レベルゼロとは

オープンなバックエンド・アプローチでどこでも計算を実現

Robert Mueller-Albrecht インテル コーポレーション プロダクト・マーケティング・エンジニア

この記事では、レベルゼロ（英語）のアプリケーション・インターフェイス、目的、およびビジョンの概要を説明します。レベルゼロの基本的なアーキテクチャーと、計算ユニットリソースへの低レベルのアクセス制御の利点を紹介します。レベルゼロは、OpenMP*（英語）や SYCL*（英語）のような言語拡張とともに使用できます。レベルゼロと SYCL* C++ 言語拡張抽象化レイヤーの間の相互作用がアプリケーション・ソフトウェア開発者にどのように明らかにされるか簡単に説明します。

レベルゼロは、任意の数のオフロードデバイスへのアクセスを設定および管理する低レベル API として設計されています。プログラムフローに干渉することなく、C++ 標準規格に準拠したヘテロジニアス計算を行うことができる抽象化レイヤーも提供します。この機能により、異なるランタイム環境の間でコードを移植できます。レベルゼロ API バックエンドを考慮することにより、SYCL* や OpenMP* 言語拡張の抽象化を超えて、制御のレベルを向上できます。

レベルゼロ・インターフェイス（英語）は、oneAPI 仕様（英語）の一部です。CPU、GPU、およびアクセラレーターへのベアメタルアクセスにより、oneAPI の API ベースのプログラミング・モデルとダイレクト・プログラミング・モデルを補完します。インテル® oneAPI ベース・ツールキットの一部としてインテル® GPU をターゲットにしたインテルのリファレンス実装（英語）とインテル® oneAPI DPC++/C++ コンパイラーでの使用法も文書化されています。この記事を読めば、レベルゼロランタイムの使用を開始したり、独自のランタイムを検討するために必要なことが分かるでしょう。

ヘテロジニアス・コンピューティングの能力を引き出す

レベルゼロのインテルの最初の実装はインテル® GPU をターゲットにしていますが、レベルゼロのビジョンと可能性は非常に広範であり、特定のデバイス要件に合わせて調整された抽象化を作成することが可能です。関数ポインター、メモリー、I/O など、幅広い言語機能をサポートするように適合させることができます。API は、CPU、GPU、フィールド・プログラマブル・ゲート・アレイ（FPGA）、その他のアクセラレーター・アーキテクチャーを含む、さまざまな計算デバイスで動作するように設計されています。

レベルゼロは、OpenCL* や Vulkan* などのほかの低レベル API と共存できます。ただし、高レベルの oneAPI および SYCL* 開発者の経験がハードウェアに依存せず、可能な限りアーキテクチャーに依存しないで柔軟になるように、独立して進化することを意図しています。SYCL* を利用する高レベルのランタイム API とライブラリーで必要となる明示的な制御も提供します。レベルゼロは完全なオープンソースで、その仕様（英語）、ソース・リポジトリー（英語）、およびインテル® GPU 計算ランタイムの実装（英語）は、GitHub* で公開されています。

つまり、レベルゼロは、ヘテロジニアス・コンピューティングを広く開放し、オフロード計算の選択を実現する、柔軟なオープン・バックエンドを提供します。システムレベルのインターフェイスを明示的に制御する、次のような機能を提供します。

- デバイス検出
- メモリー割り当て
- ピアツーピア通信
- プロセス間共有
- カーネルのサブミット
- 非同期実行とスケジューリング
- 同期プリミティブ
- メトリックレポート
- システム管理

レベルゼロの主要な概念から始めましょう。

レベルゼロの基本

レベルゼロは、アプリケーション層の下に位置していて、C++ アプリケーションとターゲット・デバイス・プロパティーの間の抽象化インターフェイスとして採用できます（**図 1**）。CPU やその他の計算デバイスが含まれることがあります。開発者は共有デバイスのリソースとシームレスにやり取りして、レベルゼロドライバーがサポートする特定のデバイスにワークロードをディスパッチできます。ドライバーは、サポートされているデバイスを利用可能なデバイスリストに追加し、任意の SYCL* キューをそのデバイスにマップしてワークを送信できます。特定のデバイス・プロパティーや複数のレベルゼロデバイス間で共有されるリソースにアクセスする必要がない場合、それらのデバイスは異なる SYCL* バックエンド API を使用するほかのデバイスと同じように動作します。

