ヘテロジニアス・プログラミング向けのレベルゼロ API の紹介

この記事は英国マンチェスター大学の Dr. Juan Fumero のブログで公開されている「Introduction to Level Zero API for Heterogeneous Programming」を著者の許可を得て翻訳した日本語参考訳です。

重要な点と概要

- レベルゼロは、ヘテロジニアス・アーキテクチャーのプログラミング向けのベアメタル API に近いと言えます。
- レベルゼロは、インテル® oneAPI の一部でありスタンドアロン API として利用できます。
- この記事では、SPIR-V* を使用してインテル® HD グラフィックスで行列乗算をディスパッチする基本アーキテクチャー、使用目的、および例を紹介します。
- レベルゼロを数カ月使用してみましたが、この記事では不足している、または改善が必要と思われる点についても触れています。

1 - はじめに

2020 年 3 月 (英語)、インテル社はレベルゼロ (Level-Zero) (英語) と呼ばれるヘテロジニアス・コンピューティング・アーキテクチャー向けの新しい低レベル・プログラミング API をリリースしました。レベルゼロという名称が示すように、これは GPU、FPGA、およびその他のアーキテクチャーのヘテロジニアス (異種) デバイスにアクセスして、実行を管理する低レベル API です。このブログでは、レベルゼロが何であるか、それを使用して何ができるかを説明し、いくつかの例と使用した経験を皆さんと共有したいと思います。

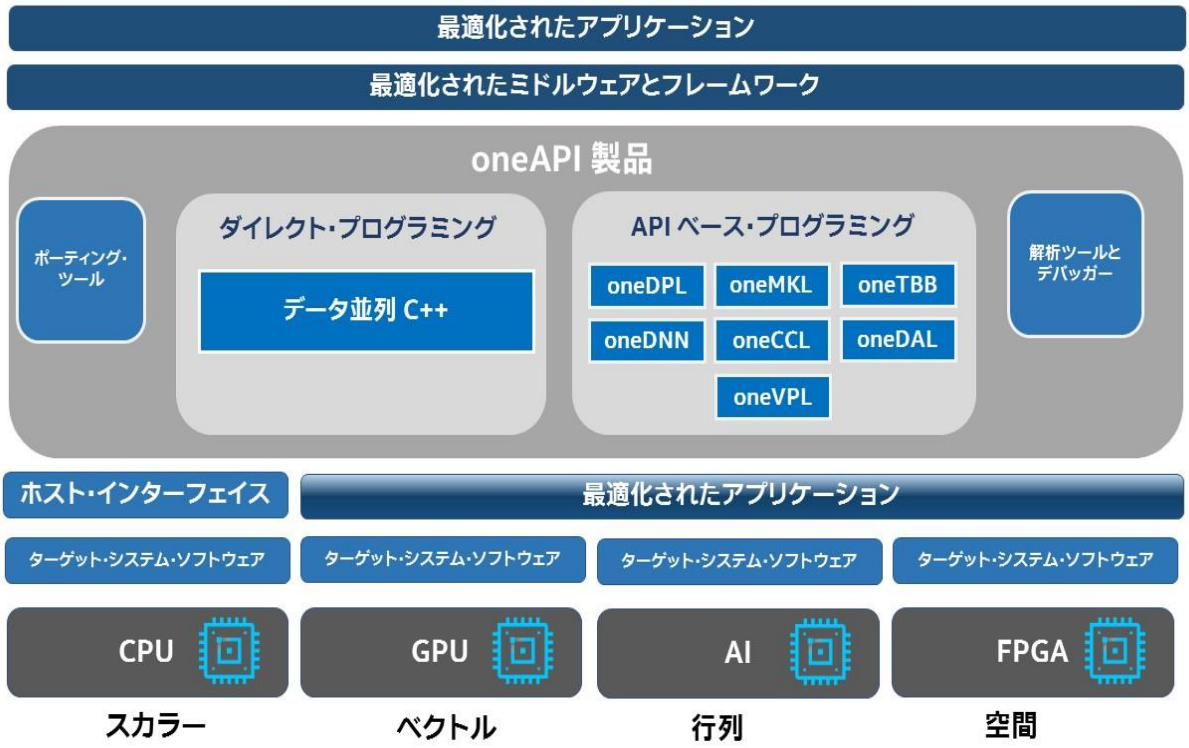
レベルゼロ API は、インテル® oneAPI (英語) の一部として出荷されています。これには、SYCL* 標準 (DPC++) を実装する高レベルの並列プログラミング API と、マシンラーニング、ディープラーニング、コンピューター・ビジョンなど向けのライブラリーとツールキット (英語) のセットも含まれています。

