OpenCL* 向けの設計を DPC++ へ移行

この記事は、インテル® デベロッパー・ゾーンに公開されている「Migrating OpenCL™ Designs to DPC++」の日本語参考訳です。

概要

この記事は、Khronos OpenCL* とデータ並列 C++ (DPC++) 標準の類似点と相違点について説明し、開発者が既存の OpenCL* アプリケーションを DPC++ に簡単に移行できるようにします。この記事は、読者が OpenCL* に習熟していることを前提としています。

データ並列 C++

DPC++ は、オープンで標準化ベースのクロスアーキテクチャー・プログラミング言語です。この言語により、開発者は CPU、GPU、および FPGA などのアクセラレーターを含むさまざまなハードウェア・ターゲットから高いパフォーマンスを引き出すことができます。同時に機能的な移植性を提供するため、ユーザーはこれらのアーキテクチャー全体でコードを再利用できます。DPC++ は、業界を超えたオープンな標準化ベースの統合プログラミング・モデルである oneAPI の一部です。

DPC++ は、最新の ISO C++ 標準、および Khronos SYCL* 標準をベースに構築され、プログラムの可能性と最適化を容易にする機能が拡張されています。これにより DPC++ は、最新の C++ の生産性の利点と、ヘテロジニアス環境でのデータとタスク並列処理のパフォーマンスの利点をもたらします。

OpenCL* との基本的な比較

DPC++ には OpenCL* をベースに構築された上位レベルの抽象化レイヤーである SYCL* が含まれいるため、DPC++ と OpenCL* を比べるとほとんどの基本概念は同じであり、OpenCL* と DPC++ 間で同等の構造を容易にマッピングできます。さらに、DPC++ は単一ソースの C++ の利便性、生産性、および柔軟性を備えています。カーネルコードがホストコードに埋め込まれているため、プログラマーはコーディングが容易になり、コンパイラーはコードが実行されるデバイスにかかわりなくプログラム全体を解析して最適化することができます。

DPC++ は、これを単一ソースの複数コンパイラー・パス (SMCP) で実現します。SMCP (Single-source Multiple-Compiler Pass) では、単一のソースファイルがさまざまなコンパイラーによって、さまざまなデバイス向けに解析され、バイナリーが生成されます。多くの場合、これらのバイナリーは単一の実行ファイルに結合されます。例えば、ホスト・コンパイラーはアプリケーションのネイティブコードを生成し、デバイス・コンパイラーは同じソースファイルを解析してカーネル向けのデバイスバイナリーを生成します。

OpenCL* から DPC++ への移行

OpenCL* と DPC++ は、どちらの言語も同様のプログラミング・モデルと機能を持つため、移行作業は比較的容易です。重要なホスト API 関数の呼び出しが少ないため、通常、DPC++ では、ホスト API 関数への呼び出しが少ないため、カーネルを実行するコードの行数は少なくなります。

ほとんどの OpenCL* アプリケーション開発者は、デバイスへのカーネルオフロードに伴う冗長なセットアップ・コードの必要性を認識しているでしょう。DPC++ を使用すると、OpenCL* C コードに関連する大部分のセットアップ・コードなしで、シンプルで現代的な C++ ベースのアプリケーションを開発できます。これにより、習得の労力が軽減され、並列化の実装に集中できます。

単純な例を見てみましょう。以下に OpenCL* と DPC++ の両方で、ベクトル加算カーネルとアプリケーション・プログラムを示します。OpenCL* および DPC++ のコードは、カーネルをデフォルトデバイスへオフロードする同じタスクを実行します。しかし DPC++ では、DPC++ のデフォルトにより、コードが大幅に簡素化されています。

