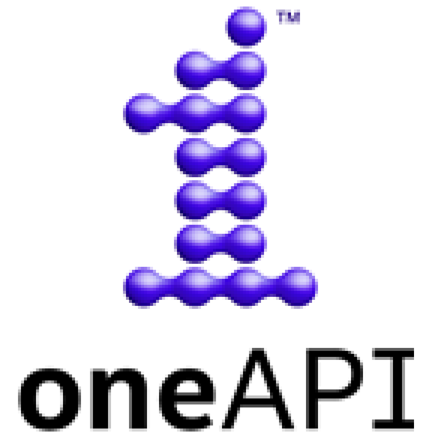


データ並列 C++ と SYCL 2020

データ並列 C++ と SYCL 2020 機能による
単一ソースの異種プログラミング

インテル コーポレーション
デベロッパー・エバンジェリスト
Rakshith Krishnappa、Praveen Kundurthy

iSUS 編集部　すがわら きよふみ

内容

- DPC++ と SYCL の背景と最新情報
- DPC++ プログラムの構造
- コードの例
  - バッファー・メモリー・モデルと同期
  - USM の暗黙的および明示的なデータ移動
  - サブグループのシャッフルと集合
  - DPC++ のリダクション

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

DPC++ = ISO C++ と Khronos SYCL

パフォーマンス、移植性、生産性

- ハードウェアの機能を活用して高速なコンピューティングを実現
- ターゲット・ハードウェア間でコードの再利用が可能、特定のアクセラレーター向けのカスタム・チューニングを行うことが可能
- 単一アーキテクチャー専用のソリューションに代わる、オープンな業界共通のソリューションを提供

C++ および SYCL ベース

- C++ の生産性の利点を提供、一般的で使い慣れた C および C++ 構造を使用
- The Khronos Group の SYCL を継承し、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・プロジェクトを通じて言語の拡張を推進

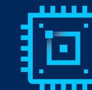
- データ並列プログラミングを簡素化する拡張機能を提供
- オープンな共同開発により継続的に進化

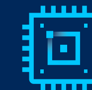
oneAPI DPC++/C++ コンパイラーとランタイム

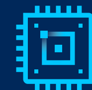
SYCL ソースコード + DPC++ 拡張

Clang/LLVM

DPC++ ランタイム


CPU


GPU


FPGA

インテル® oneAPI DPC++ ライブラリー (インテル® oneDPL)

- 3 つのコンポーネント:
  1. **標準 C++ API:** DPC++ カーネルでテストおよびサポートされる
  2. **Parallel STL:** DPC++ の実行ポリシーで拡張された C++17 アルゴリズム
  3. **STL 拡張:** 追加のアルゴリズム、クラス、イテレーター

```
sycl::queue q;
std::vector<int> v(N);
std::sort(oneapi::dpl::execution::make_device_policy(q), v.begin(), v.end());
```

- C++17 アルゴリズムを使用するコード、または Thrust ライブラリーなどに推奨

https://spec.oneapi.com/versions/latest/elements/oneDPL/source/index.html (英語) を参照

intel.

インテル® DPC++ 互換性ツール

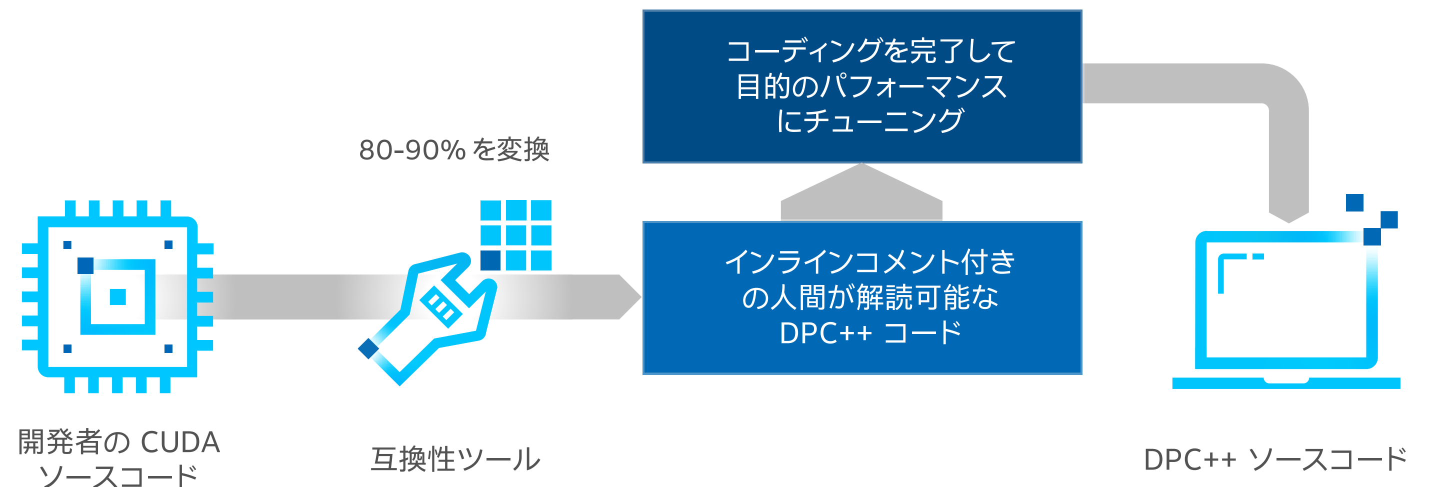
コードの移行時間を最小化

すでに CUDA で記述されているコードを DPC++ に移行する開発者を支援、可能な場合は人間が解読可能なコードを生成

通常はコードの 80-90% を自動的に移行

開発者がアプリケーションの移植を完了できるように支援するインラインコメントを提供

インテル® DPC++ 互換性ツールの使用フロー



SYCL 2020 仕様

- 2021 年 2 月に SYCL 2020 仕様がリリースされました
- 以下を含む主要機能がサポートされます:
  - 統合共有メモリー
  - リダクション
  - モダンアトミック
  - サブグループ
  - グループ·アルゴリズム (リダクション、スキャンなど)
  - 拡張性と相互運用性メカニズム
- (まだ) 利用可能な SYCL 2020 準拠の実装は存在しません