図 1. oneAPI バックエンド・アーキテクチャーのコンテキストのレベルゼロ・インターフェイス

レベルゼロの真の強みは、デバイス固有のメモリー共有または同期コンテキスト・オブジェクトに対する低レベルの制御とサポートです。デバイスの透明性が向上するだけでなく、レベルゼロ API にヘテロジニアス・オフロード・ターゲット・デバイスの構成可能性が追加されます。

レベルゼロデバイスの検出と選択の一連の流れを次に示します。

レベルゼローダー

オフロードデバイスまたはアクセラレーターへのアクセスは、レベルゼローダーから開始します。システムのデバイス向けのレベルゼロドライバーを検出します。ローダー・プロジェクトには、レベルゼロ実装をビルドして操作できるようにするレベルゼロヘッダーとライブラリー（英語）も含まれています。ドライバーの初期化と検出のコードサンプルを次に示します。

```
// ドライバーを初期化
zeInit(0);

// ドライバー・インスタンスをすべて検出
uint32_t driverCount = 0;
zeDriverGet(&driverCount, nullptr);

ze_driver_handle_t* allDrivers = allocate(driverCount *
                                          sizeof(ze_driver_handle_t));
zeDriverGet(&driverCount, allDrivers);

// GPU デバイスを含むドライバー・インスタンスを検索
ze_driver_handle_t hDriver = nullptr;
ze_device_handle_t hDevice = nullptr;
for(i = 0; i < driverCount; ++i)
{
    uint32_t deviceCount = 0;
    zeDeviceGet(allDrivers[i], &deviceCount, nullptr);
```

```
        ze_device_handle_t* allDevices = allocate(deviceCount *
                                                  sizeof(ze_device_handle_t));
        zeDeviceGet(allDrivers[i], &deviceCount, allDevices);

        for(d = 0; d < deviceCount; ++d)
        {
            ze_device_properties_t device_properties {};
            device_properties.stype = ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;
            zeDeviceGetProperties(allDevices[d], &device_properties);

            if(ZE_DEVICE_TYPE_GPU == device_properties.type)
            {
                hDriver = allDrivers[i];
                hDevice = allDevices[d];
                break;
            }
        }

        free(allDevices);
        if(nullptr != hDriver)
        {
            break;
        }
    }

    free(allDrivers);
    if(nullptr == hDevice)
        return; // GPU デバイスなし
```

続いて、メモリー、コマンドキュー、モジュール、同期オブジェクトなどを管理するコンテキスト・オブジェクトを作成します。複数のデバイスで共有できるシステムリソースを管理する場合、コンテキストの使用は特に重要になります。共有メモリーのシナリオの簡単な例を次に示します。

```
// コンテキストを作成
zeContextCreate(hDriver, &ctxtDesc, &hContextA);
zeContextCreate(hDriver, &ctxtDesc, &hContextB);

zeMemAllocHost(hContextA, &desc, 80, 0, &ptrA);
zeMemAllocHost(hContextB, &desc, 88, 0, &ptrB);
```