ベクトル加算の例

OpenCL*

```
const char *kernelsource =  (R"(__kernel void vecAdd(  __global int *a,
                                                       __global int *b,
                                                       __global int *c) {
                                 int i = get_global_id(0);
                                 c[i] = a[i] + b[i];
                              } )");

void opencl_code(int* a, int* b, int* c, int N) {
cl_int err;
cl_platform_id myplatform;
cl_device_id mydevice;
// 最初のプラットフォームとデバイスを選択
err=clGetPlatformIDs(1, &myplatform, NULL);
err=clGetDeviceIDs(myplatform, CL_DEVICE_TYPE_DEFAULT, 1, &mydevice, NULL);
// コンテキストとキューを設定
cl_context mycontext = clCreateContext(0, 1, &mydevice, NULL, NULL, &err);
cl_command_queue myq = clCreateCommandQueue(mycontext, mydevice, 0, &err);
// ソースコードからプログラムを作成して、コンパイルして、カーネル・オブジェクトを生成
cl_program myprogram = clCreateProgramWithSource(mycontext, 1,
    (const char **) & kernelsource, NULL, &err);
 clBuildProgram(myprogram, 1, &mydevice, NULL, NULL, NULL);
 cl_kernel mykernel = clCreateKernel(myprogram, "vecAdd", &err);

// バッファーを作成し、A と B をデバイスへ書き込み
size_t nbytes=N*sizeof(int);
cl_mem buf_a= clCreateBuffer(mycontext, CL_MEM_READ_ONLY, nbytes, NULL, &err);
cl_mem buf_b= clCreateBuffer(mycontext, CL_MEM_READ_ONLY, nbytes, NULL, &err);
cl_mem buf_c= clCreateBuffer(mycontext, CL_MEM_WRITE_ONLY, nbytes, NULL, &err);
err=clEnqueueWriteBuffer(myq, buf_a, CL_TRUE, 0, nbytes,  a, 0, NULL, NULL);
err=clEnqueueWriteBuffer(myq, buf_b, CL_TRUE, 0, nbytes,  b, 0, NULL, NULL);
err = clSetKernelArg(mykernel, 0, sizeof(cl_mem), &buf_a);
err = clSetKernelArg(mykernel, 1, sizeof(cl_mem), &buf_b);
err = clSetKernelArg(mykernel, 2, sizeof(cl_mem), &buf_c);
// カーネルを実行し、完了したらデータをホストにコピー
err = clEnqueueNDRangeKernel(myq, mykernel, 1, NULL, &N, NULL, 0, NULL, NULL);
clFinish(myq);
clEnqueueReadBuffer(myq, buf_c, CL_TRUE, 0, nbytes, c, 0, NULL, NULL);
}
```

DPC++

```
void dpcpp_code(int* a, int* b, int* c, int N) {
{
   // キューとバッファーを作成
   queue q;
   buffer<int,1> buf_a(a, range<1>(N));
   buffer<int,1> buf_b(b, range<1>(N));
   buffer<int,1> buf_c(c, range<1>(N));

   q.submit([&](handler &h){
      // アクセサーはカーネルからバッファーへのアクセスを提供
```

```
      auto A=buf_a.get_access<access::mode::read>(h);
      auto B=buf_b.get_access<access::mode::read>(h);
      auto C=buf_c.get_access<access::mode::write>(h);

      // N 要素の 1D 範囲を持つカーネルを並列に起動
      h.parallel_for(range<1>(N), [=](item<1> i) {
         C[i] = A[i]+B[i];
      });
   });
}
```

OpenCL* との相互利用

DPC++ では、すべての OpenCL* の機能は SYCL* API を介してサポートできます。さらに、SYCL* から OpenCL* API を直接呼び出すことができる相互運用性もあります。例えば、SYCL* バッファーは OpenCL* バッファーから構築でき、OpenCL* キューは SYCL* キューから取得できます。そして OpenCL* カーネルは SYCL* プログラムから呼び出すことができます。記事の最後にこれらの例を示します。

プログラミング・モデル

DPC++ プログラミング・モデルは、OpenCL* と非常によく似ており、多くの点で共通です。

プラットフォーム・モデル

DPC++ プラットフォーム・モデルは、OpenCL* をベースにしていますが、追加の抽象化技術が含まれています。DPC++ と OpenCL* の両方で、プラットフォーム・モデルは、1 つ以上のデバイスで実行される計算ワークを調整および制御するホストを指定します。デバイスには、CPU、GPU、FPGA、そして他のアクセラレーターを含めることができます。DPC++ では、常にホストに対応するデバイスがあるため、デバイス・カーネル・コードのターゲットがいつでも利用できることが保証されます。最も効率が良いプログラミングのアプローチは、ホストとデバイスの両方を利用してデバイスとの間のデータ移動レイテンシーを隠匿し、デバイス上で適切なワークロードをアクセラレートします。