SYCL* 2020 仕様 (リビジョン3) 日本語版 : https://www.isus.jp/wp-content/uploads/pdf/sycl-2020_JA.pdf

SYCL に移行する理由

- プラグマ/ディレクティブから (OpenMP や OpenACC など)
  - SYCL の並列処理は明示的であり、コンパイラーによるループ変換の結果ではありません
  - SYCL は C++ の機能 (イテレーター、テンプレート、ラムダ関数) と容易に統合できます
- 独自のプログラミング・モデルから (CUDA、HIP など)
  - SYCL は業界標準ですが、ハードウェア・ベンダーによって主導されるものではありません
- 移植性フレームワークから (Kokkos、RAJA など)
  - 複数企業によるコンパイラー開発とサポート (インテル、Xilinx、Codeplay)
  - SPIR-V を受け入れるターゲットに移植でき、Clang バックエンドにも移植可能

SYCL 1.2.1、DPC++ および SYCL 2020

DPC++ 拡張	説明	SYCL 2020 での等価な機能
C ++標準ライブラリー	std:: classes (例: std::complex) のサポート	いいえ
データ·フロー·パイプ	FPGA パフォーマンス·チューニング	いいえ
明示的な SIMD	SIMD アーキテクチャーのチューニング	いいえ
拡張アトミック	atomic_ref	はい
グループ·アルゴリズム	ワークグループとサブグループのブロードキャスト、リダクション、スキャンなど	はい
順序付けされたキュー	インオーダー·キュー·プロパティー	はい
ParallelFor の簡略化 キューのショートカット 名前なしカーネルラムダ	一般的なケースの簡略化構文	はい
リダクション	リダクション·カーネル	はい
サブグループ	パフォーマンス·チューニング向けのサブグループ·クラス	はい
USM	統合アドレス空間 (ポインターを使用)、管理されたメモリー、明示的な割り当てと転送	はい

拡張の完全なリストは、https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions (英語) を参照

intel.

デコーダーリング: OpenMP、CUDA、OpenCL、SYCL

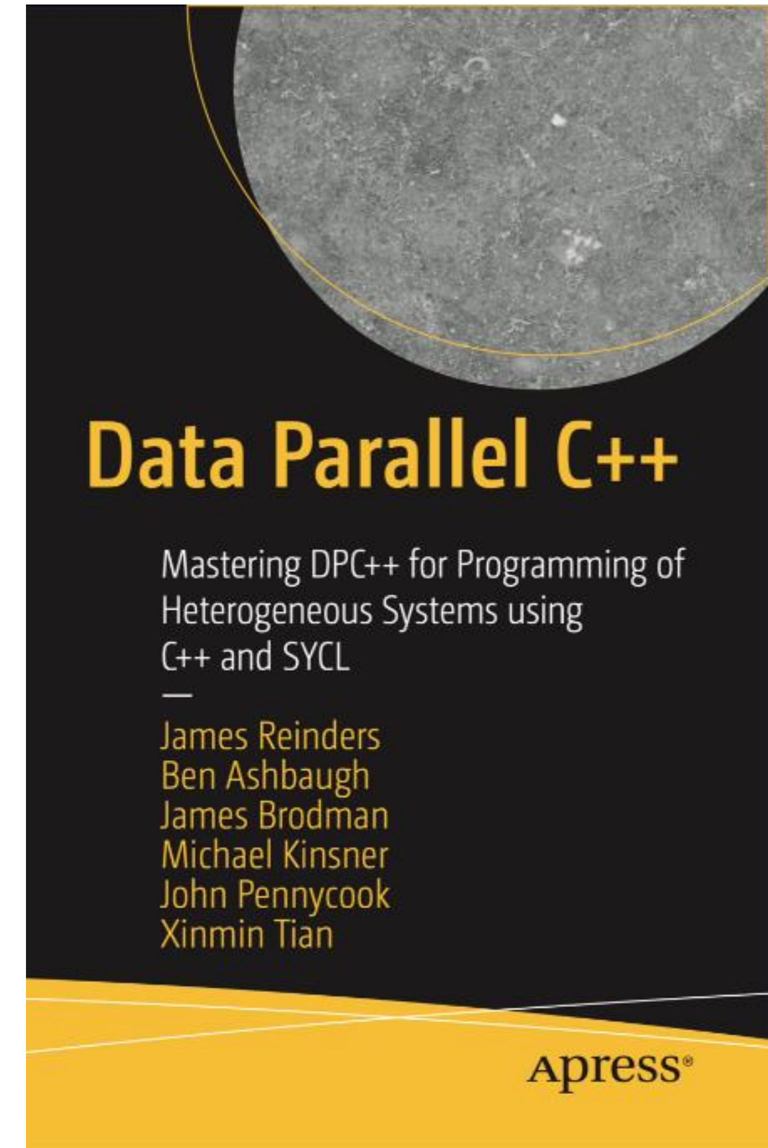
OpenMP (従来の)	OpenMP (Clang SPMD)	CUDA	OpenCL	SYCL	ハードウェアへのマッピング
チーム	チーム	スレッドブロック	ワークグループ	ワークグループ sycl::group	同じ「場所」で実行されるハードウェア・スレッドのチーム
スレッド	N/A	ワープ	サブグループ	サブグループ sycl::sub_group	ハードウェア・スレッド
SIMD ループの反復	スレッド	スレッド	ワーク項目	ワーク項目 sycl::nd_item	SIMD レーン (または、ハードウェア・スレッドごとにインターリーブされた「実行スレッド」)

- 上記は大まかな同等機能を示しています。ベンダー間で「コア」と「ハードウェア・スレッド」など、ハードウェアに相当する用語の対応付けが正確ではありません
- OpenMP の実装は分かれており、OpenMP スレッドを「実行スレッド」に割り当てているものもあることに注意してください

データ並列 C++ ブック

- 入手可能な書籍とコード例:
  - https://www.apress.com/us/book/9781484255735 (英語)

