oneAPI レベルゼロ・インターフェイスの使用

レベルゼロ API の概要紹介

Rama Kishan Malladi Amazon Web Services ソリューション・アーキテクト
Nitya Hariharan インテル コーポレーション アプリケーション・エンジニア

oneAPI 仕様（英語）は、アクセラレーター・アーキテクチャー間で同じ言語、API、プログラミング・モデルを提供することで、ソフトウェア開発を簡素化するものです。この仕様では、さまざまなアーキテクチャーに共通するデータ並列ドメイン用の API セットを定義しています。API とダイレクト・プログラミングのいずれのアプローチも、データ並列処理（各データ要素に対して同じ計算を行うこと）を基本としています。oneAPI プラットフォームは、ホストとアクセラレーター・デバイスで構成されています（**図 1**）。API プログラミング・モデルは、oneMKL、oneDPL、oneDNN、oneCCL などのライブラリーを使用して実装されています。ダイレクト・プログラミングは DPC++ で行われます。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

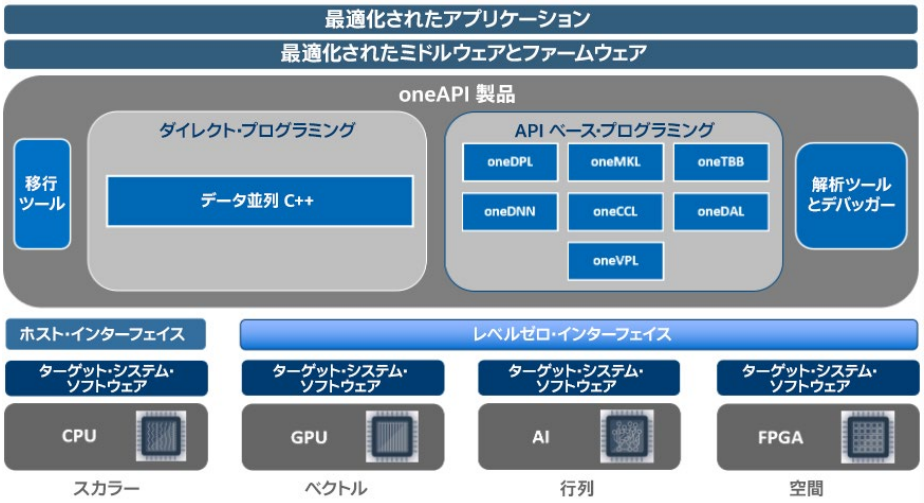

図 1. oneAPI プラットフォームの実行モデル。インテルの oneAPI 仕様の実装には、プログラミング・ツールも含まれています。

レベルゼロ：はじめに

oneAPI レベルゼロは、oneAPI プラットフォームのデバイスに対して、低レベルのメタルに直結したインターフェイスを提供します。レベルゼロは、デバイスの検出、メモリー割り当て、プロセス間通信、カーネルのサブミット、同期、メトリックのレポートを行う、きめ細かい明示的な制御 /API に加えて、広範な言語機能のサポートを提供します。また、デバイスの論理レベルと物理レベルの抽象化の両方を利用できる API を備えています。ほかの低レベル API（OpenCL* など）の影響を大きく受けていますが、レベルゼロは独自に進化するように設計されており、GPU や FPGA のようなほかの計算デバイスをサポートしています。ほとんどのアプリケーションでは、レベルゼロ API が提供する追加の制御は必要ありません。レベルゼロ API は、次のような、上位のランタイム API やライブラリーで必要とされる明示的な制御を想定しています。

- デバイスの検出と切り分け
- カーネルの実行とスケジュール
- ピアツーピア通信
- メトリックとプロファイル
- カーネル・プロファイル、インストルメンテーション
- システム管理、電力やパフォーマンスの照会

