レベルゼロ仕様ドキュメント

oneAPI レベルゼロ: 1.14.33

このドキュメントは、oneAPI GitHub で公開されている「Level Zero Specification documentation 1.14.33」（2025-10-10）の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

本ドキュメントはレイアウト調整および校閲を行っておりません。誤字脱字、製品名や用語の表記、レイアウト等の不具合が含まれる可能性があることを予めご了承ください。

日本語版の改版履歴

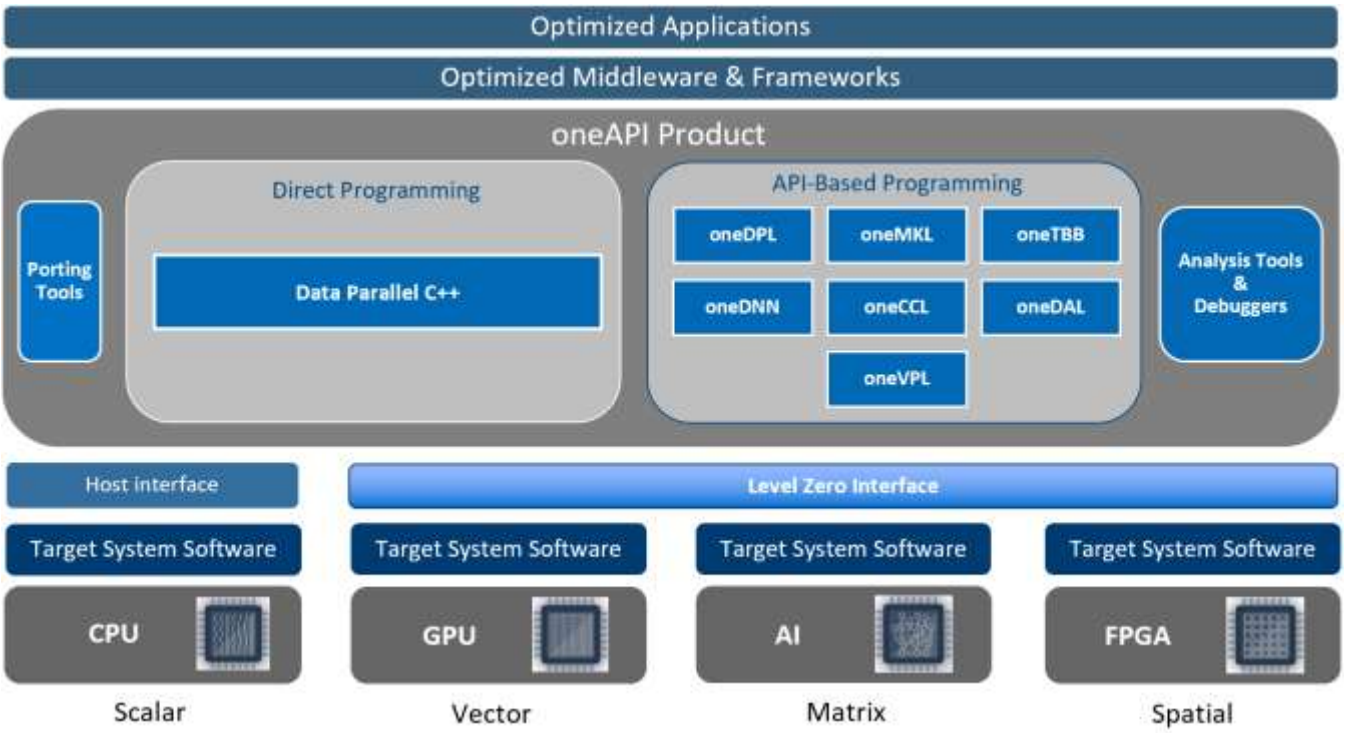
バージョン	日付	説明
1.13.0	2025年11月19日	日本語版の最初のバージョンを公開
1.14.33	2025 年11月27日	1.14.33 のリリースに伴い内容を更新。追加されたセクションの左側に緑色のバーで項目を明示

目次

はじめに

目的

oneAPI レベルゼロ・アプリケーション・プログラミング・インターフェイス（API）の目的は、オフロード・アクセラレーター・デバイスへのダイレクト・メタル・インターフェイスを提供することです。プログラミング・インターフェイスは、あらゆるデバイスのニーズに合わせてカスタマイズでき、関数ポインター、仮想関数、統合メモリー、I/O 機能などのより広範な言語機能をサポートするように適応できます。



ほとんどのアプリケーションでは、レベルゼロ API によって提供される機能の追加制御は必要ありません。レベルゼロ API は、より高レベルのランタイム API およびライブラリーに必要な明示的な制御を提供することを目的としています。

レベルゼロ API は、当初、OpenCL や Vulkan など他の低レベル API の影響を受けていましたが、独立して進化するように設計されています。また、レベルゼロ API は、GPU アーキテクチャーの影響も受けていましたが、FPGA やその他のタイプのアクセラレーター・アーキテクチャーなど、さまざまな計算デバイス・アーキテクチャーでサポートできるように設計されています。

コア

レベルゼロコア API は、以下に示す最も低レベルで、きめ細やか、かつ明示的な制御を提供します:

- デバイスの検出とパーティション化
- メモリー割り当て、可視性およびキャッシュ
- カーネルの実行とスケジュール

- ピアツーピア通信
- プロセス間共有

詳細についてはコア・プログラミング・ガイドを参照してください。

ツール

レベルゼロツール API は、直接的なアプリケーションの使用とサードパーティー・ツールの両方をサポートするため、デバイス機能への低レベルのアクセスを提供します:

- メトリックの検出とレポート
- カーネルのプロファイル、インストルメント、デバッグ

詳細についてはツール・プログラミング・ガイドを参照してください。

システム管理

レベルゼロ Sysman API は、各アクセラレーター・デバイスに対し、以下の機能へのインバンドアクセスを提供します:

- アクセラレーター・リソースのパフォーマンス、電力、および状態を照会
- アクセラレーター・リソースのパフォーマンスと電力プロファイルを制御
- ハードウェア診断の実行、ファームウェアのアップデート、デバイスのリセットなどの保守機能

デフォルトでは、管理者ユーザーのみがリソースの制御操作を行う権限を持っています。ほとんどのクエリーはすべてのユーザーが利用可能ですが、サイドチャネル攻撃に利用される可能性のあるクエリーは例外が発生します。システム管理者は、デフォルトのアクセス権限を強化または緩和できます。

詳細については、Sysman プログラミング・ガイドを参照してください。

基礎原理

次のセクションでは、API 設計の基礎について説明します。詳細については、プログラミング・ガイドおよび仕様ページを参照してください。

ヘッダーファイルは `oneapi-src/level-zero`（英語）にあります。

用語

この仕様では、要件レベルを示すため RFC2119（英語）に基づくキーワードを使用します。特に、この仕様の実装アクションを説明するため次の単語が使用されます:

- **May（してもよい）** - 単語 *may*、または形容詞 *optional（選択できる）* は、準拠した実装では説明どおりに動作することが許可されていますが、必ずしもそうする必要はないことを意味します。
- **Should（する必要がある）** - 単語 *should* または形容詞 *recommended（推奨される）* は、実装が説明されている動作から逸脱する理由がある可能性があるが、そのような逸脱は避けるべきであることを意味します。

- **Must（しなければならない）** - 単語 *must*、または用語 *required（要求されている）* もしくは *shall（することになる）* は、説明されている動作が仕様の絶対的な要件であることを意味します。

命名規則

次の命名規則に従う必要があります:

- すべての関数には `ze` というプリフィクスを付ける必要があります
- すべての関数はキャメルケース（最初を大文字表す）の `zeObjectAction` 規則を使用する必要があります
- すべてのマクロは大文字の `ZE_NAME` 規則を使用する必要があります
- すべての構造体、列挙体、その他の型は、`ze_name_t` スネークケース（単語の区切りにアンダースコアを使用）規則に従う必要があります
- すべての構造体メンバーと関数パラメーターにはキャメルケース規則を使用する必要があります
- すべての列挙値は大文字の `ZE_ENUM_ETOR_NAME` 規則を使用する必要があります
- すべてのハンドルタイプは `handle_t` で終わる必要があります
- すべての記述子構造体は `desc_t` で終わる必要があります
- すべてのプロパティー構造体は `properties_t` で終わる必要があります
- すべてのフラグ列挙子は `flags_t` で終わる必要があります

次のコーディング規則に従わなければなりません:

- すべての記述子構造体は、*:ref:`ze-base-desc-t`* から派生している必要があります
- すべてのプロパティー構造は、*:ref:`ze-base-properties-t`* から派生する必要があります
- すべての関数の入力パラメーターは出力パラメーターの前になければなりません
- すべての関数は `ze result t` を返す必要があります

バージョン管理

互換性を判断するためアプリケーションで使用する必要があるバージョンは複数あります:

API バージョン - デバイスでサポートされている API のバージョンです。

- これは通常、デバイスがアプリケーションに必要な最小限の API セットをサポートしているか判断するために使用されます
- API のコレクション全体を表す単一の 32 ビット値があります
- 値は 16 ビットのメジャー部分と 16 ビットのマイナー部分にエンコードされます
- メジャーバージョンの増分は、廃止予定の機能を含む変更された機能で構成され、下位の互換性が損なわれる可能性があります
- マイナーバージョンの増分には、昇格された拡張機能を含む追加機能が含まれ、下位の互換性を維持する必要があります
- 値は `zeDriverGetApiVersion` の呼び出しで決定されます
- 返される値は、デバイスでサポートされ、ドライバーが認識する `ze api version t` の最小値になります

- ドライバーによって報告されるサポート対象の API バージョンは、報告されたバージョンまでのすべての API が、少なくともサポート対象外であることを示す空の実装を備える最小バージョンを示すものでなければなりません。
- ドライバーが API バージョンを報告する場合、そのバージョンを含むそれまでのすべての API の実装を提供する必要があります。

ドライバーバージョン - システムにインストールされているドライバーのバージョンです。

- 通常、これはある機能に対するドライバー実装の問題を軽減するために使用されます
- 値のエンコードはベンダー固有のものですが、単調なインクリメントである必要があります
- 値は `zeDriverGetProperties` の呼び出しで決定されます

エラー処理

ホスト側のオーバーヘッドを削減するため、次の設計理念が採用されています:

- デフォルトでは、ドライバーの実装はいかなる種類のパラメーター検証も行われない可能性があります
  - これは検証レイヤーによって処理される必要があります
- デフォルトでは、ドライバーおよびデバイスは以下のいずれに対しても保護機能を提供しません:
  - 無効な API プログラミング
  - 無効な関数の引数
  - 関数の無限ループまたは再帰
  - 同期プリミティブ・デッドロック
  - ホストまたはデバイスによる非可視メモリーアクセス
  - デバイスによる非常駐メモリーアクセス
- ドライバーの実装は、いかなる種類の API 検証も実行する必要は**ありません**:
  - ドライバーは、正常に動作するアプリケーションが、動作しないアプリケーションに必要なオーバーヘッドの負担を負わないようにする必要があります
  - 特に指定がない限り、API が不適切に使用されたときのドライバーの動作は未定義です
  - デバッグ用途で、ローダーの検証レイヤーを介して API の検証を有効にできます
- すべての API 関数は `ze_result_t` を返します
  - この列挙子には、レベルゼロ API と検証レイヤーのエラーコードが含まれます
  - これにより、アプリケーション側でエラーをキャッチする一貫したパターンが可能になります。特に検証レイヤーが有効になっている場合です。

マルチスレッドと並行性

ホスト側のオーバーヘッドを削減するため、次の設計理念が採用されています:

- ドライバーのオブジェクト・ハンドルが異なる場合、API はフリースレッド化されます。
  - ドライバーはこれらの API 呼び出しのスレッドロックを回避する必要があります。
- 明示的に指定される場合を除き、ドライバーのオブジェクト・ハンドルが同じである場合、API はスレッドセーフではありません。

    - アプリケーションは、ハンドルが同一の場合に複数のスレッドが API に入らないようにする必要があります
- API は、同じドライバーのオブジェクト・ハンドルを使用する他の API とはスレッドセーフではありません
    - アプリケーションは、ハンドルが同一の場合に複数のスレッドがこれらの API に入らないようにする必要があります
- API は、ハンドルの参照カウントをサポートしていません。
    - アプリケーションは所有権を追跡し、ハンドルとメモリーを明示的に解放する必要があります
    - アプリケーションは、ハンドルとメモリーを解放する前に、すべてのドライバー・オブジェクトとメモリーがデバイスによって使用されていないことを確認する必要があります。使用中である場合、ホストまたはデバイスでエラーが発生する可能性があります
    - ドライバーは暗黙的なガベージ・コレクションをサポートしていません

一般に、API はスレッドセーフではなくフリースレッドとして設計されています。これにより、マルチスレッド・アプリケーションはスレッドとロックの両方を完全に制御できるようになります。これにより、シングル・スレッド・アプリケーションや非常に低いレイテンシーにおける不要なドライバーのオーバーヘッドも排除されます。

この規則の例外は、すべてのメモリー割り当て API が単一のグローバル・メモリー・ プールから割り当てられるため、スレッドセーフであるということです。アプリケーションでロックフリーのメモリー割り当てが必要な場合には、スレッドごとにプールを割り当てる独自のサブ・アロケーターを実装できます。

各 API 関数では、呼び出しのマルチスレッド要件の詳細を文書化する必要があります。

これらの規則によって有効になる主な使用モデルは次のとおりです:

- 独立したドライバー・オブジェクト上で、複数の同時スレッドが暗黙のスレッドロックなしで動作する可能性があります。
- ドライバーのオブジェクト・ハンドルは、暗黙のスレッドロックなしで複数のスレッド間で渡され、使用される可能性もあります

アプリケーション・バイナリー・インターフェイス

レベルゼロ 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。C/C++ アプリケーションには“`ze_api.h`”がインクルードされ、“`ze_api.lib`”とリンクされる必要があります。レベルゼロ C デバイス・ドライバー・インターフェイス（DDI）は、共有ローダーおよびドライバー・ライブラリーによってインポート・ライブラリーに提供されます。C/C++ ローダーとドライバーでは “`ze_ddi.h`”インクルードする必要があります。

これらのライブラリーの実装では、プラットフォームの標準 C コンパイラーのデフォルトのアプリケーション・バイナリー・インターフェイス（ABI）を使用する必要があります。ここで ABI とは、C データタイプのサイズ、アラインメント、レイアウト、プロシージャー呼び出し規約、および C 関数に対応する共有ライブラリー・シンボルの命名規則を意味します。ABI は、新しい関数の追加、新しい列挙子の追加、ビットフィールドでの予約ビットの使用など、API のマイナーバージョンの増分に対して下位互換性があります。ABI は、既存の関数シグネチャーや構造の変更、関数や構造の削除など、API のメジャーバージョンの増分に対して下位互換性があることは保証されません。

レベルゼロが共有ライブラリーとして提供されるプラットフォームでは、“ze”、“zet”、または “zes” で始まり、その後に数字または大文字が続くライブラリー・シンボルは、実装向けに予約されています。レベルゼロを使用するアプリケーションは、これらのシンボル定義を提供してはなりません。これにより、既存のアプリケーションとシンボル競合が発生することなく、新しい API バージョンまたは拡張機能の追加シンボルでレベルゼロが共有ライブラリーを更新できるようになります。

コア・プログラミング・ガイド

ドライバーとデバイス

API アーキテクチャーは、ベースとなるデバイス機能の物理的および論理的な抽象化の両方を公開します。デバイス、サブデバイスおよびメモリーは、物理レベルで定義されますが、コマンドキュー、イベントおよび同期メソッドは論理エンティティーとして定義されます。すべての論理エンティティーは、デバイスレベルの物理機能にバインドされます。

デバイス検出 API は、アクセラレーターの機能を列挙します。これらの API は、デバイスまたはサブデバイス内の計算ユニット数、計算で使用可能なメモリーとアフィニティー、ユーザーが管理するキャッシュサイズ、ワーク送信コマンドキューなどの情報を照会するインターフェイスを提供します。

ドライバー

ドライバー・オブジェクトは、同じレベルゼロ（L0）のドライバーでアクセスされるシステム内の物理デバイスのコレクションを表します。

- アプリケーションは、`zeInitDrivers` 関数を使用して、システムにインストールされているレベルゼロドライバーの数とそれぞれのハンドルを取得できます。`zeDriverGet` 関数は L0 仕様のバージョン 1.10 以降では非推奨となっています。
- `zeInitDrivers` は、L0 仕様のバージョン 1.10 以降、`zeInit` と `zeDriverGet` の両方の機能を置き換えるものであり、ドライバーの初期化とドライバー数の照会を同時に行います。
- `zeInitDrivers` と `zeDriverGet` は相互に排他的であり、両方を同時に使用してはなりません。これらを併用すると、未定義の動作となります。
- システムでは複数のドライバーが利用できる場合があります。たとえば、1 つのドライバーでは 1 つのベンダーの GPU を 2 つサポートし、別のドライバーでは異なるベンダーの GPU を 1 つサポートし、さらに別のドライバーでは NPU をサポートする場合があります。
- ドライバー・オブジェクトは読み取り専用のグローバル構造です。つまり、`zeInitDrivers` を複数回呼び出すと、同一のドライバーハンドルが返されます。
- ドライバーハンドルは、主にデバイスの検出時とコンテキストの作成および管理時に使用されます。

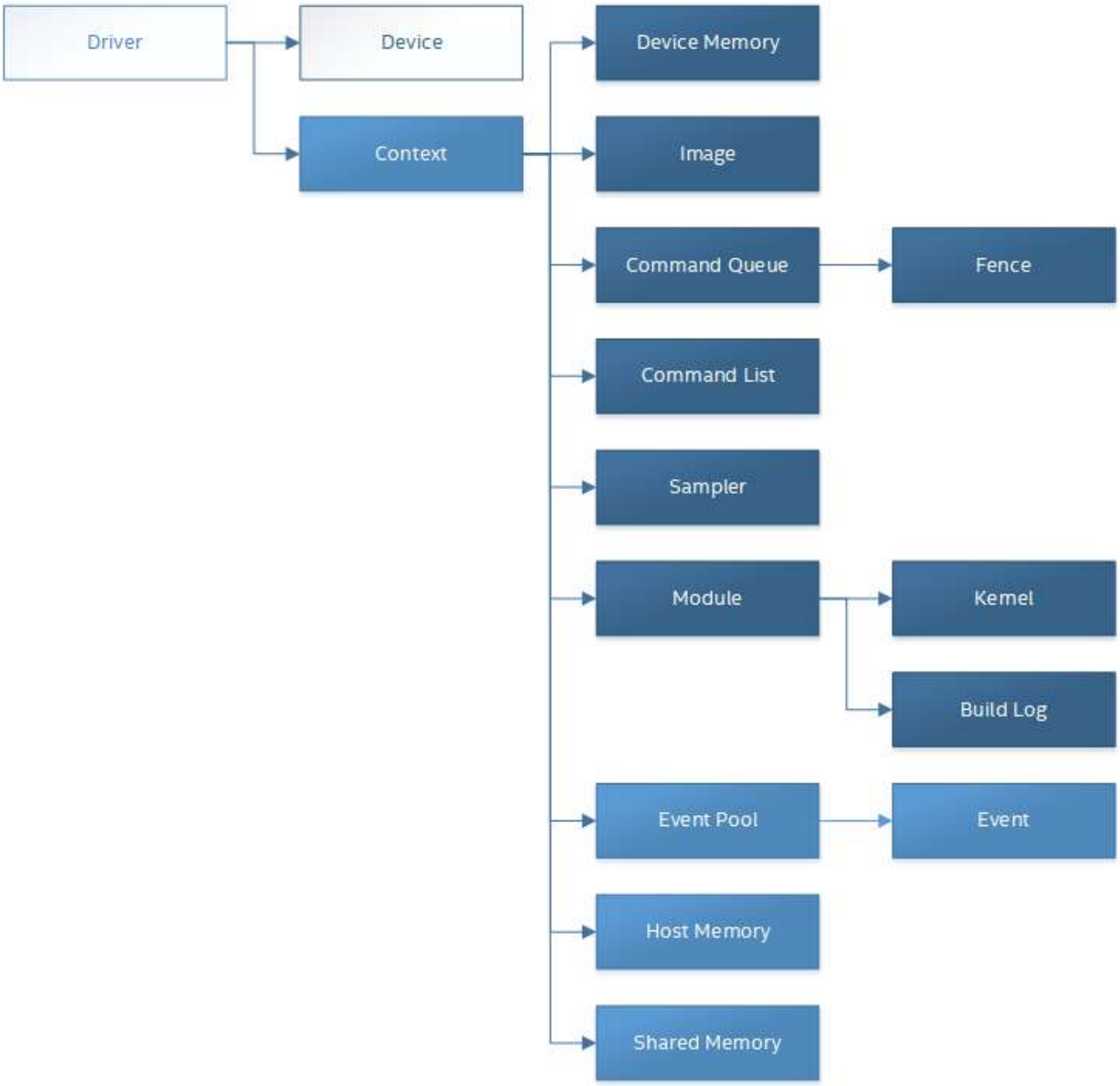
デバイス

デバイス・オブジェクトは、レベルゼロをサポートするシステム内の物理デバイスを表します。

- アプリケーションは、`zeDeviceGet` を使用して、ドライバーでサポートされているデバイスの数とそれぞれのハンドルを照会できます。
- デバイス・オブジェクトは読み取り専用のグローバル構造です。つまり、`zeDeviceGet` を複数回呼び出すと、同一のデバイスハンドルが返されます。

- デバイスハンドルは主に、デバイス固有のリソースの作成および管理に使用されます。
- このアプリケーションは、複数のデバイス間でのメモリー共有、明示的なデータ送信、および同期を行います。
- デバイスは、デバイスの物理パーティションまたは論理パーティションを細かく制御できるサブデバイスを公開している場合があります。

次の図は、このドキュメントで説明されているドライバー、デバイス、およびその他のオブジェクト間の関係を示しています。



レベルゼロのデバイスモデル階層は、**ルートデバイス**と**サブデバイス**で構成されています: ルートデバイスは 2 つ以上のサブデバイスを含むことができ、サブデバイスは 1 つのルートデバイスにのみ属します。ルートデバイスは、単一のサブデバイスを含むことはできません。その理由は、結局同じルートデバイスになるからです。ルートデバイ

スは、サブデバイスを持たないデバイスであることもあります。

ルートデバイスに属するサブデバイスは、zeDeviceGetSubDevices 関数を使用して照会できます。サブデバイスのルートデバイスは、zeDeviceGetRootDevice 関数を使用して照会できます。特定デバイスのルートデバイスとサブデバイスの定義は、実装によって異なります。

初期化と検出

レベルゼロ API を使用する際は、他の API 関数を呼び出す前に、必ず zeInitDrivers 関数を呼び出して初期化する必要があります。注: zeInit 関数は、L0 仕様のバージョン 1.10 以降では非推奨です。これらの関数は、システム内のすべてのレベルゼロドライバーを現在のプロセスのメモリーにロードし、すべてのホストスレッドで使用できるようにします。zeInitDrivers の同時呼び出しはスレッドセーフであり、各ドライバーのインスタンスは 1 つだけロードされます。

以下の擬似コードは、コアドライバーにおける基本的な初期化およびデバイス検出シーケンスを示しています:

```
// すべてのドライバー・インスタンスを検出
ze_init_driver_type_desc_t desc = {ZE_STRUCTURE_TYPE_INIT_DRIVER_TYPE_DESC};
desc.pNext = nullptr;
desc.driverType = UINT32_MAX; // 要求されたすべてのドライバータイプ
uint32_t driverCount = 0;
ze_result_t result = zeInitDrivers(&driverCount, nullptr, &desc); // ドライバーの数を照会
if (result != ZE_RESULT_SUCCESS) {
    return result; // ドライバーが見つかりません
}
ze_driver_handle_t* allDrivers = allocate(driverCount * sizeof(ze_driver_handle_t));
result = zeInitDrivers(&driverCount, allDrivers, &desc); // ドライバーハンドルをリード
if (result != ZE_RESULT_SUCCESS) {
    return result; // ドライバーハンドルが見つかりません
}

// GPU デバイスタイプをサポートするドライバーハンドルを検出
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 デバイスが見つかりません

...
```

コンテキスト

コンテキストとは、ドライバーがすべてのメモリー、コマンドキュー/リスト、モジュール、同期オブジェクトなどの管理に使用する論理オブジェクトです。

- コンテキストハンドルは、主に複数のデバイスで使用される可能性のあるリソースの作成および管理に使用されます。
- 例えば、メモリーはドライバーがサポートするすべてのデバイス間で暗黙的に共有されるわけではありませんが、明示的に共有することはできます。

以下の擬似コードは、基本的なコンテキスト作成の手順を示しています:

```
// コンテキスト作成
ze_context_desc_t ctxtDesc = {
   ZE_STRUCTURE_TYPE_CONTEXT_DESC,
   nullptr,
   0
};
zeContextCreate(hDriver, &ctxtDesc, &hContext);
```

アプリケーションは、`zeContextCreate` を使用して複数のコンテキストを任意に作成できます。

- 複数のコンテキストにおける主な使用モデルは、同一プロセス内で複数のライブラリーのメモリーとオブジェクトを分離することです。
- 同じコンテキストを複数のホストスレッドで同時に使用することができます。

以下の擬似コードは、基本的なコンテキストの作成とアクティベーションの手順を示しています:

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

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

memcpy(ptrA, ptrB, 0xe); // OK:
zeMemGetAllocProperties(hContextA, ptrB, &props, &hDevice); // illegal: コンテキスト A は ptrB について何も情報を持っていません
```

デバイスがハングアップしたりリセットされた場合、コンテキストは有効ではなくなり、そのコンテキストに関連付けられたオブジェクトが使用されると、API は `ZE_RESULT_ERROR_DEVICE_LOST` を返します。このコンテキストで作成されたメモリー割り当てへのすべてのポインター、およびオブジェクト（他のコンテキストを含む）へのハンドルはすべて無効になり、以降は使用できません。アプリケーションは、`zeContextGetStatus` 関数をいつでも使用してコンテキストの状態を確認できます。

復旧するには、`zeContextDestroy` を使用してコンテキストを破棄する必要があります。デバイスがリセットされた後、アプリケーションは新しいコンテキストを作成して動作を続行できます。アプリケーションは、デバイスがリセットされたことを確認し、デバイスハンドルに関連付けられた OS ハンドルを更新するため、`zeDeviceGetStatus` を呼び出す必要があります。そうしないと、デバイスのリセット後の `zeContextCreate` 呼び出しは失敗します。

メモリーとイメージ

上位レベルのソフトウェア・スタックからメモリーは、ホストと特定のデバイス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認識されます。

GPU 向けに、この API はデバイスメモリー階層の 2 レベルを公開しています:

1. ローカル・デバイス・メモリー: デバイスレベルおよび/またはサブデバイス・レベルで管理できます。
2. デバイスキャッシュ:
    - ラスト・レベル・キャッシュ（L3 キャッシュ）は、メモリー割り当て API を通じて制御できます。
    - ラスト・レベル・キャッシュ（L1 キャッシュ）は、プログラミング言語の組込み関数を通じて制御できます。

この API を使用すると、デバイスおよびサブデバイスの粒度でバッファーと画像を割り当てることができ、完全な

キャッシュ可能性のヒントも利用できます。

- バッファーは、仮想アドレスポインターを介してアクセスされる透過的なメモリーです
- イメージは、ハンドルを介してアクセスされる不透明なオブジェクトです

メモリー API は、デバイスメモリー、ホストメモリー、または共有メモリーを割り当てる割り当てメソッドを提供します。これらの API により、アプリケーションまたはランタイムによるリソースの暗黙的および明示的な管理が可能になります。このインターフェイスは、すべてのメモリー・オブジェクトに対する照会機能も提供します。

次の 2 つ割り当てタイプがあります:

1. メモリー - ホストとデバイスの両方から直接アクセスできる、線形かつフォーマットなしメモリー割り当て。
2. イメージ - デバイスからの直接アクセスに対応した、非線形かつフォーマット済みのメモリー割り当て。

メモリー

線形かつ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はポインターとして表現されます。ホスト上のポインターサイズは、デバイス上のポインターサイズと同じです。

タイプ

3 種類の割り当てがサポートされています。割り当ての種類は、その割り当ての*所有権*を表します:

1. **ホスト**割り当ては、ホストが所有し、システムメモリーから割り当てることを意図しています。
    - ホスト割り当ては、ホストと 1 つ以上のデバイスからアクセスできます。
    - 同じホスト割り当てへのポインターは、ホストとサポートされるすべてのデバイスで利用でき、これらの*アドレスは同一*です。
    - ホスト割り当ては、システムメモリーとデバイス・ローカル・メモリー間で移行されることは想定されていません。
    - ホスト割り当ては、広範なアクセス性とデータ転送のメリットを享受できる反面、PCI Express などを介したアクセスでは、アクセスごとのコストが高くなる可能性があるというトレードオフの関係にあります。
2. **デバイス**割り当ては、デバイスが所有し、デバイス・ローカル・メモリー外に割り当てることを意図しています。
    - デバイスの割り当ては、一般にアクセス制限と引き換えにパフォーマンスの向上を実現します。
    - ごく一部の例外を除き、デバイス割り当て領域には、割り当てられた特定のデバイスからのみアクセスできるか、あるいは別のデバイスまたはホストの割り当て領域にコピーされるかのいずれかです。
    - 同じデバイス割り当てへのポインターは、サポートされているいずれのデバイスでも使用できます。
3. **共有**割り当ては、所有権を共有し、ホストと 1 つ以上のデバイス間で移行することを意図しています。
    - 共有割り当てには、ホストと関連付けられたデバイスの両方からアクセス可能です。

- 共有割り当ては、場合によっては他のデバイスからアクセスされる可能性があります。
- 共有割り当ては、転送コストとアクセスごとのメリットとのトレードオフがあります。
- 共有割り当てへのポインターは、ホストとデバイスで同じポインターを使用できます。

共有システム割り当ては、**共有**割り当てのサブクラスであり、メモリーは割り当て API ではなく、*システム・アロケーター*（`malloc` や `new` など）を使用します。共有システム割り当ては、関連するデバイスを持たず、本質的にクロスデバイスです。他の共有割り当てと同様に、共有システム割り当てはホストとサポートされているデバイス間で移行することを目的としており、共有システム割り当てへの同じポインターは、ホストとサポートされているすべてのデバイスで使用できます。

以下に概要を説明します。

名称	初期の場所	アクセス元		移行先	
ホスト	ホスト	ホスト	はい	ホスト	N/A
		任意のデバイス	はい（PCIe 経由）	デバイス	いいえ
デバイス	特定のデバイス	ホスト	いいえ	ホスト	いいえ
		特定のデバイス	はい	デバイス	N/A
		その他のデバイス	オプション（p2p が必要）	その他のデバイス	いいえ
共有	ホスト、特定のデバイス、または未指定	ホスト	はい	ホスト	はい
		特定のデバイス	はい	デバイス	はい
		その他のデバイス	オプション（p2p が必要）	その他のデバイス	オプション
共有システム	ホスト	ホスト	はい	ホスト	はい
		デバイス	はい	デバイス	はい

少なくても、ドライバーは各デバイスおよび共有メモリー割り当てに対して、それぞれ固有の物理ページを割り当てる必要があります。ただし、アプリケーションが要求された割り当てサイズを超えるメモリーにアクセスすると、未定義の動作となります。割り当てに使用される実際のページサイズは、`zeMemGetAllocProperties` 関数を使用して `ze_memory_allocation_properties_t.pageSize` から取得できます。アプリケーションは、デバイス・メモリー・プールから用途に応じたアロケーター（例えば、小規模または固定サイズの割り当て、ロックフリーなど）を実装する必要があります。

さらに、ドライバーは一部の**共有**割り当てを*過剰*に要求する可能性があります。このような割り当て超過がいつどのように発生するのか、またワーキングセットが変化した際にどの割り当てが解放されるかといったことは、実装の詳細事項とみなされます。

アクセス機能

デバイスは、割り当ての種類ごとに異なるアクセス機能をサポートする場合があります。サポートされる機能を以下に示します:

1. **ホスト割り当て**: ホスト上で `zeMemAllocHost` によって割り当てられ、デバイス `hDevice` からアクセスされるバッファーを想定します:
    - `ZE_MEMORY_ACCESS_CAP_FLAG_RW`: バッファーは、`hDevice` からアクセスできるだけでなく、ホストからもアクセス（読み取りおよび書き込み）できます。
    - `ZE_MEMORY_ACCESS_CAP_FLAG_ATOMIC`: バッファーには、`hDevice` からアトミックにアクセスできます。アトミック操作には、緩和された一貫性を持つ読み取り・変更・書き込みアトミック操作や、非アトミック操作に対するメモリーの一貫性を強制するアトミック操作が含まれる場合があります。
    - `ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT`: バッファーには、同時アクセスをサポートする他のデバイスやホスト自身からも、`hDevice` を介して同時にアクセスできます。同時アクセスは、割り当て全体単位で行われます。この機能は、一貫性やメモリーの整合性を保証するものではありません。同時アクセスをサポートしていないデバイスから割り当てられた領域に対して同時アクセスが行われると、未定義の動作が発生します。同時アクセスをサポートしていても、アトミックな同時アクセスをサポートしないデバイスは、データ競合やそれに伴う未定義の動作を避けるため、重複しないメモリー位置に書き込む必要があります。
    - `ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT_ATOMIC`: バッファーには、同時アクセスをサポートする他のデバイスやホスト自身からも、hDevice を介して同時にアトミックアクセスができます。同時アトミックアクセスは、割り当て全体単位で行われます。メモリーの一貫性は、アトミック操作を用いることで、同時アトミックアクセスをサポートするホストとデバイス間で確保できます。同時アトミックアクセスをサポートしていないデバイスから割り当てられた領域に対して、同時アトミックアクセスが行われると、未定義の動作が発生します。
2. **デバイス割り当て**: `zeMemAllocDevice` を介してデバイス `hDevice` 上に割り当てられたバッファーを想定します:
    - `ZE_MEMORY_ACCESS_CAP_FLAG_RW`: バッファーは、`hDevice` からアクセス（読み取りおよび書き込み）できます。
    - `ZE_MEMORY_ACCESS_CAP_FLAG_ATOMIC`: バッファーには、`hDevice` からアトミックにアクセスできます。アトミック操作には、緩和された一貫性を持つ読み取り・変更・書き込みアトミック操作や、非アトミック操作に対するメモリーの一貫性を強制するアトミック操作が含まれる場合があります。
    - `ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT`: バッファーには、同時アクセスをサポートする他のデバイスからも、hDevice を介して同時にアクセスできます。対称性により、バッファーはどちらのデバイスにも配置可能であり、両方のデバイスから同時にアクセスできます。同時アクセスは、割り当て全体単位で行われます。この機能は、一貫性やメモリーの整合性を保証するものではありません。同時アクセスをサポートしていないデバイスから割り当てられた領域に対して同時アクセスが行われると、未定義の動作が発生します。同時アクセスをサポートしていても、アトミックな同時アクセスをサポートしないデバイスは、データ競合やそれに伴う未定義の動作を避けるため、重複しないメモリー位置に書き込む必要があります。デバイスは、両方のデバイスが同時アクセスをサポートし、かつ両方のデバイスがピアツーピア・アクセスもサポートしている場合に限り、別のデバ

イス上のバッファーに同時にアクセスできます。あるデバイスが同時アクセスを許可せず、ピアツーピア・アクセスを許可する場合、そのデバイスはピアツーピア・アクセスをサポートしますが、同一バッファーへの同時アクセスはサポートしません。

- `ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT_ATOMIC`: バッファーには、同時アトミックアクセスをサポートする他のデバイスからも、hDevice を介してアトミックアクセスできます。対称性により、バッファーはどちらのデバイスにも配置可能であり、両方のデバイスから同時にアトミックアクセスできます。同時アトミックアクセスは、割り当て全体単位で行われます。メモリーの一貫性は、アトミック操作を用いることで、同時アトミックアクセスをサポートするデバイス間で確保できます。同時アトミックアクセスをサポートしていないデバイスから割り当てられた領域に対して、同時アトミックアクセスが行われると、未定義の動作が発生します。デバイスは、両方のデバイスが同時アクセスをサポートし、かつ両方のデバイスがピアツーピア・アクセスもサポートしている場合に限り、別のデバイス上のバッファーにアトミックアクセスできます。あるデバイスが同時アトミックアクセスを許可せず、ピアツーピア・アトミック・アクセスを許可する場合、そのデバイスはピアツーピア・アトミック・アクセスをサポートしますが、同一バッファーへの同時アクセスはサポートしません。

3. **共有シングルデバイス割り当て**: `zeMemAllocShared` を介して作成されたホストとデバイス間で共有される `hDevice` を想定します:

   - `ZE_MEMORY_ACCESS_CAP_FLAG_RW`: バッファーは、`hDevice` からアクセスできるだけでなく、ホストからもアクセス（読み取りおよび書き込み）できます。
   - `ZE_MEMORY_ACCESS_CAP_FLAG_ATOMIC`: バッファーには、`hDevice` からだけでなくホストからもアトミックにアクセスできます。アトミック操作には、緩和された一貫性を持つ読み取り・変更・書き込みアトミック操作や、非アトミック操作に対するメモリーの一貫性を強制するアトミック操作が含まれる場合があります。
   - `ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT`: バッファーには、ホストと同時に `hDevice` からもにアクセスできます。同時アクセスは、割り当て全体単位で行われます。この機能は、一貫性やメモリーの整合性を保証するものではありません。ホストと `hDevice` から割り当て領域への同時アクセスが行われ、`hDevice` が同時アクセスをサポートしていない場合、未定義の動作となります。同時アクセスをサポートしていても、アトミックな同時アクセスをサポートしないデバイスは、データ競合やそれに伴う未定義の動作を避けるため、重複しないメモリー位置に書き込む必要があります。
   - `ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT_ATOMIC`: バッファーには、ホストと同時に hDevice からもアトミックにアクセスできます。同時アトミックアクセスは、割り当て全体単位で行われます。メモリーの一貫性は、アトミック操作を用いることで、同時アトミックアクセスをサポートするデバイス間で確保できます。ホストおよび `hDevice` が同時アトミックアクセスをサポートしていない場合、割り当て領域に対してホストと `hDevice` から同時アトミックアクセスが行われると未定義の動作が発生します。

4. **デバイス間共有割り当て**: ホストと `zeMemAllocShared` によって作成されたデバイス間共有アクセス機能をサポートするデバイス間で共有されるメモリー領域を想定します。このメモリー領域は、デバイス `hDevice` からアクセスされます:

- ZE_MEMORY_ACCESS_CAP_FLAG_RW: バッファーは、hDevice からアクセスできるだけでなく、ホストからもアクセス（読み取りおよび書き込み）できます。
- ZE_MEMORY_ACCESS_CAP_FLAG_ATOMIC: バッファーには、hDevice からだけでなくホストからもアトミックにアクセスできます。アトミック操作には、緩和された一貫性を持つ読み取り・変更・書き込みアトミック操作や、非アトミック操作に対するメモリーの一貫性を強制するアトミック操作が含まれる場合があります。
- ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT: バッファーには、同時アクセスをサポートする他のデバイスやホスト自身からも、hDevice と同時にアクセスできます。同時アクセスは、割り当て全体単位で行われます。この機能は、一貫性やメモリーの整合性を保証するものではありません。同時アクセスをサポートしていないデバイスから割り当てられた領域に対して同時アクセスが行われると、未定義の動作が発生します。同時アクセスをサポートしていても、アトミックな同時アクセスをサポートしないデバイスは、データ競合やそれに伴う未定義の動作を避けるため、重複しないメモリー位置に書き込む必要があります。
- ZE_MEMORY_ACCESS_CAP_FLAG_CONCURRENT_ATOMIC: バッファーには、同時アクセスをサポートする他のデバイスやホスト自身からも、hDevice と同時にアトミックアクセスができます。同時アトミックアクセスは、割り当て全体単位で行われます。メモリーの一貫性は、アトミック操作を用いることで、同時アトミックアクセスをサポートするデバイス間で確保できます。同時アトミックアクセスをサポートしていないデバイスから割り当てられた領域に対して、同時アトミックアクセスが行われると、未定義の動作が発生します。

必要とされるマトリックスは以下のとおりです:

割り当てタイプ	RW アクセス	アトミックアクセス	同時アクセス	同時アトミックアクセス
ホスト	必須	オプション	オプション	オプション
デバイス	必須	オプション	オプション	オプション
共有	必須	オプション	オプション	オプション
共有 （デバイス間）	オプション	オプション	オプション	オプション
共有システム （デバイス間）	オプション	オプション	オプション	オプション

キャッシュヒント、プリフェッチ、およびメモリーに関するアドバイス

メモリー割り当て時に、ホスト側とデバイス側それぞれの割り当てフラグを個別に指定することで、キャッシュの可能性に関するヒントを提供できる場合があります。

共有割り当て領域には、zeCommandListAppendMemoryPrefetch API を介して、対応するデバイスにプリフェッチできます。プリフェッチにより、メモリー転送を他の計算処理と並行して実行できるようになり、パフォーマンスが向上する可能性があります。

さらに、アプリケーションは、zeCommandListAppendMemAdvise API を介して**共有**割り当てメモリーに関するアドバイスを提供することで、ドライバーのヒューリスティックや移行ポリシーをオーバーライドできます。メモリーに関するアドバイスは、不要または非効率的なメモリー転送を回避し、パフォーマンス向上に役立つ可能性があります。

プリフェッチとメモリーアドバイスはどちらも非同期操作であり、コマンドリストに追加されます。

予約済みデバイス割り当て

アプリケーションがデバイス割り当てのため物理メモリー消費量をより細かく制御する必要がある場合、仮想アドレス空間の一部を予約し、必要に応じて物理メモリーにマッピングできます。これにより、アプリケーションは、時間の経過とともにサイズが変化する大規模な動的データ構造を、物理メモリーの使用量を最適に管理する柔軟性を得ることができます。

仮想アドレス空間の予約

仮想メモリーは `zeVirtualMemReserve` によって確保できます。予約領域の開始アドレスとサイズは、ページ境界に合わせる必要があります。アプリケーションは、`zeVirtualMemQueryPageSize` を使用して、割り当てページサイズを照会しなければなりません。

以下の擬似コードは、仮想メモリーを確保する基本的な手順を示しています:

```
// 1MB のメモリー割り当てに必要なページサイズを照会
size_t pageSize;
size_t allocationSize = 1048576;
zeVirtualMemQueryPageSize(hContext, hDevice, allocationSize, &pageSize);

// 仮想アドレス空間を 1MB 確保
size_t reserveSize = align(allocationSize, pageSize);

void* ptr = nullptr;
zeVirtualMemReserve(hContext, nullptr, reserveSize, &ptr);
```

仮想アドレス予約の増加

アプリケーションは、特定のアドレスから始まるアドレス範囲を予約したい場合があります。これは、予約数を増やす場合に役立つ可能性があります。ただし、実装が要求された開始アドレスに新しい割り当て領域を確保できない場合、別の開始アドレスを持つ適切な範囲を新たに検索することになります。アプリケーションが特定のアドレスから開始する場合、`zeVirtualMemReserve` からの戻りアドレスが、要求する開始アドレスと一致することを確認する必要があります。両者が異なる場合、アプリケーションはより大きな新しい予約領域を作成し、最初の予約領域からこの新しい予約領域に物理メモリーを再マッピングし、古い予約領域を解放します。

```
// 以前確保した領域に隣接し、さらに 1MB の仮想アドレス空間を確保
void* newptr = (uint8_t*)ptr + reserveSize;
void* retptr;
zeVirtualMemReserve(hContext, newptr, reserveSize, &retptr);

if (retptr != newptr)
{
    // 開始アドレスが異なっているため、不要になった新しい予約を解放
    zeVirtualMemFree(hContext, retptr, reserveSize);
```

```
    // 新たに 2MB より大きなメモリー領域を確保し、物理ページを再マッピング
    size_t pageSize;
    size_t largerAllocationSize = 2097152;
    zeVirtualMemQueryPageSize(hContext, hDevice, largerAllocationSize, &pageSize);

    // 仮想アドレス空間を 2MB 確保
    size_t largerReserveSize = align(largerAllocationSize, pageSize);

    void* ptr = nullptr;
    zeVirtualMemReserve(hContext, nullptr, largerReserveSize, &ptr);

    // 物理ページを元の予約領域から新しく大きな予約領域に再マッピング
    ...

    // 拡大しようとしていた元の予約領域を解放
    zeVirtualMemFree(hContext, ptr, reserveSize);
}
```

物理メモリー

物理メモリーは、API において物理ページを予約する物理メモリー・オブジェクトとして明示的に表現されます。このアプリケーションは、`zePhysicalMemCreate` 関数を使用して物理メモリー・オブジェクトを作成します。

以下の擬似コードは、物理メモリー・オブジェクトを作成する基本的な手順を示しています:

```
// 1MB の物理メモリー・オブジェクトを作成
ze_physical_mem_handle_t hPhysicalAlloc;
size_t physicalSize = align(allocationSize, pageSize);
ze_physical_mem_desc_t pmemDesc = {
    ZE_STRUCTURE_TYPE_PHYSICAL_MEM_DESC,
    nullptr,
    0, // フラグ
    physicalSize // サイズ
};

zePhysicalMemCreate(hContext, hDevice, &pmemDesc, &hPhysicalAlloc);
```

仮想メモリーページのマッピング

予約済みの仮想メモリーページは、`zeVirtualMemMap` を使用して物理メモリーにマッピングできます。アプリケーションは、予約済みの仮想アドレス範囲全体をマッピングすることも、1 つ以上の物理メモリー・オブジェクトを使用して予約済みの仮想アドレス範囲を部分的にマッピングすることもできます。マッピングが完了すると、物理メモリー・オブジェクトに対応する物理ページは、オンデマンド・ページングをサポートするデバイスにページイン（読

み込み）できます。さらに、この常駐 API を使用することで、これらの物理ページの常駐状態を制御できます。

以下の擬似コードは、1MB のメモリー領域を物理メモリーにマッピングする手順を示しています:

```
// 1MB の予約領域全体をマッピングし、アクセス権を読み書き可能に設定
zeVirtualMemMap(hContext, ptr, reserveSize, hPhysicalAlloc, 0,
    ZE_MEMORY_ACCESS_ATTRIBUTE_READWRITE);
```

アクセス属性

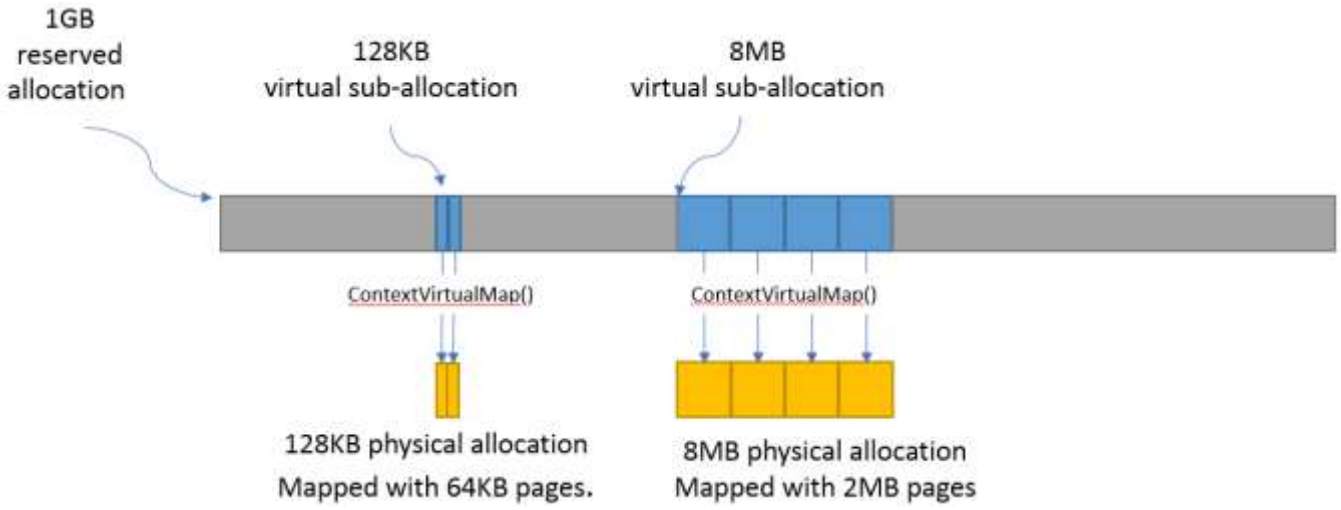
アクセス属性は、zeVirtualMemMap または zeVirtualMemSetAccessAttribute を使用して仮想メモリーページをマッピングする際に、一連のページに対して設定できます。さらに、アプリケーションは、ページ境界に整列された仮想メモリー範囲のアクセス属性を照会できます。

```
size_t accessRangeSize;
ze_memory_access_attribute_t access;
zeVirtualMemGetAccessAttribute(hContext, ptr, reserveSize, &access, &accessRangeSize);

// 範囲全体が同じアクセス属性を持ち、読み書き可能であることを想定しています
assert(accessRangeSize == reserveSize);
assert(access == ZE_MEMORY_ACCESS_ATTRIBUTE_READWRITE);
```

スパースマッピング

アプリケーションは、独自のメモリー割り当てで使用するため、大きな仮想アドレス範囲を確保する場合があります。これらのメモリー領域は、1 つまたは複数の物理メモリー・オブジェクトを使用して、スパース（疎）にマッピングできます。アプリケーションは、各サブ割り当てのページサイズを照会することを推奨します。これにより、実装は、開始アドレス、サイズ、そしてアライメントに基づいて、マッピングに最適なページサイズを使用できます。

以下の例では、まず 1GB の予約領域を確保し、次に 128KB と 8MB のサブ領域を割り当てています。




```
// 管理のため仮想アドレス空間を 1GB 確保
size_t pageSize;
size_t allocationSize = 1048576000;
```

```
zeVirtualMemQueryPageSize(hContext, hDevice, allocationSize, &pageSize);

size_t reserveSize = align(allocationSize, pageSize);

void* ptr = nullptr;
zeVirtualMemReserve(hContext, nullptr, reserveSize, &ptr);

...

// 1GB の割り当て領域の中から 128KB をサブ割り当て
size_t subAllocSize = 131072;
zeVirtualMemQueryPageSize(hContext, hDevice, subAllocSize, &pageSize);

// 128KB のサブ割り当て用の物理メモリー・オブジェクトを作成
size_t subAllocAlignedSize = align(subAllocSize, pageSize);
ze_physical_mem_desc_t pmemDesc = {
    ZE_STRUCTURE_TYPE_PHYSICAL_MEM_DESC,
    nullptr,
    0, // フラグ
    subAllocAlignedSize // サイズ
};
ze_physical_mem_handle_t hPhysicalAlloc;
zePhysicalMemCreate(hContext, hDevice, &pmemDesc, &hPhysicalAlloc);

// ページ・アライメントに一致する適切な 128KB のサブ割り当て領域を検索
...

zeVirtualMemMap(hContext, subAllocPtr, subAllocAlignedSize, hPhysicalAlloc, 0,
    ZE_MEMORY_ACCESS_ATTRIBUTE_READWRITE);

...

// 1GB の割り当て領域の中から 8MB をサブ割り当て
size_t subAllocDiffSize = 8388608;
zeVirtualMemQueryPageSize(hContext, hDevice, subAllocDiffSize, &pageSize);
...
```

イメージ

イメージは、多次元かつフォーマットが定義されたメモリーを保存するために使用されます。イメージの内容は、デバイスへのアクセスを最適化するため、実装固有のエンコーディングとレイアウトでメモリーに保存される場合があります（例：タイル状のスウィズルパターン、ロスレス圧縮など）。イメージの内容へのホストからの直接アクセスは

サポートされていません。しかし、イメージがホストからアクセス可能なメモリー領域にコピーされる場合、その内容は実装に依存しないよう暗黙的にデコードされます。

```
// 単一コンポーネントの FLOAT32 形式を指定
ze_image_format_t format = {
    ZE_IMAGE_FORMAT_LAYOUT_32, ZE_IMAGE_FORMAT_TYPE_FLOAT,
    ZE_IMAGE_FORMAT_SWIZZLE_R, ZE_IMAGE_FORMAT_SWIZZLE_0, ZE_IMAGE_FORMAT_SWIZZLE_0,
ZE_IMAGE_FORMAT_SWIZZLE_1
};

ze_image_desc_t imageDesc = {
    ZE_STRUCTURE_TYPE_IMAGE_DESC,
    nullptr,
    0, // 読み取り専用
    ZE_IMAGE_TYPE_2D,
    format,
    128, 128, 0, 0, 0
};
ze_image_handle_t hImage;
zeImageCreate(hContext, hDevice, &imageDesc, &hImage);

// ホストポインターからコンテンツをアップロード
zeCommandListAppendImageCopyFromMemory(hCommandList, hImage, nullptr, pImageData, nullptr,
0, nullptr);
...
```

フォーマット記述子は、フォーマット・レイアウト、タイプ、およびスウィズルを組み合わせたものです。このフォーマット・レイアウトには、コンポーネントの数とそれに対応するビット幅を記述します。このタイプは、以下に説明するいくつかの例外を除き、これらのすべてのコンポーネントのデータタイプを記述しています。スウィズルは、イメージ・コンポーネントがカーネルの XYZW/RGBA チャンネルにどのようにマッピングされるか示します。コンポーネントをチャンネル内に複製することは許されます。

以下の表は、各レイアウトに必要なタイプを示しています。

フォーマット・レイアウト	UINT	SINT	UNORM	SNORM	FLOAT
8	必須	必須	必須	必須	未サポート
8_8	必須	必須	必須	必須	未サポート
8_8_8_8	必須	必須	必須	必須	未サポート
16	必須	必須	必須	必須	必須
16_16	必須	必須	必須	必須	必須
16_16_16_16	必須	必須	必須	必須	必須

フォーマット・レイアウト	UINT	SINT	UNORM	SNORM	FLOAT
32	必須	必須	必須	必須	必須
32_32	必須	必須	必須	必須	必須
32_32_32_32	必須	必須	必須	必須	必須
10_10_10_2	必須	必須	必須	必須	必須
11_11_10	未サポート	未サポート	未サポート	未サポート	必須
5_6_5	未サポート	未サポート	必須	未サポート	未サポート
5_5_5_1	未サポート	未サポート	必須	未サポート	未サポート
4_4_4_4	未サポート	未サポート	必須	未サポート	未サポート

デバイスキャッシュ設定

デバイスとカーネルのキャッシュを制御する方法は 2 つあります:

1. キャッシュサイズ設定: カーネル・インスタンスごとに、SLM とデータに対してより大きなサイズを設定できる機能。
2. アプリケーションがデバイスキャッシュへのアクセスを許可する否かのヒント/設定。GPU デバイスの場合、これは 2 つの方法で提供されます:
    - フラグによるイメージの作成中
    - カーネル命令

以下の擬似コードは、キャッシュサイズ設定の基本的な手順を示しています:

```
// より大きな SLM をサポートするようにキャッシュを構成
// 注: キャッシュ設定は各カーネルに適用されます。
zeKernelSetCacheConfig(hKernel, ZE_CACHE_CONFIG_FLAG_LARGE_SLM);
```

外部メモリーのインポートとエクスポート

外部メモリーハンドルは、他の API からインポートしたり、他の API で使用するためエクスポートしたりできます。外部メモリーのインポートおよびエクスポートはオプション機能です。デバイスは、`zeDeviceGetExternalMemoryProperties` を使用して、サポートする外部メモリーハンドルのタイプを記述できます。

外部メモリーのインポートおよびエクスポートは、デバイスメモリーおよびホストメモリーの割り当て、ならびにイメージに対してサポートされます。

以下の擬似コードは、デバイスメモリー割り当て用の外部メモリーハンドルを Linux* の `dma_buf` として割り当ててエクスポートする方法を示しています:

```
// エクスポート可能な割り当てのリクエストを設定
ze_external_memory_export_desc_t export_desc = {
    ZE_STRUCTURE_TYPE_EXTERNAL_MEMORY_EXPORT_DESC,
    nullptr, // pNext
```

```
    ZE_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF
};

// 要求を割り当て記述子にリンクして割り当て
alloc_desc.pNext = &export_desc;
zeMemAllocDevice(hContext, &alloc_desc, size, alignment, hDevice, &ptr);

...

// 外部メモリーハンドルのエクスポート要求を設定
ze_external_memory_export_fd_t export_fd = {
    ZE_STRUCTURE_TYPE_EXTERNAL_MEMORY_EXPORT_FD,
    nullptr, // pNext
    ZE_EXTERNAL_MEMORY_TYPE_FLAG_OPAQUE_FD,
    0 // [out] fd
};

// エクスポート要求をクエリーにリンク
alloc_props.pNext = &export_fd;
zeMemGetAllocProperties(hContext, ptr, &alloc_props, nullptr);
```

以下の擬似コードは、Linux の `dma_buf` をデバイスメモリー割り当て用の外部メモリーハンドルとしてインポートする方法を示しています:

```
// 外部メモリーハンドルのインポート要求を設定
ze_external_memory_import_fd_t import_fd = {
    ZE_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMPORT_FD,
    nullptr, // pNext
    ZE_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF,
    fd
};

// 要求を割り当て記述子にリンクして割り当て
alloc_desc.pNext = &import_fd;
zeMemAllocDevice(hContext, &alloc_desc, size, alignment, hDevice, &ptr);
```

もう一つの例として、以下の擬似コードは、Linux の `dma_buf` をイメージ用の外部メモリーハンドルとしてインポートする方法を示しています:

```
// 外部メモリーハンドルのインポート要求を設定
ze_external_memory_import_fd_t import_fd = {
    ZE_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMPORT_FD,
    nullptr, // pNext
```

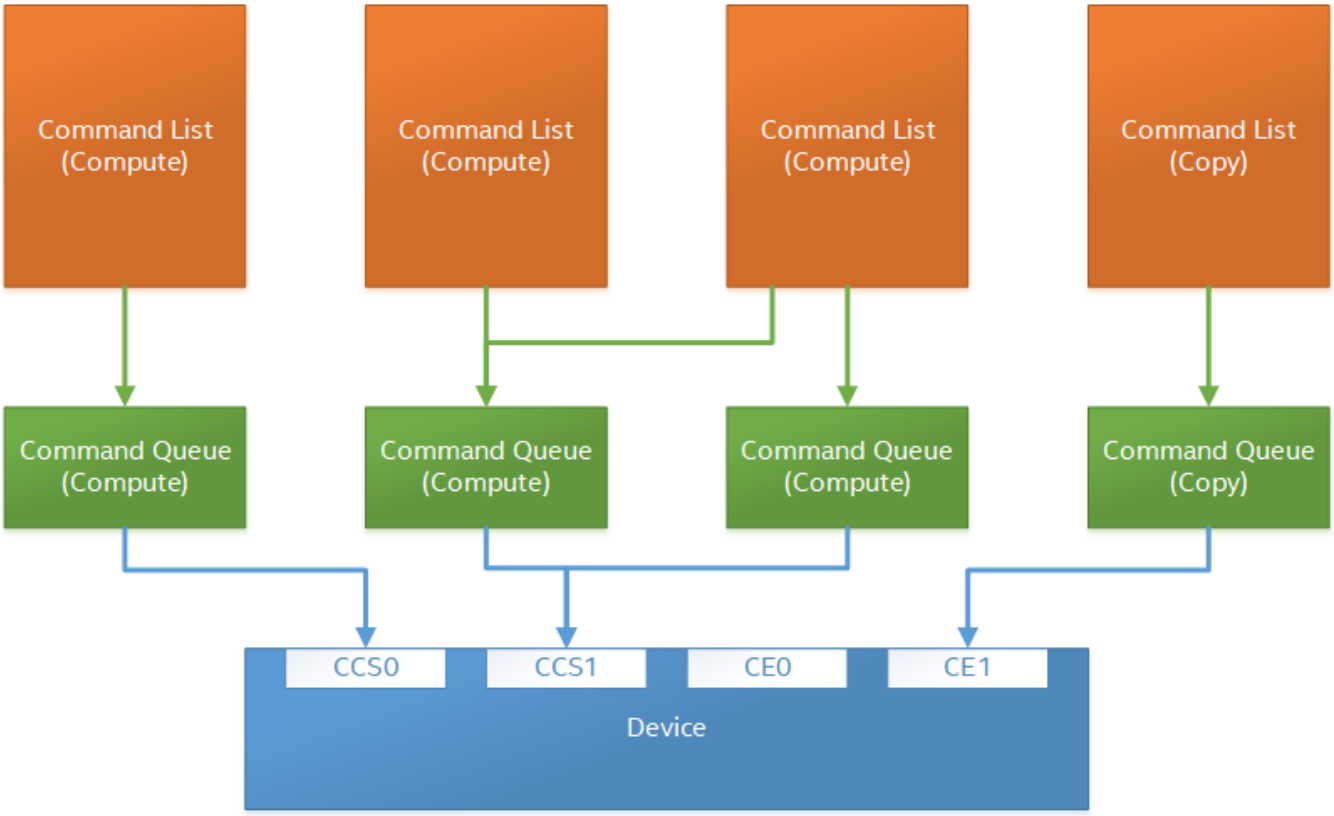
```
    ZE_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF,
    fd
};

// 要求を割り当て記述子にリンクして割り当て
image_desc.pNext = &import_fd; // extend ze_image_desc_t

// インポートしたイメージに合わせてイメージ・プロパティーを設定
image_desc.width = import_width;
...

zeImageCreate(hContext, hDevice, &image_desc, &hImage);
```

コマンドキューとコマンドリスト

コマンドキューとコマンドリストを分離する理由は以下のとおりです:

- コマンドキューは、入力ストリーム数など、主に物理デバイスの特性に関連付けられています。
- コマンドキューは、デバイスへの（ほぼ）ゼロ・レイテンシーのアクセスを提供します。
- コマンドリストは、主に同時構築のためホストスレッドに関連付けられています。
- コマンドリストは、コマンドキューへの送信とは独立して構築できます。

以下の図は、デバイスへのコマンドリストとコマンドキューの階層構造を示しています:



コマンド・キュー・グループ

コマンド・キュー・グループは、物理的な入力ストリームを表し、これは 1 つ以上の物理デバイスエンジンを表します。

検出

- コマンド・キュー・グループの数と特性は、`zeDeviceGetCommandQueueGroupProperties` 関数を使用して照会できます。
- グループ内の物理エンジンの数は、`ze_command_queue_group_properties_t.numQueues` から取得されます。
- グループがサポートするコマンドのタイプは、`ze_command_queue_group_properties_t.flags` から取得されます。
- 例えば、コマンドリストをコピー専用エンジンに送信する場合、`ZE_COMMAND_QUEUE_GROUP_PROPERTY_FLAG_COPY` が設定され、`ZE_COMMAND_QUEUE_GROUP_PROPERTY_FLAG_COMPUTE` が設定されていないコマンド・キュー・グループの順番でコマンドリストを作成し、同じ順番で作成されたコマンドキューに送信する必要があります。

以下の擬似コードは、コマンド・キュー・グループを検出する基本的な手順を示しています:

```
// すべてのコマンド・キュー・グループを検出
uint32_t cmdqueueGroupCount = 0;
```

```
zeDeviceGetCommandQueueGroupProperties(hDevice, &cmdqueueGroupCount, nullptr);

ze_command_queue_group_properties_t* cmdqueueGroupProperties =
(ze_command_queue_group_properties_t*)
    allocate(cmdqueueGroupCount * sizeof(ze_command_queue_group_properties_t));
cmdqueueGroupProperties[ i ].stype = ZE_STRUCTURE_TYPE_COMMAND_QUEUE_GROUP_PROPERTIES;
cmdqueueGroupProperties[ i ].pNext = nullptr;
zeDeviceGetCommandQueueGroupProperties(hDevice, &cmdqueueGroupCount,
cmdqueueGroupProperties);

// 計算をサポートするコマンド・キュー・タイプを検出
uint32_t computeQueueGroupOrdinal = cmdqueueGroupCount;
for( uint32_t i = 0; i < cmdqueueGroupCount; ++i ) {
    if( cmdqueueGroupProperties[ i ].flags & ZE_COMMAND_QUEUE_GROUP_PROPERTY_FLAG_COMPUTE )
{
        computeQueueGroupOrdinal = i;
        break;
    }
}

if(computeQueueGroupOrdinal == cmdqueueGroupCount)
    return; // 計算キューが見つかりません
```

コマンドキュー

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。

作成

- コマンドキューは、作成時にその順序でコマンド・キュー・グループに明示的にバインドされます。
- 同じコマンド・キュー・グループを使用する複数のコマンドキューを作成できます。例えば、アプリケーションは、ホストスレッドごとに異なるスケジューラー優先順位を持つコマンドキューを作成することができます。
- 同じコンテキスト上の同じコマンド・キュー・グループに対して作成された複数のコマンドキューは、同じ物理ハードウェア・コンテキストを共有する場合があります。
- アプリケーションが作成できるコマンドキューの最大数は、デバイス固有のリソースによって制限されます。例えば、デバイスがサポートする論理ハードウェア・コンテキストの最大数などがありますす。これは、`ze_device_properties_t.maxHardwareContexts` から取得できます。
- コマンド・キュー・グループ内のコマンドキューが実行される物理エンジンは、そのインデックスを介して仮想化されており、その数はコマンド・キュー・グループのタイプに応じた物理エンジンの数、つまり

ze_command_queue_group_properties_t.numQueues によって制限されます。

- コマンド・キュー・インデックスは、アプリケーションがどのコマンドキューを同時に実行できるか（異なるインデックス）を示すメカニズムを提供します。
- 同じインデックスを共有しないコマンドキューは、同時に起動および実行できます。
- 同じインデックスを共有するコマンドキューは順番に起動されますが、同時に実行される場合もあります。
- コマンドキュー上で実行されるすべてのコマンドリストは、それが割り当てられているコマンド・キュー・グループ内のエンジンで**のみ**実行されることが保証されています。例えば、計算コマンドリスト/キュー内のコピーコマンドは、コピーエンジンではなく計算エンジンを介して実行されます。
- インデックスの異なるコマンドキューに送信されたコマンドリストが同時に実行される保証はなく、同時に実行される可能性がだけです。

以下の擬似コードは、コマンドキューを作成する基本的な手順を示しています:

```
// コマンドキューの作成
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_queue_handle_t hCommandQueue;
zeCommandQueueCreate(hContext, hDevice, &commandQueueDesc, &hCommandQueue);
...
```

実行

- コマンドキューに送信されたコマンドリストは、**直ちに**デバイスに送信され実行されます。
- 複数のコマンドリストをまとめて送信することで、実装ではコマンドリスト全体にわたって最適化を行う可能性があります。
- コマンドキューへの送信はフリースレッド方式で行われるため、複数のホストスレッドが同じコマンドキューを共有できます。
- 複数のホストスレッドが同時に同じコマンドキューにアクセスする場合、実行順序は不定となります。
- コマンドリストは、同一のコマンド・キュー・グループ番号を持つコマンドキュー上でのみ実行できます。

破棄

- アプリケーションは、デバイスが実行中でないことを確認してからコマンドキューから削除する必要があります。これは通常、コマンドキューのフェンスを追跡することで行われますが、

`zeCommandQueueSynchronize` を呼び出して処理することも可能です。

コマンドリスト

コマンドリストは、コマンドキュー上で実行される一連のコマンドを表します。

作成

- デバイス固有のコマンドを追加できるようにするため、デバイス用のコマンドリストが作成されます。
- コマンドリストは、コマンド・キュー・グループの順序で指定された、特定タイプのコマンドキュー上で実行するために作成されます。
- コマンドリストをコピーして、別のコマンドリストを作成することができます。アプリケーションは、これを利用してコマンドリストをコピーし、別のデバイスで使用できます。

追加

- コマンドリストとホストスレッドの間には、暗黙的な関連付けは存在しません。したがって、アプリケーションは、複数のホストスレッド間でコマンド・リスト・ハンドルを共有することができます。ただし、複数のホストスレッドが同じコマンドリストに同時にアクセスしないようにするのは、アプリケーションの責任です。
- デフォルトでは、コマンドは追加された順序で実行されます。ただし、アプリケーションによっては、`ZE_COMMAND_LIST_FLAG_RELAXED_ORDERING` フラグを使用することで、ドライバーが順序付けを最適化することも可能です。命令の並べ替えは、バリアと同期プリミティブ間でのみ発生することが保証されています。
- デフォルトでは、コマンドリストに送信されたコマンドは、デバイスのスループットとホストのレイテンシーの両方のバランスを取ることで、実行が最適化されるように処理されます。
- 非常に低レイテンシーが求められる使用モデルの場合、アプリケーションは即時コマンドリストを使用する必要があります。
- 最大スループットが求められる使用モデルでは、アプリケーションは `ZE_COMMAND_LIST_FLAG_MAXIMIZE_THROUGHPUT` を使用する必要があります。このフラグは、ドライバーに対して、デバイス固有の最適化を実行できる可能性があることを示します。

以下の擬似コードは、コマンドリストを作成する基本的な手順を示しています:

```
// コマンドリストの作成
ze_command_list_desc_t commandListDesc = {
    ZE_STRUCTURE_TYPE_COMMAND_LIST_DESC,
    nullptr,
    computeQueueGroupOrdinal,
    0 // フラグ
};
ze_command_list_handle_t hCommandList;
zeCommandListCreate(hContext, hDevice, &commandListDesc, &hCommandList);
```

```
...
```

送信

- コマンドリストとコマンドキューの間には暗黙的な関連性はありません。したがって、コマンドリストは、いずれか 1 つまたは複数のコマンドキューに送信できます。
- しかし、定義上、コマンドリストは複数のコマンドキューで同時に実行することはできません。
- アプリケーションは、コマンドキューに送信する前に `close` 関数を呼び出す必要があります。
- コマンドリストは、同じコマンドキューで実行される他のコマンドリストからの状態を継承しません。つまり、各コマンドリストはそれぞれ独自のデフォルト状態から実行を開始します。
- コマンドリストは複数回送信することができます。コマンドリストを複数回実行できることを保証するのは、アプリケーションの責任です。例えば、イベントは再実行する前に明示的にリセットする必要があります。

以下の擬似コードは、コマンドリストを介してコマンドキューにコマンドを送信する手順を示しています:

```
...
// コマンドの追加が完了（通常は別のスレッドで行われます）
zeCommandListClose(hCommandList);

// コマンドキュー内のコマンドリストを実行
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);

// ホストとデバイスを同期
zeCommandQueueSynchronize(hCommandQueue, UINT32_MAX);

// 新しいコマンドのためにコマンドリストをリセット（再利用）
zeCommandListReset(hCommandList);
...
```

再利用

- コマンドリストは、頻繁な生成と破棄のオーバーヘッドを避けるため再利用できます。
- アプリケーションは、デバイスがリセットされる前に、現在のコマンドリストから実行されていないことを確認する必要があります。これは、コマンドリストに関連付けられた完了イベントを追跡することで処理されます。
- アプリケーションは、デバイスが削除される前に、現在のコマンドリストから実行されていないことを確認する必要があります。これは、コマンドリストに関連付けられた完了イベントを追跡することで処理されます。

低レイテンシー即時コマンドリスト

非常に低レイテンシーな送信を必要とする使用モデルには、特殊タイプのコマンドリストを使用できます。

- 即時コマンドリストは、コマンドリストであると同時に暗黙的なコマンドキューでもあります。
- コマンドキュー記述子を使用して、即時実行コマンドリストが作成されます。
- 即時コマンドリストに追加されたコマンドは、デバイス上で直ちに実行されます。
- 即時コマンドリストに追加されたコマンドは、コマンドが完了するまで処理をブロックすることで、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、クローズやリセットする必要はありません。ただし、利用規約は遵守され、期待される振舞いは順守されるものとします。

以下の擬似コードは、即時コマンドリストの作成と使用に関する基本的な手順を示しています:

```
// コマンドリストを即座に作成
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);

// カーネルを直ちにデバイスに送信
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);
...
```

追加パラメーターを持つカーネルの追加

- カーネルを追加するときに追加のパラメーターを渡す新しい関数が追加されました。
- 新しいパラメーターは拡張機能から渡すことができ、ベンダー固有のものになります。
- この関数は、専用の記述子を使用して協調カーネルを渡すことを可能にします。
- 複数の追加パラメーターを記述子のリンクリストとして渡すことができます。
- 追加のパラメーターまたはその組み合わせがサポートされていない場合、ドライバーはエラーを返すことがあります。

次の擬似コードは、通常のカーネルと協調カーネルの両方を追加する方法を示しています:

```
// 既存のコマンドリスト
ze_command_list_handle_t hCommandList;

// 通常のカーネルを追加する場合は、拡張引数に null ポインターを渡すだけです
void *pNext = nullptr;
zeCommandListAppendLaunchKernelWithParameters(hCommandList, hKernel, &launchArgs, pNext,
nullptr, 0, nullptr);

// 協調カーネルを追加するときに協調記述子を作成
ze_command_list_append_launch_kernel_param_cooperative_desc_t cooperativeDesc = {
```

```
    ZE_STRUCTURE_TYPE_COMMAND_LIST_APPEND_PARAM_COOPERATIVE_DESC,
    nullptr,
    true
};
void *pNext = &cooperativeDesc;
zeCommandListAppendLaunchKernelWithParameters(hCommandList, hKernel, &launchArgs, pNext,
nullptr, 0, nullptr);
...
```

引数付きカーネルの追加

- カーネルを追加するときにグループサイズ、引数、追加の拡張子を渡す新しい関数が追加されました。
- カーネル・オブジェクトの状態は、別の zeKernelSetGroupSize 関数と zeKernelSetArgumentValue 関数が呼び出されたかのように、新しいワーク・グループ・サイズと新しい引数で更新されます。
- カーネル引数はポインターリストとして渡され、各引数は特定のインデックスの引数値へのポインターを表します。
- すべてのカーネル引数を指定する必要があります。
- 引数が SLM（サイズ）の場合、このリソースの SLM サイズ（バイト単位）が特定のインデックスのポインターの下に提供され、そのタイプは size_t です。
- 引数が即値タイプ（つまり構造体、非ポインター型）の場合、ポインターの下の値には即値タイプの完全なサイズが含まれている必要があります。
- 追加の拡張機能は拡張機能から渡すことができ、ベンダー固有のものになります。
- 複数の追加拡張機能を記述子のリンクリストとして渡すことができます。
- 追加の拡張機能またはその組み合わせがサポートされていない場合、ドライバーはエラーを返す必要があります。

次の擬似コードは、ポインター、SLM、および即値タイプの引数を持つカーネルの追加を示しています:

```
// カーネル・シグネチャー
__kernel void foo(__global unsigned int *dstBuff, __local unsigned int *localArray,
unsigned int addValue);

// 既存のコマンドリスト
ze_command_list_handle_t hCommandList;

// 既存のカーネル
ze_command_list_handle_t hKernel;

// 出力バッファー
void *dstBuff;

// 配列の SLM サイズ
size_t localArraySizeInBytes;
```

```
// 即時引数
unsigned int addValue;

void *args[] = { &dstBuff, &localArraySizeInBytes, &addValue};
ze_group_count_t groupCounts = {1,2,3};
ze_group_size_t groupSizes = {1,2,3};
zeCommandListAppendLaunchKernelWithArguments(hCommandList, hKernel, groupCounts, groupSizes,
args, nullptr, nullptr, 0, nullptr);

...
```

同期プリミティブ

次の 2 つの同期プリミティブがあります:

1. フェンス - コマンドキューの実行が完了したことをホストに伝えるために使用されます。
2. イベント - きめ細かいホスト間、ホストとデバイス間、またはデバイス間の実行およびメモリー依存関係として使用されます。

以下は、様々なタイプの同期プリミティブを分離する理由です:

- 特定のタイプのプリミティブに対して、デバイス固有の最適化を可能にします:
  - フェンスは、同じコマンドキュー内の他のすべてのフェンスとデバイスメモリーを共有する場合があります。
  - イベントは、プログラム実行の一環として、パイプライン処理を用いて実装される場合があります。
  - フェンスは、暗黙的かつ粗粒度な実行およびメモリーバリアです。
  - イベントによっては、きめ細かな実行バリアおよびメモリーバリアが発生する場合があります。
- どのタイプのプリミティブをデバイス間で共有できるか区別できるようにします。

一般的にイベントは、フェンスを含む様々な使用モデルで使用できる汎用同期プリミティブです。しかし、このような汎用性には、メモリー使用量の増加や効率の低下といった代償が伴います。

フェンス

フェンスは、コマンドキューの実行が完了したことをホストに伝えるために使用されるコストの高い同期プリミティブです。

- フェンスは、単一のコマンドキューに関連付けられます。
- フェンスは、デバイスのコマンドキューからのみシグナルを送信でき（例えば、コマンドリストの実行間など）、ホスト側でのみ待機できます。
- フェンスは、シグナルが発せられる前に、デバイスとホスト間で、実行の完了とメモリーの一貫性の両方を保証します。
- フェンスには、シグナルが発せられていない状態と、発せられている状態の 2 つしかありません。

- フェンスは自動的にリセットされるわけではありません。既にシグナルが送られているフェンスにシグナルを送る（あるいはシグナルが送られていないフェンスをリセットする）ことは有効であり、フェンスの状態には影響しません。
- フェンスはホスト側からのみリセットできます。
- また、フェンスはプロセス間で共有することはできません。

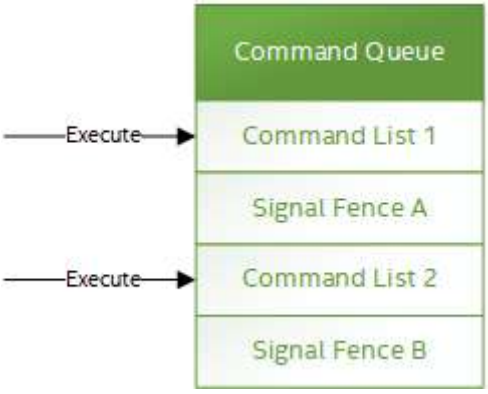
以下の擬似コードは、フェンスの作成、送信、およびクエリー実行の一連の手順を示しています:

```
// フェンスの作成
ze_fence_desc_t fenceDesc = {
    ZE_STRUCTURE_TYPE_FENCE_DESC,
    nullptr,
    0 // フラグ
};
ze_fence_handle_t hFence;
zeFenceCreate(hCommandQueue, &fenceDesc, &hFence);

// フェンスのシグナルを使用してコマンドリストを実行
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, hFence);

// フェンスがシグナルされるまで待機
zeFenceHostSynchronize(hFence, UINT32_MAX);
zeFenceReset(hFence);
...
```

フェンスの主な使用モデルは、コマンドリストの実行が完了したときにホストに通知することであり、それによって以下のことが可能になります:

- メモリーとイメージの再利用
- コマンドリストの再利用
- その他の同期プリミティブの再利用
- 明示的なメモリーの常駐

以下の図は、コマンドリスト実行後に通知されるフェンスを示しています:



イベント

イベントは、ホストとデバイス間、デバイスとホスト間、またはデバイスとデバイス間のきめ細かい依存関係が完了したことを通知するために使用されます。

- イベントは以下ができます:
    - デバイスのコマンドリスト内でシグナルが発信され、同じコマンドリスト内でそのシグナルが待機されます
    - デバイスのコマンドリスト内からシグナルが送られ、ホスト、別のコマンドキュー、または別のデバイスで待機されます
    - ホスト側からシグナルが送られ、デバイスのコマンドリスト内で待機状態となります。
- イベントには、シグナルが通知されていない状態と、通知済みの 2 つの状態しかありません。
- イベントは暗黙的にリセットされません。既に通知済みのイベントに通知信号を送る（または通知されていないイベントをリセットする）ことは有効であり、イベントの状態には影響しません。
- イベントは、ホストまたはデバイスから明示的にリセットできます。
- 1 つのイベントを複数のコマンドリストに同時に追加することもできます。
- イベントは、複数のデバイスやプロセス間で共有できます。
- イベントは、実行バリアやメモリーバリアを発生させる可能性がありますが、デバイスの利用率低下を避けるため、これらは控える必要があります。
- 循環待機シナリオなど、デッドロックを引き起こす事象に対する保護策は存在しません。
    - これらの問題は、アプリケーションで回避する必要があります。
- コマンドキューへのコマンドリストの送信後、ホスト、別のコマンドキュー、または別のデバイスによって通知されることを意図したイベントは、コマンドキュー内でのその後の処理の進行を妨げる可能性があります。
    - 適切にスケジュールされない場合、パイプライン内でバブルが発生したり、デッドロック状態に陥る場合があります。

イベントプールは、個々のイベントを作成するために使用されます:

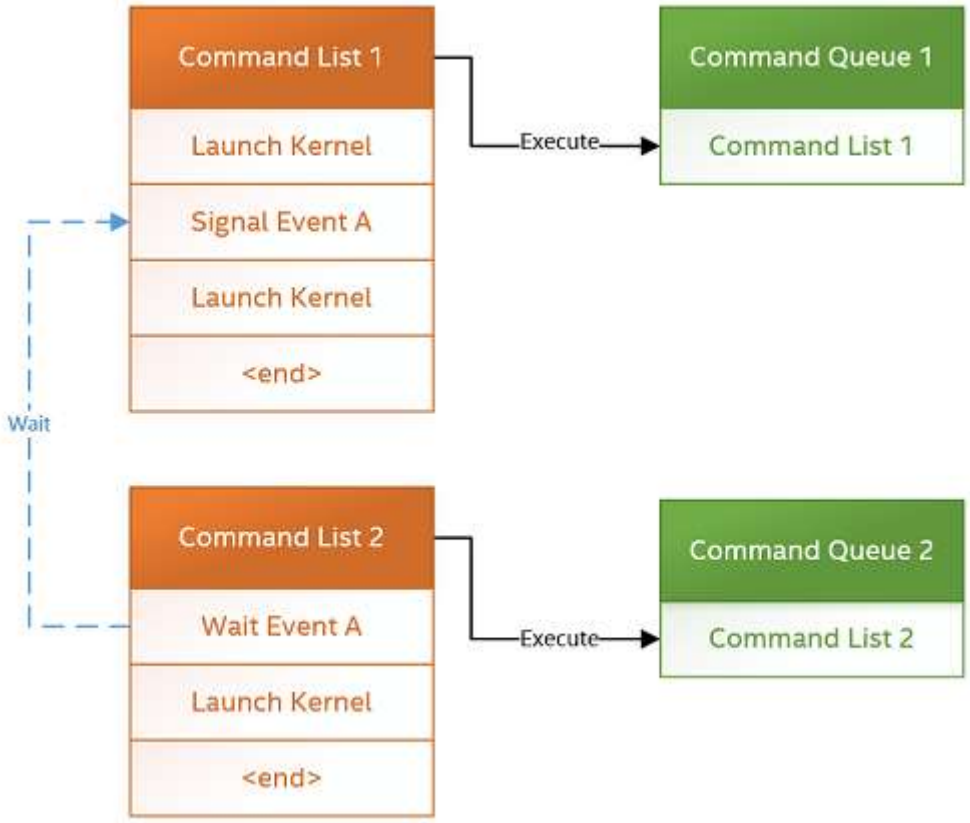
- イベントプールを使用して、複数のイベントを作成する際のコストを削減できます。これは、同じプロパティーを持つイベント間で基盤となるデバイス割り当てを共有できるためです
- イベントプールはプロセス間通信によって共有でき、個々のイベントを共有するのではなく、イベントのブロック単位で共有することが可能になります

以下の擬似コードは、イベントの作成と送信の手順を示しています:

```
// イベントプールの作成
ze_event_pool_desc_t eventPoolDesc = {
    ZE_STRUCTURE_TYPE_EVENT_POOL_DESC,
    nullptr,
    ZE_EVENT_POOL_FLAG_HOST_VISIBLE, // プール内のすべてのイベントはホストから閲覧可能です
    1 // カウント
```

```
};
ze_event_pool_handle_t hEventPool;
zeEventPoolCreate(hContext, &eventPoolDesc, 0, nullptr, &hEventPool);

ze_event_desc_t eventDesc = {
    ZE_STRUCTURE_TYPE_EVENT_DESC,
    nullptr,
    0, // インデックス
    0, // シグナルに追加のメモリー/キャッシュの一貫性維持は不要
    ZE_EVENT_SCOPE_FLAG_HOST  // イベント完了後、デバイスとホスト間でメモリーの一貫性を確保
};
ze_event_handle_t hEvent;
zeEventCreate(hEventPool, &eventDesc, &hEvent);

// カーネル実行後、イベントのシグナルをコマンドリストに追加
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, hEvent, 0, nullptr);

// シグナルでコマンドリストを実行
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);

// イベントが完了するまで待機
zeEventHostSynchronize(hEvent, 0);
...
```

以下の図は、イベントを使用したコマンドリスト間の依存関係を示しています:



カーネル・タイムスタンプ・イベント

カーネル・タイムスタンプ・イベントは、カーネルの実行開始時と終了時にデバイスのタイムスタンプを記録する特殊なタイプのイベントです。カーネル・タイムスタンプの主な目的は、実行時間の長さを示すことです。一貫性と直交性を確保するため、カーネル以外の操作においてもカーネル・タイムスタンプがサポートされています。カーネル・タイムスタンプはデバイスのタイムラインに沿って実行されますが、範囲が限られているため、予期せずオーバーフローする可能性があります。そのため、START/END の値が一致している場合でも、2 つのカーネル・タイムスタンプの時間的な順序を想定すべきではありません。`zeCommandListAppendWriteGlobalTimestamp` は同様のメカニズムを提供しますが、より広い範囲に対応しています。`zeCommandListAppendWriteGlobalTimestamp` によって記録されたタイムスタンプとカーネル・タイムスタンプのイベントは、たとえ同じ範囲内で報告されたとしても、同等であると考えるべきではありません。

- `zeCommandListAppendSignalEvent` および `zeEventHostSignal` におけるカーネル・タイムスタンプの期間は未定義です。ただし、一貫性と直交性を確保するため、このイベントは他のイベント API 機能で使用される場合、シグナル順に正しく報告されます。
- カーネル・タイムスタンプ・イベントの結果は、`zeEventQueryKernelTimestamp` または `zeCommandListAppendQueryKernelTimestamps` を使用して照会できます。
- `ze_kernel_timestamp_result_t` 構造体には、カーネル実行の開始時と終了時におけるコンテキストごとのタイムスタンプ値とグローバル・タイムスタンプ値の両方が含まれています。
- これらのカウンターは 32 ビットしかないため、アプリケーションは実行時間を計算する際に、カウンターのオーバーフローを検出して処理する必要があります

```
// タイムスタンプの周波数を取得
const double timestampFreq = NS_IN_SEC / device_properties.timerResolution;
const uint64_t timestampMaxValue = ~(-1L << device_properties.kernelTimestampValidBits);

// イベントプールの作成
ze_event_pool_desc_t tsEventPoolDesc = {
    ZE_STRUCTURE_TYPE_EVENT_POOL_DESC,
    nullptr,
    ZE_EVENT_POOL_FLAG_KERNEL_TIMESTAMP, // all events in pool are kernel timestamps
    1 // カウント
};
ze_event_pool_handle_t hTSEventPool;
zeEventPoolCreate(hContext, &tsEventPoolDesc, 0, nullptr, &hTSEventPool);

ze_event_desc_t tsEventDesc = {
    ZE_STRUCTURE_TYPE_EVENT_DESC,
    nullptr,
    0, // インデックス
    0, // シグナルに追加のメモリー/キャッシュの一貫性維持は不要
    0 // 待機に追加のメモリー/キャッシュの一貫性維持は不要
};
ze_event_handle_t hTSEvent;
zeEventCreate(hEventPool, &tsEventDesc, &hTSEvent);

// 結果用のメモリーを割り当て
ze_device_mem_alloc_desc_t tsResultDesc = {
    ZE_STRUCTURE_TYPE_DEVICE_MEM_ALLOC_DESC,
    nullptr,
    0, // フラグ
    0  // 順序
};
ze_kernel_timestamp_result_t* tsResult = nullptr;
zeMemAllocDevice(hContext, &tsResultDesc, sizeof(ze_kernel_timestamp_result_t),
sizeof(uint32_t), hDevice, &tsResult);

// カーネル実行後、タイムスタンプ・イベントのシグナルをコマンドリストに追加
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, hTSEvent, 0, nullptr);

// タイムスタンプ・イベントのクエリーをコマンドリストに追加
zeCommandListAppendQueryKernelTimestamps(hCommandList, 1, &hTSEvent, tsResult, nullptr,
hEvent, 1, &hTSEvent);
```

```
// シグナルでコマンドリストを実行
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);

// イベントが完了するまで待機
zeEventHostSynchronize(hEvent, 0);

// 実行時間（秒）を計算
double globalTimeInNs = ( tsResult->global.kernelEnd >= tsResult->global.kernelStart )
    ? ( tsResult->global.kernelEnd - tsResult->global.kernelStart ) * timestampFreq
    : (( timestampMaxValue - tsResult->global.kernelStart) + tsResult->global.kernelEnd +
1 ) * timestampFreq;

double contextTimeInNs = ( tsResult->context.kernelEnd >= tsResult->context.kernelStart )
    ? ( tsResult->context.kernelEnd - tsResult->context.kernelStart ) * timestampFreq
    : (( timestampMaxValue - tsResult->context.kernelStart) + tsResult->context.kernelEnd +
1 ) * timestampFreq;
...
```

バリア

次の 2 つバリアタイプがあります:

1. **実行バリア** - コマンドリスト内、あるいはコマンドキュー、デバイス、ホスト間におけるコマンド間の実行依存関係を伝達するために使用されます。
2. **メモリーバリア** - コマンドリスト内、またはコマンドキュー、デバイス、ホスト間におけるコマンド間でのメモリーの一貫性に関する依存関係を伝達するために使用されます。

以下の擬似コードは、ブルートフォース実行とグローバル・メモリー・バリアの送信手順を示しています:

```
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);

// コマンドリストにバリアを追加して、hKernel1 が完了してから hKernel2 が開始されるようにします
zeCommandListAppendBarrier(hCommandList, nullptr, 0, nullptr);

zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);
...
```

実行バリア

コマンドリスト上で実行されるコマンドは、投入された順序と同じ順序で開始されることが保証されているだけであり、完了順序に暗黙的な定義は存在しません。

- フェンスは、暗黙的かつ大まかな制御を提供し、フェンスがシグナルを発する前に、それ以前のすべてのコ

マンドが完了しなければならないことを示します。

- イベントを使用することで、コマンド間の実行依存関係をきめ細かく制御できるようになり、並列実行の可能性が増え、デバイスの利用率が向上します。

以下の擬似コードは、イベントを使用してきめ細かい実行専用の依存関係を送信する手順を示しています:

```
ze_event_desc_t event1Desc = {
    ZE_STRUCTURE_TYPE_EVENT_DESC,
    nullptr,
    0, // インデックス
    0, // シグナルに追加のメモリー/キャッシュの一貫性維持は不要
    0 // 待機に追加のメモリー/キャッシュの一貫性維持は不要
};
ze_event_handle_t hEvent1;
zeEventCreate(hEventPool, &event1Desc, &hEvent1);

// hEvent1 に通知する前に、hKernel1 が完了していることを確認
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, hEvent1, 0, nullptr);

// hKernel2 を開始する前に、hEvent1 が通知されていることを確認
zeCommandListAppendLaunchKernel(hCommandList, hKernel2, &launchArgs, nullptr, 1, &hEvent1);
...
```

メモリーバリア

コマンドリストで実行されるコマンドは、他のコマンドとのメモリーの一貫性は保証*されません*。つまり、暗黙的なメモリーまたはキャッシュの一貫性はありません。

- フェンスは、フェンスがシグナルされる前に、すべてのキャッシュとメモリーがデバイスとホスト全体で一貫性があることを示す、暗黙的で粗粒度の制御を提供します。
- イベントを使用することで、コマンド間のキャッシュとメモリーの一貫性の依存関係をきめ細かく制御できるようになり、並列実行の可能性が増え、デバイスの利用率が向上します。

以下の擬似コードは、イベントを使用してきめ細かいメモリーの依存関係を送信する手順を示しています:

```
ze_event_desc_t event1Desc = {
    ZE_STRUCTURE_TYPE_EVENT_DESC,
    nullptr,
    0, // インデックス
    ZE_EVENT_SCOPE_FLAG_DEVICE, // イベントが通知される前に、デバイス間でメモリの一貫性が確保されていることを確認
    0 // 待機に追加のメモリー/キャッシュの一貫性維持は不要
};
```

```
ze_event_handle_t hEvent1;
zeEventCreate(hEventPool, &event1Desc, &hEvent1);

// hEvent1 を通知する前に、hKernel1 によるメモリー書き込みがデバイス全体で完全に整合していることを確認
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, hEvent1, 0, nullptr);

// hKernel2 を開始する前に、hEvent1 が通知されていることを確認
zeCommandListAppendLaunchKernel(hCommandList, hKernel2, &launchArgs, nullptr, 1, &hEvent1);
...
```

範囲ベースのメモリーバリア

範囲ベースのメモリーバリアは、どのキャッシュラインに一貫性が必要かを明示的に制御します。

以下の擬似コードは、範囲ベースのメモリーバリアを送信する手順を示しています:

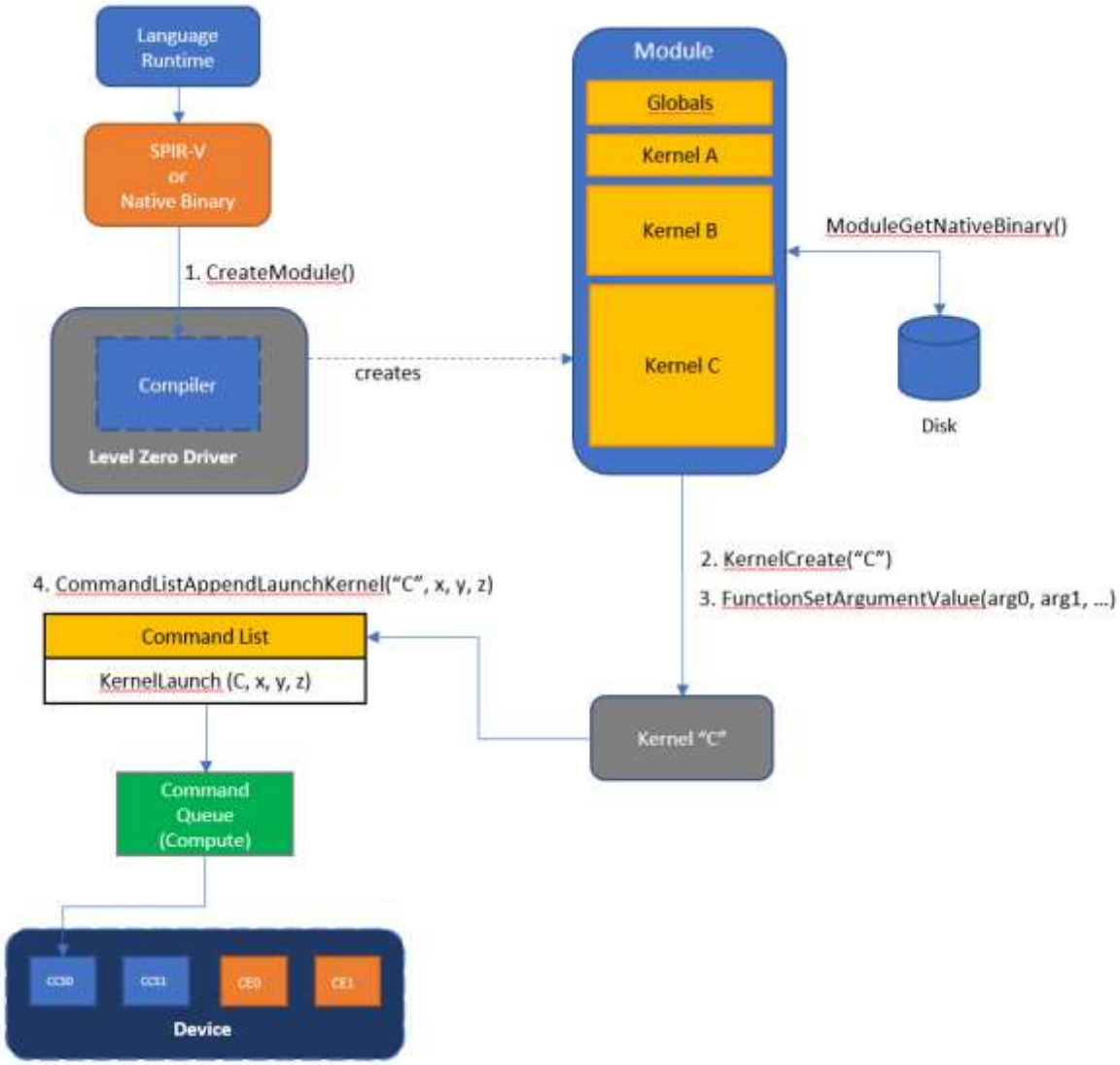
```
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, nullptr, 0, nullptr);

// hKernel1 の実行後、hKernel2 の実行前に、メモリー範囲全体がデバイス全体で完全に整合していることを確認
zeCommandListAppendMemoryRangesBarrier(hCommandList, 1, &size, &ptr, nullptr, 0, nullptr);

zeCommandListAppendLaunchKernel(hCommandList, hKernel2, &launchArgs, nullptr, 0, nullptr);
...
```

モジュールとカーネル

デバイス上でカーネルを実行するには、複数のレベルの構造が必要です:

1. モジュールは、一緒にコンパイルされたカーネルで構成される単一の翻訳単位を表します。
2. カーネルは、コマンドリストから直接起動されるモジュール内のカーネルを表します。

次の図は、システムの主要部分の概要を示しています。



モジュール

モジュールは、IL から作成することも、`zeModuleCreate` を使用してネイティブ形式から直接作成することもできます。

- `zeModuleCreate` は、入力形式を指定する `format` 引数を取ります。
- `zeModuleCreate` は、IL 形式のコンパイルステップを実行します。

以下の擬似コードは、OpenCL* カーネルからモジュールを作成する手順を示しています:

```
__kernel void image_scaling( __read_only  image2d_t src_img,
                         __write_only image2d_t dest_img,
                                      uint WIDTH,     // サイズ変更された幅
                                      uint HEIGHT )   // サイズ変更された高さ
{
   int2      coor = (int2)( get_global_id(0), get_global_id(1) );
   float2 normCoor = convert_float2(coor) / (float2)( WIDTH, HEIGHT );
   float4    color = read_imagef( src_img, SMPL_PREF, normCoor );
   write_imagef( dest_img, coor, color );
```

```
}
...
// OpenCL C カーネルは SPIRV IL (pImageScalingIL) にコンパイルされます
ze_module_desc_t moduleDesc = {
    ZE_STRUCTURE_TYPE_MODULE_DESC,
    nullptr,
    ZE_MODULE_FORMAT_IL_SPIRV,
    ilSize,
    pImageScalingIL,
    nullptr,
    nullptr
};
ze_module_handle_t hModule;
zeModuleCreate(hContext, hDevice, &moduleDesc, &hModule, nullptr);
...
```

モジュールのビルドオプション

モジュールのビルドオプションは、ze_module_desc_t を使用して文字列として渡すことができます。

ビルドオプション	説明	デフォルト	デバイスサポート
-ze-opt-disable	最適化を無効にします。	無効	すべて
-ze-opt-level	コンパイラーの最適化レベルを指定します。レベルは実装によって異なります。 0 は最適化なし (ze-opt-disable と同等) 1 は最小限に最適化します (2 と同じになる場合があります) 2 はより最適化します (デフォルト)	2	すべて
-ze-opt-greater-than-4GB-buffer-required	バッファーには 64 ビットのオフセット計算を使用します。	無効	GPU
-ze-opt-large-register-file	スレッドで使用できるレジスター数を増やします。	無効	GPU
-ze-opt-has-buffer-offset-arg	追加のオフセット (例: 32 ビットのオフセットを持つバインドテーブルへの 64 ビットポインター) を使用して、ステートレスからステートフルへの最適化をより多くのケースに拡張します。	無効	GPU
-g	デバッグ情報を含めます。	無効	GPU

モジュールの特殊化定数

SPIR-V は、実行時に特定の定数を新しい値に更新できる特殊化定数をサポートしています。SPIR-V の各特殊化定数には、識別子とデフォルト値があります。zeModuleCreate 関数を使用すると、定数の配列とそれに対応する識別子を渡して、SPIR-V モジュール内の定数をオーバーライドできます。

```
// グループサイズの仕様定数オーバーライド
ze_module_constants_t specConstants = {
    3,
    pGroupSizeIds,
    pGroupSizeValues
};
// OpenCL C カーネルは SPIRV IL (pImageScalingIL) にコンパイルされます
ze_module_desc_t moduleDesc = {
    ZE_STRUCTURE_TYPE_MODULE_DESC,
    nullptr,
    ZE_MODULE_FORMAT_IL_SPIRV,
    ilSize,
    pImageScalingIL,
    nullptr,
    &specConstants
};
ze_module_handle_t hModule;
zeModuleCreate(hContext, hDevice, &moduleDesc, &hModule, nullptr);
...
```

注: 特殊化定数はモジュールの作成時にのみ処理されるため、変更する場合は新しいモジュールをコンパイルする必要があります。

モジュールのビルドログ

zeModuleCreate 関数は、オプションでビルド・ログ・オブジェクト ze_module_build_log_handle_t を生成できます。

```
...
ze_module_build_log_handle_t buildlog;
ze_result_t result = zeModuleCreate(hContext, hDevice, &desc, &module, &buildlog);

// モジュール作成エラーのビルドログのみを保存
if (result != ZE_RESULT_SUCCESS)
{
    size_t szLog = 0;
    zeModuleBuildLogGetString(buildlog, &szLog, nullptr);
```

```
    char_t* strLog = allocate(szLog);
    zeModuleBuildLogGetString(buildlog, &szLog, strLog);

    // ログをディスクに保存
    ...

    free(strLog);
}

zeModuleBuildLogDestroy(buildlog);
```

動的リンクモジュール

モジュールは相互に依存する場合があります。つまり、モジュールには、異なるモジュールによって使用および定義される関数とグローバル変数が含まれる場合があります。このようなモジュールには、インポートとエクスポートの両方のリンク要件があると言われます。プライベート変数はリンクされたモジュール間で転送できません。つまり、その可視性は定義されているモジュール内に制限されます。リンクされたモジュール間では、リンクされた関数に渡されるグローバル変数または静的値のみが表示されます。モジュールからカーネルを作成するには、そのモジュールのすべてのインポートリンク要件を満たす必要があります。インポートのないモジュールはリンクする必要はありません。モジュールを動的にリンクするには、`zeModuleDynamicLink` を使用します。モジュールはあいまいなインポート依存関係を持つことはできません。つまり、インポートされた関数とグローバル変数は、`zeModuleDynamicLink` に渡される特定のモジュールセット内で 1 度だけ定義する必要があります。インポートは 1 度だけリンクされます。モジュールのすべてのインポート依存関係がリンクされると、完全にインポートがリンクされたモジュールを `zeModuleDynamicLink` への後続の呼び出しで使用しても、モジュールのインポートは再リンクされません。

`zeModuleDynamicLink` 関数は、オプションでリンク・ログ・オブジェクト `ze_module_build_log_handle_t` を生成できます。

```
...
ze_module_build_log_handle_t linklog;
ze_result_t result = zeModuleDynamicLink(numModules, &hModules, &hLinklog);

// リンクエラーがあるか確認
if (result == ZE_RESULT_ERROR_MODULE_LINK_FAILURE) {
  size_t szLog = 0;
  zeModuleBuildLogGetString(linklog, &szLog, nullptr);

  char_t* strLog = allocate(szLog);
  zeModuleBuildLogGetString(linklog, &szLog, strLog);

  // ログをディスクに保存
  ...
```

```
  free(strLog);
}

zeModuleBuildLogDestroy(linklog);
```

ネイティブバイナリーによるモジュールキャッシュ

モジュールのディスクキャッシュはドライバーではサポートされていません。モジュールのディスクキャッシュが必要な場合、`zeModuleGetNativeBinary` を使用して実装するのはアプリケーションの役割です。

```
...
// pIL のハッシュを計算しキャッシュをチェック
...

if (cacheUpdateNeeded)
{
    size_t szBinary = 0;
    zeModuleGetNativeBinary(hModule, &szBinary, nullptr);

    uint8_t* pBinary = allocate(szBinary);
    zeModuleGetNativeBinary(hModule, &szBinary, pBinary);

    // 対応する IL の pBinary をキャッシュ
    ...

    free(pBinary);
}
```

また、ネイティブバイナリーは、モジュールに関連付けられているすべてのデバッグ情報を保持することに注意してください。これにより、ネイティブバイナリーから作成されたモジュールのデバッグが可能になります。

ビルトインカーネル

ビルトインカーネルはサポートされていませんが、ネイティブ・バイナリー・インターフェイスを使用する上位レベルのランタイムまたはライブラリーによって実装できます。

カーネル

カーネルはモジュール内のカーネルへの参照であり、起動に必要なデータとともに明示的および暗黙的なカーネル引数の両方をサポートします。

以下の擬似コードは、モジュールからカーネルを作成する手順を示しています:

```
ze_kernel_desc_t kernelDesc = {
    ZE_STRUCTURE_TYPE_KERNEL_DESC,
```

```
    nullptr,
    0, // フラグ
    "image_scaling"
};
ze_kernel_handle_t hKernel;
ze_result_t result = zeKernelCreate(hModule, &kernelDesc, &hKernel);

// 未解決のインポートがあるかどうかを確認
if (result == ZE_RESULT_ERROR_INVALID_MODULE_UNLINKED) {
   // モジュール内に解決できないインポート依存関係を検出
   ...
}

// 指定されたモジュールにカーネル "image_scaling" が見つかったか確認
if (result == ZE_RESULT_ERROR_INVALID_KERNEL_NAME) {
   // モジュール内にカーネル "image_scaling" が見つかりません
   ...
}

...
```

カーネル・プロパティー

zeKernelGetProperties を使用して、カーネル・オブジェクトから不変プロパティーを照会します。

```
...
ze_kernel_properties_t kernelProperties;
zeKernelGetProperties(hKernel, &kernelProperties);
...
```

カーネル・プロパティーの詳細については、ze_kernel_properties_t を参照してください。

実行

カーネル・グループ・サイズ

カーネルのグループサイズは、zeKernelSetGroupSize を使用して設定できます。カーネルをコマンドリストに追加する前にグループサイズが設定されていない場合、デフォルトが選択されます。グループサイズは、一連の追加操作で更新できます。ドライバーは、カーネルをコマンドリストに追加する際に、グループサイズ情報をコピーします。

```
zeKernelSetGroupSize(hKernel, groupSizeX, groupSizeY, 1);

...
```

API は、グローバルサイズを提供するときに、推奨されるグループサイズの照会をサポートします。この関数は、`zeKernelSetGroupSize` を使用してカーネルに設定されたグループサイズを無視します。

```
// イメージを処理する推奨グループサイズを検出
uint32_t groupSizeX;
uint32_t groupSizeY;
zeKernelSuggestGroupSize(hKernel, imageWidth, imageHeight, 1, &groupSizeX, &groupSizeY,
nullptr);

zeKernelSetGroupSize(hKernel, groupSizeX, groupSizeY, 1);

...
```

カーネル引数

カーネル引数は、括弧内の明示的なカーネル引数のみを表します（例: `func(arg1, arg2, …)`）。

- カーネル起動の引数を設定するには、`zeKernelSetArgumentValue` を使用します。
- `zeCommandListAppendLaunchKernel` などの関数は、デバイスに送信するカーネル引数のコピーを作成します。
- カーネル引数はいつでも更新でき、複数の追加呼び出しでも使用できます。

イメージを引数として使用する場合、実装はイメージ形式が SPIRv モジュールへの引数として有効であるかを確認できます。イメージ形式が無効な場合、実装は `ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT` を返す場合があります。

割り当てられたイメージタイプが有効な場合、実装はイメージ作成時に未サポートを返すことはできません。これらのイメージは、SPIRv イメージタイプに限定されないカーネルで使用される場合があります。例えば、SPIRv よりも多くのイメージタイプをサポートし、チャンネル・データ・タイプ引数を使用しない VC ランタイムビルドのネイティブバイナリーなどがあります。

SPIRv チャンネルタイプと OpenCL イメージは同じチャンネル・データ・タイプの制限を共有するため、実装では OpenCL タイプチェックを再利用して、イメージを SPIRv モジュールのカーネルの引数として設定できるか確認できます。

以下の擬似コードは、カーネル引数を設定してカーネルを起動する手順を示しています:

```
// 引数をバインド
zeKernelSetArgumentValue(hKernel, 0, sizeof(ze_image_handle_t), &src_image);
zeKernelSetArgumentValue(hKernel, 1, sizeof(ze_image_handle_t), &dest_image);
zeKernelSetArgumentValue(hKernel, 2, sizeof(uint32_t), &width);
zeKernelSetArgumentValue(hKernel, 3, sizeof(uint32_t), &height);
```

```
ze_group_count_t launchArgs = { numGroupsX, numGroupsY, 1 };

// 起動カーネルを追加
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);

// 次のイメージをコピーして拡大縮小するためイメージポインターを更新
zeKernelSetArgumentValue(hKernel, 0, sizeof(ze_image_handle_t), &src2_image);
zeKernelSetArgumentValue(hKernel, 1, sizeof(ze_image_handle_t), &dest2_image);

// 起動カーネルを追加
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);

...
```

カーネルの起動

デバイス上でカーネルを起動するには、アプリケーションはコマンドリストの `AppendLaunchKernel` スタイルの関数の 1 つを呼び出す必要があります。これらの最も基本的なバージョンは、コマンドリスト、カーネルハンドル、起動引数、および完了を通知するために使用されるオプションの同期イベントを受け取る `zeCommandListAppendLaunchKernel` です。起動引数にはスレッドグループの次元が含まれます。

```
// イメージサイズとグループサイズに基づいて起動するグループ数を計算
uint32_t numGroupsX = imageWidth / groupSizeX;
uint32_t numGroupsY = imageHeight / groupSizeY;

ze_group_count_t launchArgs = { numGroupsX, numGroupsY, 1 };

// 起動カーネルを追加
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);
```

関数 `zeCommandListAppendLaunchKernelIndirect` を使用すると、コマンドの代わりにデバイスが読み取るバッファーで起動パラメーターを間接的に供給できます。これにより、デバイス上の以前の操作でパラメーターを生成できるようになります。

```
ze_group_count_t* pIndirectArgs;
...
zeMemAllocDevice(hContext, &desc, sizeof(ze_group_count_t), sizeof(uint32_t), hDevice,
&pIndirectArgs);

// 起動カーネルを追加 - 間接
zeCommandListAppendLaunchKernelIndirect(hCommandList, hKernel, &pIndirectArgs, nullptr, 0,
nullptr);
```

協調カーネル

協調カーネルにより、起動されたすべてのグループ間でのデータの共有と同期が安全に行えます。これをサポートする、相互に協力できるグループの起動を可能にする `zeCommandListAppendLaunchCooperativeKernel` があります。コマンドリストは、`ZE_COMMAND_QUEUE_GROUP_PROPERTY_FLAG_COOPERATIVE_KERNELS` フラグが設定されたコマンド・キュー・グループの順番で作成されたコマンドキューに送信する必要があります。協調カーネル起動の最大グループ数は、`zeKernelSuggestMaxCooperativeGroupCount` を呼び出して決定できます。

```
// カーネルの最大協調カーネル起動を照会
uint32_t maxGroupCount;
zeKernelSuggestMaxCooperativeGroupCount(hKernel, &maxGroupCount);

// グループの合計数は、照会された最大値より小さくなければなりません
assert(numGroupsX * numGroupsY * numGroupsZ < maxGroupCount);

ze_group_count_t launchArgs = { numGroupsX, numGroupsY, numGroupsZ };

// 起動協調カーネルを追加
zeCommandListAppendLaunchCooperativeKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,
nullptr);
```

サンプラー

API は、カーネル内からイメージをサンプリングするのに必要な状態を表すサンプラー・オブジェクトをサポートします。`zeSamplerCreate` 関数は、サンプラー記述子（`ze_sampler_desc_t`）を受け取ります:

サンプラーフィールド	説明
アドレスモード	範囲外アクセスの処理方法を決定します。`ze_sampler_address_mode_t` を参照。
フィルターモード	使用するフィルターモードを指定します。`ze_sampler_filter_mode_t` を参照。
正規化	イメージのアドレス指定の座標を `[0,1]` に正規化するかどうかを指定します。

次の疑似コードは、サンプラー・オブジェクトを作成し、カーネル引数として渡す方法を示しています:

```
// 線形フィルター用のサンプラーを設定し、エッジへの境界外アクセスをクランプ
ze_sampler_desc_t desc = {
    ZE_STRUCTURE_TYPE_SAMPLER_DESC,
    nullptr,
    ZE_SAMPLER_ADDRESS_MODE_CLAMP,
    ZE_SAMPLER_FILTER_MODE_LINEAR,
    false
    };
ze_sampler_handle_t sampler;
zeSamplerCreate(hContext, hDevice, &desc, &sampler);
```

```
...

// サンプラーはカーネル引数として渡すことができます
zeKernelSetArgumentValue(hKernel, 0, sizeof(ze_sampler_handle_t), &sampler);

// 起動カーネルを追加
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);
```

書式付き出力

API は、`printf` などの関数を使用して、カーネルから書式付き出力をプリントする機能をサポートしています。フォーマットされた出力をプリントする呼び出しにより、データが内部バッファーに書き込まれます。内部バッファーのサイズは、`ze_device_module_properties_t.printfBufferSize` によって指定されます。内部バッファーがいっぱいになると、書式付き出力をプリントする追加呼び出しでエラーコードが返されます。

書式付き出力の順序には保証がありません。複数のワーク項目が `printf` を複数回呼び出す場合、1 つのワーク項目からの出力が他のワーク項目からの出力と混在して表示されることがあります。

一部のデバイスでは、内部バッファーに書式付き出力自体が含まれず、代わりにホスト上で書式化が行われる場合があります。さらに、カーネルの起動に関連付けられたイベントが完了するまで、最終的な書式化が行われず、出力が出力ストリームにフラッシュされない場合があります。すべての出力が出力ストリームにフラッシュされたことを確認するには、カーネルの起動に関連付けられたイベントを待機するか、`zeFenceHostSynchronize` や `zeCommandQueueSynchronize` など粒度の粗い同期を使用してカーネルの起動が完了するまで待機します。

高度

環境変数

次の表は、デフォルトの機能動作をオーバーライドするためにサポートされているノブを示しています。

カテゴリー	名称	値	説明
デバイス	ZE_FLAT_DEVICE_HIERARCHY	{COMPOSITE, FLAT, COMBINED}	レベルゼロドライバーの実装によって公開されるデバイス階層モデルを定義します
	ZE_AFFINITY_MASK	リスト	ドライバーが値で指定されたデバイス（およびサブデバイス）のみを報告するように強制します。
	ZE_ENABLE_PCI_ID_DEVICE_ORDER	{0, 1}	ドライバーに、PCI バス ID の低いものから高いものの順にデバイスを報告するように強制します
メモリー	ZE_SHARED_FORCE_DEVICE_ALLOC	{0, 1}	すべての共有割り当てをデバイスメモリーに強制します

カテゴリー	名称	値	説明
ドライバー	ZEL_DRIVERS_ORDER	文字列	ユーザーに報告するドライバーの順序を定義します。構文の詳細については、ドライバーの順序セクションを参照してください。

ドライバーの順序

レベルゼロランタイムは、環境変数を通じて "zer" レベルゼロランタイム API で使用されるデフォルトのドライバーを変更し、ドライバーの選択と順序付けを可能にする機能を提供します。

この環境変数は、レベルゼロローダーによって読み取られ、ドライバーの初期化および使用順序を決定します。

次の構文を使用して、異なるタイプの複数のドライバーを考慮した堅牢なドライバーセレクターが作成されます:

1. **特定のタイプとそのタイプ内のインデックスを指定します:** `ZEL_DRIVERS_ORDER=<driver_type>:<driver_index>[,<driver_type>:<driver_index>]`
2. **特定のタイプを指定します:** `ZEL_DRIVERS_ORDER=<driver_type>[,<driver_type>]`
3. **ドライバー・インデックスのみを指定する（元のグローバル・ドライバー・インデックスを参照):** `ZEL_DRIVERS_ORDER=<driver_index>[,<driver_index>]`

サポートされているドライバーのタイプ:

- `DISCRETE_GPU_ONLY`
- `GPU`
- `INTEGRATED_GPU_ONLY`
- `NPU`

これにより、すべてのドライバーを順序付けしたり、先頭に指定したドライバーを基準にドライバーを並べ替えたりすることができます。他のライブラリーではドライバーが“マスク”されないため、ローダーレベルで読み込まれる環境変数によってドライバーとデバイスをマスクすることはできません。`ZEL_DRIVERS_ORDER` 環境変数で明示的に指定されないデバイスやドライバータイプは、レベルゼロローダーによって引き続き公開されますが、デフォルトの順序ではドライバーリストの末尾に表示されます。

使用例

1. `ZEL_DRIVERS_ORDER = DISCRETE_GPU_ONLY:1, NPU`

   2 つの GPU ドライバー（discrete:0、discrete:1）と 1 つの NPU ドライバーを持つシステムでは、デフォルトの順序は Discrete:0、Discrete:1、NPU ですが、この設定により順序が次のように変更されます:

   - Discrete:1、NPU:0、Discrete:0
   - zer で使用されるインデックス 0 == Discrete:1

2. `ZEL_DRIVERS_ORDER = 2,0`

   2 つの GPU ドライバー（Discrete、Integrated）と 1 つの NPU ドライバーを備えたシステムでは、デフォルトの順序は Discrete、Integrated、NPU ですが、この設定により順序が次のように変更されます:

   - NPU、Discrete、Integrated
   - zer で使用されるインデックス 0 == NPU

3. `ZEL_DRIVERS_ORDER = GPU:1, NPU:0`

   2 つの GPU ドライバー（Discrete、Integrated）と 1 つの NPU ドライバーを備えたシステムでは、デフォルトの順序は Discrete、Integrated、NPU で、タイプごとのインデックスは次のとおりです:

   - GPU: Discrete、Integrated
   - NPU: NPU

   結果の順序:

   - Integrated、NPU、Discrete
   - zer で使用されるインデックス 0 == Integrated

4. `ZEL_DRIVERS_ORDER = NPU`

   2 つの GPU ドライバー（Discrete、Integrated）と 1 つの NPU ドライバーを備えたシステムでは、デフォルトの順序は Discrete、Integrated、NPU ですが、この設定により順序が次のように変更されます:

   - NPU、Discrete、Integrated
   - zer で使用されるインデックス 0 == NPU

デバイス検出ツール

デバッグやシステム検査の目的で、インテル® コンピュート・ベンチマーク・リポジトリーの `show_devices_l0` ツールを使用して、システム上のドライバー情報やデバイス・プロパティーを簡単に表示できます:

```
# コンピュート・ベンチマーク・リポジトリーのクローンとビルド
git clone intel/compute-benchmarks.git
cd compute-benchmarks
mkdir build && cd build
cmake ..
make show_devices_l0

# ツールを実行すると、ドライバーのインデックス、ドライバーのタイプ、デバイス情報が表示されます
./source/tools/show_devices_l0
```

このツールは以下を表示します:

- 各レベルゼロドライバーのドライバー・インデックスとドライバータイプ
- デバイスタイプ、名前、機能などのデバイス・プロパティー
- サブデバイスの情報（利用可能な場合）
- メモリーのプロパティーやその他のデバイス固有の詳細

`show_devices_l0` ツールは、レベルゼロドライバーとデバイス階層を検査する便利な方法を提供します。これは、`ZEL_DRIVERS_ORDER` 環境変数を使用してドライバーの順序を構成する場合に特に便利です。

デバイス階層

`ZE_FLAT_DEVICE_HIERARCHY` を使用すると、基盤となるハードウェアが公開されるデバイス階層モデルと、`zeDeviceGet` で返されるデバイスのタイプをユーザーが選択できます。

`COMPOSITE` に設定すると、`zeDeviceGet` はルートデバイスを持たないすべてのデバイスを返します。デバイス階層を走査するには、`zeDeviceGetSubDevices` を使用してサブデバイスを照会し、`zeDeviceGetRootDevice` を使用してルートデバイスを照会します。ドライバー実装では、ルートデバイスで行われた送信と割り当てに対して暗黙的な最適化を行う場合があります。

`FLAT` に設定すると、`zeDeviceGet` はサブデバイスを持たないすべてのデバイスを返します。デバイス階層を走査することは**できません。**`zeDeviceGetSubDevices` は常に 0 個のデバイスハンドルを返し、`zeDeviceGetRootDevice` は nullptr を返します。このモードでは、レベルゼロのドライバー実装は、同じアプリケーションでルートデバイスとサブデバイスを同時に使用するために必要なオーバーヘッドを削除することで、実行とメモリーの割り当てを最適化できます。

`COMBINED` に設定すると、`zeDeviceGet` はサブデバイスを持たないすべてのデバイスを返します。デバイス階層を走査するには、`zeDeviceGetSubDevices` を使用してサブデバイスを照会し、`zeDeviceGetRootDevice` を使用してルートデバイスを照会します。ドライバー実装では、ルートデバイスで行われた送信と割り当てに対して暗黙的な最適化を行う場合があります。

SYSMAN API によって返されるデバイスは、`ZE_FLAT_DEVICE_HIERARCHY` の影響を受けず、常に物理デバイスに対応する最上位のデバイスハンドルを返します。

アフィニティー・マスク

アフィニティー・マスクを使用すると、アプリケーションまたはツールは、サードパーティーのライブラリーまたは別のプロセス内のアプリケーションから見えるデバイスとサブデバイスをそれぞれ制限できます。アフィニティー・マスクは、`zeDeviceGet` および `zeDeviceGetSubDevices` から返されるハンドル数に影響します。アフィニティー・マスクは、デバイスおよび/またはサブデバイスの順番をカンマで区切ったリストとして環境変数で指定されます。値はシステム構成に固有です（デバイス数、各デバイスのサブデバイス数など）。値は、ドライバーによってデバイスが報告される順序に固有です。つまり、最初のデバイスは順序 0 にマップされ、2 番目のデバイスは順序 1 にマップされ、以下同様に続きます。アフィニティー・マスクが設定されていない場合は、デフォルトの動作として、すべてのデバイスとサブデバイスが報告されます。

アフィニティー・マスクは、`ZE_FLAT_DEVICE_HIERARCHY` 環境変数に設定された値によって定義されたデバイスをマスクします。つまり、レベルゼロドライバーは最初に `ZE_FLAT_DEVICE_HIERARCHY` を読み取って、アプリ

ケーションで使用されるデバイスハンドルを決定し、次に選択されたデバイスモデルに基づいて ZE_AFFINITY_MASK で渡された値を解釈します。

zeDeviceGet によって報告されるデバイスの順序は実装固有であり、アフィニティー・マスク内のデバイス順序の影響を受けません。

ZE_ENABLE_PCI_ID_DEVICE_ORDER 環境変数を設定することで、zeDeviceGet によって報告されるデバイスの順序を強制することができます。

次の例は、2 つの物理デバイスで構成され、各デバイスがさらに 4 つの小さなデバイスに分割できるシステム構成の使用方法を示しています。この例では、2 つの物理デバイスを*親デバイス*、小さいサブデバイスを*タイル*と呼びます。

ZE_AFFINITY_MASK を異なる値に設定し、ZE_FLAT_DEVICE_HIERARCHY を *COMPOSITE* に設定すると、次のシナリオが発生する可能性があります:

ZE_AFFINITY_MASK = *0, 1*: すべての親デバイスとタイルが報告されます（デフォルトと同じ）:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	はい	デバイスハンドル 0、サブデバイス・ハンドル 0
0	1	はい	デバイスハンドル 0、サブデバイス・ハンドル 1
0	2	はい	デバイスハンドル 0、サブデバイス・ハンドル 2
0	3	はい	デバイスハンドル 0、サブデバイス・ハンドル 3
1	0	はい	デバイスハンドル 1、サブデバイス・ハンドル 0
1	1	はい	デバイスハンドル 1、サブデバイス・ハンドル 1
1	2	はい	デバイスハンドル 1、サブデバイス・ハンドル 2
1	3	はい	デバイスハンドル 1、サブデバイス・ハンドル 3

ZE_AFFINITY_MASK = *0*: 親デバイス 0 のみがデバイスハンドル 0 として報告され、そのすべてのタイルがサブデバイス・ハンドルとして報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	はい	デバイスハンドル 0、サブデバイス・ハンドル 0
0	1	はい	デバイスハンドル 0、サブデバイス・ハンドル 1
0	2	はい	デバイスハンドル 0、サブデバイス・ハンドル 2
0	3	はい	デバイスハンドル 0、サブデバイス・ハンドル 3
1	0	いいえ	
1	1	いいえ	
1	2	いいえ	
1	3	いいえ	

ZE_AFFINITY_MASK = *1*: 親デバイス 1 のみがデバイスハンドル 0 として報告され、そのすべてのタイルがサ

ブデバイス・ハンドルとして報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	いいえ	
0	1	いいえ	
0	2	いいえ	
0	3	いいえ	
1	0	はい	デバイスハンドル 0、サブデバイス・ハンドル 0
1	1	はい	デバイスハンドル 0、サブデバイス・ハンドル 1
1	2	はい	デバイスハンドル 0、サブデバイス・ハンドル 2
1	3	はい	デバイスハンドル 0、サブデバイス・ハンドル 3

`ZE_AFFINITY_MASK = 0.0`: 親デバイス 0 のタイル 0 のみがデバイスハンドル 0 として報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	はい	デバイスハンドル 0
0	1	いいえ	
0	2	いいえ	
0	3	いいえ	
1	0	いいえ	
1	1	いいえ	
1	2	いいえ	
1	3	いいえ	

`ZE_AFFINITY_MASK = 0.0`: 親デバイス 1 のみがデバイスハンドル 0 として報告され、そのタイル 1 と 2 はそれぞれサブデバイス 0 と 1 として報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	いいえ	
0	1	いいえ	
0	2	いいえ	
0	3	いいえ	
1	0	いいえ	
1	1	はい	デバイスハンドル 0、サブデバイス・ハンドル 0
1	2	はい	デバイスハンドル 0、サブデバイス・ハンドル 1
1	3	いいえ	

`ZE_AFFINITY_MASK = 0.2, 1.3, 1.0, 0.3`: 親デバイス 0 と 1 は、それぞれデバイスハンドル 0 と 1 として報告されます。親デバイス 0 は、タイル 2 と 3 をそれぞれサブデバイス 0 と 1 として報告します。親デバイス 1 は、タイル 0 と 3 をそれぞれサブデバイス 0 と 1 として報告します。順序は変更されません:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	いいえ	
0	1	いいえ	
0	2	はい	デバイスハンドル 0、サブデバイス・ハンドル 0
0	3	はい	デバイスハンドル 0、サブデバイス・ハンドル 1
1	0	はい	デバイスハンドル 1、サブデバイス・ハンドル 0
1	1	いいえ	
1	2	いいえ	
1	3	はい	デバイスハンドル 1、サブデバイス・ハンドル 1

次の例は、2 つの親デバイスとそれぞれ 4 つのタイルを備えた同じシステムで、`ZE_FLAT_DEVICE_HIERARCHY` を *FLAT* に設定するときに、`ZE_AFFINITY_MASK` で異なる値を使用する方法を示しています。`ZE_FLAT_DEVICE_HIERARCHY` を *FLAT* に設定すると、`zeDeviceGet` によってタイルのみが報告されます。つまり、このシステムでは、`zeDeviceGet` は最大 8 つのデバイスハンドルを報告し、デバイスハンドル 0 ～ 3 は親デバイス 0 の 4 つのタイルに、デバイスハンドル 4 ～ 7 は親デバイス 1 の 4 つのタイルに対応します:

`ZE_AFFINITY_MASK = 0, 1, 2, 3, 4, 5, 6, 7`: すべてのタイルは `zeDeviceGet` によってデバイスハンドルとして報告されます（デフォルトと同じ）:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	はい	デバイスハンドル 0
0	1	はい	デバイスハンドル 1
0	2	はい	デバイスハンドル 2
0	3	はい	デバイスハンドル 3
1	0	はい	デバイスハンドル 4
1	1	はい	デバイスハンドル 5
1	2	はい	デバイスハンドル 6
1	3	はい	デバイスハンドル 7

`ZE_AFFINITY_MASK = 0`: 親デバイス 0 のタイル 0 のみがデバイスハンドル 0 として報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	はい	デバイスハンドル 0
0	1	いいえ	

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	2	いいえ	
0	3	いいえ	
1	0	いいえ	
1	1	いいえ	
1	2	いいえ	
1	3	いいえ	

`ZE_AFFINITY_MASK = 1`: 親デバイス 0 のタイル 1 のみがデバイスハンドル 0 として報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	いいえ	
0	1	はい	デバイスハンドル 0
0	2	いいえ	
0	3	いいえ	
1	0	いいえ	
1	1	いいえ	
1	2	いいえ	
1	3	いいえ	

`ZE_AFFINITY_MASK = 0, 4`: 親デバイス 0 のタイル 0 はデバイスハンドル 0 として報告され、親デバイス 1 のタイル 0 はデバイスハンドル 1 として報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	はい	デバイスハンドル 0
0	1	いいえ	
0	2	いいえ	
0	3	いいえ	
1	0	はい	デバイスハンドル 1
1	1	いいえ	
1	2	いいえ	
1	3	いいえ	

`ZE_AFFINITY_MASK = 1, 2, 7`: 親デバイス 0 のタイル 1 はデバイスハンドル 0 として報告され、親デバイス 0 のタイル 2 はデバイスハンドル 1 として報告され、親デバイス 1 のタイル 3 はデバイスハンドル 2 として報告されます:

親デバイス	タイル	エクスポーズ	デバイスハンドル使用
0	0	いいえ	
0	1	はい	デバイスハンドル 0
0	2	はい	デバイスハンドル 1
0	3	いいえ	
1	0	いいえ	
1	1	いいえ	
1	2	いいえ	
1	3	はい	デバイスハンドル 2

`ZE_AFFINITY_MASK = 0.0`: は無効です。`ZE_FLAT_DEVICE_HIERARCHY` が *FLAT* に設定されている場合、`zeDeviceGet` によって報告されるデバイスハンドルには、サブデバイスが含まれません。

サブデバイスのサポート

API によりサブデバイスのサポートが可能になり、デバイスのサブ・パーティションへのスケジュールとメモリー割り当てを詳細に制御できるようになります。サブデバイスを照会して取得する関数はありますが、これらの関数外ではサブデバイスとデバイスを区別しません。サブデバイスはデバイスの固有のパーティションを表す必要はありません。つまり、複数のサブデバイスが同じ物理ハードウェアを共有できます。さらに、サブデバイスは、たとえば 1 つのスライスにまで、さらに多くのサブデバイスに分割できます。

`zeDeviceGetSubDevices` を使用して、サブデバイスがサポートされていることを確認し、サブデバイス・ハンドルを取得します。サブデバイスの `ze_device_properties_t` には追加のデバイス・プロパティーがあります。これらは、デバイスがサブデバイスであることを確認したり、サブデバイス ID を照会したりするために使用できます。ライブラリーは、入力デバイスハンドルがデバイスを表すか、サブデバイスを表すのか判断するために使用できます。

親デバイス上のデバイスメモリー割り当てをサブデバイスが認識するには、ドライバーが必要です。ただし、サブデバイス・ハンドルを使用する場合、ドライバーは、サブデバイスに接続されているローカルメモリーにデバイスメモリー割り当てを配置しようとします。これらの割り当ては、サブデバイス、そのサブデバイスでのみ可視です。API 呼び出しが `ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY` を返す場合、アプリケーションは親デバイスを使用して再試行を試みる可能性があります。

サブデバイスのコマンドキューを作成する場合、アプリケーションはサブデバイス・ハンドルを使用して `zeDeviceGetCommandQueueGroupProperties` を呼び出して順番を決定する必要があります。詳細は `ze_command_queue_desc_t` を参照してください。

`zeDeviceGetProperties` を使用して、デバイスまたはサブデバイスの 16 バイトの一意のデバイス識別子（uuid）を取得できます。