スケジューリング・モデル

図 2 に示すように、すべてのコマンドがスケジュールされ、レベルゼロデバイスにディスパッチされます。

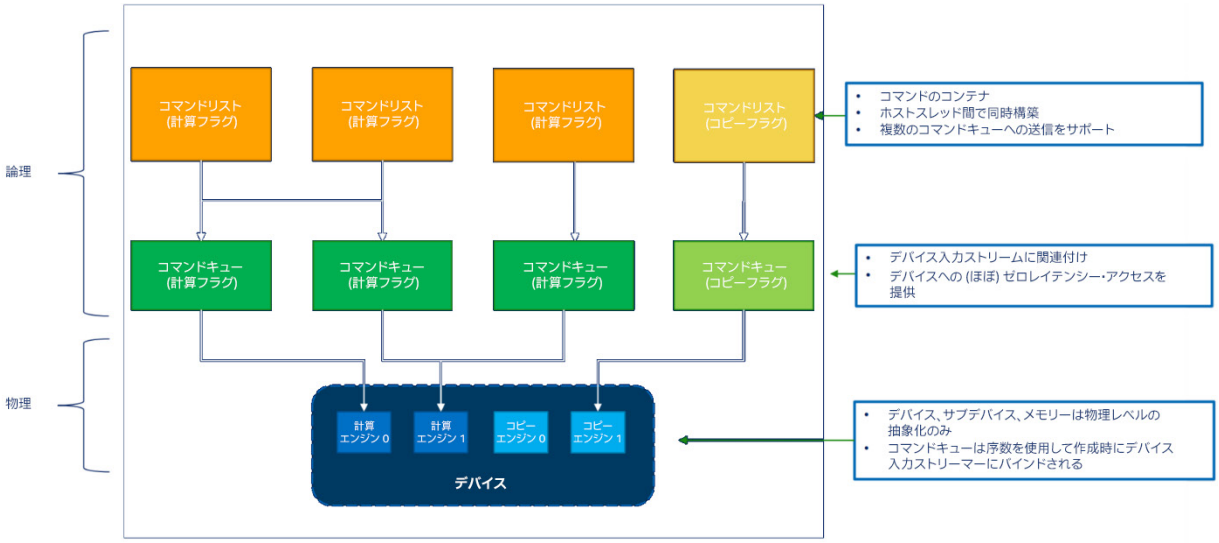

図 2. レベルゼロ・スケジューリング・モデル

コマンドは、オフロード計算ユニットまたはアクセラレーターで実行される一連のコマンドを表すコマンドリストに追加されます。コマンドリストはリセットして再利用できます。リストを再度作成する必要はありません。同じ一連のコマンドの送信に再利用できます。コマンドを再度追加する必要はありません。

コマンドリストは、実行のためコマンドキューに送信されます。キューはデバイスの物理入力ストリームに関連付けられた論理オブジェクトで、同期または非同期として設定でき、キューグループに構成できます。このスケジューリング・モデルは、**図 3** に示すソースコードのフローに変換されます。

図 3. レベルゼロ・スケジューリング実行フロー

即時コマンドリスト

コマンドリストの処理を最適化して、レイテンシーの管理に役立てることができます。保証された応答時間が必要な優先度の高いタスクは、低レイテンシーの即時コマンドリストを使用して処理できます。これは、非常に低レイテンシーの送信使用モデル専用の、特別なタイプのコマンドリストです。

コマンドリストとその暗黙的なコマンドキューは、コマンドキュー記述子を使用して作成します。即時コマンドリストに追加したコマンドは、デバイス上でただちに実行されます。即時コマンドリストに追加したコマンドは、コマンドが完了するまでブロックされ同期的に実行される場合があります。即時コマンドリストは、完了後に閉じたりリセットする必要はありません。使用法は尊重され、想定された動作に従います。次の疑似コードは、即時コマンドリストの作成と使用の基本的な手順を示しています。

```
// 即時コマンドリストを作成
ze_command_queue_desc_t commandQueueDesc = {
    ZE_STRUCTURE_TYPE_COMMAND_QUEUE_DESC,
    nullptr,
    computeQueueGroupOrdinal,
    0, // インデックス
    0, // フラグ
    ZE_COMMAND_QUEUE_MODE_DEFAULT,
    ZE_COMMAND_QUEUE_PRIORITY_NORMAL
};
ze_command_list_handle_t hCommandList;
zeCommandListCreateImmediate(hContext, hDevice, &commandQueueDesc,
                                                &hCommandList);

// カーネルをデバイスにただちにサブミット
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);
…
```

産業用システムのエラー状態を判断して直ちに対処する必要がある閉フィードバック・ループのユースケースが豊富にあると想像してみてください。即時コマンドリストの概念により、保証された応答時間が必要なユースケースに GPU オフロード計算が追加されます。