デバイスに関連する用語の同等性

次の表は、OpenCL* と DPC++ に関連する用語を GPU ハードウェアにマッピングしています。表から分かるように、OpenCL* と DPC++ は同じ用語を使用しています。

OpenCL*	DPC++	GPU ハードウェア
計算ユニット	計算ユニット	サブスライス (インテル) [デュアル] 計算ユニット (AMD) ストリーミング・マルチプロセッサー (Nvidia)
処理要素	処理要素	実行ユニット (インテル) SIMD ユニット (AMD) SM コア (Nvidia)

実行モデルの同等性

OpenCL* と DPC++ の両方で、階層的な並列実行が可能です。ワークグループ (Work-group)、サブグループ (Subgroup)、およびワーク項目 (Work-item) の概念は、両方の言語で同じです。ワークグループとワーク項目

の間にあるサブグループは、ワークグループ内のワーク項目のグループを定義します。通常、サブグループはサポートされる単一命令複数データ (SIMD) ハードウェアにマッピングされます。サブグループ内のワーク項目の同期は、ほかのサブグループ内のワーク項目とは関連なく発生する可能性があり、サブグループはグループ内のワーク項目間の通信操作を発生させます。

OpenCL*	DPC++	GPU ハードウェア
Work-group	Work-group	スレッドグループ
Subgroup	Subgroup	実行ユニットスレッド (ベクトル・ハードウェア)
Work-item	Work-item	SIMD レーン (チャネル)

ホスト API

ホスト API は OpenCL* と DPC++ の両方で使用され、プラットフォームとデバイスを管理します。次の表は、ホスト API を定義するインクルード・ファイルを示します。

インクルード・ファイル	
OpenCL*	**DPC++**
CL/opencl.h または CL/cl.h CL/cl.hpp (C++ バインディング) CL/cl2.hpp(C++ バインディング v2.x)	CL/sycl.hpp

プラットフォーム・レイヤー API

プラットフォーム・レイヤー API はデバイスの実行環境を設定します。プラットフォーム・レイヤー API によって実行されるタスクには、次のものがあります。

1. ホストからデバイスと機能の検出を許可
2. 計算デバイスの照会、選択、および初期化
3. 計算コンテキストの生成

OpenCL* では、プラットフォーム (ベンダー)、デバイス (アクセラレーター)、およびコンテキストを選択するため、明示的にプラットフォーム・レイヤー API を呼び出す必要があります。DPC++ では、OpenCL* と同様のことを行うか、DPC++ ランタイムによりデフォルトのプラットフォームとデバイスを選択することができます。DPC++ で利用可能なアクセラレーター・デバイスのカテゴリーは、OpenCL* と似ています。

DPC++ でデバイスを明示的に指定するには、**sycl::device_selector** 抽象化クラスから派生するサブクラスを使用します。DPC++ は、組込みのデバイスセレクターを提供することでコードをすばやく実行できるようにします。

利用可能な組込みデバイスセレクター:

- **default_selector**: 実装で定義され、利用可能なデバイスが存在しない場合はホストを選択します。
- **host_selector**: ホストデバイスを選択して、常に有効なデバイスを返します。

- **cpu_selector**: CPU デバイスの選択を試みます。
- **gpu_selector**: GPU デバイスの選択を試みます。
- **intel::fpga_selector**: FPGA デバイスの選択を試みます。
- **intel::fpga_emulator_selector**: FPGA カーネルのデバッグ向けに、FPGA エミュレーション・デバイスの選択を試みます。
- **accelerator_selector**: その他の利用可能なアクセラレーターの選択を試みます。

アプリケーションがシステムで利用可能な GPU の中から特定の GPU など、特定のデバイスを選択する必要がある場合、**sycl::device_selector** クラスから継承した独自のデバイスセレクターを作成することもできます。

プラットフォーム・レイヤー API	
OpenCL*	**DPC++**

`// 最初のプラットフォーム ID を取得`
`cl_platform_id myp;`
`err=clGetPlatformIDs(1, &myp, NULL);`
 	
`// デバイスを取得`
`cl_device_id mydev;`
`err=clGetDeviceIDs(myp,`
`  CL_DEVICE_TYPE_GPU, 1, &mydev, NULL);`
 	