```
// デバイスのすべてのサブデバイスを照会
uint32_t subdeviceCount = 0;
zeDeviceGetSubDevices(hDevice, &subdeviceCount, nullptr);
```

```
ze_device_handle_t* allSubDevices = allocate(subdeviceCount * sizeof(ze_device_handle_t));
zeDeviceGetSubDevices(hDevice, &subdeviceCount, &allSubDevices);

// サブデバイス 2 にワークを割り当ててディスパッチすることが目的です
assert(subdeviceCount >= 3);
ze_device_handle_t hSubdevice = allSubDevices[2];

// サブデバイスのプロパティーを照会
ze_device_properties_t subdeviceProps {};
subDeviceProps.stype = ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;
zeDeviceGetProperties(hSubdevice, &subdeviceProps);

assert(subdeviceProps.flags & ZE_DEVICE_PROPERTY_FLAG_SUBDEVICE); // サブデバイスへのハンドルがあることを確認
assert(subdeviceProps.subdeviceId == 2);    // 要求したサブデバイスへのハンドルがあることを確認

void* pMemForSubDevice2;
zeMemAllocDevice(hContext, &desc, memSize, sizeof(uint32_t), hSubdevice,
&pMemForSubDevice2);
...
```

デバイスの常駐

ページフォールトをサポートしていないデバイスの場合、ドライバーは、プログラムの実行前にカーネルによってアクセスされるすべてのページが常駐していることを確認する必要があります。これは、`ze_device_properties_t.flags` で `ZE_DEVICE_PROPERTY_FLAG_ONDEMANDPAGING` を確認することで判断できます。

ほとんどの場合、ドライバーはデバイスアクセスの割り当ての常駐を暗黙的に処理します。これは、カーネル引数を含む API パラメーターを検査することによって行われます。ただし、デバイスがページフォールトを**サポートしておらず**、ドライバーが複数レベルの間接参照など、デバイスが割り当てにアクセスするか判断できない場合は、次の 2 つの方法があります:

1. アプリケーションは、プログラム作成中に `ZE_KERNEL_FLAG_FORCE_RESIDENCY` フラグを設定して、実行中にすべてのデバイス割り当てを強制的に常駐させることができます。
    - アプリケーションは、`zeKernelSetIndirectAccess` と次のフラグを使用して、間接的にアクセスする割り当てタイプを指定し、どの割り当てを常駐させるかを最適化する必要があります。
        - `ZE_KERNEL_INDIRECT_ACCESS_FLAG_HOST`
        - `ZE_KERNEL_INDIRECT_ACCESS_FLAG_DEVICE`
        - `ZE_KERNEL_INDIRECT_ACCESS_FLAG_SHARED`

- ドライバーがすべての割り当てを常駐させることができない場合、`zeCommandQueueExecuteCommandLists` の呼び出しは `ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY` を返します

2. アプリケーションが必要に応じて動的に常駐を変更できるように、明示的な `zeContextMakeMemoryResident` API が含まれています。

   - アプリケーションがデバイスメモリーをオーバーコミットした場合、`zeContextMakeMemoryResident` 呼び出しは `ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY` を返します

アプリケーションがこのようなケースで常駐を適切に管理しないと、デバイスで回復不可能なページフォールトが発生する可能性があります。

次の疑似コードは、間接引数に対して粗粒度の常駐制御を使用する手順を示しています:

```
struct node {
   node* next;
};
node* begin = nullptr;
zeMemAllocHost(hContext, &desc, sizeof(node), 1, &begin);
zeMemAllocHost(hContext, &desc, sizeof(node), 1, &begin->next);
zeMemAllocHost(hContext, &desc, sizeof(node), 1, &begin->next->next);

// 'begin' はカーネル引数として渡され、コマンドリストに追加されます
zeKernelSetIndirectAccess(hKernel, ZE_KERNEL_INDIRECT_ACCESS_FLAG_HOST);
zeKernelSetArgumentValue(hKernel, 0, sizeof(node*), &begin);
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);

...

zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);
...
次の疑似コードは、間接引数に対して細粒度の常駐制御を使用する手順を示しています:
struct node {
   node* next;
};
node* begin = nullptr;
zeMemAllocHost(hContext, &desc, sizeof(node), 1, &begin);
zeMemAllocHost(hContext, &desc, sizeof(node), 1, &begin->next);
zeMemAllocHost(hContext, &desc, sizeof(node), 1, &begin->next->next);

// 'begin' はカーネル引数として渡され、コマンドリストに追加されます
zeKernelSetArgumentValue(hKernel, 0, sizeof(node*), &begin);
```

```
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);
...

// 間接割り当てをキューに入れる前に常駐させます
zeContextMakeMemoryResident(hContext, hDevice, begin->next, sizeof(node));
zeContextMakeMemoryResident(hContext, hDevice, begin->next->next, sizeof(node));

zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, hFence);

// 完了を待機
zeFenceHostSynchronize(hFence, UINT32_MAX);

// 最後に、デバイスリソースを解放するために排出
zeContextEvictMemory(hContext, hDevice, begin->next, sizeof(node));
zeContextEvictMemory(hContext, hDevice, begin->next->next, sizeof(node));
...
```

他の API との相互運用性

レベルゼロには、メモリー割り当て（イメージとデバイスメモリーの両方）とモジュールの汎用相互運用性メカニズムが含まれています。

メモリー割り当ては、外部メモリーのインポートとエクスポートを介して、レベルゼロと他の API 間で共有できます。レベルゼロは、他の API で使用するメモリー割り当てのエクスポートと、他の API からのメモリー割り当てのインポートをサポートしています。

モジュールは、ネイティブ形式のバイナリーを介してレベルゼロと他の API 間で共有できます。`zeModuleGetNativeBinary` と `ZE_MODULE_FORMAT_NATIVE` を参照してください。

次の疑似コードは、OpenCL `cl_program` からレベルゼロカーネルまでの OpenCL との相互運用性を示しています:

```
void* clDeviceBinary;
size_t clDeviceBinarySize;
clGetProgramInfo(cl_program, CL_PROGRAM_BINARIES, clDeviceBinary, &clDeviceBinarySize);

ze_module_desc_t desc = {
    ZE_STRUCTURE_TYPE_MODULE_DESC,
    nullptr,
    ZE_MODULE_FORMAT_NATIVE,
    clDeviceBinarySize,
    clDeviceBinary
};
zeModuleCreate(hContext, hDevice, &desc, &hModule, nullptr);
```

```
zeKernelCreate(hModule, nullptr, hKernel); // レベルゼロの OpenCL と同じカーネル
```

プロセス間通信

API を使用すると、異なるデバイスプロセス間でメモリー・オブジェクトを共有できます。各プロセスには独自の仮想アドレス空間があるため、メモリー・オブジェクトが新しいプロセスで共有されるときに同じ仮想アドレスを使用できる保証はありません。メモリー・オブジェクトを簡単に共有する一連の API があります。

プロセス間でレベルゼロ割り当てを使用するプロセス間通信（IPC）API には、次の 2 種類があります:

1. メモリー
2. イベント

メモリー

次のコード例は、メモリー IPC API の使用方法を示しています:

1. まず、割り当てが行われ、パッケージ化され、送信プロセスで送信されます:

```
void* dptr = nullptr;
zeMemAllocDevice(hContext, &desc, size, alignment, hDevice, &dptr);

ze_ipc_mem_handle_t hIPC;
zeMemGetIpcHandle(hContext, dptr, &hIPC);

// 受信プロセスへの送信方法はレベルゼロでは定義されていません:
send_to_receiving_process(hIPC);
次に、割り当てが受信プロセスで受信され、パッケージが解除されます:
// 送信プロセスからの受信方法はレベルゼロでは定義されていません:
ze_ipc_mem_handle_t hIPC;
hIPC = receive_from_sending_process();

void* dptr = nullptr;
zeMemOpenIpcHandle(hContext, hDevice, hIPC, 0, &dptr);
```

3. 各プロセスは、`dptr` を介して同じデバイスメモリー割り当てを参照できるようになりました。各プロセスの `dptr` の値のアドレスが等価であることは保証されないことに注意してください。
4. クリーンアップするには、まず受信プロセスでハンドルをクローズします:

```
zeMemCloseIpcHandle(hContext, dptr);
```

5. 最後に、`zeMemPutIpcHandle` を使用して IPC ハンドルをドライバーに返し、送信プロセスでデバイスポインターを解放します。`zeMemPutIpcHandle` が呼び出されない場合、その呼び出しによって実行されるすべてのアクションは最終的に `zeMemFree` によって実行されます。

```
zeMemPutIpcHandle(hContext, hIpc);
zeMemFree(hContext, dptr);
```

イベント

次のコード例は、イベント IPC API の使用方法を示しています:

1. まず、イベントプールが作成され、パッケージ化され、送信プロセスで送信されます:

```
// イベントプールの作成
ze_event_pool_desc_t eventPoolDesc = {
    ZE_STRUCTURE_TYPE_EVENT_POOL_DESC,
    nullptr,
    ZE_EVENT_POOL_FLAG_IPC | ZE_EVENT_POOL_FLAG_HOST_VISIBLE,
    10 // カウント
};
ze_event_pool_handle_t hEventPool;
zeEventPoolCreate(hContext, &eventPoolDesc, 1, &hDevice, &hEventPool);

// IPC ハンドルを取得して別のプロセスに送信
ze_ipc_event_pool_handle_t hIpcEvent;
zeEventPoolGetIpcHandle(hEventPool, &hIpcEventPool);
send_to_receiving_process(hIpcEventPool);
```

2. 次に、イベントプールが受信プロセスで受信され、パッケージが解除されます:

```
// 他のプロセスから IPC ハンドルを取得
ze_ipc_event_pool_handle_t hIpcEventPool;
receive_from_sending_process(&hIpcEventPool);

// イベントプールを開く
ze_event_pool_handle_t hEventPool;
zeEventPoolOpenIpcHandle(hContext, hIpcEventPool, &hEventPool);
```

3. 各プロセスは、ハンドルを介して同じデバイスメモリー割り当てを参照できるようになりました。

受信プロセスはロケーションでイベントを作成します

```
ze_event_handle_t hEvent;
ze_event_desc_t eventDesc = {
    ZE_STRUCTURE_TYPE_EVENT_DESC,
    nullptr,
    5, // インデックス
    0, // シグナルに追加のメモリー/キャッシュの一貫性維持は不要
    ZE_EVENT_SCOPE_FLAG_HOST  // イベントが通知された後、デバイスとホスト間のメモリの一貫性を確保
};
zeEventCreate(hEventPool, &eventDesc, &hEvent);
```

```
// カーネルを送信し、完了したらイベントを通知
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &args, hEvent, 0, nullptr);
zeCommandListClose(hCommandList);
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);
```

送信プロセスは同じロケーションでイベントを作成

```
ze_event_handle_t hEvent;
ze_event_desc_t eventDesc = {
    ZE_STRUCTURE_TYPE_EVENT_DESC,
    nullptr,
    5,
    0, // シグナルに追加のメモリー/キャッシュの一貫性維持は不要
    ZE_EVENT_SCOPE_FLAG_HOST  // イベントが通知された後、デバイスとホスト間のメモリの一貫性を確保
};
zeEventCreate(hEventPool, &eventDesc, &hEvent);


zeEventHostSynchronize(hEvent, UINT32_MAX);
```

各プロセスの `hEvent` の値のアドレスが等価であることは保証されないことに注意してください。

4. クリーンアップするには、まず受信プロセスでプールのハンドルをクローズします:

```
zeEventDestroy(hEvent);
zeEventPoolCloseIpcHandle(&hEventPool);
```

5. 最後に、zeEventPoolPutIpcHandle を使用して IPC ハンドルをドライバーに返し、送信プロセスでイベントプールを解放します。zeEventPoolPutIpcHandle が呼び出されない場合、その呼び出しによって実行されるすべてのアクションは最終的に zeEventPoolDestroy によって実行されます。

```
zeEventDestroy(hEvent);
zeEventPoolPutIpcHandle(hContext, hIpcEventPool);
zeEventPoolDestroy(hEventPool);
```

ピアツーピア・アクセスとクエリー

ピアツーピア API は、ホストからデバイス、デバイスからホスト、デバイスからデバイス間でデータを変換する機能を提供します。データ・マーシャリング API は、非同期操作としてスケジュールすることも、コマンドキューを介してカーネル実行と同期することもできます。データの一貫性は、アプリケーションからの明示的な関与なしにドライバーによって維持されます。

カードは、スケールアップ・ファブリックによってノード内でリンクされる可能性があり、構成に応じて、ファブリックはリモートアクセス、アトミック、およびデータコピーをサポートします。

次のピアツーピア機能が API を通じて提供されます:

- 2 つのデバイス/サブデバイス間のリモート・メモリー・アクセス機能を確認します: `zeDeviceCanAccessPeer`

`zeDeviceCanAccessPeer` クエリーには以下のルールが適用されます

- デバイス/サブデバイスは常にそれ自身のピアであり、常に自分自身にアクセスできます。

- ピアツーピアのリモート・メモリー・アクセス、アトミック機能、論理および物理帯域幅とレイテンシーを照会します: `zeDeviceGetP2PProperties` + 帯域幅拡張プロパティー。

`zeDeviceGetP2PProperties` クエリーには以下のルールが適用されます

- デバイス/サブデバイスは常にそれ自身のピアであり、常に自分自身にアクセスでき、またアトミックにアクセスすることもできます。

- ピアツーピア・ファブリックを介してデバイス間でデータをコピーします: `zeCommandListAppendMemoryCopy`

`zeDeviceCanAccessPeer` と `zeDeviceGetP2PProperties` はどちらも同じ情報を返します。2 つのデバイスはピアツーピア・アクセスをサポートしていますか? `zeDeviceGetP2PProperties` は、アトミック、帯域幅、レイテンシーなどのサポートなど、`zeDeviceCanAccessPeer` よりも詳細な情報を提供します。

ツール・プログラミング・ガイド

初期化

環境変数

それぞれの機能について、`zeInit` で次の環境変数を有効にする必要があります。

カテゴリー	名称	値	説明
ツール	`ZET_ENABLE_API_TRACING_EXP`	`{0, 1}`	注：`API` トレース用のドライバー・インストルメントを有効にします。このトレース環境変数は非推奨です。代わりにトレース・ローダー・レイヤーを使用してください。
	`ZET_ENABLE_METRICS`	`{0, 1}`	デバイスメトリックのドライバー・インストルメントと依存関係を有効にします
	`ZET_ENABLE_PROGRAM_INSTRUMENTATION`	`{0, 1}`	ドライバー・インストルメントとプログラム・インストルメントの依存関係を有効にします
	`ZET_ENABLE_PROGRAM_DEBUGGING`	`{0, 1}`	プログラムのデバッグのためドライバーのインストルメントと依存関係を有効にします

ローダーレイヤー経由の API トレースのサポート

ローダーレイヤーは、アプリケーションによって行われたレベルゼロ（L0）API 呼び出しをツールが監視および操作するインフラを提供します。これは、API 実行中に特定のポイントで呼び出されるコールバック関数をツールが登録できるようにする、トレースメカニズムを通じて実現されます。これらのコールバックは、L0 API 関数の入力および出力パラメーターに直接アクセスし、検証と変更を可能にします。さらに、ツールはこれらのコールバックを利用して、API 呼び出しをブロックしたりコマンドストリームに挿入することができます（たとえば、メトリックを収集したり、デバッグ機能実装のため）。

トレースは、1 つ以上のトレーサーの作成によって容易になります。各トレーサーは一意のハンドルで表され、プロローグ関数とエピローグ関数のセットに関連付けられます。プロローグ関数は対応する L0 API 呼び出しの前に実行され、エピローグ関数は呼び出しが完了した後に実行されます。このデュアルフック・モデルにより、ツールは開始ポイントと終了ポイントの両方で API の動作を監視および操作できます。

要約すると、ローダーレイヤーは、次の関数を公開することでトレースを可能にします:

- トレーサーを作成し、トレーサーハンドルを取得
- 各トレーサーのプロローグとエピローグのコールバックを登録

- トレースを動的に有効または無効にする
- 不要になったトレーサーを破棄

この設計により、ツール開発者はトレース機能をワークフローに統合できる柔軟で拡張可能なメカニズムを利用できます。

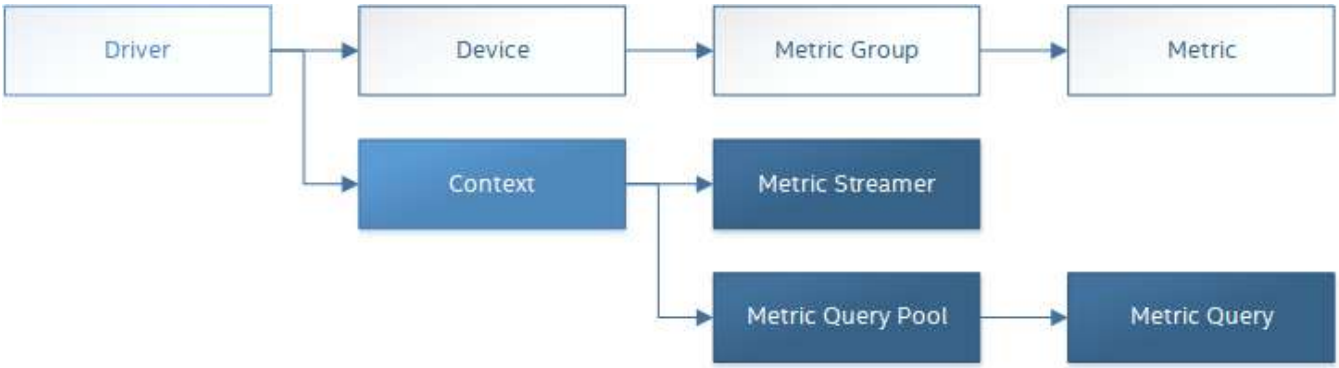
ローダーレイヤーの完全な機能概要については、oneapi-src/level-zero ドキュメントをご覧ください。

メトリック

はじめに

デバイスは、パフォーマンスのデバッグをサポートするように設計されたプログラム可能なインフラを提供します。このドキュメントで説明されている API は、これらのデバイスメトリックへのアクセスを提供します。この API の目的はパフォーマンスのデバッグをサポートすることです。全体のパフォーマンスに影響する可能性があるため、通常の実行で使用することは推奨しません。

次の図は、このドキュメントで説明されているメトリック・オブジェクト間の関係を示しています。



詳細なメトリックのほとんどは、使用前にデバイスを適切にプログラムする必要があります。ほとんどの場合、デバイスのプログラミングはグローバルであることを理解することが重要です。これは通常、ソフトウェア・ツールまたはアプリケーションがメトリックを使用している場合、他のアプリケーションが同じデバイスリソースを利用できないことを意味します。

メトリックグループ

デバイス・インフラストラクチャーは、事前定義された非プログラマブル・カウンターのセットと、別のカウンターセットやその他のタイプのカウンターと連携するプログラマブルな接続ネットワークで構成されます。簡略化のために、構成の最小単位はメトリックグループです。メトリックグループは、ワークロードのパフォーマンスに関する特定の視点を提供するメトリックのセットです。グループはメトリックを集約し、デバイスのプログラミングと利用可能な収集方法を定義します。アプリケーションは、すべてのメトリックグループが異なるドメインに属している場合、複数のメトリックグループからのデータ収集を選択できます。ドメインは、安全に同時利用できる独立したデバイスリソースのソフトウェア表現として使用されます。

サンプルタイプ

サンプルタイプは、メトリック値の読み取りに関するデバイス機能のソフトウェア表現です。各メトリックグループは、

サポートされるサンプルタイプに関する情報を提供します。時間ベースとイベントベースの各サンプルタイプをサポートする個別の API セットがあります。

利用可能なすべてのサンプルタイプは、zet_metric_group_sampling_type_flags_t で定義されています。

- 特定のメトリックグループでサポートされているサンプルタイプに関する情報は、zet_metric_group_properties_t.samplingType で提供されます。
- デバイスが、同じ名前で異なるサンプルタイプを持つ複数のメトリックグループを提供する場合があります。
- 列挙するときは、目的のサンプルタイプをサポートするメトリックグループを選択することが重要です。

ドメイン

すべてのメトリックグループは、特定のドメイン（zet_metric_group_properties_t.domain）に属します。

- メトリックグループは通常、測定に使用される均一なデバイスカウンター構成を定義します。
- 各ドメインは、メトリックグループによって使用される排他的なリソースを表します。
- 2 つの異なるメトリックグループのデータを同時に収集することは可能ですが、それらのメトリックグループが異なるドメインに属している場合に限ります。つまり、同時に収集できるメトリックグループは異なるドメインの値を持ちます。

列挙子

利用可能なすべてのメトリックはメトリックグループに整理されます。

- データ収集中は、メトリックグループ全体のデータが収集されます。
- 利用可能なメトリックグループとそのメトリックのリストはデバイスによって異なります。

次の API は、識別と利用に必要なすべての情報を提供します。

- メトリックグループのプロパティーには、zet_metric_group_properties_t を返す関数 zetMetricGroupGetProperties を介してアクセスします。
- メトリックのプロパティーには、zet_metric_properties_t を返す関数 zetMetricGetProperties を介してアクセスします。

一般的なツールフローは、特定のメトリックグループを探してメトリックを列挙することです。特定のシナリオに必要なメトリックに応じて、ツールはワークロードを複数回実行し、そのたびに異なるメトリックグループのセットを記録する場合があります。通常、異なる実行からのメトリックを共に使用する場合、実行間の安定性と結果の再現性を確保することに注意する必要があります。目的のメトリックグループを見つけるのにメトリックを列挙する場合、どのサンプルタイプが使用されるかを事前に知っておくことが重要です。

メトリックツリーを列挙するには以下を行います:

1. メトリックグループ数を取得するには、zetMetricGroupGet を呼び出します。

2. すべてのメトリックグループを取得するには、zetMetricGroupGet を呼び出します。
3. 利用可能なすべてのメトリックグループを反復処理します。
    - この時点で、メトリックグループ名、ドメイン、またはサンプルタイプを確認できます。
    - メトリックグループ名は一意ではない可能性があります。
4. メトリック・グループ・ハンドル（zet_metric_group_handle_t）を使用して zetMetricGroupGetProperties を呼び出し、zet_metric_group_properties_t.metricCount をチェックして、各メトリックグループのメトリック数を取得します。
5. 親メトリックグループ（zet_metric_group_handle_t）を指定した zetMetricGet を使用して、利用可能なメトリックを反復処理します。
6. 親メトリック（zet_metric_handle_t）を使用して zetMetricGetProperties を呼び出して、メトリックのプロパティー（名前、説明など）を確認します。

次の疑似コードは、利用可能なすべてのメトリックグループとそのメトリックの基本的な列挙を示しています: さらに、選択した名前とサンプルタイプを持つメトリックグループを返します。同様のコードを使用して、特定タイプの測定に対して優先されるメトリックグループを選択できます。

```
ze_result_t FindMetricGroup( ze_device_handle_t hDevice,
                             char* pMetricGroupName,
                             uint32_t desiredSamplingType,
                             zet_metric_group_handle_t* phMetricGroup )
{
    // 特定のデバイスで利用可能なメトリックグループを取得
    uint32_t metricGroupCount = 0;
    zetMetricGroupGet( hDevice, &metricGroupCount, nullptr );

    zet_metric_group_handle_t* phMetricGroups = malloc(metricGroupCount *
sizeof(zet_metric_group_handle_t));
    zetMetricGroupGet( hDevice, &metricGroupCount, phMetricGroups );

    // 利用可能なすべてのメトリックグループを反復処理
    for( i = 0; i < metricGroupCount; i++ )
    {
        // インデックス 'i' の下のメトリックグループとそのプロパティーを取得
        zet_metric_group_properties_t metricGroupProperties {};
        metricGroupProperties.stype = ZET_STRUCTURE_TYPE_METRIC_GROUP_PROPERTIES;
        zetMetricGroupGetProperties( phMetricGroups[i], &metricGroupProperties );

        printf("Metric Group: %sn", metricGroupProperties.name);

        // 取得したメトリックグループが目的のサンプルタイプをサポートするか確認
```

```
        if((metricGroupProperties.samplingType & desiredSamplingType) == desiredSamplingType)
        {
            // 取得したメトリックグループが目的の名前を持っているか確認
            if( strcmp( pMetricGroupName, metricGroupProperties.name ) == 0 )
            {
                *phMetricGroup = phMetricGroups[i];
                break;
            }
        }
    }

    free(phMetricGroups);
}
```

設定

`zetContextActivateMetricGroups` API 呼び出しを使用して、データ収集用にデバイスを構成します。

- この関数を再度呼び出すと、アクティブ化の対象として選択されていないメトリックグループのデバイス・プログラミングが無効になります。
- 未定義の結果を回避するには、データを収集していないときにのみ `zetContextActivateMetricGroups` を呼び出します。

プログラミングの制限:

- すべてのメトリックグループが異なる `zet_metric_group_properties_t.domain` を持つ限り、メトリックグループを任意に組み合わせて同時に構成できます。
- `MetricGroup` は、`zetMetricStreamerClose` と最後の `zetCommandListAppendMetricQueryEnd` が完了するまでアクティブでなければなりません。

収集

サポートされているメトリック収集モードには、時間ベースとイベントベースがあります。

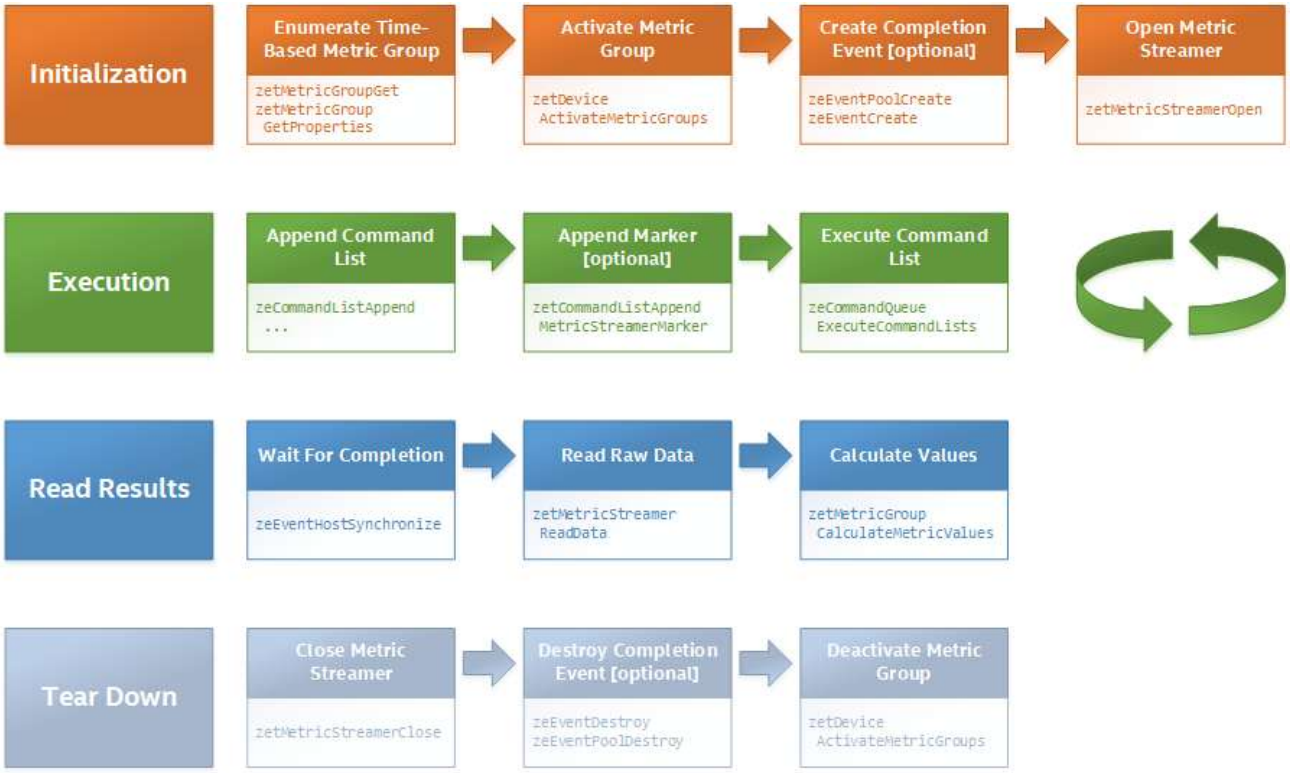
- 時間ベースの収集では、タイマーや他のイベントを使用してデータサンプルを保存します。メトリック・ストリーマー・インターフェイスは、構成と収集向けのソフトウェア・インターフェイスです。
- イベントベースの収集は、コマンドリストに追加された Begin/End イベントのペアで構成されます。メトリック・クエリー・インターフェイスは、構成と収集向けのソフトウェア・インターフェイスです。

メトリック・ストリーマー

時間ベースの収集では、単純な「開く」、「読み取り」、「閉じる」スキームが使用されます:

- `zetMetricStreamerOpen` はストリーマーを開きます。

- `zetMetricStreamerReadData` は、後に `zetMetricGroupCalculateMetricValues` によって処理される生データを読み取ります。
- `zetMetricStreamerClose` はストリーマーを閉じます。



以下の擬似コードは、時間ベースの収集の基本的な手順を示しています:

```
ze_result_t TimeBasedUsageExample( ze_context_handle_t hContext,
                                   ze_device_handle_t hDevice )
{
    zet_metric_group_handle_t     hMetricGroup         = nullptr;
    ze_event_handle_t             hNotificationEvent   = nullptr;
    ze_event_pool_handle_t        hEventPool           = nullptr;
    ze_event_pool_desc_t          eventPoolDesc        = {ZE_STRUCTURE_TYPE_EVENT_POOL_DESC,
nullptr, 0, 1};
    ze_event_desc_t               eventDesc            = {ZE_STRUCTURE_TYPE_EVENT_DESC};
    zet_metric_streamer_handle_t  hMetricStreamer      = nullptr;
    zet_metric_streamer_desc_t    metricStreamerDesc   =
{ZET_STRUCTURE_TYPE_METRIC_STREAMER_DESC};

    // 時間ベースの収集に適した "ComputeBasic" メトリックグループを検索します
    FindMetricGroup( hDevice, "ComputeBasic",
ZET_METRIC_GROUP_SAMPLING_TYPE_FLAG_TIME_BASED, &hMetricGroup );

    // HW を設定
```

```
    zetContextActivateMetricGroups( hContext, hDevice, /* count= */ 1, &hMetricGroup );

    // 通知イベントを作成
    zeEventPoolCreate( hContext, &eventPoolDesc, 1, &hDevice, &hEventPool );
    eventDesc.index  = 0;
    eventDesc.signal = ZE_EVENT_SCOPE_FLAG_HOST;
    eventDesc.wait   = ZE_EVENT_SCOPE_FLAG_HOST;
    zeEventCreate( hEventPool, &eventDesc, &hNotificationEvent );

    // メトリックのストリーマーをオープン
    metricStreamerDesc.samplingPeriod      = 1000;
    metricStreamerDesc.notifyEveryNReports  = 32768;
    zetMetricStreamerOpen( hContext, hDevice, hMetricGroup, &metricStreamerDesc,
hNotificationEvent, &hMetricStreamer );