レベルゼロ API は、OpenCL* と Vulkan* API、およびそれぞれのプログラミング・モデルの影響を強く受けていますが、レベルゼロは独立して進化し、最新のインテル® GPU アーキテクチャーに対応する柔軟性を備えています。

レベルゼロが OpenCL* などの既存のフレームワークと異なる点は何でしょう? 例を挙げると、レベルゼロ API は仮想関数、関数ポインター、統合メモリー、デバイスのパーティション化、インストルメントとデバッグ、および電力管理のコントロール、周波数のコントロール、ハードウェアの診断をサポートします。このレベルのコントロールは、システム・プログラミング、ランタイムシステム、およびコンパイラーにとって有用であり、ヘテロジニアス・ハードウェアをよりプログラマブル/アクセシブルにします。

レベルゼロは、インテルのコンピューティング・エコシステムのどこに当てはまりますか? 次の図 (レベルゼロ仕様 1.1.2) は、インテル® oneAPI 製品とレベルゼロの関連を示しています。最も下層のレベルに、対応するドライバー (ターゲット・ソフトウェア・システム) を備えた各種ハードウェア・タイプがあります。これは、レベルゼロ API を使用して GPU、FPGA、および AI アーキテクチャーをプログラムできる可能性があることを意味しま

す。仕様には FPGA と AI アーキテクチャーが含まれていますが、現在のインテル実装 (2021 年 6 月) にはインテル® GPU 以外のハードウェアは含まれていません。レベレゼロ API の上位には、oneAPI と DPC++ (データ並列 C++) プログラミング・フレームワークとすべてのツールキットがあります。ただし、DPC++ でコーディングしたり、ツールキットを使用しなくてもレベルゼロで直接プログラムできます。このブログでは基本的な原理を説明し、その例を示します。