レベルゼロの C API は、共有インポート・ライブラリーによってアプリケーションに提供されます。そのため、C/C++ アプリケーションでは「`ze_api.h`」をインクルードして、「`ze_api.lib`」（または共有ライブラリー）とリンクする必要があります。

レベルゼロ：API

レベルゼロ API は、コア、ツール、およびシステム・プログラミングに分類されますが、この記事ではコア・プログラミング API（**図 2**）のみを説明します。コア・プログラミング API は、デバイス、ドライバー、コンテキスト、メモリー、コマンドキュー / リスト、同期、バリア、モジュール、カーネルをサポートしています。**表 1** と**表 2** は、よく使用される API の一覧です。**図 3** は、レベルゼロを使用した実行フローです。

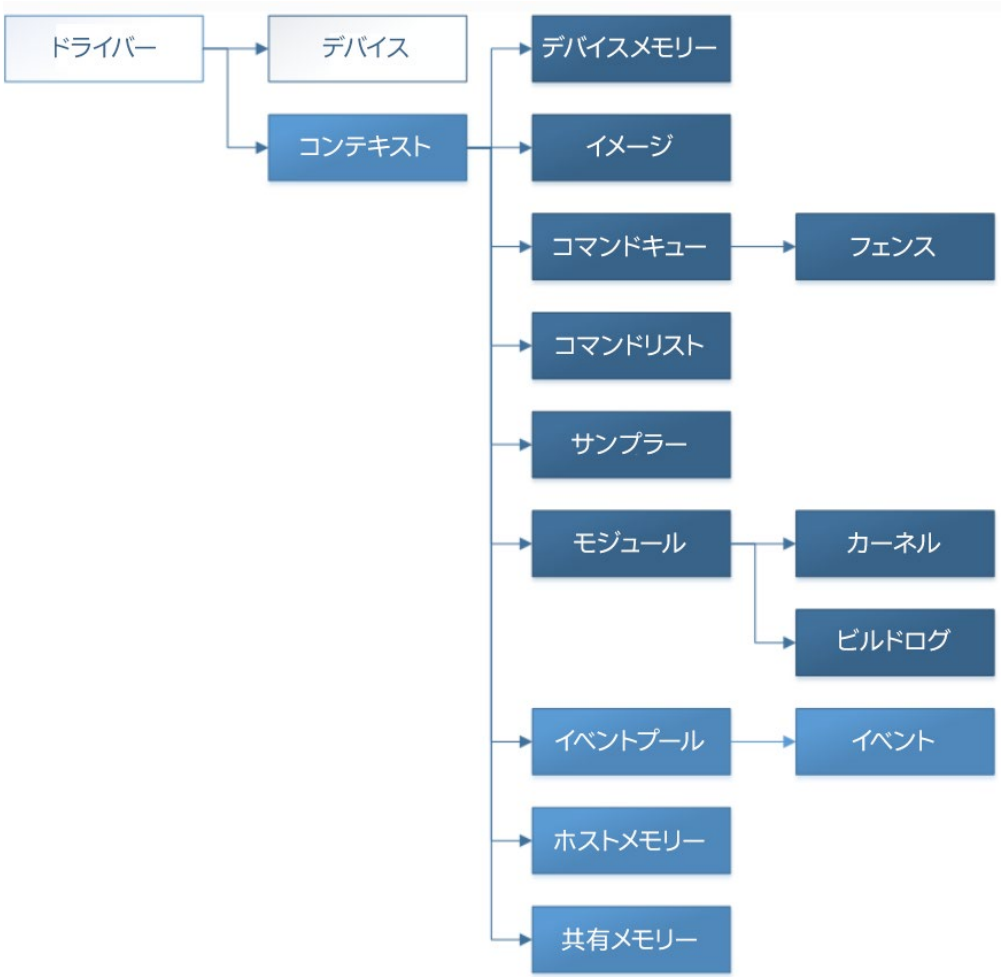

図 2. コア・プログラミング API のコンポーネント

デバイス、コンテキスト、キューの API	説明
zeInit、zeDriverGet	すべてのドライバーを初期化して検出します。
zeDeviceGet、zeDeviceGetProperties	DEVICE_TYPE のドライバー・インスタンスを探します。
zeContextCreate	コンテキストを作成します。