`// コンテキストを作成`
`cl_context context;`
`context = clCreateContext(NULL, 1,`
`  &mydev, NULL, NULL, &err);`
	`using namespace sycl;`
	
	`// (オプション) デバイスセレクターを準備`
	`default_selector selector;`
	`// または`
	`// 開発およびデバッグ向け`
	`host_selector selector;`
	`// または`
	`cpu_selector selector;`
	`// または`
	`gpu_selector selector;`
	`// または`
	`intel::fpga_selector selector;`
	`// または独自セレクターを記述`
	`my_custom_selector selector;`

コマンドキュー

コマンドキューは、OpenCL* と DPC++ でホストがデバイスのアクションを要求するメカニズムです。それぞれのコマンドキューは 1 つのデバイスに関連付けられ、ホストはコマンドをキューに送信します。同じデバイスに複数のキューを割り当てることができます。

OpenCL* では、コマンドキューは **deviceID** から作成され、タスクは **clEnqueue**… コマンドによりデバイスに送信できます。

DPC++ では、キューはデバイスセレクターから作成されます。デバイスセレクターが用意されていない場合、ランタイムは単純に **default_selector** を使用します。その後、タスクは submit コマンドを使用してキューに送信できます。

キュー	
OpenCL*	**DPC++**

`cl_command_queue q;`
`q = clCreateCommandQueue(…);`
`clEnqueue…(q, …);`
	`// キュークラス`
	`queue q(selector);`
	`q.submit([&](handler& h) {`
	`  // コマンド・グループ・コード`
	`});`

データ管理

OpenCL* でデータを管理するには、デバイスメモリーの抽象化されたビューを示すバッファーやイメージを作成する必要があり、関数を使用してデバイスメモリーの読み取り、書き込み、またはコピーを行います。DPC++ でもバッファーとイメージを操作できますが、統合共有メモリー (USM) を使用してポインターを介してデバイスメモリーにアクセスすることができます。DPC++ でバッファーとイメージを使用する場合、カーネル内のデータにアクセスするメカニズムとしてアクセサーが使用されます。アクセサーを使用すると、暗黙的なデータ依存関係が検出され、それに応じてカーネル実行がスケジュールされます。

データ管理	
OpenCL*	**DPC++**
clCreateBuffer()	バッファークラス
clEnqueueReadBuffer()	アクセサーで暗黙的に処理されるか、handler::copy() または queue::memcpy() で明示的に処理されます
clEnqueueWriteBuffer()	
clEnqueueCopyBuffer()	
N/A	アクセサークラス

データ管理の例

OpenCL*

```
cl_mem buf_a = clCreateBuffer(context, CL_MEM_READ_ONLY, N*sizeof(int), NULL, &err);
cl_mem buf_b = clCreateBuffer(context, CL_MEM_WRITE_ONLY, N*sizeof(int), NULL, &err);

clEnqueueWriteBuffer(q, buf_a, CL_TRUE, 0, N*sizeof(int),  a, 0, NULL, NULL);
// カーネルを起動するコード
…
clEnqueueReadBuffer(q, buf_b, CL_TRUE, 0, N*sizeof(int), b, 0, NULL, NULL);
```

DPC++

```
{
   // バッファーを作成
   // スコープを介して自動的に取得されたデータの読み書き
   buffer<int,1> buf_a(a, range<1>(N));
   buffer<int,1> buf_b(b, range<1>(N));
   q.submit([&](handler &h){
        auto A=buf_a.get_access<access::mode::read>(h);
        auto B=buf_b.get_access<access::mode::write>(h);
        h.parallel_for…  {          // カーネルを起動
             B[i] = process(A[i]);
        }
   });
}
```

統合共有メモリー (USM)

DPC++ には、ソースコードで明示的にアクセサーを指定しなくてもホストとデバイス間でメモリーを共有可能にしてプログラミングを簡素化する USM のサポートも含まれます。USM はポインターベースのバッファーへの代替手段を提供します。これにより、DPC++ ではポインターが利用できるため、C++ のポインターベースのプログラムを DPC++ に容易に移行できます。USM をサポートするデバイスは仮想アドレス空間をサポートする必要があり、ホスト上の USM 割り当てルーチンが返すポインター値は、デバイス上で有効であることが保証