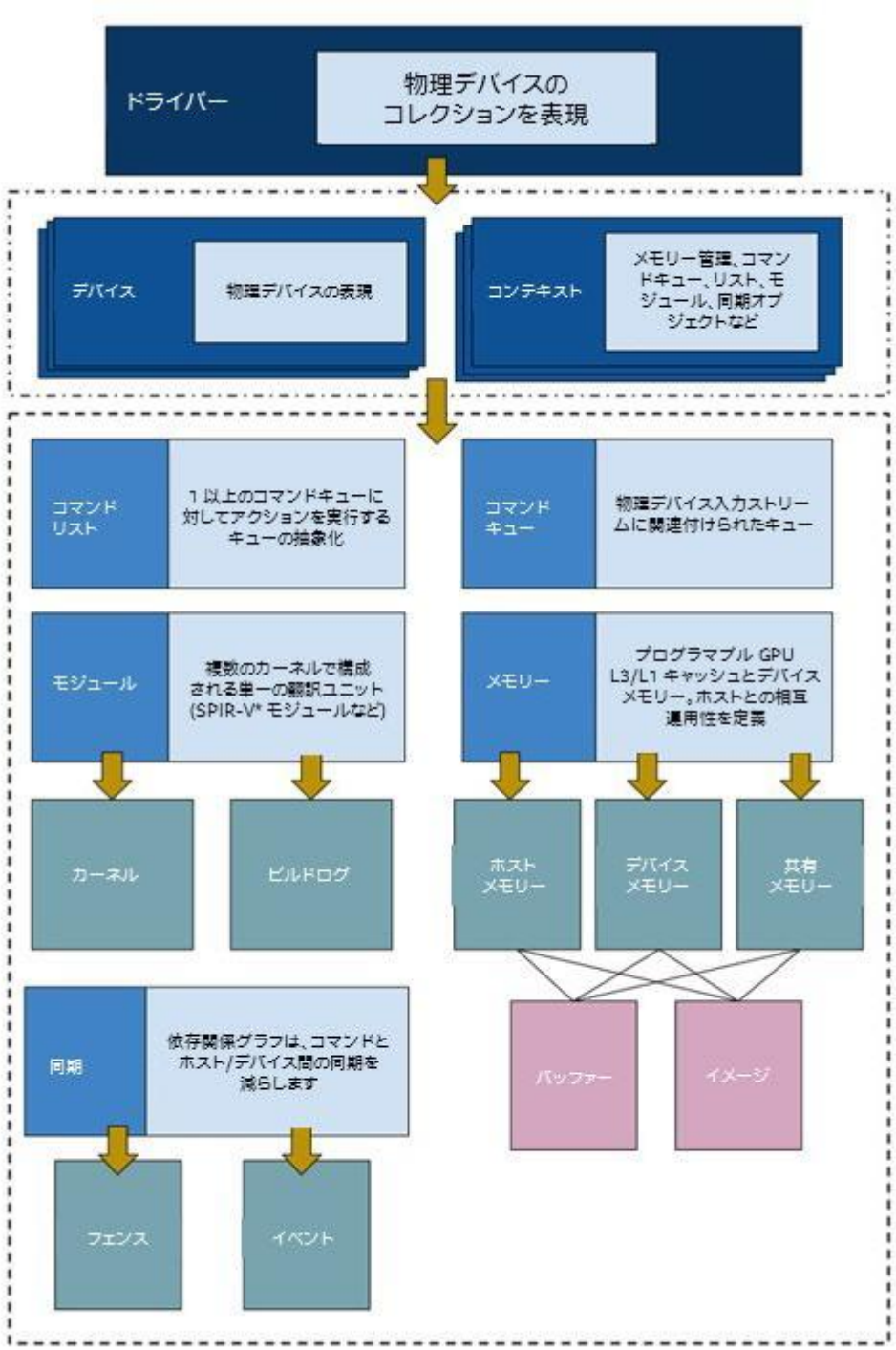


イメージの出典: インテルのレベルゼロ仕様 1.1.2: https://bit.ly/3yJIOUW (英語)

2 - 大所高所からのレベルゼロ・アーキテクチャー

レベルゼロは OpenCL* と類似しています。OpenCL* をすでに理解していると、レベルゼロの設計は非常に似ているように思われるでしょう。OpenCL* を習得していない場合、レベルゼロの学習曲線は OpenCL* よりもはるかに高いと思われます。これは、低レベルで冗長的であるためです。

次の図は、レベルゼロ・アーキテクチャーの概要、および主要コンポーネントとレベルゼロ・オブジェクトを示しています。上位レベルには、物理デバイスのコレクション (GPU コレクション) を表すオブジェクトであるレベルゼロドライバーがあります。(OpenCL* と同様に) システムでは複数のドライバーが利用できる可能性があることに注意してください。`zeDriverGet` 関数を呼び出してドライバー・オブジェクトをインスタンス化できます。



ドライバー・オブジェクトがインスタンス化されると、コンテキストとデバイス・オブジェクトを作成できます。コンテキスト (OpenCL* と同様) は、ヘテロジニアス・デバイスでの実行を処理するオブジェクトであり、メモリー、コマンドの管理、カーネルの起動、および同期に使用されます。

デバイス・オブジェクトは、システム内の物理デバイスを表します。デバイス・オブジェクトを介して、利用可能なメモリー、スレッドの最大数、デバイス名などのデバイス・プロパティーを照会できます。

レベルゼロでは、ハンドラーと記述子の 2 つの基本データタイプを管理することが重要です。ハンドラーは、コントローラー・オブジェクト (コンテキスト・ハンドラー、ドライバーハンドラーなど) を表します。記述子は、特定

のハンドラーの動作を変更する際に使用されるオブジェクトです。例えば、フラグ、メモリー・プロパティー、同期プロパティーなどを渡します。レベルゼロのコンテキストを作成するには、次のようにします。