zeMemAllocHost、zeMemAllocDevice、zeMemAllocShared	ホスト、デバイス、または共有メモリー上にメモリーを割り当てます。
zeCommandQueueCreate	コマンドキューを作成します。
zeCommandListCreate	コマンドリストを作成します。
zeCommandQueueExecuteCommandLists	コマンドキューのコマンドリストを実行します。
zeCommandQueueSynchronize	ホストとデバイスを同期します。

表 1. デバイス、コンテキスト、およびキューのレベルゼロ API

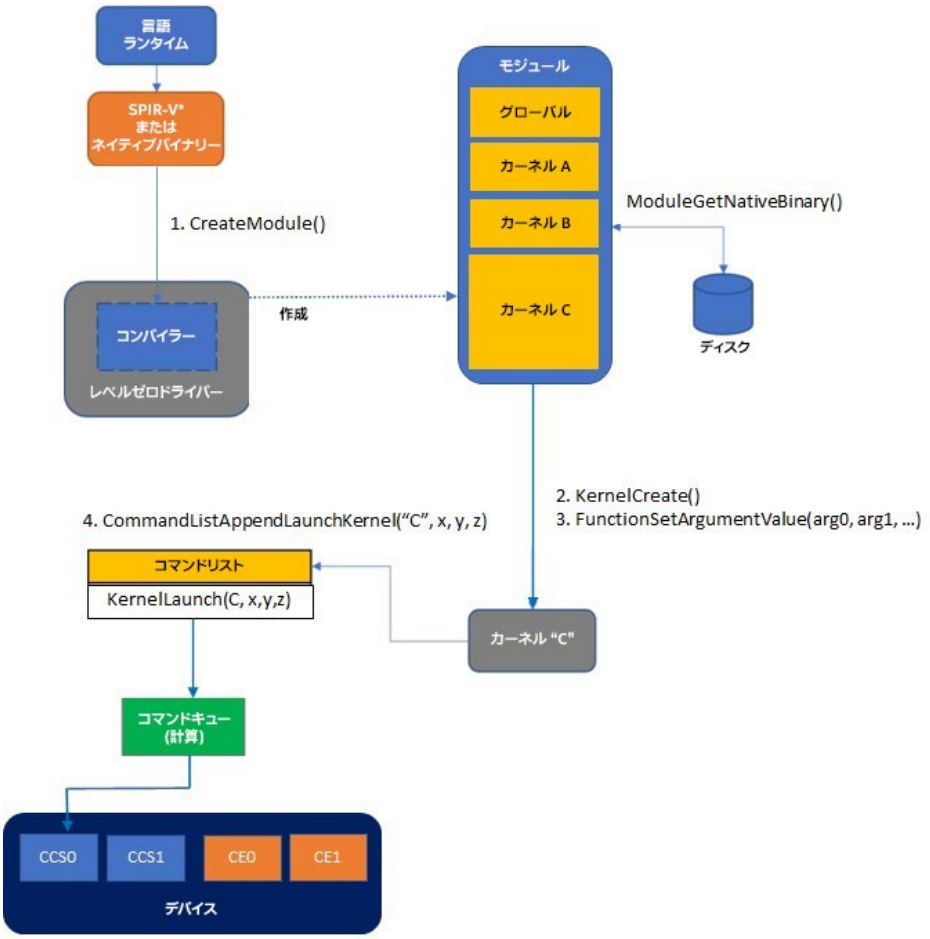

図 3. 高レベルのフロー：コマンドリスト、キュー、モジュール、およびデバイス上のカーネル実行

同期、モジュール、カーネルの API	説明
zeEventPoolCreate	イベントプールを作成します。
zeCommandListAppendLaunchKernel	カーネル、引数、シグナルをコマンドリストに追加します。
zeEventHostSynchronize	イベントの完了を待機します。
zeModuleCreate	IL コードをコンパイルするか、ネイティブバイナリーをロードしてモジュールを作成します。
zeKernelCreate	モジュール内のカーネルを参照します。
zeKernelSetArgumentValue	カーネル起動時の引数を設定します。