されます。USM を使用すると、プログラマーは関数を使用してアクセスを管理したり、イベントを待機するかイベント間の依存関係を通知して依存関係を強制できます。

カーネルの起動

DPC++ では単一ソースに記述できるため、カーネルを指定、コンパイル、および起動する手順は、OpenCL* と DPC++ では異なります。OpenCL* では、通常、カーネルは異なるソースファイルまたはプリコンパイルされたバイナリーにあります。DPC++ では、カーネル関数は、統合されたソースファイルのコマンドスコープ内のカーネルスコープで指定します。DPC++ カーネルは、通常、C++ ラムダ式や関数オブジェクト (ファンクター) 形式で表現されます。

OpenCL* でカーネルを起動するには、最初にソースコードまたはプリコンパイルされたバイナリーからプログラム・オブジェクトを作成し、**clBuildProgram** を実行してカーネルをコンパイルします。次に、**clCreateKernel** コマンドを使用して、コンパイルされたプログラムから特定のカーネルを抽出します。そして、**clSetKernelArg** コマンドでそれぞれのカーネル引数をバッファー、イメージ、または変数にマップします。最後に、**clEnqueueNDRangeKernel** コマンドを使用してカーネルを起動できます。階層的な NDRange の起動が必要な場合、ワーク項目の総数である **global_work_size** と、各ワークグループ内のワーク項目の総数である **local_work_size** を指定する必要があります。

DPC++ でカーネルを起動するのははるかに簡単です。必要なことは、**queue::submit()** 呼び出しで作成されたコマンドグループのスコープ内に、ラムダ式またはファンクターとしてカーネルを記述するだけです。カーネルを実行するコマンドグループの処理関数 (例えば、**parallel_for**、**single_task**、または **parallel_for_work_group**) でラムダ式やファンクターに渡されるカーネル関数を呼び出すと、カーネル関数がデバイスで実行されます。並列カーネルの場合、グローバル範囲、またはグローバルとローカル両方の実行範囲を指定する **nd_range** のいずれかを指定する必要があります。

カーネルの起動	
OpenCL*	**DPC++**
clCreateProgramWithSource/Binary() clBuildProgram() clCreateKernel() clSetKernelArg() clEnqueueNDRangeKernel()	queue::submit() parallel_for() parallel_for_work_group() parallel_for_work_item() single_task()
global_work_size, local_work_size 変数	range クラス nd_range クラス

カーネルの起動の例

OpenCL*

```
cl_program myprogram = clCreateProgramWithSource(…);
clBuildProgram(myprogram…);
cl_kernel mykernel = clCreateKernel(program, “kernel_name”, &err);
clSetKernelArg(mykernel, 0, sizeof(cl_mem), (void *) &a_buf);
clSetKernelArg(mykernel, 1, sizeof(cl_mem), (void *) &b_buf);
clEnqueueNDRangeKernel(queue, mykernel, 2, NULL, global_size, local_size, …);
```

DPC++

```
q.submit([&](handler &h){
     range<2> global(N,N);
     range<2> local(B,B);
     h.parallel_for(nd_range<2>(global,local), [=](nd_item<2> item)
     {
          // カーネルコード
     });
});
```

同期

ホストとデバイスに対応する各種コマンドキュー間で実行する同期機能は、OpenCL* と DPC++ で共通です。ただし、DPC++ では、アクセサーを使用すると暗黙の依存関係が存在します。2 つのカーネルが同じバッファーを使用すると、アクセサーは自動的に SYCL* グラフでデータの依存関係を作成し、2 番目のカーネルは最初のカーネルの完了を待機するように調整します。明示的な同期は、OpenCL* と DPC++ の両方で使用できます。次の表に等価性を示します。

カーネルの起動	
OpenCL*	**DPC++**
clFinish()	queue::wait()
clWaitForEvents()	event::wait()

デバイス・カーネル・コード

OpenCL* と DPC++ のカーネルコードは非常によく似ています。ここでは違いについて説明します。

最初の違いは、カーネル修飾子とアドレス空間修飾子の使い方です。これらの修飾子は、すべてランタイムで抽象化されるため DPC++ では必要ありません。

修飾子	
OpenCL*	**DPC++**
__kernel	N/A
__constant	N/A
__global	N/A
__local	N/A
__private	N/A