アプリケーション開発者にとっての利点

レベルゼロの主要な設計原則の説明に続いて、アプリケーション開発者が SYCL* ベースのアプリケーションからレベルゼロと対話する方法を見てみましょう。

デバイスの選択

レベルゼロドライバーが実装されたデバイスはすべて、アプリケーション開発者が初期化して使用できます。インテルのレベルゼロ実装は、GitHub*（英語）で公開されており、ほかのデバイスのリファレンスとして使用して、oneAPI クロスアーキテクチャーのヘテロジニアス計算サポートを活用できます。

特定のオフロードデバイスを選択するには、SYCL_DEVICE_FILTER 環境変数［**翻訳者注：**2023.1 ではこの環境変数は非推奨となり、ONEAPI_DEVICE_SELECTOR に置き換えられています］を使用します。この環境変数は、すべてのデバイスクエリー機能とデバイスセレクターに影響します。実行中のシステムで SYCL* を使用するデバイスが利用可能かどうかを確認するには、sycl-ls コマンドを使用します。

```
$ sycl-ls
[opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA Emulation Device 1.2
[2022.13.3.0.16_160000]
[opencl:cpu:1] Intel(R) OpenCL, 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz 3.0
[2022.13.3.0.16_160000]
[opencl:gpu:2] Intel(R) OpenCL HD Graphics, Intel(R) Iris(R) Xe Graphics 3.0 [31.0.101.3358]
[ext_oneapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) Iris(R) Xe Graphics 1.3 [1.3.23828]
[host:host:0] SYCL host platform, SYCL host device 1.2 [1.2]
```

デバイスのプロパティーに関して、次のように考えることができます。ドライバー・オブジェクトは、システムの物理デバイスのコレクションを表します。複数のドライバーを利用できる場合があります。例えば、あるドライバーがあるベンダーのアクセラレーターをサポートし、別のドライバーが別のベンダーのアクセラレーターをサポートするようなケースです。コンテキスト・オブジェクトは、デバイスまたはシステムリソースを表します。その主な目的は、複数のデバイスで使用される可能性があるリソースの作成と管理です。デバイス・オブジェクトは、システムの物理デバイスを表します。デバイス検出 API は、システムのデバイスを列挙するために使用されます。zeDeviceGet() 関数は、ドライバーがサポートするレベルゼロデバイスの数を照会し、読み取り専用のグローバル構造であるデバイス・オブジェクトを取得するために使用されます。すべてのデバイスには、16 バイトの汎用一意識別子（UUID）が割り当てられます。デバイスハンドルは、デバイス固有のリソースの作成および管理中に使用されます。

レベルゼロの追加機能を利用するには、まず SYCL* 対応の C++ コードとレベルゼロ API 間の相互運用性を有効にする必要があります。C++ アプリケーション内からレベルゼロのデバイス固有のコンテキスト・オブジェクトを直接操作する場合は、ヘッダーファイルをソースコードに次の順序で含める必要があります。

```
#include “level_zero/ze_api.h”
#include “sycl/backend/level_zero.hpp”
```

レベルゼロ・バックエンドは、sycl::backend 列挙子に追加されます。

```
enum class backend {
  // ...
  ext_oneapi_level_zero,
  // ...
};
```

この方法で、SYCL* 名前空間から sycl:get-native API を使用して、SYCL* オブジェクトの基礎となるレベルゼロのデータ構造をリクエストできます。

```
template <backend BackendName, class SyclObjectT>
auto get_native(const SyclObjectT &Obj)
    -> backend_return_t<BackendName, SyclObjectT>
  ext_oneapi_level_zero,
```

詳細は、インテル® oneAPI レベルゼロ・バックエンド仕様（英語）を参照してください。指定された SYCL* キューは、システムの利用可能なデバイスに接続されます。

```
try {
    vector<device> sub_devices = ...;
    for (auto &d : sub_devices)
    {
      // 各キューは自身のコンテキストにあり、データを共有しない
      auto q = queue(d);
      q.submit([&](handler& cgh) {...});
    }
}
```