表 2. 同期およびカーネル関連関数のレベルゼロ API

リスト 1 は、ドライバーとデバイスの検出を行う main プログラムです。**リスト 2** は、関数 RunTest のカーネルの実行を示しています。

```
// ドライバーコード(main)
int main(int argc, char *argv[])
{
   zeInit(ZE_INIT_FLAG_GPU_ONLY);

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

   uint32_t deviceCount = 1;
   ze_device_handle_t device;
   zeDeviceGet(driverHandle, &deviceCount, &device);

   ze_device_properties_t deviceProperties = {};
   zeDeviceGetProperties(device, &deviceProperties);

   uint32_t subDeviceCount = 0;
   zeDeviceGetSubDevices(device, &subDeviceCount, nullptr);
   ze_device_handle_t subDevices[2] = {};
   zeDeviceGetSubDevices(device, &subDeviceCount, subDevices);

  for (uint32_t i = 0; i < subDeviceCount; i++) {
      ze_device_properties_t deviceProperties = {};
      zeDeviceGetProperties(subDevices[i], &deviceProperties);
   }

   RunTest(driverHandle, subDevices, device, subDeviceCount, outputValidBool);
}
```

リスト 1. ドライバーとデバイス検出のレベルゼロの例

```
void RunTest(ze_driver_handle_t &driverHandle, ze_device_handle_t *subDevice,
             ze_device_handle_t rootDevice, uint32_t subDeviceCount,
             bool &validRet)
{
  // 変数の初期化、ホストメモリーの割り当て
  ...
  // コンテキスト、コマンドキュー、リストの作成
  zeContextCreate(driverHandle, &contextDesc, &context);
  ...
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeCommandQueueCreate(context, subDevice[i], &cmdQueueDesc, &cmdQueue[i]);
    zeCommandListCreate(context, subDevice[i], &cmdListDesc, &cmdList[i]);
  }
  ...
  // デバイスで実行するカーネルが含まれる IL ファイル(SPIR-V* 形式)のロード
  const char *modulePath = "Gpu_Module_Kernel.spv";
  uint32_t spirvSize = 0;
  auto spirvModule = readBinaryFile(modulePath, spirvSize);
  ...
  zeModuleCreate(context, subDevice[i], &moduleDesc, &module[i], nullptr);
  zeKernelCreate(module[i], &kernelDesc, &kernel[i]);
  zeKernelSetGroupSize(kernel[i], groupSizeX, groupSizeY, groupSizeZ);
  ...
  // デバイスメモリーの割り当て、コマンドリストへのメモリーコピー操作の追加
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeMemAllocDevice(context, &deviceDesc, bufferWidth*sizeof(float), 0,
                     subDevice[i], &d_input[i]);
    zeCommandListAppendMemoryCopy(cmdList[i], d_input[i], input[i],
                     bufferWidth*sizeof(float), nullptr, 0, nullptr);
    ...
  }
    // ホストからデバイスへのデータのコピー(割り当てとコピーコマンドを実行)
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListClose(cmdList[i]);
    zeCommandQueueExecuteCommandLists(cmdQueue[i], 1, &cmdList[i], nullptr);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandQueueSynchronize(cmdQueue[i], UINT32_MAX);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListReset(cmdList[i]);
  }
  ...
  // カーネル引数の設定
  for (uint32_t i=0; i<num_tiles; i++)
  {
    arg_indx = 0;
    start_idx = i * segment_size;

    zeKernelSetArgumentValue(kernel[i], arg_indx++, sizeof(d_input[i]),
                             &d_input[i]);
    zeKernelSetArgumentValue(kernel[i], arg_indx++, sizeof(d_input[i]),
                             &d_input[i + 1 == num_tiles ? 0 : i + 1]);
    ...
  }
  // イベントプールの作成、カーネル起動コマンドへの追加
  ze_event_pool_handle_t eventPool;
  zeCommandListAppendLaunchKernel(cmdList[i], kernel[i], &group_count,
                                  kernelTsEvent[i], 0, nullptr);
  ... // (続く)
```