- 幅広い話題をカバーしています
  - SYCL の主要コンセプトを紹介
    (カーネル、バッファー、アクセサーなど)
  - 各種アーキテクチャー (CPU、GPU、FPGA) の紹介とプログラミング方法
  - DPC++ メモリーモデルのような複雑なトピックを詳しく説明します



無料の電子書籍と有料の印刷物の両方で利用できます

まとめ

- SYCL は C++ にヘテロジニアス･コンピューティングをもたらします
  - Khronos Group による標準化
  - 負担なしで OpenCL の最大の機能を提供します
  - エコシステムが成熟するにつれて、HPC 向けの機能/機能拡張が増加することが期待されます
- インテルはオープンソースの SYCL コンパイラー (DPC++) 開発に取り組んでいます
  - https://github.com/intel/llvm (英語) で提供されます
  - 最終目的は Clang/LLVM の更新です

DPC++ プログラムの構造

- **内容**
  - コードを実行する場所を決定します
  - データ転送と同期
  - DPC++ 実行モデルとメモリーモデル
- **演習**
  - 虚数乗法

DPC++ コードの構造

```
void dpcpp_code(int* a, int* b, int* c) {
  // デバイスキューの設定
  queue q;
  // 入出力ベクトル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));
  // キューへコマンドグループ関数オブジェクトを送信します
  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: デバイスキューを作成
(開発者は、デバイスセレクターを介してデバイスタイプを指定するか、デフォルトセレクターを使用できます)

ステップ 2: バッファーを作成
(ホストとデバイスの両方のメモリーを表現できます)

ステップ 3: (非同期) 実行向けにコマンドグループを送信します

ステップ 4: デバイスで使用されるバッファーを表現するアクセサーを作成します

ステップ 5: カーネル関数と起動パラメーター (グループサイズなど) を指定します

ステップ 6: デバイスで実行するコードを指定します

カーネル呼び出しは並列に実行されます

カーネルはレンジ内の要素ごとに呼び出されます

カーネル呼び出しは呼び出し ID にアクセスできます

完了です!
結果は buf_c バッファーが破棄されると、ベクトル c にコピーされます

デバイスへ送信します

- **デバイス**はシステム内の特定のアクセラレーターを表します
- ワークはデバイスに直接送信されるのではなく､デバイスに関連付けられた**キュー**に送信されます
- 特定のデバイス向けのキューを作成するには **device_selector** が必要です

```
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;
```

メモリーモデル

バッファーはホストとデバイス間で共有されるデータをカプセル化します

アクセサーは、バッファーに格納されているデータへのアクセスを提供し、グラフのデータ依存関係を作成します

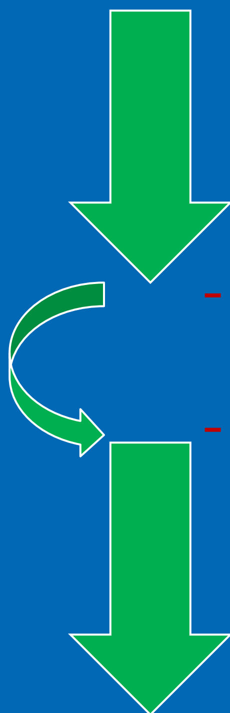
統合共有メモリー (USM) は、メモリーを管理する代替のポインターベースのメカニズムを提供します。詳細については以降で説明します

```
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] << " ";
```

非同期実行

ホスト

ホストコードの実行

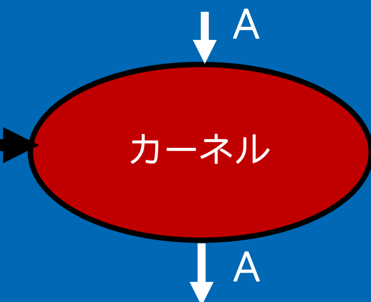


カーネルをキューに投入してグラフ化して処理を続行します

```
#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は非同期に実行されます



非同期実行

```
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

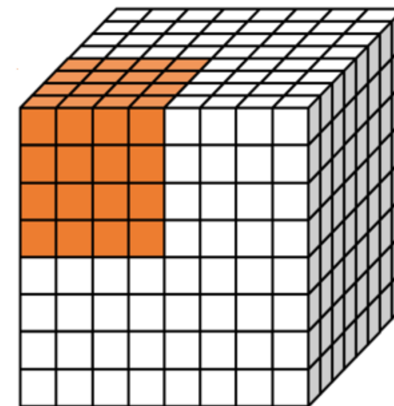
} カーネル 3

} カーネル 4

データとコントロールの依存関係は
ランタイムによって解決されます

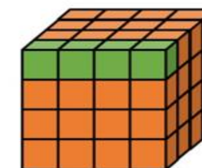
ハードウェアへの割り当て (インテルの GEN11 グラフィックス)

ワークグループ内のすべてのワーク項目は、固有のローカルメモリーを持つ単一のサブスライスにスケジュールされます



サブグループ内のすべてのワーク項目は、単一の EU スレッドで実行されます

サブグループ内の各ワーク項目は、SIMD レーン/チャネルに割り当てられます



要約: 重要なクラス

クラス	機能
sycl::device	SYCL カーネルを実行できる CPU、GPU、FPGA、または他のデバイスを表します
sycl::queue	カーネルを送信 (エンキュー) できるキューを表します 同じ sycl::device に複数のキューがマップされることがあります
sycl::buffer	ランタイムがホストとデバイス間で転送できる割り当てをカプセル化します
sycl::handler	バッファーをカーネルに接続するコマンド·グループ·スコープを定義する際に使用します
sycl::accessor	特定のカーネルのアクセス要件 (リード、ライト、リードライトなど) を定義する際に使用されます
sycl::range, sycl::nd_range, sycl::id, sycl::item, sycl::nd_item	実行レンジとそのレンジ内の個別の実行エージェントを表現します

ハンズオン・コーディング

バッファー・メモリー・モデルと同期

DPC++/SYCL 2020 の新機能

- **内容**
  - 統合共有メモリー (USM)
  - サブグループ
  - リダクション
- **演習**
  - USM
  - サブグループ集合とシャッフル
  - リダクション·カーネル

DPC++ 構文と SYCL 2020 構文

- SYCL 1.2.1 への DPC++ 拡張構文と SYCL 2020 で採用される構文は異なることがあります
- ハンズオンの資料では、現在の DPC コンパイラーとの互換性のため、DPC++ 拡張構文を使用します
- SYCL 2020 でサポートされる一部の機能は、オープンソース・コンパイラーですでに利用できます

統合共有メモリー (USM)

USM を使用すると、ポインターを介した割り当てが可能となり、ホストとデバイスの両方で同じポインターを使用できます

統合共有メモリーの設定

ホストが初期化します

デバイスが変更できます

ホストが出力します