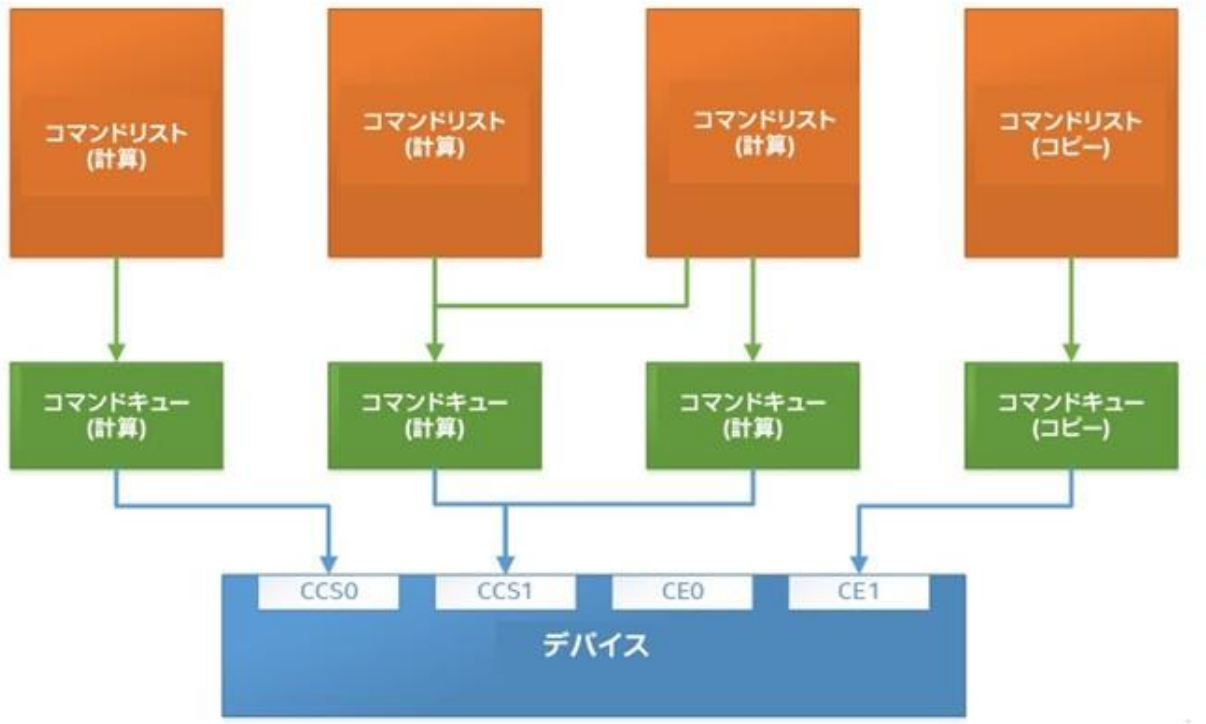
```
// 記述子
ze_context_desc_t contextDescription = {};
contextDescription.stype = ZE_STRUCTURE_TYPE_CONTEXT_DESC;
contextDescription.flags = 0;

// ハンドラー
ze_context_handle_t context;
zeContextCreate(driverHandle, &contextDescription, &context);
```

デバイス・オブジェクトとコンテキスト・オブジェクトを使用することで、レベルゼロ・アプリケーションの実行に必要なオブジェクト (コマンドキューとコマンドリスト、モジュールとカーネルなど) を作成したり、バッファーとイメージ・オブジェクトを作成することができます。

コマンドキューは、「デバイスへの論理入力ストリーム」として定義され、物理入力ストリームに関連付けられます。複数のコマンドキューを異なる入力ストリームに関連付けて持つことができます。コマンドリストは、コマンドキューを介して実行されるアクションのリストを示します。同じコマンドリストを介して異なるコマンドキューにコマンドを送信したり、別のコマンドリストにコマンドを送信したりできます。

次の図 (レベルゼロ 1.1.2 仕様 (英語)) は、コマンドリストとコマンドキューの関連を示しています。図に示すように、コマンドには 2 つのタイプ (計算とコピー用) があります。コマンドリストはコマンドをコマンドキューに送信できます。コマンドキューは、物理デバイスに近い低レベルの抽象化を表現します。低レイテンシーのコマンドキューと呼ばれる特殊タイプのコマンドキューもあります。これにより、複数のコマンドリストを記述しなくても済みます。ここでの説明は、あくまで概略であることに留意してください。詳しくは仕様をご覧になることをお勧めします。



イメージの出典: インテルのレベルゼロ仕様: https://bit.ly/3yJIOUW (英語)

コンテキストとデバイス・オブジェクトを使用して、デバイスメモリー、ホストメモリー、共有メモリーに明示的なバッファーを割り当てたり、明示的にイメージ (多次元およびユーザー定義メモリー) を作成して、メモリーを予約することができます。

レベルゼロはまた、計算のため SPIR-V* カーネルをディスパッチできます。そのため、レベルゼロは SPIR-V* モジュールを構築し、カーネルのディスパッチに必要なカーネル・オブジェクトを作成する関数を提供します。次のセクションで例を示します。

同期もまたレベルゼロ・アプリケーションでは重要な要素です。これには、イベントとフェンスの 2 つのタイプのオブジェクトがあります。これらのオブジェクトを使用することで、開発者はバリアを挿入して、コマンドリスト内のコマンドとコマンドキュー間の依存関係を調整できます。この概念は OpenCL* に非常に良く似ています。

ここまでレベルゼロ・アーキテクチャーの概要を説明してきましたが、ここで簡単な例を紹介します。共有メモリーを使用して統合 HD グラフィックスで行列乗算を実行するため、SPIR-V* カーネルをディスパッチします。