カーネル内のワークグループとワーク項目にインデックスを付ける考え方は、OpenCL* と DPC++ で同じです。ただし、インデックス関数の呼び出し方法は若干異なります。DPC++ では、**nd_item** クラスは各種インデックス、サイズ、範囲を返す関数を提供します。**nd_item** クラスには、特定の次元内のインデックスではなく、グローバルまたはローカルの線形インデックスを返す機能が含まれています。**nd_item** クラスは、ワークグループとサブグループに関連する機能をカプセル化する **group** オブジェクトと **sub_group** オブジェクトを返すこともできます。

また DPC++ は、**group** クラスと **sub_group** クラスを通して、プロードキャスト (broadcast)、任意 (any)、すべて (all)、レデュース (reduce)、排他的スキャン (exclusive scan)、包括的スキャン (inclusive scan)、シャッフル (shuffles) などの機能を、ワークグループおよびサブグループに提供します。

カーネルの照会	
OpenCL*	**DPC++**
get_global_id()	nd_item::get_global_id()
get_local_id()	nd_item::get_local_id()
get_group_id()	nd_item::get_group_id()
get_global_size()	nd_item::get_global_range()
get_local_size()	nd_item::get_local_range()
get_num_group()	nd_item::get_num_group()
N/A	get_group()
N/A	get_sub_group()
N/A	get_global_linear_id()
N/A	get_local_linear_id()

インデックス付けと同様に、DPC++ のカーネル同期機能も **nd_item** クラスで提供されます。各関数では、local、global、または global_and_local に設定可能な **fence_space** を渡します。

同期	
OpenCL*	**DPC++**
barrier()	nd_item::barrier()
mem_fence()	nd_item::mem_fence()
read_mem_fence()	nd_item::mem_fence()
write_mem_fence()	nd_item::mem_fence()

カーネルの例

ここでは、行列 c = a x b を計算するタイル化された行列乗算の OpenCL* と DPC++ の実装を比較します。ローカルメモリー **a_tile** と **b_tile** は、グローバルメモリーへのロードとストアを最小化するために使用されます。カーネルは、N x N の 2D グローバルサイズと B x B の 2D ローカルサイズの ND-range カーネルとして起動されます。DPC++ では、最初に **parallel_for_work_group** を使用してワークグループの並列処理を有効にし、次に **parallel_for_work_item** によりワーク項目の並列処理を有効にします。これらの構造と対応するスコープには、バリアが暗黙的に配置されます。

行列乗算カーネルの例

OpenCL*

```
__kernel void matrix_mul(__global float *restrict a,
                         __global float *restrict b,
                         __global float *restrict c)
```

```
{
     __local float a_tile[B][B];
     __local float b_tile[B][B];
     int j=get_global_id(0);
     int i=get_global_id(1);
     int lj=get_local_id(0);
     int li=get_local_id(1);
     for (int kb=0; kb < N/B; ++kb)
          // A と B の行列タイルをローカルメモリーにロード
          a_tile[lj][li] = a[j][kb*B+li];
          b_tile[lj][li] = b[kb*B+lj][i];

          // ローカルメモリーへのロード完了を待機
          barrier(CLK_LOCAL_MEM_FENSE);

          // ローカルメモリーを使用して行列乗算を計算
          for (int k=0; k < B; ++k)
          {
               c[j][i] += a_tile[lj][k] + b_tile[k][li];
          }

          // すべてのワーク項目を確実に完了するためのバリア
          barrier(CLK_LOCAL_MEM_FENCE);
     }
}
```