リスト 2. カーネル実行のレベルゼロの例

```
  // コマンドリストの実行、キューのコマンド実行の同期
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListClose(cmdList[i]);
    zeCommandQueueExecuteCommandLists(cmdQueue[i], 1, &cmdList[i], nullptr);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandQueueSynchronize(cmdQueue[i], UINT32_MAX);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListReset(cmdList[i]);
  }
  // カーネルイベント統計の取得、実行時間の計算
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeEventQueryKernelTimestamp(kernelTsEvent[i], &kernelTsResults);
    uint64_t kernelDuration = kernelTsResults.context.kernelEnd -
                              kernelTsResults.context.kernelStart;
  }
  // デバイスからホストへデータをコピー
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeCommandListAppendMemoryCopy(cmdList[i], output[i], d_output[i],
                      bufferWidth*sizeof(float), nullptr, 0, nullptr);
  }
  ...
  // カーネル、メモリー、コンテキスト、イベント、およびその他のオブジェクトの破棄
  zeEventPoolDestroy(eventPool);
  for (size_t i=0; i<num_tiles; i++) {
    zeMemFree(context, d_input[i]);
    zeKernelDestroy(kernel[i]);
    zeCommandListDestroy(cmdList[i]);
    ...
  }
  zeContextDestroy(context);
  ...
}
```

リスト 2. カーネル実行のレベルゼロの例（続き）

OpenMP* の例

レベルゼロ API は、OpenMP* オフロードコードのコンパイル時にもバックエンドで生成されます。これらの API 呼び出しは、`LIBOMPTARGET_DEBUG` 環境変数と `LIBOMPTARGET_INFO` 環境変数が `1` 以上に設定されている場合はダンプされます。**リスト 3** は、デバイス上に構造体配列（AoS）を割り当て、ホスト上で初期化したデータをコピーして、デバイス上で更新した後、ホストへ転送する例を示します。ログは、前述の 2 つの環境変数を `99` に設定して生成されたものです。

リスト 4 は、ログから抜粋したレベルゼロ API 呼び出しの一部です。`zeMemAllocDevice` 呼び出しでデバイス上にデータを割り当て、続く `zeCommandListAppendMemoryCopy` 呼び出しでホストからデバイスへデータをコピーします。カーネルの計算が終了すると、2 つ目の `zeCommandListAppendMemoryCopy` 呼び出しで、デバイスからホストへ更新されたデータをコピーします。そして、`zeMemFree` 呼び出しで AoS を削除し、デバイスメモリーをメモリープールに返します。これらの呼び出しは、**リスト 2** と似ています。ここでは、OpenMP* プラグマに直接関連するレベルゼロ呼び出しに注目します。ログには、ホストからデバイスへのポインターのコピー、メモリーブロックのプロパティーの取得、ホストからデバイスポインターへのマッピング、終了時のデバイスのクリーンアップに使用される多くの呼び出しが表示されます。このほかにも、レベルゼロ API は、メモリーの割り当て、ホストからデバイスへのコピー、またはホストとデバイスの間での共有をより細かく制御できる呼び出しを提供します。

```
struct force_data
{
  float Mass;
  int index;
};

#pragma omp declare target
struct force_data *myData;
#pragma omp end declare target

int main()
{
  int a[10], max=10;

  // ホスト上に構造体配列を割り当て
  myData = (struct force_data*) malloc(max  * sizeof(struct force_data));

  for(int i = 0; i < max; i++)
  {
    myData[i].index = 1; a[i] = 2;
  }

 // 1. デバイス上にデータを割り当て
  #pragma omp target enter data map(alloc:myData[0:max])
  {
// 2. デバイス上のデータを更新
  #pragma omp target teams distribute parallel for map(to:a)
    for(int i=0; i < max; i++)
      myData[i].index = myData[i].index + a[i];
  }

// 3. デバイス上のデータを削除
  #pragma omp target exit data map(delete:myData[0:max])
  for(int i=0; i < max; i++)
     printf("%d\n", myData[i].index);
}
```