3 - 例: 行列乗算と共有メモリー向けに SPIR-V* カーネルをディスパッチ

A) レベルゼロとビルド・プロジェクトを入手

レベルゼロ・アプリケーションを実行するために必要なもの:

a) ドライバーサポート: https://github.com/intel/compute-runtime/releases/ (英語)
b) レベルゼローダー (以下の手順を参照)

レベルゼロのローダーとダイナミック・ライブラリーをビルドするには、プロジェクトのクローンを作成します。

```
git clone https://github.com/oneapi-src/level-zero
```

次に以下を行います。

```
mkdir build
cd build
cmake ..
cmake --build . --config Release
```

注: ここでは CentOS* 7.9 を使用しています。その他の Linux* ディストリビューションでもビルドの手順は同じです。GCC が 9.0 以上であることを確認してください。ここでは、以下を実行します。

```
scl enable devtoolset-9 bash
```

Windows* (英語) を使用する場合、Visual Studio* で Spectre 軽減ライブラリー (`/Qspectre`)を有効にします。

次に、ヘッダーとダイナミック・ライブラリーの `PATH` を更新します。

```
export CPLUS_INCLUDE_PATH=$LEVEL_ZERO_PATH/include:$CPLUS_INCLUDE_PATH
export LD_LIBRARY_PATH=$LEVEL_ZERO_PATH/build/lib:$LD_LIBRARY_PATH
```

これで、コーディングを開始できます。

B) 行列乗算のレベルゼロ・アプリケーション

ここでは、行列乗算の例を使用して、レベルゼロ・アプリケーションの最も代表的な部分を説明します。この例は、インテルの計算ランタイムドライバー (英語) で利用可能なレベルゼロのテストをベースにしています。行列乗算のすべてのソースコードは、GitHub* (英語) から入手できます。これを実行するには、次の手順に従ってください。

最初に、ドライバーを初期化して、デバイスの検索を行います。

```
// 初期化
VALIDATECALL(zeInit(ZE_INIT_FLAG_GPU_ONLY));

// ドライバーを取得
uint32_t driverCount = 0;
VALIDATECALL(zeDriverGet(&driverCount, nullptr));

ze_driver_handle_t driverHandle;
VALIDATECALL(zeDriverGet(&driverCount, &driverHandle));
```

VALIDATECALL マクロを使用して、それぞれの呼び出しの状態を確認することを忘れないでください。

```
#define VALIDATECALL(myZeCall) \
   if (myZeCall != ZE_RESULT_SUCCESS){ \
       std::cout << "Error at "        \
           << #myZeCall << ": "        \
           << __FUNCTION__ << ": "     \
           << __LINE__ << std::endl;   \
       std::terminate(); \
   }
```

常にエラーをチェックする必要がありますが、ここでは簡略化のためエラー・コントロールを省略します。

これで、レベルゼロのコンテキストを作成できます。

```
// コンテキストの生成
ze_context_desc_t contextDescription = {};
contextDescription.stype = ZE_STRUCTURE_TYPE_CONTEXT_DESC;
ze_context_handle_t context;
zeContextCreate(driverHandle, &contextDescription, &context);
```

次のステップでは、デバイス・オブジェクトをインスタンス化します。まず、ドライバーに関連するデバイス数を取得して、デバイスリストを作成します。

```
// デバイスを取得
uint32_t deviceCount & 0;
zeDeviceGet(driverHandle, &deviceCount, nullptr);
```

```
ze_device_handle_t device;
zeDeviceGet(driverHandle, &deviceCount, &device);
```

この時点で、デバイスに関連するいくつかの基本情報を出力できます。

```
ze_device_properties_t deviceProperties & {};
zeDeviceGetProperties(device, &deviceProperties);
std::cout << "Device   : " << deviceProperties.name << "\n"
    << "Type     : " << ((deviceProperties.type == ZE_DEVICE_TYPE_GPU) ? "GPU" :
"FPGA") << "\n "
    << "Vendor ID: " << std::hex << deviceProperties.vendorId << std::dec <<
"\n ";
```

作成したアプリケーションを実行すると、次のような情報を取得できます。

```
Device   : Intel(R) HD Graphics 630 [0x591b]
Type     : GPU
Vendor ID: 8086
```

これまでの手順を見ると、すべてが OpenCL* API に非常に類似していることが分かります。次にコマンドキューとコマンドリストを作成します。前述したようにここからは状況が異なります。