    // ワークロードを実行します。この例では、実験全体のデータがデバイスバッファーに収まると想定
    Workload(hDevice);
    // ワークロード実行中にオプションでマーカーを挿入
    //zetCommandListAppendMetricStreamerMarker( hCommandList, hMetricStreamer,
tool_marker_value );

    // データを待機します。この例では、すでにすべてのワークロードが実行されているためオプションです
    //zeEventHostSynchronize( hNotificationEvent, 1000 /*timeout*/ );
    // イベントが発生した場合はリセット

    // 生データを読み取り
    size_t rawSize = 0;
    zetMetricStreamerReadData( hMetricStreamer, UINT32_MAX, &rawSize, nullptr );
    uint8_t* rawData = malloc(rawSize);
    zetMetricStreamerReadData( hMetricStreamer, UINT32_MAX, &rawSize, rawData );

    // メトリックのストリーマーをクローズ
    zetMetricStreamerClose( hMetricStreamer );
    zeEventDestroy( hNotificationEvent );
    zeEventPoolDestroy( hEventPool );

    // デバイスの設定を解除
    zetContextActivateMetricGroups( hContext, hDevice, 0, nullptr );

    // メトリックデータを計算
    CalculateMetricsExample( hMetricGroup, rawSize, rawData );
```

```
    free(rawData);
}
```

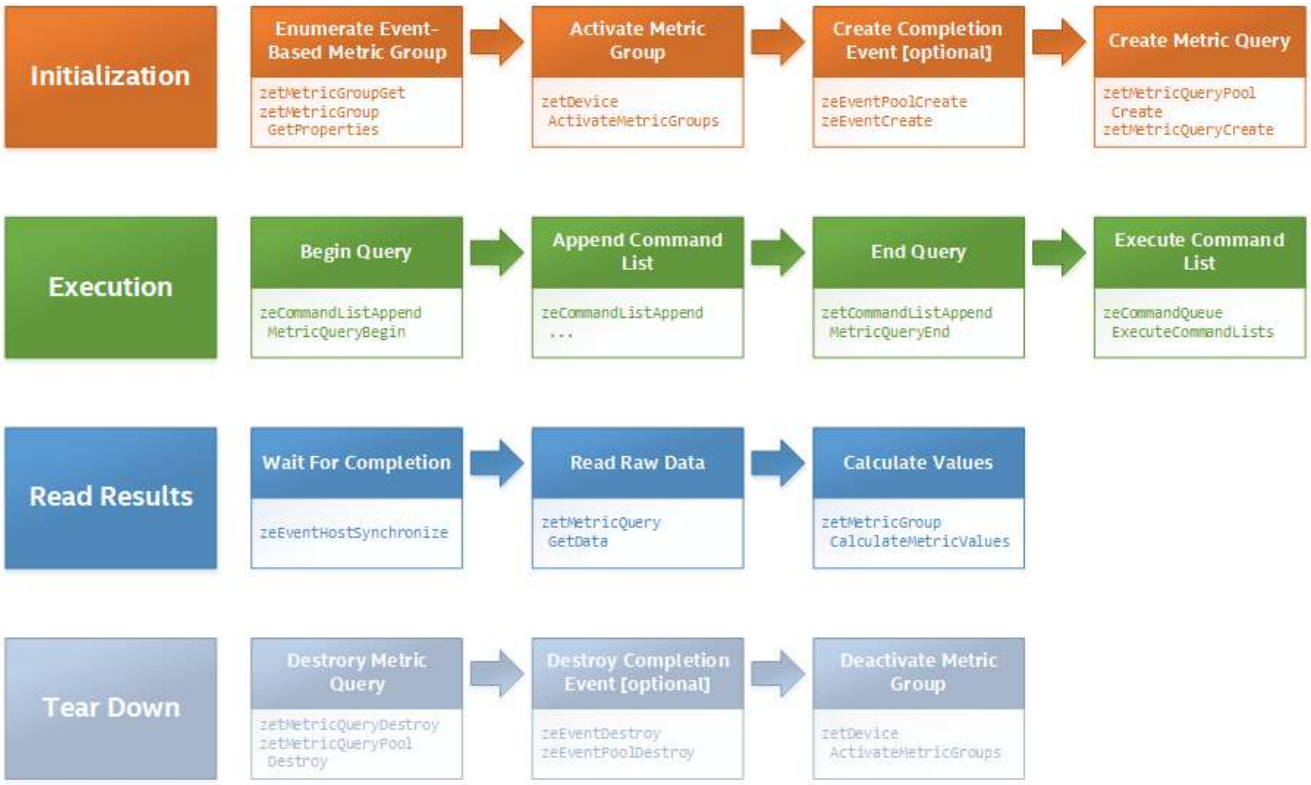
メトリックの照会

イベントベースの収集では、単純な Begin、End、GetData スキームが使用されます:

- `zetCommandListAppendMetricQueryBegin` はカウント開始イベントを定義します。
- `zetCommandListAppendMetricQueryEnd` はカウント終了イベントを定義します。
- `zetMetricQueryGetData` は、後に `zetMetricGroupCalculateMetricValues` によって処理される生データを読み取ります。

通常、ワークロードを特徴付けるため複数のクエリーが使用され、再利用されます。クエリープールは、複数のクエリーに対してデバイスメモリーを効率良く再利用するのに使用されます。

- `zetMetricQueryPoolCreate` は、同種のクエリープールを作成します。
- `zetMetricQueryPoolDestroy` はプールを解放します。アプリケーションは、プールを解放する前に、プール内のクエリーが使用されていないことを確認しなければなりません。
- `zetMetricQueryCreate` は、プール内の一意の場所のハンドルを取得します。
- `zetMetricQueryReset` を使用すると、プール内の場所を低コストで再利用できます。



以下の擬似コードは、クエリーベースの収集の基本的な手順を示しています:

```
ze_result_t MetricQueryUsageExample( ze_context_handle_t hContext,
```

```
                                ze_device_handle_t hDevice )
{
    zet_metric_group_handle_t     hMetricGroup         = nullptr;
    ze_event_handle_t           hCompletionEvent     = nullptr;
    ze_event_pool_desc_t         eventPoolDesc        = {ZE_STRUCTURE_TYPE_EVENT_POOL_DESC,
nullptr};
    ze_event_desc_t             eventDesc           = {ZE_STRUCTURE_TYPE_EVENT_DESC,
nullptr};
    ze_event_pool_handle_t       hEventPool           = nullptr;
    zet_metric_query_pool_handle_t hMetricQueryPool     = nullptr;
    zet_metric_query_handle_t     hMetricQuery         = nullptr;
    zet_metric_query_pool_desc_t   queryPoolDesc        =
{ZET_STRUCTURE_TYPE_METRIC_QUERY_POOL_DESC, nullptr};

    // イベントベースの収集に適した "ComputeBasic" メトリックグループを検索します
    FindMetricGroup( hDevice, "ComputeBasic",
ZET_METRIC_GROUP_SAMPLING_TYPE_FLAG_EVENT_BASED, &hMetricGroup );

    // HW を設定
    zetContextActivateMetricGroups( hContext, hDevice, 1 /* count */, &hMetricGroup );

    // メトリック・クエリー・プールと完了イベントを作成
    queryPoolDesc.type        = ZET_METRIC_QUERY_POOL_TYPE_PERFORMANCE;
    queryPoolDesc.count       = 1000;
    zetMetricQueryPoolCreate( hContext, hDevice, hMetricGroup, &queryPoolDesc,
&hMetricQueryPool );
    eventPoolDesc.flags = 0;
    eventPoolDesc.count = 1000;
    zeEventPoolCreate( hContext, &eventPoolDesc, 1, &hDevice, &hEventPool );

    // コマンドリストに BEGIN メトリッククエリーを書き込む
    zetMetricQueryCreate( hMetricQueryPool, 0 /*slot*/, &hMetricQuery );
    zetCommandListAppendMetricQueryBegin( hCommandList, hMetricQuery );

    // コマンドリストをビルド
    ...

    // コマンドリストに END メトリッククエリーを書き込み、イベントを使用してデータが利用可能であるか判断
    eventDesc.index  = 0;
    eventDesc.signal = ZE_EVENT_SCOPE_FLAG_HOST;
    eventDesc.wait   = ZE_EVENT_SCOPE_FLAG_HOST;
```

```
    zeEventCreate( hEventPool, &eventDesc, &hCompletionEvent);
    zetCommandListAppendMetricQueryEnd( hCommandList, hMetricQuery, hCompletionEvent, 0,
nullptr );

    // zeCommandQueueExecuteCommandLists( , , , ) を使用してワークロードをデバイスに送信

    // データを待機
    zeEventHostSynchronize( hCompletionEvent, 1000 /*timeout*/ );

    // 生データを読み取り
    size_t rawSize = 0;
    zetMetricQueryGetData( hMetricQuery, &rawSize, nullptr );
    uint8_t* rawData = malloc(rawSize);
    zetMetricQueryGetData( hMetricQuery, &rawSize, rawData );

    // リソースを解放
    zeEventDestroy( hCompletionEvent );
    zeEventPoolDestroy( hEventPool );
    zetMetricQueryPoolDestroy( hMetricQueryPool );

    // HW 構成を解除
    zetContextActivateMetricGroups( hContext, hDevice, 0, nullptr );

    // メトリックデータを計算
    CalculateMetricsExample( hMetricGroup, rawSize, rawData );
    free(rawData);
}
```

計算

`MetricStreamer` と `MetricQuery` はどちらも、アプリケーションの処理には適さないデバイス固有の生の形式でデータを収集します。メトリック値を計算するには、`zetMetricGroupCalculateMetricValues` を使用します。

以下の擬似コードは、メトリックの計算と解釈の基本的な手順を示しています:

```
ze_result_t CalculateMetricsExample( zet_metric_group_handle_t hMetricGroup,
                                     size_t rawSize, uint8_t* rawData )
{
    // メトリックデータを計算
    uint32_t numMetricValues = 0;
    zet_metric_group_calculation_type_t calculationType =
ZET_METRIC_GROUP_CALCULATION_TYPE_METRIC_VALUES;
```

```
    zetMetricGroupCalculateMetricValues( hMetricGroup, calculationType, rawSize, rawData,
&numMetricValues, nullptr );
    zet_typed_value_t* metricValues = malloc( numMetricValues * sizeof(zet_typed_value_t) );
    zetMetricGroupCalculateMetricValues( hMetricGroup, calculationType, rawSize, rawData,
&numMetricValues, metricValues );

    // 特定のメトリックグループで利用可能なメトリックを取得
    uint32_t metricCount = 0;
    zetMetricGet( hMetricGroup, &metricCount, nullptr );

    zet_metric_handle_t* phMetrics = malloc(metricCount * sizeof(zet_metric_handle_t));
    zetMetricGet( hMetricGroup, &metricCount, phMetrics );

    // 結果メトリックをプリント
    uint32_t numReports = numMetricValues / metricCount;
    for( uint32_t report = 0; report < numReports; ++report )
    {
        printf("Report: %dn", report);

        for( uint32_t metric = 0; metric < metricCount; ++metric )
        {
            zet_typed_value_t data = metricValues[report * metricCount + metric];

            zet_metric_properties_t metricProperties {};
            metricProperties.stype = ZET_STRUCTURE_TYPE_METRIC_PROPERTIES;
            zetMetricGetProperties( phMetrics[ metric ], &metricProperties );

            printf("Metric: %sn", metricProperties.name );

            switch( data.type )
            {
            case ZET_VALUE_TYPE_UINT32:
                printf(" Value: %lun", data.value.ui32 );
                break;
            case ZET_VALUE_TYPE_UINT64:
                printf(" Value: %llun", data.value.ui64 );
                break;
            case ZET_VALUE_TYPE_FLOAT32:
                printf(" Value: %fn", data.value.fp32 );
                break;
            case ZET_VALUE_TYPE_FLOAT64:
```

```
            printf(" Value: %fn", data.value.fp64 );
            break;
        case ZET_VALUE_TYPE_BOOL8:
            if( data.value.ui32 )
                printf(" Value: truen" );
            else
                printf(" Value: falsen" );
            break;
        default:
            break;
        };
    }
  }

  free(metricValues);
  free(phMetrics);
}
```

プログラムのインストルメント

はじめに

プログラム・インストルメント API は、プログラムの直接インストルメントを可能にすることで、デバイスカーネルの低レベル・プロファイルの基本フレームワークをツールに提供します。これらの機能は、既に提供されている機能やカスタム・ローダー・レイヤーと組み合わせることで、より高度なフレームワークを開発できます。

次の 2 つのインストルメント機能があります:

1. 関数間のインストルメント - 関数呼び出しのインターセプトとリダイレクト
2. 関数内のインストルメント - 関数内に新しい命令を挿入

関数間のインストルメント

次の機能により、ツールは関数呼び出しをインターセプトしてリダイレクトできます:

- モジュール間の関数呼び出し - 異なるモジュール間で関数を呼び出す機能（例:アプリケーションのモジュールとツールのモジュール）
- カスタム・ローダー・レイヤー - API 呼び出しをインターセプトして挿入する機能

たとえば、ツールは次の方法でカスタム・ローダー・レイヤーを使用する場合があります:

- `zeModuleCreate` - すべての関数モジュールハンドルをインストルメントされたモジュールハンドルに置き換えます
- `zeKernelCreate` - すべての呼び出しサイトでカーネルハンドルをインストルメントされたカーネルハン

ドルに置き換えます

- zeModuleGetFunctionPointer - すべての呼び出しサイトで関数ポインターをインストルメントされた関数ポインターに置き換えます
- zeCommandListAppendLaunchKernel - 呼び出しサイトでカーネルハンドルをインストルメントされたカーネルハンドルに置き換えます

関数内のインストルメント

次の機能により、ツールはカーネル内に命令を挿入できるようになります:

- zetModuleGetDebugInfo - ツールがアプリケーション・モジュールの標準的なデバッグ情報を照会できるようにします
- zetKernelGetProfileInfo - ツールがカーネルのさまざまな面に関する詳細情報を照会できるようにします
- zeModuleGetNativeBinary - ツールがアプリケーションのモジュールのネイティブバイナリーを取得してインストルメントし、インストルメントされたバージョンを使用して新しいモジュールを作成できるようにします
- カスタム・ローダー・レイヤー - 上記の関数間のインストルメントと同じ使用法

コンパイル

モジュールは、コンパイラーが適切なプロファイル・メタデータを生成するため、インストルメントが実行されることを事前に認識した上でコンパイルする必要があります。インストルメント・レイヤーが有効になっている場合、新しいビルドフラグがサポートされます: “-zet-profile-flags <value>”、ここで <value> は 16 進数の zet_profile_flags_t の組み合わせである必要があります。

たとえば、ツールはカスタム・ローダー・レイヤーを使用して、ツールがインストルメントする各 zeModuleCreate 呼び出しにこのビルドフラグを挿入できます。別の例として、ツールはビルドフラグを使用してモジュールを再コンパイルし、カスタム・ローダー・レイヤーを使用してアプリケーションのモジュールハンドルを独自のものに置き換えることができます。

インストルメント

インストルメントを有効にしてモジュールをコンパイルすると、ツールは zetModuleGetDebugInfo と zetKernelGetProfileInfo を使用してアプリケーションの命令をデコードし、モジュール内の各関数の状況を登録できるようになります。

ツールで追加機能を使用する場合、他のモジュールを作成し、zeModuleGetFunctionPointer を使用してアプリケーション・モジュールとツール・モジュール間の関数を呼び出すことができます。ツールは、zeModuleGetFunctionPointer を使用して、モジュール内の各関数のホストとデバイスのアドレスを取得できます。

実際の計測用に API は提供されていません。代わりに、アプリケーション・モジュールのネイティブバイナリーをデコードし、ネイティブ命令を挿入する役割はツールに任されています。このモデルは、インストルメントがコンパイ

ラーによって操作されるのを防ぎます。

実行

ツールでアプリケーションの関数アドレスを変更する場合、関数ポインターを処理する API 呼び出しをインターセプトするためカスタム・ローダー・レイヤーを使用する必要があります。たとえば、`zeModuleGetFunctionPointer` および `zeCommandListAppendLaunchKernel` のすべてのフレーバーです。

プログラムのデバッグ

はじめに

プログラムデバッグ API は、デバイスコードをデバッグする基本的なフレームワークをツールに提供します。

デバッグ API は単一のデバイスでのみ動作します。マルチデバイス・システムをデバッグする場合、ツールは各デバイスを個別にデバッグする必要があります。

デバッグ API は、単一のホストプロセスのコンテキストでのみ動作します。複数のホストプロセスを同時にデバッグする場合、ツールは各ホストプロセスによって送信されたデバイスコードを個別にデバッグする必要があります。

デバイスのデバッグ・プロパティー

ツールは、`zetDeviceGetDebugProperties` を呼び出してデバイスのデバッグ・プロパティーを照会できます。

デバッグセッションを開始するには、ツールは接続先のデバイスのデバッグ・プロパティーを照会する必要があります。デバッガーのアタッチサポートは、`zet_device_debug_properties_t` の `ZET_DEVICE_DEBUG_PROPERTY_FLAG_ATTACH` フラグによって示されます。

```
zet_device_debug_properties_t props {};
props.stype = ZET_STRUCTURE_TYPE_DEVICE_DEBUG_PROPERTIES;
zetDeviceGetDebugProperties(hDevice, &props);


if (ZET_DEVICE_DEBUG_PROPERTY_FLAG_ATTACH & props.flags == 0)
   return; // デバッグはサポートされていません
```

アタッチとデタッチ

ツールは、`zetDebugAttach` を呼び出してデバイスに接続する必要があります。ライブラリーは次のプロパティーをチェックします:

- デバイスはデバッガーの接続をサポートしている必要があります。
- 要求されたホストプロセスが存在する必要があります。
- ツールプロセスは、要求されたホストプロセスをデバッグできるようにする必要があります。
- ツールをホストプロセスに接続する必要がないことに注意してください。
- 同時に他のツールを接続しないでください。
- このシステムではデバイスのデバッグを有効にする必要があります。

許可された場合、`zet_debug_session_handle_t` が提供されます。デバッグセッションのハンドルは、ツール

が再度デタッチされるまで、他のプログラムデバッグ API で使用できます。

デバッグセッションを終了するには、ツールは対応する zetDebugDetach 呼び出しで提供された zet_debug_session_handle_t を渡して zetDebugAttach を呼び出します。

次のサンプルコードは、アタッチとデタッチを示しています:

```
zet_debug_session_handle_t hDebug;

zet_debug_config_t config;
memset(&config, 0, sizeof(config));
config.pid = ...;

errcode = zetDebugAttach(hDevice, &config, &hDebug);
if (errcode)
    return errcode;

...

errcode = zetDebugDetach(hDebug);
if (errcode)
    return errcode;
```

デバイスとサブデバイス

ツールは任意のデバイスに接続でき、そのデバイスのすべてのサブデバイスに暗黙的に接続されます。

サブデバイスごとに個別のコードセグメントを使用する実装では、サブデバイスへの個別の接続も可能です。これは、サブデバイス・ハンドルを使用して zetDeviceGetDebugProperties を呼び出し、zet_device_debug_properties_t の ZET_DEVICE_DEBUG_PROPERTY_FLAG_ATTACH フラグで確認できます。その場合、ツールはデバイスに接続するか、1 つ以上のサブデバイスに接続するか選択できます。

サブデバイスに接続されている場合、コードセグメントへの書き込みは、同じアドレス空間範囲を共有する場合でも、他のサブデバイスにブロードキャストされることはありません。これにより、ブレークポイントを 1 つのサブデバイス内に設定できます。

ツールがサブデバイスに接続されている時に親デバイスに接続しようとすると、ZE_RESULT_ERROR_NOT_AVAILABLE が返されます。

サブデバイス間でコードセグメントを共有する実装では、デバイスへの接続のみが許されます。サブデバイスに接続しようとすると、ZE_RESULT_ERROR_NOT_AVAILABLE が返されます。

デバイススレッドの識別

デバイススレッドは、スライス、サブスライス、EU、およびスレッド番号によって識別されます。これらの番号は、0 から ze_device_properties_t によって報告されたそれぞれの番号から 1 を引いた値の範囲になります。

ツールがデバイスにアタッチされている場合、そのデバイス内のすべてのサブデバイスについて、デバイスのスレッ

ドが列挙されます。

デバイス上のスレッドの合計数は、次のサンプルコードに示すように、デバイス・プロパティーを使用して計算できます:

```
ze_device_properties_t properties {};
properties.stype = ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;
uint64_t num_threads;

zeDeviceGetProperties(hDevice, &properties);

num_threads = properties.numSlices * properties.numSubslicesPerSlice *
   properties.numEUsPerSubslice * properties.numThreadsPerEU;
```

ツールは、スライス、サブスライス、EU、およびスレッド数を反復処理することにより、デバイスのプロパティーに基づいてすべての利用可能なスレッド識別子を列挙できます。

スレッドの可用性

すべてのスレッドが常に利用できるわけではなく、一部のスレッドは利用できない場合があります。これには次のような理由が考えられます:

- スレッドがアイドル状態になっている可能性があります
- スレッドが別のプロセスに割り当てられる可能性があります

このデバッグツール API の目的上、スレッドは次の 3 つの状態のいずれかになります:

- 実行中
- 停止中
- 利用不可

デバッグイベント

デバッグセッションが開始されるとすぐに、デバイスからデバッグイベントを受信します。FIFO の最上位のイベントを読み取るには、ツールは `zetDebugReadEvent` を呼び出す必要があります。

次のサンプルコードは、イベントの読み取りを示しています:

```
zet_debug_event_t event;
errcode = zetDebugReadEvent(hDebug, UINT64_MAX, &event);
if (errcode)
   return errcode;

...

if (event.flags & ZET_DEBUG_EVENT_FLAG_NEED_ACK) {
   errcode = zetDebugAcknowledgeEvent(hDebug, &event);
```

```
    if (errcode)
        return errcode;
}
```

デバッグイベントは、zet_debug_event_t 構造体によって記述され、次のものが含まれます:

- イベントタイプは zet_debug_event_type_t です。
- zet_debug_event_flags_t のビットベクトル。次のいずれかになります:
    - ZET_DEBUG_EVENT_FLAG_NEED_ACK は、zetDebugAcknowledgeEvent を呼び出してイベントを確認する必要があることを示します。これにより、ツールはイベントに応じて任意のアクションを実行し、イベントを確認することで完了を示すことができます。

      実装では、前のイベントが確認されるまで、新しいイベントの読み取りをブロックできます。実装がさらにイベントの読み取りを許可する場合、順不同でのイベントの承認を許可する必要があります。

共通フィールドの後に、イベント・オブジェクトには、イベントのタイプに応じてイベント固有のフィールドが含まれます。すべてのイベントにイベント固有のフィールドがあるわけではありません。

- ZET_DEBUG_EVENT_TYPE_DETACHED: ツールがデタッチされました。
    - デタッチ理由（zet_debug_detach_reason_t）は、以下のいずれかの可能性があります:
        - ZET_DEBUG_DETACH_REASON_HOST_EXIT は、ホストプロセスが終了したことを示します。
- ZET_DEBUG_EVENT_TYPE_PROCESS_ENTRY: ホストプロセスによって、デバイス上に 1 つ以上のコマンドキューが作成されました。
- ZET_DEBUG_EVENT_TYPE_PROCESS_EXIT: ホストプロセスはデバイス上のすべてのコマンドキューを破棄しました。
- ZET_DEBUG_EVENT_TYPE_MODULE_LOAD: メモリー内のモジュールがデバイスにロードされました。
- ZET_DEBUG_EVENT_TYPE_MODULE_UNLOAD: メモリー内モジュールがデバイスからアンロードされようとしています。
- ZET_DEBUG_EVENT_TYPE_THREAD_STOPPED: デバイス例外によりスレッドが停止しました。

  報告されたスレッドは、zetDebugResume 呼び出しによって再開されるまで停止したままになります。

- ZET_DEBUG_EVENT_TYPE_THREAD_UNAVAILABLE: スレッドは利用できないため中断できません。

  要求されたスレッドのいずれも割り込み可能でない場合、割り込み要求に応答してイベントが生成されます。

- ZET_DEBUG_EVENT_TYPE_PAGE_FAULT: デバイスでページフォールトが発生しました。

  イベントは、ページフォールトの理由、ページサイズに合わせたフォールトアドレス、およびアライメントを指定するマスクを提供します。

  マスクは、アドレスとビット単位の AND 演算によって、指定されたフォールトアドレスと同じページサイズにアライメントされたアドレスを取得できます。

実行制御

ツールは、zetDebugInterrupt と zetDebugResume をそれぞれ呼び出すことで、デバイススレッドを中断および再開できます。

スレッド引数では、単一のスレッド、スレッドのグループ、またはデバイス上のすべてのスレッドを指定できます。すべてのスレッドを指定するには、ツールは ze_device_thread_t 内のすべてのフィールドを最大値に設定する必要があります。すべてのフィールドではなく一部のフィールドを最大値に設定することで、スレッドグループを指定できます。

zetDebugInterrupt 呼び出しはブロックされません。指定されたすべてのスレッドが停止したか、現在使用できないと判断された場合、少なくとも 1 つのスレッドを停止できた場合、ツールは ZET_DEBUG_EVENT_TYPE_THREAD_STOPPED イベントを受け取り、現在使用できるスレッドがない場合は ZET_DEBUG_EVENT_TYPE_THREAD_UNAVAILABLE イベントを受け取ります。

スレッド引数でデバイス上のスレッドグループまたはすべてのスレッドが指定された場合、イベントの前にそれぞれのスレッドで ZET_DEBUG_EVENT_TYPE_THREAD_STOPPED イベントが発生する可能性があります。

ツールは、スレッドの状態にアクセスしたり、スレッドを介してメモリーにアクセスするまで、スレッドが停止している、または使用できないことを認識しません。利用できないスレッドはいつでも利用可能になる可能性があります。

次のサンプルコードは、デバッグセッション内のすべてのスレッドを中断して再開する方法を示しています:

```
ze_device_thread_t allthreads;
allthreads.slice = UINT32_MAX;
allthreads.subslice = UINT32_MAX;
allthreads.eu = UINT32_MAX;
allthreads.thread = UINT32_MAX;

errcode = zetDebugInterrupt(hDebug, allthreads);
if (errcode)
   return errcode;

...

errcode = zetDebugResume(hDebug, allthreads);
if (errcode)
   return errcode;
```

メモリーアクセス

ツールは、停止したデバイススレッドのコンテキストで、そのスレッドがメモリーの読み取りまたは書き込みを行ったかのように、メモリーの読み取りと書き込みを行うことがあります。

メモリーはデバイス固有のメモリー空間に分割される場合があります。たとえば、GPU デバイスは、zet_debug_memory_space_type_t によって定義される次のメモリー空間をサポートします:

- `ZET_DEBUG_MEMORY_SPACE_TYPE_DEFAULT` - デフォルトのメモリー空間
- `ZET_DEBUG_MEMORY_SPACE_TYPE_SLM` - 共有ローカルメモリー空間

すべてのフィールドが最大値に設定された特別なスレッドのコンテキストで、デフォルトのメモリー空間にアクセスすることもできます。

メモリーの読み取りと書き込みを行うには、それぞれ `zetDebugReadMemory` 関数と `zetDebugWriteMemory` 関数を呼び出します。関数はそれぞれ、スレッド、メモリー空間、および入力/出力バッファーを指定します。

次の例では、1 つのデバイススレッドのコンテキスト内の場所からデフォルトメモリー空間内の別の場所に 16 バイトのメモリーコピーを行います:

```
zet_debug_memory_space_desc_t srcSpace = {
    ZET_STRUCTURE_TYPE_DEBUG_MEMORY_SPACE_DESC,
    nullptr,
    ZET_DEBUG_MEMORY_SPACE_TYPE_DEFAULT,
    srcAddress
};
zet_debug_memory_space_desc_t dstSpace = {
    ZET_STRUCTURE_TYPE_DEBUG_MEMORY_SPACE_DESC,
    nullptr,
    ZET_DEBUG_MEMORY_SPACE_TYPE_DEFAULT,
    dstAddress
};

ze_device_thread_t thread0 = {
    0、0、0、0
};

uint8_t buffer[16];
errcode = zetDebugReadMemory(hDebug, thread0, &srcSpace, sizeof(buffer), buffer);
if (errcode)
    return errcode;

...

errcode = zetDebugWriteMemory(hDebug, allthreads, &dstSpace, sizeof(buffer), buffer);
if (errcode)
    return errcode;
```

レジスター状態アクセス

ツールは、停止されたデバイススレッドのレジスター状態を読み書きする場合があります。

レジスターは、類似したレジスターセットにグループ化されます。デバイスでサポートされているレジスターセットのタイプは、zetDebugGetRegisterSetProperties および zetDebugGetThreadRegisterSetProperties を使用して照会できます。前者は、デバイスでサポートされているレジスターセットに関する一般的な情報を提供し、後者は、引数スレッドの具体的なレジスターセットを提供します。レジスターセットは動的プロパティーに依存し、停止するたびに変更される可能性があります。レジスターセットのプロパティーは、各レジスターセット内のレジスター最大数や、レジスターセットが読み取り専用かどうかなど、各レジスターセットの詳細を指定します。

実際のレジスタータイプはデバイス固有であり、デバイスベンダーによって定義されます。

次の疑似コードは、デバイスのレジスター・セット・プロパティーを取得する方法を示しています:

```
uint32_t nRegSets = 0;
zetDebugGetRegisterSetProperties(hDevice, &nRegSets, nullptr);

zet_debug_regset_properties_t* pRegSets = allocate(nRegSets *
sizeof(zet_debug_regset_properties_t));
zetDebugGetRegisterSetProperties(hDevice, &nRegSets, pRegSets);
```

次の疑似コードは、スレッドのレジスター・セット・プロパティーを取得する方法を示しています:

```
ze_device_thread_t thread0 = {
    0、0、0、0
};
uint32_t nRegSets = 0;
zetDebugGetThreadRegisterSetProperties(hDebug, thread0, &nRegSets, nullptr);

zet_debug_regset_properties_t* pRegSets = allocate(nRegSets *
sizeof(zet_debug_regset_properties_t));
zetDebugGetThreadRegisterSetProperties(hDebug, thread0, &nRegSets, pRegSets);
```

レジスターの状態を読み書きするには、それぞれ zetDebugReadRegisters 関数と zetDebugWriteRegisters 関数を使用します。

以下の擬似コードは、レジスターセットの反復処理を示しています:

```
for (i = 0; i < nRegSets; ++i) {
    void* values = allocate(pRegSets[i].count * pRegSets[i].valueSize);

    errcode = zetDebugReadRegisters(hDebug, thread0, pRegSets[i].type, 0, pRegSets[i].count,
values);
    if (errcode)
        return errcode;

    ...
```

```
    errcode = zetDebugWriteRegisters(hDebug, thread0, pRegSets[i].type, 0,
pRegSets[i].count, values);
    if (errcode)
        return errcode;

    free(values);
}
```

Sysman プログラミング・ガイド

はじめに

Sysman は、アクセラレーター・デバイスの電力とパフォーマンスを監視および制御するシステムリソース管理ライブラリーです。

高レベルの概要

環境変数

システムリソース管理ライブラリーは、`zesInit` を呼び出すことによって環境変数を使用せずに初期化できるようになりました。

互換性のため、それぞれの機能の初期化中に次の環境変数も有効にできます。sysman を初期化する場合、`zesInit` を呼び出すか、次の環境変数を使用するいずれかを推奨しますが、両方の使用は推奨されません。

カテゴリー	名称	値	説明
Sysman	ZES_ENABLE_SYSMAN	{0, 1}	システム管理のためドライバーの初期化と依存関係を有効にします
Sysman	ZES_ENABLE_SYSMAN_LOW_POWER	{0, 1}	ドライバーはデバイスを低電力モードで初期化します

初期化

デバイスの電源とパフォーマンスを管理するアプリケーションは、システムリソース管理ライブラリーを使用して、システム管理ドライバーとデバイスハンドルを列挙できます。

以下の擬似コードは、基本的な初期化とデバイスの検出手順を示しています:

```
function main( ... )
    if (zesInit(0) != ZE_RESULT_SUCCESS)
        output("Can't initialize the API")
    else
        # すべてのドライバーを検出
        uint32_t driversCount = 0
        zesDriverGet(&driversCount, nullptr)
        zes_driver_handle_t* allDrivers = allocate(driversCount *
sizeof(zes_driver_handle_t))
        zesDriverGet(&driversCount, allDrivers)

        zes_driver_handle_t hDriver = nullptr
        for(i = 0 .. driversCount-1)
            # ドライバー内のデバイスを検出
```

```
        uint32_t deviceCount = 0
        zesDeviceGet(allDrivers[i], &deviceCount, nullptr)

        zes_device_handle_t* hSysmanHandles =
            allocate_memory(deviceCount * sizeof(zes_device_handle_t))
        zesDeviceGet(allDrivers[i], &deviceCount, hSysmanHandles)