```
queue q;
int *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);
```

統合共有メモリー (USM)

USM 割り当てを作成するには 3 つの方法があります

タイプ	説明	ホストでアクセス可能か?	デバイスでアクセス可能か?
`sycl::malloc_device`	デバイスメモリーの割り当て プログラマーは、ホストとデバイス間でデータを明示的に転送する必要があります	いいえ	はい
`sycl::malloc_host`	ホストメモリーの割り当て カーネルは割り当てられたメモリーに直接アクセスできます	はい	はい
`sycl::malloc_shared`	ホストメモリーとデバイスメモリー間の割り当てを移行できます 実装が異なると、ホストとデバイスが割り当てられたメモリーに同時にアクセスできるかどうかは、異なる条件が提供されることがあります	はい	はい

USM – 明示的なデータ転送

malloc_device() はデバイスにメモリーを割り当てますが、ホストからは直接アクセスできません

q.memcpy() を使用してホストからデバイスにメモリーを明示的にコピーします

デバイスカーネルは、同じ (デバイス) ポインターを使用できます

q.memcpy() を使用してデバイスからホストにメモリーを明示的にコピーします

```
queue q;

int data[N];
for (int i = 0; i < N; i++) data[i] = 10;

int *data_device = 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] << std::endl;
free(data_device, q);
```

USM – 暗黙的なデータ転送

malloc_shared() は、ホストとデバイス間で移行できるメモリーを割り当てます

デバイスカーネルは同じポインターを使用できます

ホストは同じポインターを使用してメモリーに直接アクセスできます

```
queue q;

int *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] << std::e
ndl;
free(data, q);
```

USM – データ依存関係

- バッファーを使用する場合、カーネル間のデータ依存関係は、アクセサーの使用状況に応じて SYCL ランタイムによって追跡されます

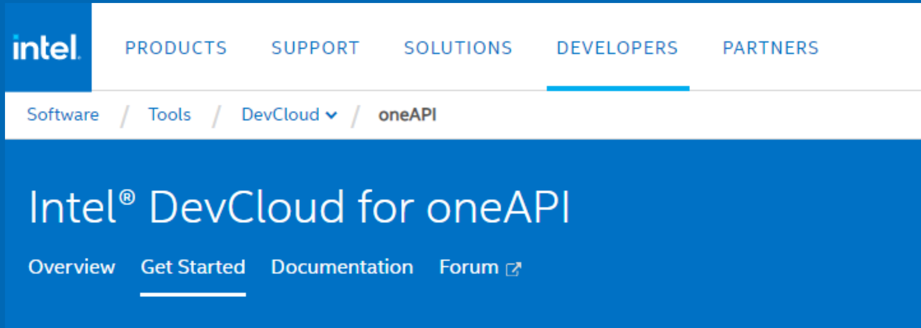
- 統合共有メモリーを使用する場合、データ依存関係は、プログラマーが解決する必要があります
  - データにアクセスする前に **q.wait()** によって明示的にホスト/デバイスを同期します
  - **sycl::event** オブジェクトを使用して、カーネル間の依存関係を指定します
    または
    インオーダー・キューを使用して、カーネル間の暗黙の依存関係を追加します

ハンズオン・コーディング

USM の暗黙的および明示的なデータ移動

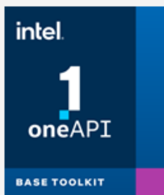
インテル® DevCloud for oneAPI へアクセスしてください

https://devcloud.intel.com/oneapi/get_started/



Explore Intel oneAPI Toolkits in the DevCloud

These toolkits are for performance-driven applications—HPC, IoT, advanced rendering, deep lear modules, and go deeper with developer guides.



Intel® oneAPI Base Toolkit

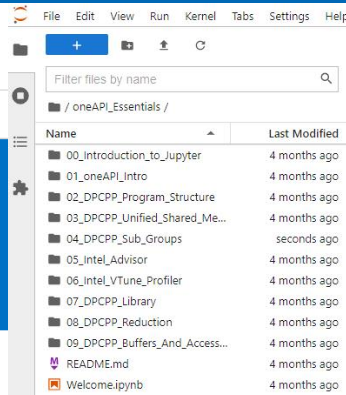
Build and deploy high-performance, data-centric applications across div

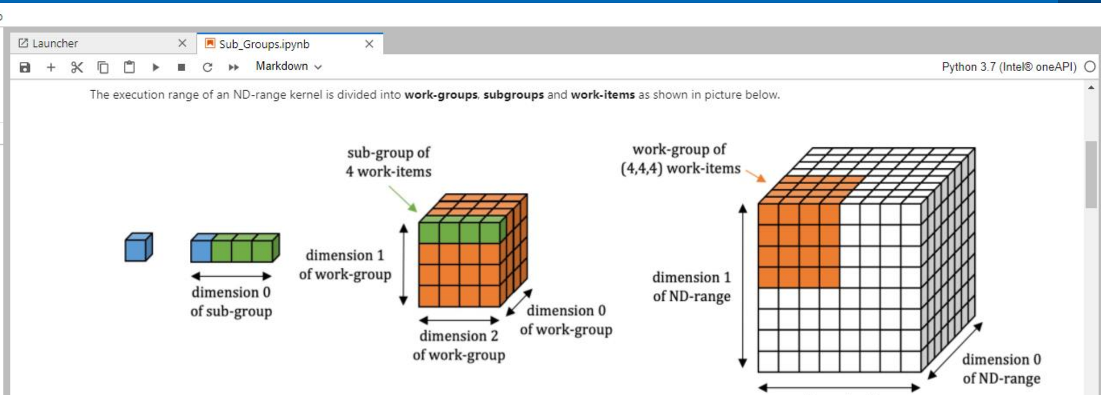
Get Started with your first Sample

View Training Modules

The toolkit includes:

- Intel® oneAPI Collective Communications Library
- Intel® oneAPI Data Analytics Library
- Intel® oneAPI Deep Neural Networks Library
- Intel® oneAPI DPC++ Compiler
- Intel® oneAPI DPC++ Library

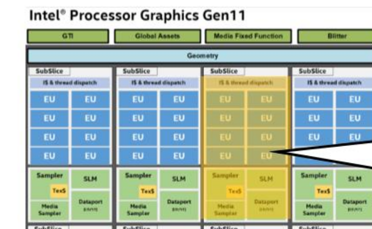


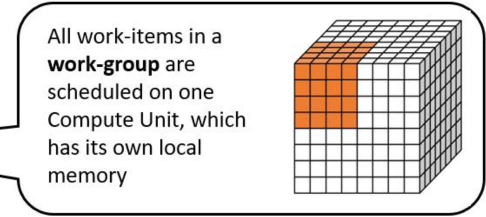


How a Subgroup Maps to Graphics Hardware

Work-item	Represents the individual instances of a kernel function.
Work-group	The entire iteration space is divided into smaller groups called work-groups, work-items within a work-group are scheduled on a single compute unit on hardware.
Subgroup	A subset of work-items within a work-group that are executed simultaneously, may be mapped to vector hardware. (DPC++)

The picture below shows how work-groups and subgroups map to **Intel® Gen11 Graphics Hardware**.





usm.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```cpp
#include <iostream>
#include <cl/sycl.hpp>

using namespace sycl;

static const int N = 256;

int main() {
  queue q{property::queue::in_order()};
  std::cout << "Device : " << q.get_devic
td::endl;

  // USM allocation and initialize
  int *data = malloc_shared<int>(N, q);
  for (int i = 0; i < N; i++) data[i] = 10;

  // 3 kernals task submit data without dependency.
  q.parallel_for (range<1>(N), [=](id<1> i){data[i] += 2; });
  q.parallel_for (range<1>(N), [=](id<1> i){data[i] += 3; });
  q.parallel_for (range<1>(N), [=](id<1> i){data[i] += 5; }).wait();

  for (int i = 0; i < N; i++)
    std::cout << data[i] << " ";
  std::cout << std::endl;

  free(data, q);
  return 0;
}
~
```