コマンドキューを作成するには、デバイスで使用可能なコマンドキューを確認して、コマンドキューのインデックスをコマンドキューとコマンドリスト記述子にアタッチする必要があります。この方法でコマンドリストをコマンドキューにマップして「リンク」することができます。コマンドキューはデバイスにアタッチされたコマンドの低レベルシーケンスを表現し、コマンドリストは送信可能なコマンドキューに関連する抽象化されたオブジェクトを表現することに注意してください。

```
// グループのクエリー数
uint32_t numQueueGroups = 0;
zeDeviceGetCommandQueueGroupProperties(device, &numQueueGroups, nullptr);
// numGroups == 0 の場合は例外をスローします
// …

std::vector<ze_command_queue_group_properties_t>
queueProperties(numQueueGroups);
zeDeviceGetCommandQueueGroupProperties(device,
                                       &numQueueGroups,
                                       queueProperties.data());

ze_command_queue_handle_t cmdQueue;
ze_command_queue_desc_t cmdQueueDesc = {};
for (uint32_t i = 0; i < numQueueGroups; i++) {
  if (queueProperties[i].flags & ZE_COMMAND_QUEUE_GROUP_PROPERTY_FLAG_COMPUTE)
{
    cmdQueueDesc.ordinal = i;
  }
}

// コマンドキューの作成
cmdQueueDesc.index = 0;
cmdQueueDesc.mode = ZE_COMMAND_QUEUE_MODE_ASYNCHRONOUS;
```

```
zeCommandQueueCreate(context, device, &cmdQueueDesc, &cmdQueue);

// コマンドリストの作成
ze_command_list_handle_t cmdList;
ze_command_list_desc_t cmdListDesc = {};
cmdListDesc.commandQueueGroupOrdinal = cmdQueueDesc.ordinal;
zeCommandListCreate(context, device, &cmdListDesc, &cmdList);
```

これで、レベルゼロ向けの共有メモリー割り当て (malloc) を呼び出して、メモリーを割り当てることができます。ここでは 3 つの行列 (a、b、c) の割り当てが必要です。分かりやすくするため 1D 表現を使用して、列と行に適切にアクセスするようインデックスを調整します。レベルゼロで共有メモリーを使用するため、ホスト側とデバイス側からメモリーにアクセスする方法を定義する 2 つのメモリー記述子が用意されています。

```
// 2 つのバッファーを作成
const uint32_t items = 1024;
constexpr size_t allocSize = items * items * sizeof(int);
ze_device_mem_alloc_desc_t memAllocDesc;
memAllocDesc.flags = ZE_DEVICE_MEM_ALLOC_FLAG_BIAS_UNCACHED;
memAllocDesc.ordinal = 0;
ze_host_mem_alloc_desc_t hostDesc;
hostDesc.flags = ZE_HOST_MEM_ALLOC_FLAG_BIAS_UNCACHED;
```

これで、行列 A、B および C の malloc が行えます。

```
void *sharedA = nullptr;
zeMemAllocShared(context,
                 &memAllocDesc,
                 &hostDesc,
                 allocSize,
                 1,
                 device,
                 &sharedA));
```

ホスト側のバッファーにアクセスする方法を示します。

```
memset(sharedA, val, allocSize);
```

行列 B と C についても同様です。

```
void *sharedB = nullptr;
zeMemAllocShared(context, &memAllocDesc, &hostDesc,
                  allocSize, 1, device, &sharedB));

void *dstResult = nullptr;
zeMemAllocShared(context, &memAllocDesc, &hostDesc,
                 allocSize, 1, device, &dstResult);
```

次に、モジュールを作成して SPIR-V* カーネルを構築します。レベルゼロを介して SPIR-V* カーネルを構築する前に、カーネルを確認してみましょう。CLANG と LLVM を使用して、OpenCL* カーネルを SPIR-V* 形式にコンパイルできます。

```
kernel void mxm(__global int* a, __global int* b, global int *c, const int n)
{
```

```
    uint idx = get_global_id(0);
    uint jdx = get_global_id(1);

    int sum = 0;
    for (int k = 0; k < n; k++) {
        sum += a[idx * n + k] * b[k * n + jdx];
    }

    c[idx * n + jdx] = sum;
}
```

このカーネルを SPIR-V* 形式にコンパイルするには、次のコマンドを使用します。これは、LLVM がインストールされていることを前提としています。ここでは、Intel/LLVM fork (英語) を使用しています。

```
$ clang -cc1 -triple spir matrixMultiply.cl -O2 -finclude-default-header -emit-
llvm-bc -o matrixMultiply.bc
$ llvm-spirv matrixMultiply.bc -o matrixMultiply.spv
```

