Range カーネル



ワークグループの実行はハードウェア上の実行ユニットにマップされる



ハードウェアの実行ユニット数に応じて、複数のワークグループの同時実行が可能

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
});
```

グローバル・サイズ (range<1>(1024))

ワークグループ・サイズ (range<1>(64))

- 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、グローバル・インデックス、ワークグループ・インデックス、グループ ID などを照会できる

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·メモリー·モデルを使用できる

- **バッファー·メモリー·モデル** – SYCL* カーネルが使用できる 1 次元、2 次元、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しなければならない
- **統合共有メモリー (USM)** – 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

buffer クラスは SYCL* カーネルが使用できる 1 次元、2 次元、3 次元の共有配列を定義し、accessor クラスを使用してアクセスしなければならない

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
- 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

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

統合共有メモリー (USM)

統合共有メモリー (USM) によりホストとデバイスで同じポインター参照を使用する
共有割り当てが可能になり、ホストとデバイス間で**暗黙的にデータが移動**される

- sycl::malloc_shared はホストとデバイスの両方がアクセスできるメモリーを割り当てる
- ホストはこのデータを変更できる
- デバイスもカーネルコードで同じデータを変更できる
- wait() メソッドは**ブロッキング呼び出し**であるため、キューが実行を完了した後にリターンする
- 結果データはホストで利用可能

```
sycl::queue q;
auto data = sycl::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] << " ";
sycl::free(data, q);
```

統合共有メモリー

統合共有メモリーでは**デバイスメモリーの割り当て**も可能で、ホストとデバイス間で**明示的にデータを移動**できる

- **sycl::malloc_device** はデバイスにメモリーを割り当てる
- **memcpy** はホストからデバイスにデータをコピーする
- デバイスはカーネルコードでデータを変更できる
- **memcpy** はデバイスからホストへデータをコピーバックする
- 結果データはホストで利用可能
- **wait()** メソッドは**ブロッキング呼び出し**であるため、キューが実行を完了した後にリターンする

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

適切なメモリーモデルを選択する

SYCL* のバッファーは強力で洗練されている

- 抽象化を問題なく適用でき、バッファーが開発の妨げにならない場合に使用
- カーネルの依存関係は暗黙的に処理される
- 2 次元/3 次元のデータ構造が扱いやすい

USM は使い慣れたポインターベースの C++ インターフェイスを提供

- C++ コードを SYCL* へ移行する際に変更を最小限に抑えることができて便利
- カーネルの依存関係は明示的に処理する必要がある
- 共有割り当ては迅速に機能を実現するが、デフォルトで最高のパフォーマンスを引き出すことを目的としていないため、パフォーマンスのためには明示的にデータを移動するデバイス割り当てが推奨される

SYCL* メモリーモデルの参考資料

以下のモジュールで詳細な SYCL* メモリーモデルのトレーニングとサンプルコードが提供されている

- SYCL* のバッファーとアクセサーの詳細
- SYCL* の統合共有メモリー
- SYCL* のグラフ･スケジュールとデータ管理

SYCL* コードの構造

- SYCL* プログラムには sycl/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 <sycl/sycl.hpp>
using namespace sycl;
```

SYCL* コードの構造

```
void sycl_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];
    });
  });
}
```

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

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

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

以下はヒューリスティックを使用したカスタム·デバイス·セレクターの例で、返される整数値が CPU やほかのアクセラレーターよりも高いため GPU が優先される

```
#include <sycl/sycl.hpp>
using namespace sycl;
class my_device_selector {
public:
  int operator()(const device& dev) const {
    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

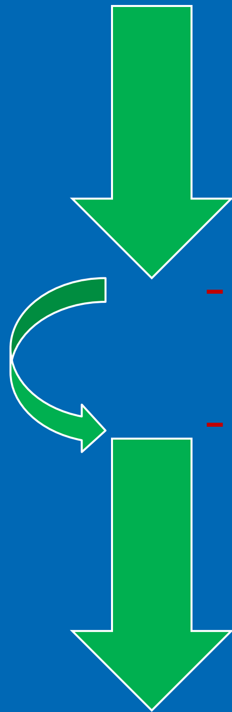
同期操作を除き、これらは独立して実行される

- 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

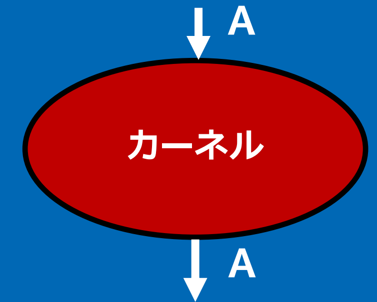
ホスト

グラフ




```
#include <sycl/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];
}
```

グラフはホストプログラムと非同期に実行



カーネルコード

```
h.parallel_for(N, [=](auto i){
    // すべてのワークアイテムのデバイスで実行するコード
    // C++
});
```

- デバイス上で実行するカーネルコードは、すべて C++ 言語機能を使用して記述できる
  - SYCL* カーネルが実行するヘテロジニアス･デバイスの制限により、カーネルコード内で使用できるベース C++ 言語機能には一定の制限がある
- SYCL* には低レベルのハードウェアにアクセスするプログラミング機能もあり、パフォーマンスのチューニングが可能

カーネルコード

SYCL* には、低レベルのハードウェアにアクセスしたり、カーネル・プログラミングを簡素化するプログラミング機能がある

共有ローカルメモリー	デバイスはローカルメモリーに専用のリソースを持っていることがあり、ローカルメモリー経由で通信するほうがグローバルメモリー経由で通信するよりもパフォーマンスが高くなる
サブグループ	同時に実行されるワークグループ内の関連するワークアイテムのサブセット
グループ・アルゴリズム	グループ・アルゴリズムは、最適化されたアルゴリズムのライブラリーを提供し、これらはワークグループとサブグループで使用できる
アトミック操作	アトミック操作により、データ競合を引き起こすことなくメモリー位置への同時アクセスが可能
カーネル・リダクション	parallel_for に reduction オブジェクトを導入してカーネルのリダクションを簡素化

SYCL* カーネル機能の参考資料

以下のモジュールで詳細な SYCL* カーネルコード機能のトレーニングとサンプルコードが提供されている

- SYCL* サブグループ
- SYCL* ローカルメモリーとアトミック操作
- SYCL* カーネル･リダクション

リソース

C++ with SYCL* プログラミングの重要なリソース

リソース

- SYCL* の基本トレーニング･モジュール
  - https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/Jupyter/oneapi-essentials-training (英語)
- oneAPI GPU 最適化ガイド
  - https://www.isus.jp/products/oneapi/oneapi-gpu-optimization-guide-released/
- SYCL* コードの例
  - https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL (英語)

リソース

- インテル® oneAPI コンパイラー、ライブラリー、ツールのダウンロードとインストール
  - https://www.xlsoft.com/jp/products/intel/oneapi/index.html
- オープンソースの SYCL* コンパイラーのビルド
  - https://github.com/intel/llvm (英語)
- SYCL* 仕様
  - https://www.isus.jp/others/sycl-spec-japanese-released/

intel®