Intel(r) oneAPI Tools

```
C> dpcpp usm.cpp

C> usm
Device : Intel(R) Graphics [0x3e98]
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 2
0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 2
0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 2
0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

キューにプロパティーを指定して、明示的に順番を制御

usm2.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```
int main() {
   queue q; // {property::queue
   std::cout << "Device : " << q
std::endl;

   // USM allocation and initia
   int *data1 = malloc_shared<in
   int *data2 = malloc_shared<in
   for (int i = 0; i < N; i++)

   // 3 kernals task submit data without dependency.
   auto e1 = q.parallel_for (range<1>(N), [=](id<1> i){data1[i] += 2; });
   auto e2 = q.parallel_for (range<1>(N), [=](id<1> i){data2[i] += 3; });
   q.parallel_for (range<1>(N), {e1, e2}, [=](id<1> i){data1[i] += data2[i]; }).w
ait();

   for (int i = 0; i < N; i++)
      std::cout << data1[i] << " ";
   std::cout << std::endl;

   free(data1, q); free(data2, q);
   return 0;
}
```

Intel(r) oneAPI Tools

```
C> dpcpp usm2.cpp

C> usm2
Device : Intel(R) Graphics [0x3e98]
25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 2
5 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 2
5 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 2
5 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
```

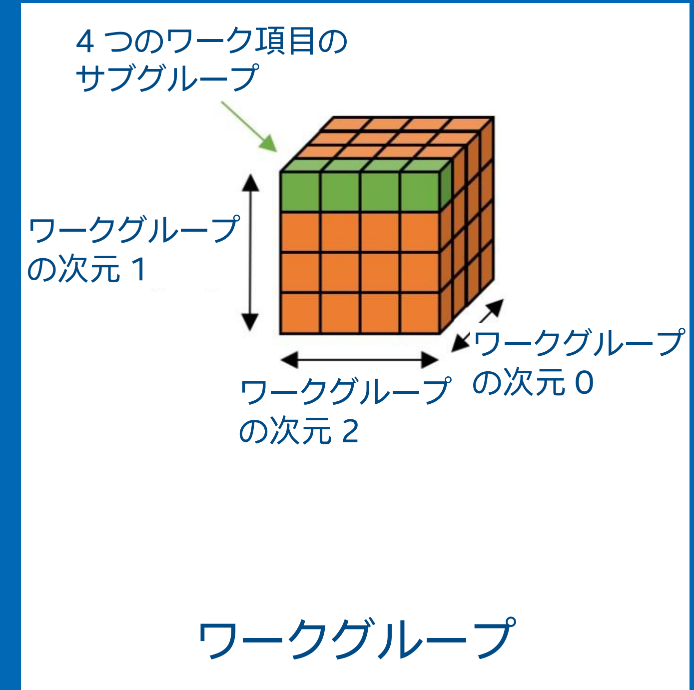
キューのプロパティーを指定
せずに、イベントを使用

サブグループ

ワークグループ内のワーク項目のサブセットは、追加の保証付きで実行され、多くの場合 SIMD ハードウェアにマップされます

サブグループ内のワーク項目は、シャッフル操作により直接通信できます

サブグループはまた、サブグループ集合へのアクセスを提供します (リダクション、スキャン、any/all など)



サブグループ

sub_group クラス

サブグループのハンドルは、**get_sub_group()** を使用して nd_item から取得できます

次のような関数を提供します:

- サブグループに関する詳細情報を**照会**します
- **シャッフル**操作を実行するか、**集合**関数を使用します

```
q.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item)
{
  auto sg = item.get_sub_group();
  // カーネルコード
});
```

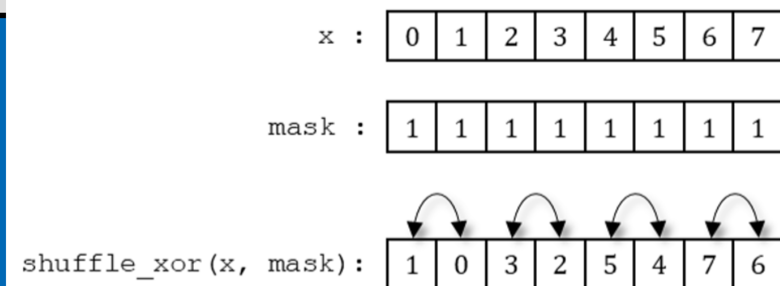
サブグループ

サブグループのシャッフル

- サブグループで最も便利な機能の１つは、**明示的なメモリー操作なしで**個々のワーク項目間で直接通信できることです

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
        auto sg = item.get_sub_group();
        size_t i = item.get_global_id(0);

        /* シャッフル */
        //data[i] = sg.shuffle(data[i], 2);
        //data[i] = sg.shuffle_up(0, data[i], 1);
        //data[i] = sg.shuffle_down(data[i], 0, 1);
        data[i] = sg.shuffle_xor(data[i], 1);
});
```



サブグループ

グループ集合

- 集合関数は、密接に関連する**一般的な並列パターン**の実装を提供します
- 集合は、ワークグループとサブグループの両方で利用できます

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
      auto sg = item.get_sub_group();
      size_t i = item.get_global_id(0);

      /* 集合 */
      data[i] = reduce(sg, data[i], ONEAPI::plus<>());
      //data[i] = reduce(sg, data[i], ONEAPI::maximum<>());
      //data[i] = reduce(sg, data[i], ONEAPI::minimum<>());
});
```

サブグループのサイズを指定

サブグループのサイズは、カーネルごとに個別に設定できます。
利用可能なサブグループ・サイズのセットは、ハードウェアに固有です

```
q.parallel_for(range<1>(N),
              [=](id<1> id) [[intel::reqd_sub_group_size(16)]] {
  // カーネルコード
});
```

サブグループのサイズは、**sub_group** クラス を使用しないカーネルでもチューニングできます (例えば、SIMD 幅とレジスターの使い方をチューニング)

SYCL 2020 のサブグループ

SYCL 2020 は、DPC++ のサブグループのシャッフルを新しいアルゴリズムで置き換えます

DPC++ 拡張

```
sycl::ONEAPI::sub_group sg = it.get_sub_group();

auto a = sg.shuffle_down(x, 1);
auto b = sg.shuffle_up(x, 1);
auto c = sg.shuffle(x, id);
auto d = sg.shuffle_xor(x, mask);
```

メンバー関数としてのシャッフル

SYCL 2020

```
sycl::sub_group sg = it.get_sub_group();

auto a = sycl::shift_group_left(sg, x, 1);
auto b = sycl::shift_group_right(sg, x, 1);
auto c = sycl::select_from_group(sg, x, id);
auto d = sycl::permute_group_by_xor(sg, x, mask);
```

フリー関数としてのシャッフル
C++ に合わせた命名規則を持ちます

https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:algorithms (英語)

ハンズオン･コーディング

サブグループのシャッフルと集合

sub.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```
static constexpr size_t N = 64; // global size
static constexpr size_t B = 64; // work-group size

int main() {
   queue q;
   std::cout << "Device : " << q.get_device().get_info<info

   q.submit ([&](handler &h){
   // setup SYCL stream class to print stdout from device
   auto out = stream(1024, 768, h);

   // nd-range kernel
      h.parallel_for (nd_range<1>(N, B),[=](nd_item<1> item){

      // get sub_group
         auto sg = item.get_sub_group();

      // query sub_group and print sub_group info one per sub_group
         if (sg.get_local_id()[0] == 0){
            out << "sub_group id: " << sg.get_group_id()[0] << " of "
                << sg.get_group_range()[0] << ", size = " << sg.get_local_range()[0]
                << endl;
         }
      });
   }).wait();
}
```

Intel(r) oneAPI Tools

```
C> dpcpp sub.cpp

C> sub
Device : Intel(R) Graphics [0x3e98]
sub_group id: 0 of 4, size = 16
sub_group id: 3 of 4, size = 16
sub_group id: 2 of 4, size = 16
sub_group id: 1 of 4, size = 16
```

サブグループ ID とサイズを表示

Intel(r) oneAPI Tools

```
C> sub
Device : 11th Gen Intel(R) Core(TM) i7-11700K @ 3.60GHz
sub_group id: 0 of 4, size = 16
sub_group id: 1 of 4, size = 16
sub_group id: 2 of 4, size = 16
sub_group id: 3 of 4, size = 16
```

Intel(r) oneAPI Tools

```
C>sub
Device : Intel(R) UHD Graphics 750
sub_group id: 0 of 4, size = 16
sub_group id: 1 of 4, size = 16
sub_group id: 2 of 4, size = 16
sub_group id: 3 of 4, size = 16
```

Intel(r) oneAPI Tools

```
C> sycl-ls
ACC : Intel(R) FPGA Emulation Platform for OpenCL(TM) 1.2 [2021.11.3.0.17_160000]
CPU : Intel(R) OpenCL 2.1 [2021.11.3.0.17_160000]
GPU : Intel(R) OpenCL HD Graphics 3.0 [30.0.100.9667]
GPU : Intel(R) Level-Zero 1.1 [1.1.0]
HOST: SYCL host platform 1.2 [1.2]
```

sub2.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```
int main() {
   queue q;
   std::cout << "Device : " << q.get_de

   auto sg_sizes = q.get_device().get_i
   std::cout << "Supported Sub-group Si
   for (int i = 0; i < sg_sizes.size()

   auto max_sg_size = std::max_element(
   std::cout << "Max Sub-group Size           << max_sg_size[0] << std::endl;

   // Initialize data array using usm
   int *data = malloc_shared<int>(N, q);
   for(int i = 0; i < N; i++) data[i] = i;

   // use oarallel_for and sub_groups
   q.parallel_for (nd_range<1>(N, B), [=](nd_item<1> item)[[intel::reqd_sub_group_size(S)]]{
      auto sg = item.get_sub_group();
      auto i = item.get_global_id(0);

      // write sub_group to ZERO except first location for each sub_group
      if (sg.get_local_id()[0] != 0) data[i] = 0;
   }).wait();
```