SPIR-V* ファイルが作成されたため、次はレベルゼロを使用して SPIR-V* カーネルをコンパイルする呼び出しを行います。

```
std::unique_ptr<char[]> spirvInput(new char[length]);
file.read(spirvInput.get(), length);
ze_module_desc_t moduleDesc = {};
ze_module_build_log_handle_t buildLog;
moduleDesc.format = ZE_MODULE_FORMAT_IL_SPIRV;    // <<< USE SPIR-V
moduleDesc.pInputModule = reinterpret_cast
                          <const uint8_t*>(spirvInput.get());
moduleDesc.inputSize = length;
moduleDesc.pBuildFlags = "";
auto status = zeModuleCreate(context, device, &moduleDesc, &module, &buildLog);
```

SPIR-V* モジュールの作成中にエラーが発生した場合、buildLog オブジェクトを調査します。詳細については、GitHub* (英語) のコードを参照してください。では、カーネル・オブジェクトを作成しましょう。これは、OpenCL* の手順とよく似ています。

```
ze_kernel_desc_t kernelDesc = {};
kernelDesc.pKernelName = "mxm";       // カーネル名
zeKernelCreate(module, &kernelDesc, &kernel);
```

これでカーネルをディスパッチする準備が整いました。ここでは、展開するスレッド数とブロック数を設定する必要があります。最初に、特定のカーネルに展開するスレッド数をレベルゼロに示し、次にそれに基づいてスレッドを設定できることに留意してください。

```
uint32_t groupSizeX = 32u;
uint32_t groupSizeY = 32u;
uint32_t groupSizeZ = 1u;
zeKernelSuggestGroupSize(kernel, items, items, 1U, &groupSizeX, &groupSizeY,
&groupSizeZ));
zeKernelSetGroupSize(kernel, groupSizeX, groupSizeY, groupSizeZ));
```

ここで、選択したブロックサイズが最終的に設定されたものではないことに気付くかもしれません。レベルゼロドライバーはより適切な値を選択する可能性があります。例えば、ここでは 32x32 のブロックサイズを設定し

ましたが、ドライバーは 256x1 を選択しました。これらの値を変更すると、パフォーマンスがどのように変化するか分かります。例えば、32x1 を選択すると、パフォーマンスは C++ シーケンシャル・コードの 10 倍になりますが、256x1 では 15 倍になりました。この値を自由に変更してみてください。経験上、レベルゼロを使用すると適切なブロックサイズが得られます。

次のようにカーネルへの引数を設定して、コマンドリストから起動します。

```
// 引数をプッシュします
zeKernelSetArgumentValue(kernel, 0, sizeof(dstResult), &dstResult);
zeKernelSetArgumentValue(kernel, 1, sizeof (sharedA), &sharedA);
zeKernelSetArgumentValue(kernel, 2, sizeof (sharedB), &sharedB);
zeKernelSetArgumentValue(kernel, 3, sizeof (int), &items);

// カーネルのスレッド・ディスパッチ
ze_group_count_t dispatch;
dispatch.groupCountX = items / groupSizeX;
dispatch.groupCountY = items / groupSizeY;
dispatch.groupCountZ = 1;

// GPU 上でカーネルを起動します
zeCommandListAppendLaunchKernel(cmdList,  kernel,  &dispatch,  nullptr,  0,
nullptr);
```

ホストとデバイス間で明示的なデータ転送が行われていないことに注目してください。これは、共有メモリーを使用しているためです。つまり、カーネルの起動前にデータ転送を明示的に行う必要はありません。

カーネル・ディスパッチは非ブロッキング呼び出しであるため、コマンドリスト内のコマンドの実行を完了するには、リストをクローズしコマンドキューを介して強制的に実行する必要があります。

```
// リストをクローズして、実行のため送信します
zeCommandListClose(cmdList));
zeCommandQueueExecuteCommandLists(cmdQueue, 1, &cmdList, nullptr);
zeCommandQueueSynchronize(cmdQueue, std::numeric_limits<uint64_t>::max());
```

これで完了です! これで、ホスト側 (共有メモリー) で結果を参照できるようになります。

アプリケーションを実行すると、デフォルトで 1024x1024 の行列乗算が実行され、次のような出力が得られます。

```
$ ./mxm
Device   : Intel(R) HD Graphics 630 [0x591b]
Type     : GPU
Vendor ID: 8086
#Queue Groups: 1
Group X: 256
Group Y: 1
GPU Kernel = 463028562 [ns]
SEQ Kernel = 6651558420 [ns]
Speedup = 14x

Matrix Multiply validation PASSED
```

この例では、レベルゼロ・アプリケーションの主要な構成を説明し、SPIR-V カーネルをディスパッチするため共有メモリーを使用しました。ソースコード全体は GitHub* (英語) で入手できます。

4 - レベルゼロに欠如しているものは?