DPC++

```
h.parallel_for_work_group<class matrix_mul>(range<2>(N/B, N/B),
                                            [=] (group<2> grp) {
float a_tile[B][B];
float b_tile[B][B];
int jb=grp.get_id(0);
int ib=grp.get_id(1);

for (int kb=0;kb<N/B; ++kb) {
     // parallel_for_work_item は、A と B のタイルをロード
     grp.parallel_for_work_item(range<2>(B,B), [&](h_item<2> item) {
          int lj=item.get_logical_local_id(0);
          int li=item.get_logical_local_id(1);
          int j=jb*B+lj;
          int i=ib*B+li;
          // A と B のタイルをローカルメモリーにロード
          a_tile[lj][li] = a[j][kb*B+li];
          b_tile[lj][li] = b[kb*B+lj][i];
     });
// ここに暗黙のバリア
     grp.parallel_for_work_item(range<2>(B,B), [&](h_item<2> item) {
          int lj=item.get_logical_local_id(0);
          int li=item.get_logical_local_id(1);
          int j=jb*B+lj;
          int i=ib*B+li;

          // ローカルメモリーを使用して行列乗算を計算
          for (int k=0; k < B; ++k)
          {
               c[j][i] += a_tile[lj][k] * b_tile[k][li];
          }
      });
// ここにも暗黙のバリア
}
});
```

OpenCL* と DPC++ の相互運用性の例

既存の OpenCL* アプリケーションを DPC++ に移行する場合、開発者は段階的に移行することを望むかもしれません。OpenCL* と DPC++ は、いくつかの方法で相互運用できます。ここでは、そのいくつかを紹介します。

DPC++ プログラムから既存の OpenCL* カーネルを実行

OpenCL* カーネルコードを DPC++ 環境で実行したい場合、SYCL* はそのためのメカニズムを提供します。次に、DPC++ プログラムから OpenCL* カーネルの取得を行う手順を示します。

- デバイスキューで使用される同一コンテキストから、**sycl::program** オブジェクトを作成します。
- **program::build_with_source()** 関数に OpenCL* カーネルのソース文字列を渡してカーネルコードをビルドします。
- DPC++ コマンドキューのスコープ内でカーネル引数のアクセサーを作成します。
- **handler::set_arg()** 関数または **handler::set_args()** 関数を使用して、アクセサーをカーネル引数に渡します。
- プログラム・オブジェクトから取得したカーネル・オブジェクトおよび範囲または NDrange を渡して、**handler::single_task()** 関数または **handler::parallel_for()** 関数を使用してカーネルを起動します。

次の例では、SYCL* 環境で OpenCL* カーネルを実行することを除き、この記事の最初の例と同じベクトル加算を実行します。

DPC++ プログラム内の OpenCL* カーネルの例

```
void dpcpp_code(int* a, int* b, int* c, int N) {
{
  queue q{gpu_selector()};       // GPU ターゲットのコマンドキューを作成
  program p(q.get_context());  // q と同じコンテキストからプログラムを作成

  // OpenCL カーネルをコンパイル。
  // これは、R" で示されるように C++ の文字列として表現される。
  p.build_with_source(R"( __kernel void vecAdd(__global int *a,
                                               __global int *b,
                                               __global int *c)
                          {
                                  int i=get_global_id(0);
                                  c[i] = a[i] + b[i];
                          } )");
    buffer<int, 1> buf_a(a, range<1>(N));
    buffer<int, 1> buf_b(b, range<1>(N));
    buffer<int, 1> buf_c(c, range<1>(N));
    q.submit([&](handler& h) {
            auto A = buf_a.get_access<access::mode::read>(h);
            auto B = buf_b.get_access<access::mode::read>(h);
            auto C = buf_c.get_access<access::mode::write>(h);
            // カーネルの引数としてバッファーを設定
            h.set_args(A, B, C);
            // N 個の要素に対し、p プログラム・オブジェクトから vecAdd カーネルを起動。
            h.parallel_for(range<1> (N), p.get_kernel("vecAdd"));
    });
}
```

DPC++ オブジェクトを OpenCL* オブジェクトに変換

OpenCL* API を使用するように DPC++ プログラムを拡張する場合、DPC++ オブジェクトを相互運用可能な OpenCL* オブジェクトに変換する必要があります。多くの DPC++ オブジェクトには、OpenCL* API で利用できる OpenCL* オブジェクトを取得する **get()** メソッドが用意されています。次の表は、これらのオブジェクトを示します。

get() と DPC++/SYCL* オブジェクト

- cl::sycl::platform::get() -> cl_platform_id
- cl::sycl::context::get() -> cl_context
- cl::sycl::device::get() -> cl_device_id
- cl::sycl::queue::get() -> cl_command_queue
- cl::sycl::event::get() -> cl_event
- cl::sycl::program::get() -> cl_program
- cl::sycl::kernel::get() -> cl_kernel