Intel(r) oneAPI Tools

```
C> sub2
Device : Intel(R) Graphics [0x3e98]
Supported Sub-group Sizes : 8 16 32
Max Sub-group Size        : 32
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 48 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 64 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 96 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 112 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 128 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 144 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 160 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 176 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
192 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 208 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 224 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 240 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

それぞれのサブグループの先頭以外にゼロを書き込む

sub3.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```
int main() {
   queue q;
   std::cout << "Device : " << q.

   auto sg_sizes = q.get_device()
   std::cout << "Supported Sub-gr
   for (int i = 0; i < sg_sizes.s

   auto max_sg_size = std::max_el
   std::cout << "Max Sub-group Si

   // Initialize data array using
   int *data = malloc_shared<int>
   for(int i = 0; i < N; i++) dat

   // use oarallel_for and sub_groups
   q.parallel_for (nd_range<1>(N, B), [=](nd_item<1> item){
      auto sg = item.get_sub_group();
      auto i = item.get_global_id(0);

      // reverse the order of items in sub_group using shuffle_xor
      data[i] = sg.shuffle_xor(data[i], sg.get_max_local_range() - 1);
   }).wait();
```

Intel(r) oneAPI Tools

```
C> sub3
Device : Intel(R) Graphics [0x3e98]
Supported Sub-group Sizes : 8 16 32
Max Sub-group Size        : 32
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 31 30 29 28 27 26 25 24 23 22 21 20 19 18
17 16 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 95 94 93 9
2 91 90 89 88 87 86 85 84 83 82 81 80 111 110 109 108 107 106 105 104 103 102 10
1 100 99 98 97 96 127 126 125 124 123 122 121 120 119 118 117 116 115 114 113 11
2 143 142 141 140 139 138 137 136 135 134 133 132 131 130 129 128 159 158 157 15
6 155 154 153 152 151 150 149 148 147 146 145 144 175 174 173 172 171 170 169 16
8 167 166 165 164 163 162 161 160 191 190 189 188 187 186 185 184 183 182 181 18
0 179 178 177 176 207 206 205 204 203 202 201 200 199 198 197 196 195 194 193 19
2 223 222 221 220 219 218 217 216 215 214 213 212 211 210 209 208 239 238 237 23
6 235 234 233 232 231 230 229 228 227 226 225 224 255 254 253 252 251 250 249 24
8 247 246 245 244 243 242 241 240
```

サブグループのデータの順番を入れ替える

リダクション

リダクションは、複数の値を組み合わせて不特定の順番で単一の値を生成します

- 計算とアクセラレーター･ハードウェアへの性質上、**リダクションの並列化**には注意が必要です
- DPC++ はヘテロジニアス･プログラミングのリダクションに**単純化**されたアプローチを導入します

グループのリダクション

ワークグループ集合を使用して、それぞれのワークグループのすべての要素の合計を計算できます

```
q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item){
  auto wg = item.get_group();
  size_t i = item.get_global_id(0);

  // ワークグループ·レデュースを使用して、ワークグループのすべての要素を加算
  int sum = reduce(wg, data[i], ONEAPI::plus<>());

  // レデュースされた値を使用して何らかの処理を行います
  ...
});
```

グループ間のリダクション (別名リダクション·カーネル)

ワークグループ集合を使用して、それぞれのワークグループのすべての要素の合計を計算できます

部分結果は**追加カーネル**を介して組み合わせできます

```
q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item){
  auto wg = item.get_group();
  size_t i = item.get_global_id(0);

  // ワークグループ·レデュースを使用して、ワークグループのすべての要素を加算
  int sum_wg = reduce(wg, data[i], ONEAPI::plus<>());

  // 各ワークグループの最初の位置にワークグループの合計を書き込みます
  if (item.get_local_id(0) == 0) data[i] = sum_wg;
});
```

```
q.single_task([=](){
  int sum = 0;
  for (int i = 0; i < N; i += B) {
      sum += data[i];
  }
  data[0] = sum;
});
```

グループ間のリダクション (別名リダクション·カーネル)

DPC++ はリダクション·カーネル専用の抽象化を導入しています

リダクション·オブジェクトをカプセル化:

1. リダクション変数
2. オプションの単位元
3. リダクション操作

```
queue q;
int *data = malloc_shared<int>(N, q);
for (int i = 0; i < N; i++) data[i] = i;

int *sum = malloc_shared<int>(1, q);
sum[0] = 0;

q.parallel_for(nd_range<1>{N, B},
                  ONEAPI::reduction(sum, 0, ONEAPI::plus<>()),
                  [=](nd_item<1> it, auto& tmp) {
     int i = it.get_global_id(0);
     tmp += data[i];
}).wait();

std::cout << "Sum = " << sum[0] << std::endl;
```

SYCL 2020 リダクション

```
myQueue.submit([&](handler& cgh) {

  // リダクションの入力値は標準アクセサー (または USM ポインター) です
  auto inputValues = accessor(valuesBuf, cgh);

  // リダクション･セマンティクスを使用して変数を表現する一時オブジェクトを作成します
  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  auto maxReduction = reduction(maxBuf, cgh, maximum<>());

  // parallel_for は 2 つのリダクション操作を行います
  cgh.parallel_for(range<1>{1024},
    sumReduction, maxReduction,
    [=](id<1> idx, auto& sum, auto& max) {
      sum += inputValues[idx];
      max.combine(inputValues[idx]);
  });
});
```

https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:reduction (英語)