その後、デバイスが使用され、ほかの SYCL* デバイスと同様に実行キューが割り当てられます。興味深いのは、デバイス固有のリソース、またはホストとオフロード実行デバイス間で共有されるリソースにアクセスする場合です。デバイス固有のリソースにアクセスする例として、統合共有メモリー（USM）を見てみましょう。以下の概要は、インテルの USM 実装の方法を反映していますが、ほかのレベルゼロ・ライブラリーにも同じ機能を実装できます。

統合共有メモリー

メモリーは、CPU と GPU の両方をカバーする単一の仮想アドレス空間を持つ統合メモリーとして上位レベルのソフトウェア・スタックに表示されます。リニアな、フォーマットされていないメモリー割り当ては、ホスト・アプリケーションでポインターとして表されます。ホストのポインターは、デバイスのポインターと同じサイズです。SYCL* 名前空間を使用してメモリーを割り当てる方法は 3 つあります。

1. `sycl::malloc_device`
   - 割り当てには、指定されたデバイスのみアクセスでき、ホストまたはコンテキストのほかのデバイスはアクセスできません。
   - データは常にデバイス上に保持され、カーネル実行に利用できる最速のデータです。
   - ホストまたはコンテキストのほかのデバイスにデータを転送するには、明示的なコピーが必要です。
2. `sycl::malloc_host`
   - 割り当てには、ホストとコンテキストのほかのデバイスからアクセスできます。
   - データは常にホスト上にあり、デバイスから PCI 経由でアクセスします。
   - データをホストまたはデバイスと同期するために明示的なコピーは必要ありません。
3. `sycl::malloc_shared`
   - 割り当てには、ホストと指定されたデバイスのみアクセスできます。
   - ホストとデバイス間でデータを移行（レベルゼロドライバーで操作）して、アクセスを高速化できます。
   - ホストとデバイス間の同期には明示的なコピーは必要ありませんが、コンテキストのほかのデバイスには必要です。

この方法で呼び出される同等の低レベルのレベルゼロ呼び出しはそれぞれ、`zeMemAllocDevice`、`zeMemAllocHost`、および `zeMemAlloc_shared` です。実装の詳細は、レベルゼロコア API 仕様（英語）を参照してください。

USM は、レベルゼロが提供する高度なデバイス認識の一例です。詳細は、レベルゼロ・コア・プログラミング・ガイド（英語）を参照してください。

まとめと次のステップ

レベルゼロは、計算ユニットとアクセラレーターでワークをスケジュールし、メモリーを管理するための豊富なインターフェイスのセットを提供します。プログラムのロードと実行、メモリーの割り当て、およびヘテロジニアス・ワークロードの管理に必要な、すべてのサービスを提供します。特定のハードウェア構成に合わせてカスタマイズできるオープンなインターフェイスを使用しながら、それほど特殊ではないセットアップでワークロードを実行するために必要な抽象化と柔軟性を維持します。

コマンドキューやコマンドリストなど、レベルゼロで定義されたオブジェクトを使用すると、基盤となるハードウェアを低レベルで制御できます。これらの利用可能な最適化手法を使用すると、高レベルのプログラミング言語とアプリケーションは、ハードウェアに近いレイテンシーでワークロードを実行して、パフォーマンスを向上できます。

SYCL* と組み合わせると、C++ を使用してシームレスに使用およびアクセスできます。さらに、Python*、Julia*、および Java* での経験を活用して、さまざまな言語でレベルゼロの優れた言語ランタイムサポートを提供しています。オープンソース・コミュニティーと業界全体がレベルゼロへの貢献度を高めることにより、レベルゼロがマルチアーキテクチャーの選択における汎用的で強力なインターフェイスとなることを期待しています。

関連情報（英語）

- oneAPI レベルゼロおよび OpenCL* ドライバー向けインテル® グラフィックス計算ランタイム
- インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス : インテル® oneAPI レベルゼロ
- oneAPI レベルゼロ仕様
- oneAPI レベルゼロのヘッダー、ローダー、および検証レイヤー