レベルゼロは素晴らしいイニシアチブであり、オープンであることがとても気に入っています。また、プロジェクトの貢献者が GitHub* で提案や議論を受け付けていること (私的な経験ですが) にも感謝しています。

とはいえ、このブログを書いている時点 (2021 年 6 月) で、いくつかの改善点があることに気が付きました。それらのいくつか (すべてではなくても) は、将来のリリースで解決/サポートされることを願っています。

→ はじめに免責事項について触れておきます。レベルゼロはオープンソースであるため、以下の「問題」は私を含めコミュニティーのメンバーからも提案される可能性があることに注意してください。これは、私が見つけたいくつかのことを指摘しているだけです。

レベルゼロに欠如しているものは、まず、特にイベント、バリア、タイマーなどを使用する際に、文書化された例が必要です。レベルゼロには、一連のテスト項目が含まれており、多くのユーティリティーがあることは理解しています。oneAPI の例 (英語) に類似したものはありますが、レベルゼロのものはありません。これらの例で、パフォーマンスを簡単に評価し、OpenCL* と比較できると便利です (例えば、OpenCL* コマンドキューとレベルゼロの低レイテンシー・コマンドリスト)。

また、OpenCL*/CUDA* 開発者からの「変換表」も欠如しています。実際のところ、仕様書にはレベルゼロの各関数に関連する豊富なドキュメントがありますが、通常は OpenCL* の同等の関数 (存在する場合) を参照していると言わざるを得ません。しかし、レベルゼロ仕様の新しいセクションとして、一般的な関数/機能の等価表があると便利です。それはきっと、新しいテクノロジーを受け入れるのに役立つでしょう。

技術的な観点から、レベルゼロのほとんどの記述データ構造に含まれる `pNext` フィールドの概要は、このテクノロジーの開発者や入門者にとっては有用なものであると思います。私の理解では、このフィールドを使用することで、開発者は各種記述子 C 構造体を利用してプロパティーを拡張できます。ただし、その可能性を示すユースケース (完全な例) と互換性の一覧がないため、どれを拡張すべきか不明です。また、拡張機能はかなりの確率でパフォーマンスに影響するため、開発者が特定のアプリケーション領域のパフォーマンスを向上させる適切な拡張機能セットはあるでしょうか?

また、より優れたエラー・コントロールが必要です。例えば、開発者が誤ってレベルゼロがすでに閉じたコマンドリストをリセットし忘れた場合 (英語)、レベルゼロは閉じたリストで連続した呼び出しを行っても何も警告しません。ただし、コマンドは実行されないため (警告も表示されませんが)、誤った結果が得られます。これは非常にエラーが発生しやすく、実際、私も何度か経験しました。開発者はコマンドリストをリセットし忘れないよう、特に注意する必要があります。

5 - まとめ

レベルゼロは、インテルによって開発され、インテル® oneAPI ツールキットの一部として出荷される低レベルのベアメタル API に近いものです。この新しい API を使用すると、開発者はヘテロジニアス・ハードウェア、特に GPU 向けのアプリケーションを微調整でき、仮想関数、関数ポインター、効率良く微調整されたメモリー管理を使用したり、電源やファームウェアを制御することができます。

このブログでは、インテルのレベルゼロ API の概要を説明しました。最初に、一般的な紹介とプログラミング・モデルの概要を示し、次に、レベルゼロを使用してインテル® HD グラフィックスに SPIR-V* カーネルをディスパッチする手順と例を示しました。最後に、レベルゼロに欠如していると私が考えるいくつかの提案と、コミュニティーがこのプロジェクトに貢献して欲しいことを述べました。

レベルゼロ API は、ランタイム、OS、システム・ソフトウェアによって制御されるヘテロジニアス・ハードウェアにアクセスする優れた方法であり、開発者がアプリケーションを微調整して、ヘテロジニアス・ハードウェアで実行する際に高レベル・アプリケーションのリソース利用率と移植性を最大化すると信じています。

謝辞

このブログに対し建設的なフィードバックをくれた Athanasios Stratikopoulos 氏 (英語) に感謝します。