ハンズオン･コーディング

DPC++ のリダクション

intel.

reduction.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```
// USM allocation and initialize
int *data = malloc_shared<int>(N, q);
for (int i = 0; i < N; i++) data[i] = i;

// 3 kernals task submit data without dependency.
q.parallel_for (nd_range<1>(N, B), [=](nd_item<1> item){
   auto wg = item.get_group();
   auto i = item.get_global_id(0);

   // Adds all elements in work-group using work-group reduce
   int sum_wg = ONEAPI::reduce(wg, data[i], ONEAPI::plus<>());

   // write work-group sum to first location for each work-group
   if (item.get_local_id(0) == 0) data[i] = sum_wg;
});

q.single_task([=](){
   int sum = 0;
   for (int i = 0; i < N; i += B) sum += data[i];
   data[0] = sum;
}).wait();

std::cout << "Sum = " << data[0] << "¥n";
```

Intel(r) oneAPI Tools

```
C> reduction
Device : Intel(R) Graphics [0x3e98]
Sum = 523776
```

USM データのリダクション 1

intel

reduction2.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```cpp
static constexpr size_t N = 1024; // global size
static constexpr size_t B = 128; // work-group size

int main() {
   queue q{property::queue::in_order()};
   std::cout << "Device : " << q.get_device().get_info<info::device::name>() << std::endl;

   // Initialize data array using usm
   int *data = malloc_shared<int>(N, q);
   for (int i = 0; i < N; i++) data[i] = i;

   // implicit usm for writing sum value
   int *sum = malloc_shared<int>(N, q);
   *sum = 0;

   // 3 kernals task submit data without dependency.
   q.parallel_for (nd_range<1>{N, B},ONEAPI::reduction(sum, 0, ONEAPI::plus<>()),[=](nd_item<1> it, auto& temp){
      auto i = it.get_global_id(0);
      temp.combine(data[i]);
   }).wait();

   std::cout << "Sum = " << *sum << "¥n";

   free(data, q); free(sum, q);
```

Intel(r) oneAPI Tools

```
C> reduction2
Device : Intel(R) Graphics [0x3e98]
Sum = 523776
```

USM データのリダクション 2

minmax.cpp (~¥Desktop¥dpcpp¥multi) - VIM

```
int main() {
  queue q{property::queue::in
  std::cout << "Device : " <<

  // Initialize data array us
  auto result = malloc_shared
  auto data = malloc_shared<int>(N, q);

  // Initialize input data with randum numbers
  srand(time(0));
  for (int i = 0; i < N; ++i) data[i] = rand() % 256;
  std::cout << "Input Data:¥n";
  for (int i = 0; i < N; i++) std::cout << data[i] << " "; std::cout << "¥n¥n";

  pair<int, int> operator_identity = {std::numeric_limits<int>::max(), std::numeric_limits<int>::min()};
  *result = operator_identity;
  auto reduction_object = ONEAPI::reduction(result, operator_identity, ONEAPI::minimum<pair<int, int>>());

  // parallel_for with user defined reduction object
  q.parallel_for (nd_range<1>{N, B}, reduction_object, [=](nd_item<1> item, auto& temp){
    auto i = item.get_global_id(0);
    temp.combine({data[i], i});
  }).wait();

  std::cout << "Minimun value and index = " << result->val << " at " << result->idx << "¥n";
```

Intel(r) oneAPI Tools

```
2 117 242 12 252 165 175 48 190 17 144 94 5 12 109 157 8 2 212 209 111 8 81 234
183 183 236 91 215 59 244 205 36 167 251 21 114 139 34 100 124 201 151 190 134 1
64 91 174 64 26 73 55 238 116 84 13 198 27 16 6 153 173 116 203 182 80 222 211 1
2 71 87 146 142 195 24 237 220 251 78 242 31 126 27 198 52 200 221 222 14 20 178
 114 85 114 60 47 83 25 161 45 17 254 214 32 189 166 30 210

Minimun value and index = 0 at 549
```

USM データの最小値とそのインデックスを取得

参考資料

- SYCL* 2020 API リファレンス･ガイド
  https://www.isus.jp/wp-content/uploads/pdf/SYCL-2020-reference-guide-JA.pdf
- SYCL* 2020 仕様 (リビジョン3)
  https://www.isus.jp/wp-content/uploads/pdf/sycl-2020_JA.pdf
- oneAPI 仕様リリース 1.0 Rev3 Part1
  https://www.isus.jp/products/oneapi/oneapi-spec-japanese/
- インテル® oneAPI プログラミング･ガイド 2021.3
  https://www.isus.jp/products/oneapi/oneapi-programming-guide-released/
- インテル® oneAPI ポーティング･ガイド
  https://www.isus.jp/products/c-compilers/oneapi-porting-guide-japanese/
- oneAPI GPU 最適化ガイド
  https://www.isus.jp/products/oneapi/oneapi-gpu-optimization-guide-released/

法務上の注意事項と最適化に関する注意事項

インテルのテクノロジーを使用するには、対応したハードウェア、ソフトウェア、またはサービスの有効化が必要となる場合があります。詳細については、OEM または販売店にお問い合わせいただくか、http://www.intel.co.jp/ を参照してください。

実際の費用と結果は異なる場合があります。

インテルは、サードパーティーのデータについて管理や監査を行っていません。ほかの情報も参考にして、正確かどうかを評価してください。

最適化に関する注意事項: インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。
注意事項の改訂 #20110804 https://software.intel.com/content/www/us/en/develop/articles/optimization-notice.html#opt-jp

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。

SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。 結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。構成の詳細は、補足資料を参照してください。性能やベンチマーク結果について、さらに詳しい情報をお知りになりたい場合は、https://www.intel.com/benchmarks (英語) を参照してください。

性能の測定結果はシステム構成の日付時点のテストに基づいています。また、現在公開中のすべてのセキュリティー・アップデートが適用されているとは限りません。詳細は、システム構成を参照してください。絶対的なセキュリティーを提供できる製品またはコンポーネントはありません。

本資料は、(明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず) いかなる知的財産権のライセンスも許諾するものではありません。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

intel®