リスト 3. OpenMP* オフロードの例

```
Libomptarget (pid:6780)  --> Entering OpenMP data region at unknown:31:31 with 1 arguments:
Libomptarget (pid:6780)  --> alloc(myData[0:10])[80]
…
Target LEVEL0 RTL (pid:10428)  --> ZE_CALLER: zeMemAllocDevice ( context, &deviceDesc, Size,
Align, Device, &mem )
…
Libomptarget (pid:10428)  --> Copying data from host to device, HstPtr=…., TgtPtr=….,
Size=80, Name=myData[0:10]
…
Target LEVEL0 RTL (pid:10428)  --> Copy Engine is used for data transfer
Target LEVEL0 RTL (pid:10428)  --> ZE_CALLER: zeCommandListAppendMemoryCopy ( cmdList, Dest,
Src, Size, nullptr, 0, nullptr )
…
Libomptarget (pid:6780)  --> Entering OpenMP kernel at unknown:41:41 with 6 arguments:
Libomptarget (pid:6780)  --> tofrom(myData)[8] (implicit)
…

Libomptarget (pid:6780)  --> Updating OpenMP data at unknown:46:46 with 1 arguments:
Libomptarget (pid:6780)  --> from(myData[0:10])[80]

Libomptarget (pid:6780)  --> Copying data from device to host, TgtPtr=0xffffd556aa640000,
HstPtr=0x0000000000d362b0, Size=80, Name=myData[0:10]

Target LEVEL0 RTL (pid:6780)  --> Copy Engine is used for data transfer
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLER: zeCommandListAppendMemoryCopy ( cmdList, Dest,
Src, Size, nullptr, 0, nullptr )

Libomptarget (pid:6780)  --> Deleting tgt data 0xffffd556aa640000 of size 80

Target LEVEL0 RTL (pid:6780)  --> Returned device memory 0xffffd556aa640000 to memory pool
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLER: zeMemFree ( Context, (void *)block->Base )
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLEE: zeMemFree (
Target LEVEL0 RTL (pid:6780)  -->     hContext = 0x0000000000d33500
Target LEVEL0 RTL (pid:6780)  -->     ptr = 0xffffd556aa640000
```

リスト 4. OpenMP* オフロード向けにバックエンドで生成されるレベルゼロ API 呼び出し

まとめ

レベルゼロ API は、主にライブラリーやフレームワーク開発者を対象としており、ほとんどのアプリケーション開発者は、レベルゼロ API が提供する追加の制御を必要としないでしょう。レベルゼロ API は、デバイス検出、メモリー管理、カーネルのサブミット、プロセス間通信などに対して、よりきめ細かい明示的な制御を提供します。この記事では、基本的な例を使用してレベルゼロのプログラミングを紹介しました。OpenMP* オフロードの例では、オフロード･アクセラレーター･デバイスへのメタルに直結したインターフェイスを提供する、バックエンドで生成される一連の呼び出しに関する情報も得られます。レベルゼロ API の詳細は、oneAPI レベルゼロ仕様（英語）を参照してください。