OpenCL* オブジェクトを DPC++ オブジェクトに変換

OpenCL* オブジェクトを SYCL* コンストラクターで使用し、対応する SYCL* オブジェクトを作成できます。この手法は、SYCL* ランタイムの機能を既存の OpenCL* ソースベースに追加する場合に使用されます。
DPC++ バッファーとイメージは、**get()** から **cl_mem** オブジェクトに変換することはできませんが、バッファーとイメージには **cl_mem** を受け付けるコンストラクターがあることに注意してください。

OpenCL* オブジェクトを使用する DPC++/SYCL* コンストラクター

- cl::sycl::platform::platform(cl_platform_id)
- cl::sycl::context::context(cl_context, …)
- cl::sycl::device::device(cl_device_id)
- cl::sycl::queue::queue(cl_command_queue,…)
- cl::sycl::event::event(cl_event, …)
- cl::sycl::program::program(context, cl_program)
- cl::sycl::kernel::kernel(cl_kernel, …)
- cl::sycl::buffer::buffer(cl_mem, …)
- cl::sycl::image::image(cl_mem, …)

次の例は、OpenCL* オブジェクトの **cl_mem**、**cl_command_queue**、**cl_kernel**、および **cl_context** を使用する DPC++ プログラムを示しています。

OpenCL* オブジェクトを使用した DPC++ プログラム

```
cl_kernel ocl_kernel = clCreateKernel(ocl_program, "vecAdd", &err);
cl_mem ocl_buf_a=clCreateBuffer(ocl_context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
cl_mem ocl_buf_b=clCreateBuffer(ocl_context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
cl_mem ocl_buf_c=clCreateBuffer(ocl_context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
clEnqueueWriteBuffer(ocl_queue, ocl_buf_a, CL_TRUE, 0, bytes, host_a, 0, NULL, NULL );
clEnqueueWriteBuffer(ocl_queue, ocl_buf_b, CL_TRUE, 0, bytes, host_b, 0, NULL, NULL );
{ // DPC++ アプリケーションのスコープ
     // コンテキスト、キュー、カーネル、およびバッファーの SYCL* バージョンを作成
     context sycl_context(ocl_context);
     queue sycl_queue(ocl_queue, sycl_context);
     kernel sycl_kernel(ocl_kernel, sycl_context);
     buffer<int, 1> sycl_buf_a(ocl_buf_a, sycl_context);
     buffer<int, 1> sycl_buf_b(ocl_buf_b, sycl_context);
     buffer<int, 1> sycl_buf_c(ocl_buf_c, sycl_context);
     sycl_queue.submit([&](handler& h) {
          // 各バッファーのアクセサーを作成
          auto a_accessor = sycl_buf_a.get_access<access::mode::read>(h);
          auto b_accessor = sycl_buf_b.get_access<access::mode::read>(h);
          auto c_accessor = sycl_buf_c.get_access<access::mode::write>(h);
          // カーネル引数をアクセサーにマッピング
```

```
            h.set_args(a_accessor, b_accessor, c_accessor);
            // カーネルを起動
            h.parallel_for(r, sycl_kernel);
        });
}
// バッファーの内容を読み取り、ホストの配列に戻す
clEnqueueReadBuffer(ocl_queue, ocl_buf_c, CL_TRUE, 0, bytes, host_c, 0, NULL, NULL );
```

まとめ

OpenCL* と DPC++ は、ともにヘテロジニアス向けの並列オープン言語であり、その機能と特性は類似しているため、既存の OpenCL* アプリケーションを簡単に DPC++ に移行してその機能を利用できます。DPC++ には、単一ソース・プログラミング、データ型に依存しないカーネルの実行、組込みのデフォルト、簡素化された同期など、最新のプログラミング機能が含まれています。これらの機能により、アプリケーションを最初からコーディングするか、既存のコードを移植して、パフォーマンスを最適化して各種アクセラレーターで実行できます。

関連情報

- インテル® DevCloud (英語) で DPC++ およびその他の oneAPI 製品を試す
- インテル® oneAPI 導入ページ (英語)
- インテル® oneAPI プログラミング・ガイド日本語版

製品とパフォーマンス情報

[1] 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