        # hSymanHandles を使用してデバイスを管理

    free_memory(...)
```

互換性のため、アプリケーションは Level0 Core API を使用して、システム内の利用可能なアクセラレーター・デバイスを列挙することもできます。アプリケーションは、各デバイスハンドルを sysman デバイスハンドルにキャストして、デバイスのシステムリソースを管理できます。

各デバイスには固有のハンドルがあります。複数のスレッドがハンドルを使用できます。ハンドルを介して同じデバイス・プロパティーに同時アクセスすると、最後の要求が優先されます。

以下の疑似コードは、システム内の GPU デバイスを列挙し、それらの Sysman ハンドルを作成する方法を示しています:

```
function main( ... )
    if (zeInit(0) != ZE_RESULT_SUCCESS)
        output("Can't initialize the API")
    else
        # すべてのドライバーを検出
        uint32_t driversCount = 0
        zeDriverGet(&driversCount, nullptr)
        ze_driver_handle_t* allDrivers = allocate(driversCount * sizeof(ze_driver_handle_t))
        zeDriverGet(&driversCount, allDrivers)

        ze_driver_handle_t hDriver = nullptr
        for(i = 0 .. driversCount-1)
            # ドライバー内のデバイスを検出
            uint32_t deviceCount = 0
            zeDeviceGet(allDrivers[i], &deviceCount, nullptr)

            ze_device_handle_t* allDevices =
                allocate_memory(deviceCount * sizeof(ze_device_handle_t))
            zeDeviceGet(allDrivers[i], &deviceCount, allDevices)

            for(devIndex = 0 .. deviceCount-1)
                ze_device_properties_t device_properties {}
```

```
            device_properties.stype = ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES
            zeDeviceGetProperties(allDevices[devIndex], &device_properties)
            if(ZE_DEVICE_TYPE_GPU != device_properties.type)
                next
            # Sysman デバイスハンドルを取得
            zes_device_handle_t hSysmanDevice = (zes_device_handle_t)allDevices[devIndex]
            # hSysmanDevice を使用してデバイスを管理

    free_memory(...)
```

グローバルデバイスの管理

デバイス全体の情報にアクセスし、デバイス全体を制御するため、次の操作が提供されています:

- デバイスの UUID、デバイス ID、サブデバイス数を取得
- ブランド/モデル/ベンダー名を取得
- このデバイスを使用しているプロセスに関する情報を照会
- デバイスをリセット
- デバイスが修復されたかどうかを照会
- デバイスをリセットする必要があるかどうか、またその理由は何か（ロックされている、修復を開始している）を照会
- PCI 情報:
    - 設定されたバーを取得
    - サポートされている最大帯域幅を取得
    - 現在の速度（GEN/レーン数）を照会
    - 現在のスループットを照会
    - パケット再試行カウンターを照会

利用可能な機能の完全なリストを以下に示します。

デバイス・コンポーネント管理

デバイスのグローバル・プロパティー管理以外にも、デバイスのパフォーマンスや電源構成を変更するために管理できるデバイス・コンポーネントが多数あります。類似のコンポーネントは**クラス**に分割され、各クラスには実行できる一連の操作があります。

たとえば、デバイスには通常、 1 つ以上の周波数ドメインがあります。Sysman API は、周波数のクラスと、管理できるすべての周波数ドメインの列挙を示します。

以下の表は、デバイスクエリーを提供するクラスと、2 つのサブデバイスを持つデバイスに対して列挙されるコンポーネントのリストをまとめた例です。この表には、各クラスのすべてのコンポーネントに提供される操作（クエリー）が表示されます。

クラス	コンポーネント	操作
電力	カード：電力	エネルギー消費量を取得

クラス	コンポーネント	操作
	パッケージ：電力 サブデバイス 0：合計電力 サブデバイス 1：合計電力	
周波数	サブデバイス 0：GPU 周波数 サブデバイス 0：メモリー周波数 サブデバイス 1：GPU 周波数 サブデバイス 1：メモリー周波数	利用可能な周波数を一覧表示します 周波数レンジを設定 周波数を取得 スロットリングの原因を取得 スロットリング時間を取得
エンジン	サブデバイス 0：すべてのエンジン サブデバイス 0：計算エンジン サブデバイス 0：メディアエンジン サブデバイス 0：コピーエンジン サブデバイス 1：すべてのエンジン サブデバイス 1：計算エンジン サブデバイス 1：メディアエンジン サブデバイス 1：コピーエンジン	ビジー時間を取得
スケジューラー	サブデバイス 0：すべてのエンジン サブデバイス 1：すべてのエンジン	スケジューラー・モードとプロパティーを取得します スケジューラー・モードとプロパティーを取得します
ファームウェア	サブデバイス 0：各ファームウェアを列挙 サブデバイス 1：各ファームウェアを列挙	ファームウェアの名前とバージョンを取得
メモリー	サブデバイス 0：メモリーモジュール サブデバイス 1：メモリーモジュール	サポートされている最大帯域幅を取得 フリーメモリーを取得 現在の帯域幅を取得
ファブリック・ポート	サブデバイス 0：各ポートを列挙 サブデバイス 1：各ポートを列挙	ポート構成を取得 (UP/DOWN) 物理リンクの詳細を取得 ポートの状態を取得 (正常/劣化/障害/無効) リモートポートを取得 ポート rx/tx の速度を取得 ポート rx/tx の帯域幅を取得
温度	パッケージ：温度 (最小、最大) サブデバイス 0：GPU 温度 (最小、最大) サブデバイス 0：メモリー温度 (最小、最大) サブデバイス 1：GPU 温度 (最小、最大) サブデバイス 1：メモリー温度 (最小、最大)	現在の温度センサーの値を取得
PSU	パッケージ： 電源	電源の詳細を取得 現在の状態 (温度、電流、ファン) を照会
ファン	パッケージ：ファン	詳細を取得 (最大ファン速度)

クラス	コンポーネント	操作
		設定を取得 (固定ファン速度、温度速度テーブル) 現在のファン速度を照会
LED	パッケージ: LED	詳細を取得 (RGB 対応) 現在の状態 (オン、色) を照会
RAS	サブデバイス 0: 1 組の RAS エラーカウンター サブデバイス 1: 1 組の RAS エラーカウンター	RAS の訂正可能および訂正不可能なエラーカウンターの合計を読み取り エラーのカテゴリー別内訳を読み取ります (リセット数、プログラミング・エラー数、プログラミング・エラー数、ドライバーエラー数、計算エラー数、キャッシュエラー数、メモリーエラー数、PCI エラー数、ディスプレイ・エラー数、非計算エラー数)
診断	パッケージ: SCAN テストスイート パッケージ: ARRAY テストスイート	すべての診断テストのリストを取得

以下の表は、デバイス制御を提供するクラスと、2 つのサブデバイスを持つデバイスに対して列挙されるコンポーネントのリストをまとめた例です。この表には、各クラスのすべてのコンポーネントに提供される操作（制御）が表示されます。

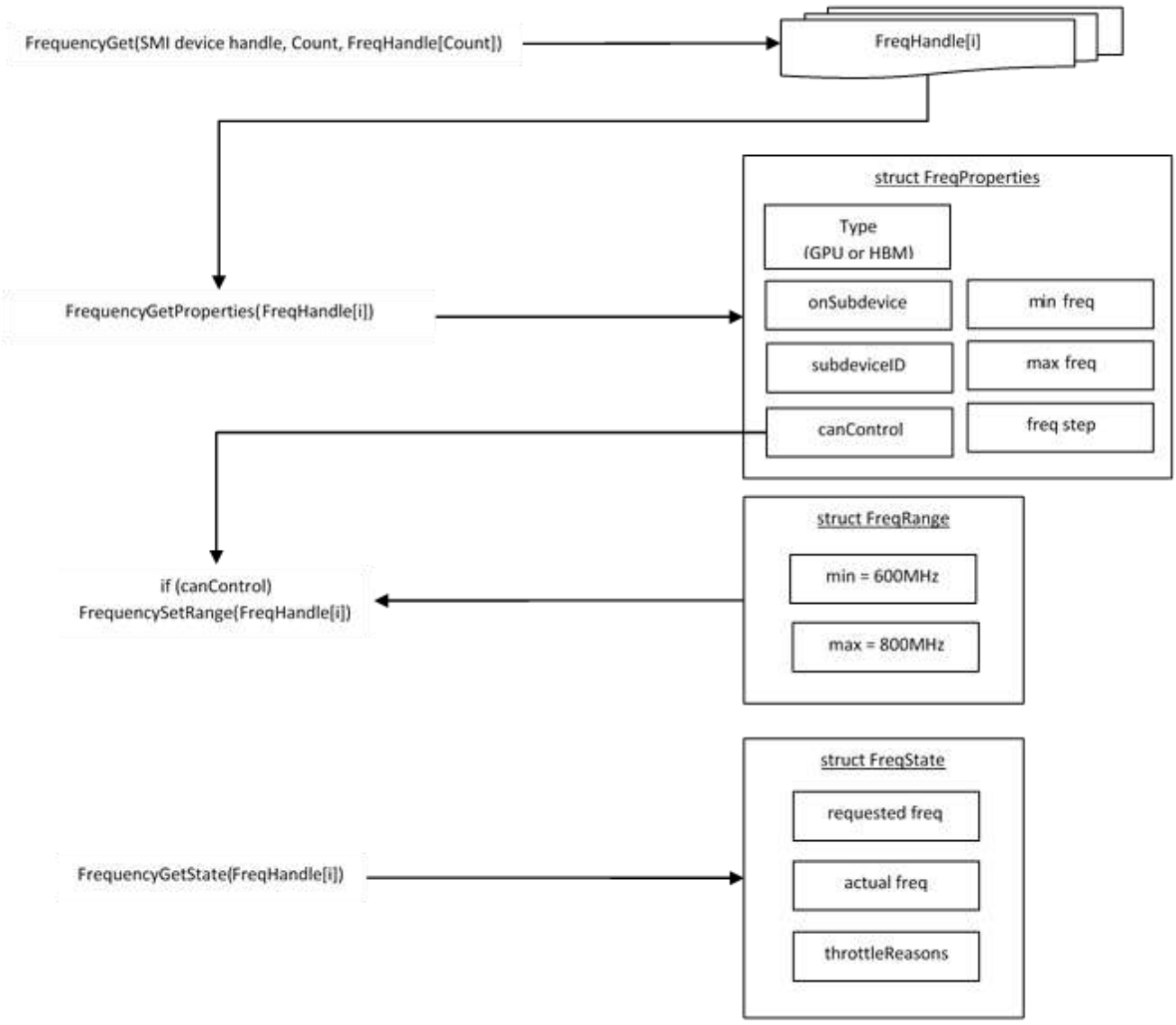
クラス	コンポーネント	操作
電力	カード: 電力 パッケージ: 電力	電力制限の持続を設定 電力制限のバーストを設定 電力制限のピークを設定
周波数	サブデバイス 0: GPU 周波数 サブデバイス 0: メモリー周波数 サブデバイス 1: GPU 周波数 サブデバイス 1: メモリー周波数	周波数レンジを設定
スケジューラー	サブデバイス 0: すべてのエンジン サブデバイス 1: すべてのエンジン	スケジューラー・モードを設定 スケジューラー・モードを設定
パフォーマンス係数	サブデバイス 0: 計算 サブデバイス 0: メディア サブデバイス 1: 計算 サブデバイス 1: メディア	ワークロードのパフォーマンスを調整
スタンバイ	サブデバイス 0: サブデバイス全体を制御 サブデバイス 1: サブデバイス全体を制御	オプチュニスティック・スタンバイを無効化
ファームウェア	サブデバイス 0: 各ファームウェアを列挙 サブデバイス 1: 各ファームウェアを列挙	新しいファームウェアをフラッシュ

クラス	コンポーネント	操作
ファブリック・ポート	サブデバイス 0：各ポートを制御 サブデバイス 1：各ポートを制御	ポート UP/DOWN を設定 ビーコンのオン/オフ
ファン	パッケージ：ファン	設定を取得（固定速度、温度速度テーブル）
LED	パッケージ：LED	LED のオン/オフと色の設定
診断	SCAN テストスイート ARRAY テストスイート	テストスイート内の診断テストのすべてまたはサブセットを実行します

デバイス・コンポーネント列挙子

Sysman API は、管理可能なクラス内のすべてのコンポーネントを列挙する関数を提供します。

たとえば、デバイスのさまざまな部分の周波数を制御するのに使用される周波数クラスがあります。ほとんどのデバイスでは、列挙子は 2 つのハンドルを提供します。1 つは GPU 周波数を制御するハンドルで、もう 1 つはデバイスのメモリー周波数を列挙するハンドルです。これを次の図に示します:



C API では、各クラスは一意のハンドルタイプに関連付けられています（例: `zes_freq_handle_t` は周波数コンポーネントを参照）。C++ API では、各クラスは C++ クラスです（例: `zes::SysmanFrequency` クラスのインスタンスは周波数コンポーネントを参照）。

以下の疑似コードは、Sysman API を使用してすべての GPU 周波数コンポーネントを列挙し、サポートされている場合はそれぞれを特定の周波数に固定する方法を示しています:

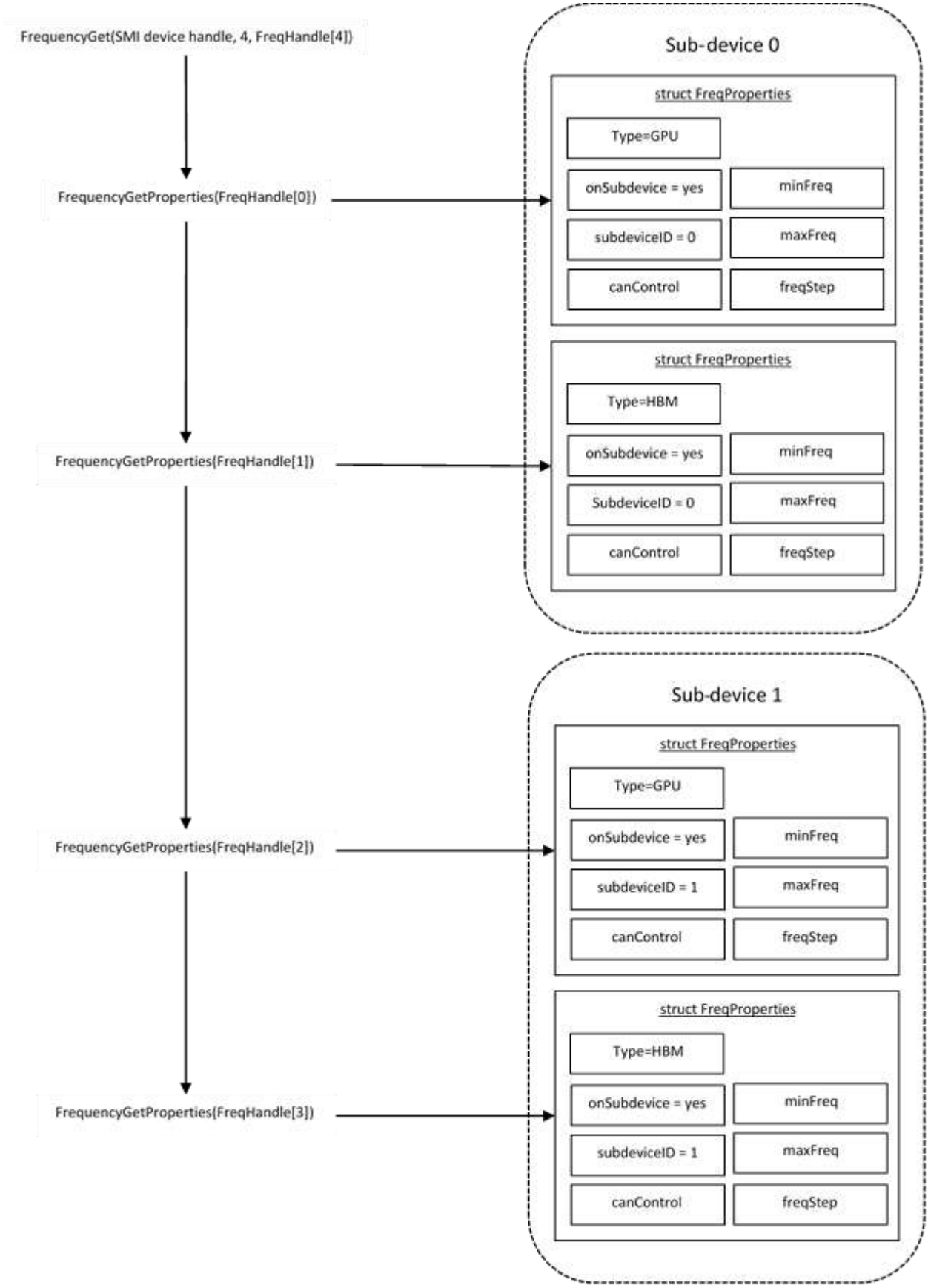
```
function FixGpuFrequency(zes_device_handle_t hSysmanDevice, double FreqMHz)
   uint32_t numFreqDomains
   if ((zesDeviceEnumFrequencyDomains(hSysmanDevice, &numFreqDomains, NULL) ==
ZE_RESULT_SUCCESS))
       zes_freq_handle_t* pFreqHandles =
          allocate_memory(numFreqDomains * sizeof(zes_freq_handle_t))
       if (zesDeviceEnumFrequencyDomains(hSysmanDevice, &numFreqDomains, pFreqHandles) ==
ZE_RESULT_SUCCESS)
          for (index = 0 .. numFreqDomains-1)
             zes_freq_properties_t props {};
             props.stype = ZES_STRUCTURE_TYPE_FREQ_PROPERTIES;
             if (zesFrequencyGetProperties(pFreqHandles[index], &props) ==
ZE_RESULT_SUCCESS)
                # ドメインの周波数を変更するのは、以下の条件を満たす場合のみです:
                # 1.このドメインは GPU アクセラレーターを制御
                # 2.ドメインの周波数は変更可能
                if (props.type == ZES_FREQ_DOMAIN_GPU
                   and props.canControl)
                      # 周波数を修正
                      zes_freq_range_t range
                      range.min = FreqMHz
                      range.max = FreqMHz
                      zesFrequencySetRange(pFreqHandles[index], &range)
   free_memory(...)
```

サブデバイス管理

Sysman デバイスハンドルはデバイスレベルで動作します。サブデバイスのデバイスハンドルが Sysman 関数のいずれかに渡されると、デバイスハンドルが使用されたような結果になります。

デバイス・コンポーネントの列挙子は、各サブデバイスに配置されているコンポーネントのリストを返します。各コンポーネントのプロパティーは、それがどのサブデバイスに配置されているか示します。1 つのサブデバイス内のコンポーネントのみを管理したいソフトウェアは、サブデバイス ID を使用して列挙されたコンポーネントをフィルターする必要があります（`ze_device_properties_t.subdeviceId` を参照）。

下の図は、2 つのサブデバイスを持つデバイス上で列挙される周波数コンポーネントを示しています。各サブデバイスには GPU とデバイスメモリーの周波数制御があります:



以下の疑似コードは、特定のサブデバイスの GPU 周波数を修正する方法を示しています（追加のサブデバイス・チェックに注意してください）:

```
function FixSubdeviceGpuFrequency(zes_device_handle_t hSysmanDevice, uint32_t subdeviceId,
double FreqMHz)
```

```
   uint32_t numFreqDomains
   if ((zesDeviceEnumFrequencyDomains(hSysmanDevice, &numFreqDomains, NULL) ==
ZE_RESULT_SUCCESS))
       zes_freq_handle_t* pFreqHandles =
          allocate_memory(numFreqDomains * sizeof(zes_freq_handle_t))
       if (zesDeviceEnumFrequencyDomains(hSysmanDevice, &numFreqDomains, pFreqHandles) ==
ZE_RESULT_SUCCESS)
          for (index = 0 .. numFreqDomains-1)
             zes_freq_properties_t props {};
             props.stype = ZES_STRUCTURE_TYPE_FREQ_PROPERTIES;
             if (zesFrequencyGetProperties(pFreqHandles[index], &props) ==
ZE_RESULT_SUCCESS)
                # ドメインの周波数を変更するのは、以下の条件を満たす場合のみです：
                # 1.このドメインは GPU アクセラレーターを制御
                # 2.ドメインの周波数は変更可能
                # 3.ドメインは指定されたサブデバイスにあります
                if (props.type == ZES_FREQ_DOMAIN_GPU
                   and props.canControl
                   and props.subdeviceId == subdeviceId)
                      # 周波数を修正
                      zes_freq_range_t range
                      range.min = FreqMHz
                      range.max = FreqMHz
                      zesFrequencySetRange(pFreqHandles[index], &range)
   free_memory(...)
```

イベント

イベントは、Sysman API をポーリングせずに、デバイス上で何らかの変更が発生したかどうかを判断する手段です（例：新しい RAS エラーの発生など）。アプリケーションは、通知を受け取りたいイベントを登録し、その後通知を待機します。アプリケーションは、通知の待機時にブロックすることもできます。これにより、新しい通知が受信されるまで、呼び出し元のアプリケーション・スレッドがスリープ状態になります。

API を使用すると、1 つの関数呼び出しで複数のデバイスからのイベントを登録し、任意のデバイスから送信されるイベント通知を待機できます。

通知が発生すると、アプリケーションはクエリー Sysman インターフェイス関数を使用して詳細を取得できます。

以下のイベントが提供されます:

- RAS エラーが発生しました

イベントを処理に使用できる関数の完全なリストを以下に示します。

テレメトリーとタイムスタンプ

多くの API 呼び出しは、基盤となるハードウェアのテレメトリー（カウンター）を返します。これらのカウンターは通常は単調であり、ビット幅の境界でラップアラウンドします。アプリケーションでは通常、 2 つのサンプル間の差分を取得する必要があります。多くの場合、カウンターの変化率が必要になります。たとえば、リンクを介して送信されたバイトのカウンターをサンプリングし、サンプル間のデルタ時間で割ると、平均帯域幅が得られます。

テレメトリーを返す場合、API には、基盤となるハードウェア・カウンターがサンプリングされたときのタイムスタンプが含まれます。各タイムスタンプは、それに関係するテレメトリーにのみ関連付けられます。テレメトリーに関連付けられた各タイムスタンプは、独自の絶対ベースを持つことができ、他のテレメトリーで返されるタイムスタンプとは異なることがあります。したがって、異なるテレメトリーから返されたタイムスタンプを基に計算を行うべきではありません。

タイムスタンプはアプリケーション間で同じベースになる保証はありません。これらは単一のアプリケーションの実行でのみ使用する必要があります。

インターフェイスの詳細

グローバル操作

デバイスのプロパティー

次の操作により、デバイス全体のプロパティーを取得できます:

関数	説明
zesDeviceGetProperties()	静的デバイス・プロパティーを取得します - デバイス UUID、サブデバイス ID、デバイスのブランド/モデル/ベンダー文字列
zesDeviceGetState()	デバイスの状態を確認する: デバイスは修復されたか、デバイスをリセットする必要があるか、その理由は何か（ウェッジされている、修復を開始している）

以下の疑似コードは、デバイスに関する一般情報を表示する方法を示しています:

```
function ShowDeviceInfo(zes_device_handle_t hSysmanDevice)
    zes_device_properties_t devProps {}
    devProps.stype = ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES
    zes_device_state_t devState
    if (zesDeviceGetProperties(hSysmanDevice, &devProps) == ZE_RESULT_SUCCESS)
        output("    UUID:          %s", devProps.core.uuid.id)
        output("    #subdevices:    %u", devProps.numSubdevices)
        output("    brand:         %s", devProps.brandName)
        output("    model:         %s", devProps.modelName)
    if (zesDeviceGetState(hSysmanDevice, &devState) == ZE_RESULT_SUCCESS)
        output("    Was repaired:   %s", (devState.repaired == ZES_REPAIR_STATUS_PERFORMED) ?
"yes" : "no")
        if (devState.reset != 0)
```

```
    {
        output("DEVICE RESET REQUIRED:")
        if (devState.reset & ZES_RESET_REASON_FLAG_WEDGED)
            output("- Hardware is wedged")
        if (devState.reset & ZES_RESET_REASON_FLAG_REPAIR)
            output("- Hardware needs to complete repairs")
    }
}
```

ホストプロセス

次の関数は、デバイスを使用しているホストプロセスに関する情報を提供します:

関数	説明
zesDeviceProcessesGetState()	このデバイスを使用しているすべてのプロセスに関する情報（プロセス Id、デバイスのメモリー割り当てサイズ、使用されているアクセラレーター）を取得します。

プロセス ID を使用すると、アプリケーションは所有者と実行可能ファイルへのパスを判別できます。この情報は API によって返されません。

デバイスのリセット

デバイスは以下の関数でリセットできます:

関数	説明
zesDeviceReset()	ドライバーがデバイスの PCI バスリセットを実行するように要求します。

PCI リンク操作

次の関数を使用すると、デバイスの PCI エンドポイントに関するデータを取得できます:

関数	説明
zesDevicePciGetProperties()	PCI ポートの静的プロパティーを取得します - BDF アドレス、バーの数、最大サポート速度
zesDevicePciGetState()	現在の PCI ポート速度（レーン数、世代）を取得
zesDevicePciGetBars()	構成された各 PCI バーに関する情報を取得
zesDevicePciGetStats()	PCI 統計情報（スループット、パケット合計、パケットリプレイ数）を取得

以下の疑似コードは、PCI BDF アドレスを出力する方法を示しています:

```
function ShowPciInfo(zes_device_handle_t hSysmanDevice)
    zes_pci_properties_t pciProps {};
    pciProps.stype = ZES_STRUCTURE_TYPE_PCI_PROPERTIES;
    if (zesDevicePciGetProperties(hSysmanDevice, &pciProps) == ZE_RESULT_SUCCESS)
```

```
    output("    PCI address:       %04u:%02u:%02u.%u",
        pciProps.address.domain,
        pciProps.address.bus,
        pciProps.address.device,
        pciProps.address.function);
```

電源ドメインの操作

PSU（電源ユニット）はデバイスに電力を供給します。デバイスが消費する電力量は電圧と周波数の関数であり、どちらもデバイス上のマイクロコントローラーである Punit によって制御されます。電圧と周波数が高すぎる場合、次の 2 つの状態が発生する可能性があります:

1. 過電流 - デバイスが消費する電流が PSU が供給できる最大電流を超えた状態です。これが起こると PSU は信号をアサートし、それが Punit によって処理されます。
2. 過熱 - デバイスが過剰な熱を発生しており、放散速度が十分ではありません。Punit は温度を監視し、センサーが最高温度がしきい値 TjMax（通常は 100 ℃）を超えると反応します。

これらのいずれかの状況が発生すると、Punit はデバイスの周波数/電圧を最小値まで抑制するため、パフォーマンスに重大な影響を与えます。Punit は電力制限を課すことでこのような深刻なスロットリングを回避します。電力制限には次の 2 種類があります:

1. リアクティブ - この場合、Punit は実際の電力（ハードウェア測定）の一定間隔にわたる移動平均を測定します。平均電力が制限を超えると、Punit は各周波数領域に対して要求できる最大周波数制限を徐々に減らし始めます。逆に、平均電力が制限を下回っている場合、Punit は各ドメインのハードウェア周波数制限まで、要求できる最大周波数制限を徐々に増やします。ユーザー/ドライバーの周波数要求が最大周波数制限を超えると、スロットリングが発生し、通常は電力が削減されます。
2. プロアクティブ - この場合、Punit はチップの現在の構成と周波数要求に基づいて計算を行い、消費される可能性のある最悪の電力を予測できます。この計算がプロアクティブ制限を超える場合、制限内に収まる最大周波数を見つける検索が行われます。

デバイスのハードウェア・スコープ・レベルで制限を適用する必要はありません。デバイスは 1 つ以上の電源ドメインに分割されます。電源ドメインは、電力消費を監視および制御できるハードウェア・スコープです。電源ドメインは、次のようなハードウェア・スコープに存在する可能性があります:

1. カードレベル - このレベルで定義された電源ドメインは、カード全体の電力消費を監視および制御します。
2. パッケージレベル - このレベルで定義された電源ドメインは、カード上の単一の物理パッケージ全体の電力消費を監視および制御します。
3. スタックレベル - このレベルで定義された電源ドメインは、パッケージ内の単一スタック全体の電力消費を監視および制御します。

特定の時点で、プラットフォームは主電源で稼働していることもあれば、ラップトップなどのプラットフォームはバッテリー電源で稼働していることもあります。これは、電力ソースと呼ばれます。制限は、デバイスが指定された電源

から電力を供給されている場合にのみ有効になるように設定できます。つまり、デバイスが主電源ではなくバッテリー電源で動作しているときは別の制限を設定できます。

プラットフォームと電源ドメインに応じて、電力制限はアンペア数またはワット数で表すことができます。API を照会して、特定の電力制限をどの単位で指定する必要があるか判断できます。

電力制限は、次のいずれかの電力レベルに対応します。

制限	ウィンドウ	説明
即時	NA	Punit は、現在の周波数要求に対する最悪の電力ケースを予測し、制限を超えると実際の周波数が低下します。
ピーク	例: 100 マイクロ秒	Punit は、短期間にわたる電力の移動平均を追跡します。プログラム可能なしきい値を超えると、Punit は周波数/電圧の調整を開始します。
バースト	例: 2 ミリ秒	Punit は、中期間にわたる電力の移動平均を追跡します。プログラム可能なしきい値を超えると、Punit は周波数/電圧の調整を開始します。
持続的	例: 28 秒	Punit は、長期間にわたる電力の移動平均を追跡します。プログラム可能なしきい値を超えると、Punit は周波数/電圧を制限します。

持続、バースト、およびピーク電力制限はリアクティブである一方、瞬間電力制限はプロアクティブであることに注意してください。

工場出荷時のデフォルト値は、デバイスが通常の温度で動作し、大きなワークロードが実行されていることを想定して調整されています:

- ピーク電力制限は、最も集中的な計算ワークロードを除くすべての PSU 過電流信号のトリップを回避するように調整されています。ほとんどのワークロードは、この条件に達することなく最大周波数で実行できるはずです。
- バースト電力制限により、ほとんどのワークロードを短時間最大周波数で実行できるようになります。
- 長時間にわたって高周波数が要求された場合（設定可能、デフォルトは 28 秒）、持続的な電力制限がトリガーされ、高周波数の要求とデバイスの使用率が継続すると、周波数が調整されます。

一部の電源ドメインでは、エネルギー消費量が特定の値を超えたときにイベント `ZES_EVENT_TYPE_FLAG_ENERGY_THRESHOLD_CROSSED` を生成する要求をサポートしています。これは、GPU がビジー状態になるまでアプリケーションを一時停止するのに便利な方法です。この手法では、何らかのデルタエネルギーしきい値で `zesPowerSetEnergyThreshold()` を呼び出し、関数 `zesDeviceEventRegister()` を使用してイベントを受信するように登録し、次に `zesDriverEventListen()` を呼び出してイベントがトリガーされるまでブロックします。呼び出しが行われてから電源ドメインで消費されたエネルギーが指定されたデルタを超えると、イベントがトリガーされ、アプリケーションが起動されます。

デバイスには複数の電源ドメインが存在する場合があります:

- PCIe カード全体で消費される電力を処理する 1 つのカードレベルの電源ドメイン。

- アクセラレーター・チップ全体で消費される電力を処理する 1 つのパッケージレベルの電源ドメイン。これには、チップ上のすべてのサブデバイスの電力が含まれます。
- 製品にサブデバイスがある場合、サブデバイスごとに 1 つ以上の電源ドメイン。

デバイスの電源を管理するため次の関数が提供されています:

関数	説明
zesDeviceEnumPowerDomains()	電源ドメインを列挙します。
zesPowerGetProperties()	特定の電源ドメインの電力制限を変更するときに指定できる最小/最大電力制限を取得します。また、部品の工場出荷時の持続電力制限も読み取ってください。
zesPowerGetEnergyCounter()	特定のドメインのエネルギー消費量を読み取ります。
zesPowerGetLimitsExt()	特定の電源ドメインのすべての電力制限を取得します。
zesPowerSetLimitsExt()	特定の電源ドメインのすべての電力制限を設定します。
zesPowerGetEnergyThreshold()	現在のエネルギーしきい値を取得します。
zesPowerSetEnergyThreshold()	エネルギーしきい値を設定します。イベント ZES_EVENT_TYPE_FLAG_ENERGY_THRESHOLD_CROSSED この関数を呼び出して、それ以降に消費されたエネルギーが指定されたしきい値を超えたときに生成されます。

以下の疑似コードは、デバイス上の各電源ドメインに関する情報を出力する方法を示しています:

```
function ShowPowerDomains(zes_device_handle_t hSysmanDevice)
   uint32_t numPowerDomains
   if (zesDeviceEnumPowerDomains(hSysmanDevice, &numPowerDomains, NULL) ==
ZE_RESULT_SUCCESS)
      zes_pwr_handle_t* phPower =
         allocate_memory(numPowerDomains * sizeof(zes_pwr_handle_t))
      if (zesDeviceEnumPowerDomains(hSysmanDevice, &numPowerDomains, phPower) ==
ZE_RESULT_SUCCESS)
         for (pwrIndex = 0 .. numPowerDomains-1)
            zes_power_properties_t props {};
            props.stype = ZES_STRUCTURE_TYPE_POWER_PROPERTIES;
            if (zesPowerGetProperties(phPower[pwrIndex], &props) == ZE_RESULT_SUCCESS)
               if (props.onSubdevice)
                  output("Sub-device %u power:n", props.subdeviceId)
                  output("    Can control: %s", props.canControl ? "yes" : "no")
                  call_function ShowPowerLimits(phPower[pwrIndex])
               else
                  output("Total package power:n")
```

```
                    output("    Can control: %s", props.canControl ? "yes" : "no")
                    call_function ShowPowerLimits(phPower[pwrIndex])
    free_memory(...)
}

function ShowPowerLimits(zes_pwr_handle_t hPower)
    uint32_t limitCount = 0
    if (zesPowerGetLimitsExt(hPower, &limitCount, nullptr) == ZE_RESULT_SUCCESS)
        zes_power_limit_ext_desc_t * allLimits = allocate(limitCount *
sizeof(zes_power_limit_ext_desc_t));
        if (zesPowerGetLimitsExt(hPower, &numLimits, allLimits) == ZE_RESULT_SUCCESS)

            for (i = 0; i < limitCount; ++i)
                output("Limit is enabled: %s", enabled)
                output("Power averaging window: %d", interval)
```

以下の疑似コードは、デバイス上で最初に見つかった電源ドメインの持続電力制限を変更する方法を示しています:

```
function SetPowerDomainLimit(zes_device_handle_t hSysmanDevice)
    uint32_t numPowerDomains
    if (zesDeviceEnumPowerDomains(hSysmanDevice, &numPowerDomains, NULL) ==
ZE_RESULT_SUCCESS)
        zes_pwr_handle_t* phPower =
            allocate_memory(numPowerDomains * sizeof(zes_pwr_handle_t))
        if (zesDeviceEnumPowerDomains(hSysmanDevice, &numPowerDomains, phPower) ==
ZE_RESULT_SUCCESS)
            for (pwrIndex = 0 .. numPowerDomains-1)
                zes_power_properties_t props {};
                props.stype = ZES_STRUCTURE_TYPE_POWER_PROPERTIES;
                if (zesPowerGetProperties(phPower[pwrIndex], &props) == ZE_RESULT_SUCCESS)
                    uint32_t limitCount = 0
                    if (zesPowerGetLimitsExt(hPower, &limitCount, nullptr) ==
ZE_RESULT_SUCCESS)
                        zes_power_limit_ext_desc_t * allLimits = allocate(limitCount *
sizeof(zes_power_limit_ext_desc_t));
                        if (zesPowerGetLimitsExt(hPower, &numLimits, allLimits) ==
ZE_RESULT_SUCCESS)
                            for (i = 0; i < limitCount; ++i)
                                if (allLimits[i].level == ZES_POWER_LEVEL_SUSTAINED)
                                    if (allLimits[i].limitValueLocked == False)
                                        allLimits[i].limit = newLimit
                        zesPowerSetLimitsExt(hPower, &numLimits, allLimits)
```

疑似コードは、平均電力を出力する方法を示しています。関数が定期的に（たとえば 100 ミリ秒ごとに）呼び出されることを前提としています。

```
function ShowAveragePower(zes_pwr_handle_t hPower, zes_power_energy_counter_t*
pPrevEnergyCounter)
    zes_power_energy_counter_t newEnergyCounter;
    if (zesPowerGetEnergyCounter(hPower, &newEnergyCounter) == ZE_RESULT_SUCCESS)
        uint64_t deltaTime = newEnergyCounter.timestamp - pPrevEnergyCounter->timestamp;
        if (deltaTime)
            output("    Average power: %.3f W", (newEnergyCounter.energy - pPrevEnergyCounter-
>energy) / deltaTime);
            *pPrevEnergyCounter = newEnergyCounter;
```

周波数ドメインの操作

ハードウェアは周波数を管理して、最高のパフォーマンスと電力消費のバランスを維持します。ほとんどのデバイスには 1 つ以上の周波数領域があります。

デバイス上の周波数領域を管理するため次の関数が提供されています:

関数	説明
zesDeviceEnumFrequencyDomains()	デバイスおよびサブデバイス上のすべての周波数領域を列挙します。
zesFrequencyGetProperties()	この周波数と最小/最大ハードウェア周波数によって制御されるドメイン zes_freq_domain_t を確認します。
zesFrequencyGetAvailableClocks()	このドメインで要求できるすべての利用可能な周波数の配列を取得します。
zesFrequencyGetRange()	周波数領域でハードウェアが動作できる現在の最小/最大周波数を取得します。
zesFrequencySetRange()	周波数領域でハードウェアが動作できる現在の最小/最大周波数を設定します。
zesFrequencyGetState()	周波数ドメインにおける現在の周波数要求、実際の周波数、TDP 周波数、およびスロットリングの原因を取得します。
zesFrequencyGetThrottleTime()	周波数領域が調整された時間を取得します。

特定の周波数領域に対してデバイス・プロパティー zes_freq_properties_t.canControl が true の場合にのみ、周波数範囲を設定できます。

最小/最大周波数範囲を同じ値に設定することにより、ソフトウェアはハードウェア制御の周波数を無効にし、Punit が過剰な電力/熱によるスロットルの必要がない限り、固定された周波数を取得します。

電力/熱条件によっては、ソフトウェアやハードウェアが要求する周波数が維持されない場合があります。この状況は、現在の周波数要求、実際の（解決された）周波数、および現在の状況に依存するその他の周波数情報を示す zesFrequencyGetState() 関数で判断できます。実際の周波数が要求された周波数を下回る場合、

zes_freq_state_t.throttleReasons は、Punit によって周波数が制限されている理由を示します。

スロットリングが開始されると、サポートされている場合
(zes_freq_properties_t.isThrottleEventSupported を参照)、イベント
ZES_EVENT_TYPE_FLAG_FREQ_THROTTLED がトリガーされます。

周波数/電圧オーバークロック

オーバークロックは電圧-周波数（V-F）曲線を変更して、ハードウェアがより高い周波数に達することを許可してパフォーマンスを向上させるか、同じ周波数で電圧を下げて効率を向上させることが可能になります。

デフォルトでは、ハードウェアは工場で調整された最大周波数と電圧周波数曲線を適用します。電圧-周波数曲線は、過電流状態に陥ることなく特定の周波数に安全に到達するのに必要な電圧を指定します。ハードウェアが過電流（IccMax）を検出すると、自身を保護するため周波数を大幅に制限します。また、ハードウェアはチップのいずれかの部分が最大温度制限（TjMax）を超えたことを検出すると、周波数を大幅に制限します。

最大のパフォーマンスを向上させるには、次の変更を加えることができます:

- 最大周波数を上げます。
- より高い周波数で安定性を確保するには、電圧を上げます。
- 最大電流（IccMax）を増やします。
- 最大温度（TjMax）を増やします。

これらすべての変更には、デバイスが破損するリスクが伴います。

デバイスの回路全体を使用しない特定のワークロードの効率を向上させるには、次の変更を加えることができます:

- 電圧を下げます

周波数のオーバークロックは、目標とする周波数ターゲットで zesFrequencyOcSetFrequencyTarget() を呼び出し、新しい電圧と電圧オフセットで zesFrequencyOcSetVoltageTarget() を呼び出して電圧を設定することで実現されます。オーバークロック時の電圧と周波数の制御方法には 3 つのモードがあります:

オーバークロック・モード	説明
ZES_OC_MODE_OVERRIDE	実際の周波数が要求された周波数より低い場合、このモードでは、周波数要求に関係なく、ユーザーが指定した固定電圧 VoltageTarget と VoltageOffset が常に適用されます。これは効率的ではありませんが、周波数の変化に伴う電源電圧の変化を回避することで安定性を向上できます。
ZES_OC_MODE_INTERPOLATIVE	このモードでは、補間される新しい電圧/周波数ポイントを使用して電圧/周波数曲線を拡張できます。既存の電圧/周波数ポイントも、固定電圧によってオフセット（上または下）できます。このモードでは、FIXED モードと OVERRIDE モードが無効になります。

オーバークロック・モード	説明
ZES_OC_MODE_FIXED	このモードでは、ハードウェアによる周波数スロットリングが無効になり、周波数と電圧が指定されたオーバークロック値にロックされます。このモードでは、OVERRIDE モードと INTERPOLATIVE モードが無効になります。このモードでは部品が破損する可能性があり、ほとんどの保護機能はこのモードでは無効になります。

オーバークロックを処理するため、次の関数が提供されています:

関数	説明
zesFrequencyOcGetCapabilities()	デバイスのオーバークロック機能を決定します。
zesFrequencyOcGetFrequencyTarget()	現在のオーバークロックのターゲット周波数セットを取得します。
zesFrequencyOcSetFrequencyTarget()	新しいオーバークロックのターゲット周波数を設定します
zesFrequencyOcGetVoltageTarget()	現在のオーバークロックのターゲット電圧セットを取得します。
zesFrequencyOcSetVoltageTarget()	新しいオーバークロックのターゲット電圧とオフセットを設定します。
zesFrequencyOcSetMode()	目標のオーバークロック・モードを設定します。
zesFrequencyOcGetMode()	目標のオーバークロック・モードを取得します。
zesFrequencyOcGetIccMax()	有効な最大電流制限を取得します。
zesFrequencyOcSetIccMax()	新しい最大電流制限を設定します。
zesFrequencyOcGetTjMax()	有効な最高温度制限を取得します。
zesFrequencyOcSetTjMax()	新しい最高温度制限を設定します。

オーバークロックをオフにするには、zesFrequencyOcSetMode() をモード ZES_OC_MODE_OFF で呼び出し、zesFrequencyOcGetIccMax() と zesFrequencyOcSetTjMax() を値 0.0 で呼び出します。

スケジューラー操作

スケジューラー・コンポーネントは、アクセラレーター・エンジンでワークロードを実行する方法と、複数のワークロードが同時に送信されたときにハードウェア・リソースを共有する方法を制御します。このポリシーはスケジューラー・モードと呼ばれます。

使用可能なスケジューラーの動作モードは、以下の表の列挙型 zes_sched_mode_t によって指定されます:

スケジューラー・モード	説明
ZES_SCHED_MODE_TIMEOUT	このモードは、ハードウェアにワークを送信する複数のアプリケーションまたはコンテキスト向けに最適化されています。より優先度の高いワークが到着すると、スケジューラーは一定のタイムアウト間隔内で現在実行中のワークを一時停止し、他のワークを送信しようとします。ウォッチドッグ・タイム

スケジューラー・モード	説明
	アウト（zes_sched_timeout_properties_t）を構成できます。これは、ワークロードがバッチワークを完了するか、終了する前に他のアプリケーションに引き継ぐまでスケジューラーが待機する最大時間を制御します。ウォッチドッグ・タイムアウトが ZES_SCHED_WATCHDOG_DISABLE に設定されていると、スケジューラーは公平性を強制しません。つまり、他に実行するワークがある場合、スケジューラーはそれを送信しようとしますが、すぐに完了しない実行中のプロセスを終了しません。
ZES_SCHED_MODE_TIMESLICE	このモードは、ハードウェアに同時にワークを送信する複数のコンテキスト間でハードウェアの実行時間を公平に共有するように最適化されています。設定可能です（zes_sched_timeslice_properties_t）。タイムスライス間隔と、スケジューラーが終了前に別のアプリケーションにワークが渡されるまで待機する時間の長さ。
ZES_SCHED_MODE_EXCLUSIVE	このモードは、単一のアプリケーション/コンテキストのユースケース向けに最適化されています。これにより、コンテキストがプリエンプトや終了されることなく、ハードウェア上で無期限に実行されることが許可されます。他のコンテキストで保留中のワークはすべて、実行中のコンテキストが完了し、それ以上のワークが送信されなくなるまで待機する必要があります。
ZES_SCHED_MODE_COMPUTE_UNIT_DEBUG	このモードはアプリケーションのデバッグ用に最適化されています。これにより、特定時点のハードウェア上でワークを実行できるコマンドキューは 1 つだけです。スケジューラーの公平性ポリシーを強制することなく、必要なだけワークを実行することが許可されます。

デバイスには複数のスケジューラー・コンポーネントを含めることができます。各スケジューラー・コンポーネントは、1 つ以上のアクセラレーター・エンジン（zes_engine_type_flags_t）上のワークロードの実行動作を制御します。各スケジューラー・コンポーネントのスケジューラー・モードを変更するには、次の関数を使用できます:

関数	説明
zesDeviceEnumSchedulers()	各スケジューラー・コンポーネントへのハンドルを取得します。
zesSchedulerGetProperties()	スケジューラー・コンポーネント（サブデバイス、このスケジューラーにリンクされているエンジン、サポートされているスケジューラー・モード）のプロパティーを取得します。

関数	説明
zesSchedulerGetCurrentMode()	現在のスケジューラー・モード（タイムアウト、タイムスライス、排他、単一コマンドキュー）を取得します。
zesSchedulerGetTimeoutModeProperties()	タイムアウト・スケジューラー・モードの設定を取得します
zesSchedulerGetTimesliceModeProperties()	タイムスライス・スケジューラー・モードの設定を取得します
zesSchedulerSetTimeoutMode()	タイムアウト・スケジューラー・モードに変更するか、プロパティーを変更します
zesSchedulerSetTimesliceMode()	タイムスライス・スケジューラー・モードに変更するか、プロパティーを変更します
zesSchedulerSetExclusiveMode()	排他スケジューラー・モードに変更するか、プロパティーを変更します
zesSchedulerSetComputeUnitDebugMode()	計算ユニットのデバッグ・スケジューラー・モードに変更するか、プロパティーを変更します

以下の疑似コードは、他のワークの実行を許可しながら公平性を強制するスケジューラーの停止方法を示しています:

```
function DisableSchedulerWatchdog(zes_device_handle_t hSysmanDevice)
   uint32_t numSched
   if ((zesDeviceEnumSchedulers(hSysmanDevice, &numSched, NULL) == ZE_RESULT_SUCCESS))
      zes_sched_handle_t* pSchedHandles =
         allocate_memory(numSched * sizeof(zes_sched_handle_t))
      if (zesDeviceEnumSchedulers(hSysmanDevice, &numSched, pSchedHandles) ==
ZE_RESULT_SUCCESS)
         for (index = 0 .. numSched-1)
            ze_result_t res
            zes_sched_mode_t currentMode
            res = zesSchedulerGetCurrentMode(pSchedHandles[index], &currentMode)
            if (res == ZE_RESULT_SUCCESS)
               ze_bool_t requireReload
               zes_sched_timeout_properties_t props
               props.watchdogTimeout = ZES_SCHED_WATCHDOG_DISABLE
               res = zesSchedulerSetTimeoutMode(pSchedHandles[index], &props,
&requireReload)
               if (res == ZE_RESULT_SUCCESS)
                  if (requireReload)
                     output("WARNING: Reload the driver to complete desired
configuration.")
                  else
```

```
                    output("Schedule mode changed successfully.")
                else if(res == ZE_RESULT_ERROR_UNSUPPORTED_FEATURE)
                    output("ERROR: The timeout scheduler mode is not supported on this
device.")
                else if(res == ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS)
                    output("ERROR: Don't have permissions to change the scheduler mode.")
                else
                    output("ERROR: Problem calling the API to change the scheduler mode.")
            else if(res == ZE_RESULT_ERROR_UNSUPPORTED_FEATURE)
                output("ERROR: Scheduler modes are not supported on this device.")
            else
                output("ERROR: Problem calling the API.")
```

ECC 構成を動的に有効化/無効化

メモリーの破損は、背景放射線、宇宙線などの自然現象によりデータ内のランダムビットが反転したときに発生します。単一のデータ値の上位ビットの 1 つが 1 ビット反転するだけで、一部のアプリケーションの動作が大幅に変わる場合もあります。金融、産業制御、重要なインフラストラクチャー、重要なデータベース分野のワークロードでは、通常、メモリー破損は許容されません。メモリー破損により、非常に望ましくない動作が発生する可能性があります。エラー訂正コード（ECC）は、メモリーのパフォーマンスと容量を犠牲にしてメモリーの破損を減らすメモリー・コントローラー・テクノロジーです。

メモリーのパフォーマンスと容量が失われるため、一部のワークロードでは ECC は望ましくありません。アプリケーション・ドメインでは、低レベルのメモリー破損の影響を受けない場合もあります。アルゴリズムは数値的安定性を考慮して設計されることもあれば、本質的に確率的であることもあり、メモリー破損の影響を受けない場合もあります。

製品は ECC 機能をサポートし、さらに ECC を動的に構成可能にする場合もあります。つまり、ECC がサポートされている場合、必要に応じてオンまたはオフにすることができます。ECC の有効状態と無効状態を切り替えるには、ウォームリセットまたはコールドリブートのいずれかで、デバイスリセットが必要になる場合があります。

ECC のサポートは、`zesDeviceEccAvailable()` 関数で確認できます。ECC がサポートされている場合は、`zesDeviceEccConfigurable()` 関数を使用して動的に ECC 制御のサポートを確認できます。現在の ECC 状態、保留中の ECC 状態、および保留中の ECC 状態に影響を与えるアクションは、`zesDeviceGetEccState()` 関数によって返される `zes_device_ecc_properties_t` 構造体を使用して判別できます。ECC 状態は、`zesDeviceSetEccState()` を呼び出すことによって変更できます。この関数は、目的の ECC 状態を入力として受け取り、現在の ECC 状態、保留中の ECC 状態、および保留中の ECC 状態に影響を与えるアクションをリストする `zes_device_ecc_properties_t` 構造体を返します。

次の疑似コードは、ECC 状態を照会し、無効から有効に変更する方法を示しています:

```
function EnableECC(zes_device_handle_t hSysmanDevice)
    ze_bool_t EccAvailable = False;
    zesDeviceEccAvailable(hSysmanDevice, &EccAvailable)
    if (EccAvailable == True) {
```

```
        ze_bool_t EccConfigurable = False;
        zesDeviceEccConfigurable(hSysmanDevice, &EccConfigurable)
        if (EccConfigurable == True) {
            zes_device_ecc_properties_t props = {ZES_DEVICE_ECC_STATE_UNAVAILABLE,
ZES_DEVICE_ECC_STATE_UNAVAILABLE, ZES_DEVICE_ACTION_NONE}
            zesDeviceGetEccState(hSysmanDevice, &props)
            if (props.currentState == ZES_DEVICE_ECC_STATE_DISABLED) {
                zes_device_ecc_desc_t newState = ZES_DEVICE_ECC_STATE_ENABLED
                zesDeviceSetEccState(hSysmanDevice, newState, &props)
            }
        }
    }
```

ワークロードのパフォーマンス調整

ハードウェアはシステムリソースを効率良くバランスさせようとしますが、外部からのパフォーマンス・ヒントの恩恵を受けられるワークロードもあります。これが可能なハードウェアでは、API はこれらのヒントを提供するために使用できる*パフォーマンス係数*ドメインを公開します。

パフォーマンス係数は、アクセラレーター・ユニットに供給されるエネルギーとサポートユニットに供給されるエネルギー間のトレードオフを表す 0 から 100 までの数値として定義されます。たとえば、計算負荷の高いワークロードでは、メモリー・コントローラーではなく計算ユニットにエネルギーをより多く配分することでメリットが得られます。あるいは、メモリー制限されたワークロードでは、計算ユニットのパフォーマンスを犠牲にしてメモリー・コントローラーのスループットを高めることでメリットが得られます。通常、ハードウェアはこのバランスを適切に維持しますが、パフォーマンス係数を使用するとバランスをさらに積極的に調整できます。上記の例では、パフォーマンス係数が 100 の場合、ワークロードは完全に計算に制限されており、メモリー・コントローラーからのサポートはほとんど必要ないことを示しています。一方、パフォーマンス係数が 0 の場合、ワークロードは完全にメモリーに制限されており、メモリー・コントローラーのパフォーマンスを向上させる必要があることを示します。

ワークロードのチューニングには、パフォーマンス係数を 0 から 100 まで変化させてアプリケーションを繰り返し実行し、最適なパフォーマンスが得られる値を選択することが含まれます。デフォルト値は 50 です。あるいは、さらに動的なアプローチとして、アクセラレーターのさまざまな利用率メトリックを監視してメモリーと計算の上限を決定し、ボトルネックを解消するためパフォーマンス係数を上下させる試みが挙げられます。

API は、パフォーマンス係数によって制御できるドメインを列挙する方法を提供します。ドメインには、この設定によってパフォーマンスの影響を受けるアクセラレーターが 1 つ以上含まれています。API は、ドメインのパフォーマンス係数を変更する関数を提供します。

利用可能な関数の概要は次のとおりです:

関数	説明
`zesDeviceEnumPerformanceFactorDomains()`	ハードウェアで使用可能なパフォーマンス係数ドメインを列挙します。

関数	説明
zesPerformanceFactorGetProperties()	パフォーマンス係数ドメインがサブデバイスに配置されているか、またいずれのアクセラレーターがそれによって影響を受けるか確認します。
zesPerformanceFactorGetConfig()	ドメインのハードウェアによって使用されている現在のパフォーマンス係数を読み取ります。
zesPerformanceFactorSetConfig()	ドメインのハードウェア・パフォーマンス係数を変更します。

エンジングループの操作

アクセラレーター・リソース（計算ユニットアレイやメディアデコーダーなど）には、エンジンによってワークが供給されます。API は、これらのエンジンの実行時間（アクティビティー）を測定する機能を提供します。エンジンのタイプは、zes_engine_group_t で定義されます。

通常、エンジンと基礎となるアクセラレーター・リソースには対の関係があります。たとえば、単一のメディア・デコード・エンジンは単一のメディア・デコーダー・ハードウェアにワークを送信しますが、他のエンジンはそれを実行できません。単一のエンジンの実行時間（アクティビティー）を測定するのは、アクセラレーター・ハードウェアの実行時間を測定するのと同じです。

複数のエンジンが同じアクセラレーター・ハードウェアにワークを送信する製品もあります。ハードウェアは各エンジンからのワークを同時に実行します。この場合、各エンジンはアクセラレーター・ハードウェアの一部のみを受信するため、各エンジンの実行時間の合計は、アクセラレーター・ハードウェアの実行時間よりも長くなります。また、API は、個々のエンジンのレベルではなく、ハードウェア・アクセラレーターのレベルで合計実行時間を測定するエンジングループも提供します。たとえば、API は ZES_ENGINE_GROUP_COMPUTE_SINGLE タイプの複数のエンジングループを列挙し、個々のエンジンのアクティビティーを測定できるようになります。ただし、共有計算リソース全体のアクティビティーを測定するため、API は ZES_ENGINE_GROUP_COMPUTE_ALL タイプのエンジングループを列挙します。この場合、アクティビティー・レポートは、2 つのスナップショットの間にいずれかの計算エンジンがアクティブであった場合に作成されます。

アクティビティー・カウンターのスナップショットを 2 つ取得することで、デバイスの各種平均利用率を計算できます。

以下の関数が提供されます:

関数	説明
zesDeviceEnumEngineGroups()	照会可能なエンジングループを列挙します。
zesEngineGetProperties()	エンジングループのプロパティーを取得します。これは、エンジングループのタイプ（zes_engine_group_t のいずれか）と、グループが測定しているサブデバイスを返します。
zesEngineGetActivity()	エンジングループのアクティビティー・カウンターを返します。

スタンバイドメインの操作

デバイスがアイドル状態のときは低電力状態になります。低電力状態からの移行にはレイテンシーが伴うため、パフォーマンスが低下する可能性があります。ハードウェアは、デバイスへのワークロードの送信間に長いアイドル時間がある場合、電力を節約し、送信間のアイドル時間が短い場合には、デバイスを起動したまま維持することの間でバランスを取ろうとします。

デバイスは、自律的に電源をスタンバイ状態にできる 1 つ以上のブロックで構成されます。ドメインのリストは `zes_standby_type_t` によって提供されます。

以下の関数を使用して、ハードウェアがスタンバイ状態に移行する方法を制御できます:

関数	説明
`zesDeviceEnumStandbyDomains()`	スタンバイドメインを列挙します。
`zesStandbyGetProperties()`	スタンバイドメインのプロパティーを取得します。これにより、このドメイン（`zes_engine_group_t` のいずれか）の影響を受けるデバイスの部分と、ドメインが配置されているサブデバイスが返されます。
`zesStandbyGetMode()`	スタンバイドメインの現在のプロモーション・モード（`zes_standby_promo_mode_t` のいずれか）を取得します。
`zesStandbySetMode()`	スタンバイドメインのプロモーション・モード（`zes_standby_promo_mode_t` のいずれか）を取得します。

ファームウェアの操作

デバイスのファームウェアを管理するため次の関数が提供されています:

関数	説明
`zesDeviceEnumFirmwares()`	デバイス上で管理できるすべてのファームウェアを列挙します。
`zesFirmwareGetProperties()`	ファームウェアの名前とバージョンを確認します。
`zesFirmwareFlash()`	新しいファームウェア・イメージをフラッシュします。

メモリーモジュールの照会

API は、すべてのデバイスのメモリーモジュールの列挙を提供します。メモリーモジュールごとに、現在の帯域幅と最大帯域幅を照会できます。API は、次のいずれかの値（`zes_mem_health_t`）にできるヘルスメトリックも提供します。

メモリー状態	説明
`ZES_MEM_HEALTH_OK`	すべてのメモリーチャネルは正常です。
`ZES_MEM_HEALTH_DEGRADED`	1 つ以上のチャネルで過剰な修正可能なエラーが検出されました。デバイスをリセットする必要があります。
`ZES_MEM_HEALTH_CRITICAL`	修復不可能なエラーが多すぎるバンクをカバーするためメモリーを減らして動作します。
`ZES_MEM_HEALTH_REPLACE`	修復不可能なエラーが多すぎるため、デバイスを交換する必要があります。

メモリーモジュールのヘルス状態が変化すると、イベント ZES_EVENT_TYPE_FLAG_MEM_HEALTH がトリガーされます。

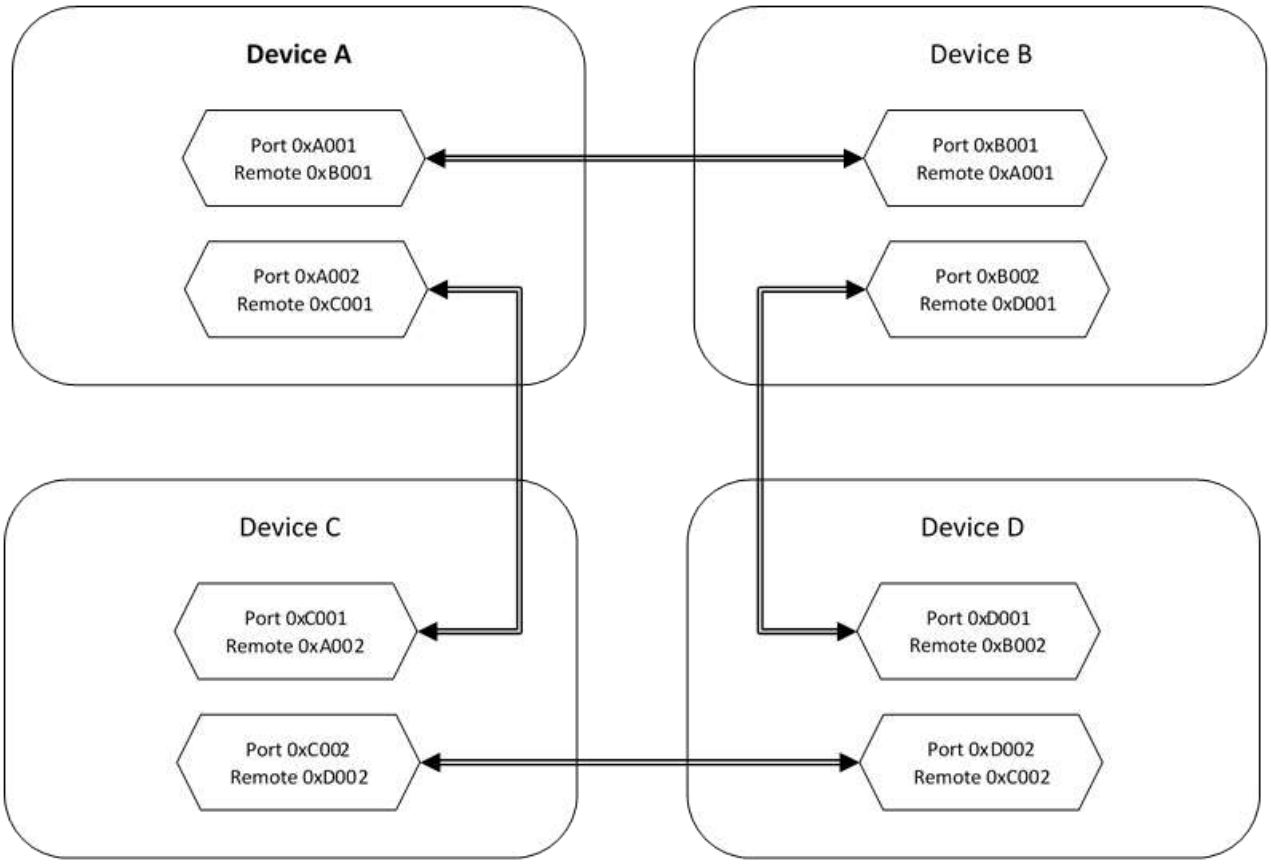
次の関数は、デバイスのメモリーモジュールに関する情報へのアクセスを提供します:

関数	説明
zesDeviceEnumMemoryModules()	メモリーモジュールを列挙します。
zesMemoryGetProperties()	モジュールのメモリーのタイプと最大物理メモリーを確認します。
zesMemoryGetBandwidth()	モジュールのメモリー帯域幅カウンターを返します。
zesMemoryGetState()	メモリーモジュールの現在の空きメモリーの状態と物理メモリーの合計を返します。

ファブリック・ポートの操作

ファブリックとは、アクセラレーター・デバイス間の高速相互接続を表す用語であり、主にリモートデバイスのメモリーへの低レイテンシーの高速アクセスを提供するために使用されます。デバイスには、物理リンクを介してデータを送受信する 1 つ以上の**ファブリック・ポート**があります。リンクはファブリック・ポートを接続し、デバイス間でデータを移動できるようにします。ルーティング・ルールは、ファブリックを通過するトラフィックのフローを決定します。

下の図は、それぞれ 2 つのファブリック・ポートを備えた 4 つのデバイスを示しています。各ポートには、別のデバイスのポートに接続するリンクがあります。この例では、デバイスはリング状に接続されています。デバイス A と D は、ファブリック・ルーティング・ルールの設定に応じて、デバイス B またはデバイス C を介して互いのメモリーにアクセスできます。デバイス B と D 間の接続がダウンした場合、ルーティング・ルールを変更して、デバイス B と D がファブリック内の 2 つのホップ（デバイス A と C）を経由して互いのメモリーにアクセスするように構成できます。



API を使用すると、デバイスで使用可能なすべてのポートを列挙できます。各ポートは、システム内で次の情報によって一意に識別されます:

- ファブリック ID: ファブリック・エンドポイントの一意の識別子
- アタッチ ID: デバイス接続ポイントの一意の識別子
- ポート番号: 論理ポート番号（通常は物理デバイスのいずれかにマークされています）

API はこの情報を `struct {t}_fabric_port_id_t` で提供します。識別子は汎用的ではありません。一意性は特定のシステム内でのみ保証され、システム構成が変更されないという条件が満たされます。

ファブリック・ポートが接続されると、API はリモート・ファブリック・ポートの一意の識別子を提供します。システム内のすべてのポートを列挙し、リモートポート ID を一致させることで、アプリケーションは接続トポロジーマップを構築できます。

各ポートについて、API はポートの構成（UP/DOWN）とヘルス（次のいずれかの値）を照会できます:

ファブリック・ポートの状態	説明
ZES_FABRIC_PORT_STATUS_HEALTHY	ポートは稼働しており、期待通りに動作しています。
ZES_FABRIC_PORT_STATUS_DEGRADED	ポートは稼働していますが、品質や帯域幅が低下しています。
ZES_FABRIC_PORT_STATUS_FAILED	不安定なポート接続により、ワークロードの進行が妨げられています。
ZES_FABRIC_PORT_STATUS_DISABLED	ポートはダウンに設定されています。

ポートが劣化状態にある場合、API は、観察されている品質劣化のタイプに関する追加情報を提供します。ポート

がレッド状態の場合、API は不安定の原因に関する追加情報を提供します。

ポートのヘルス状態が変化すると、イベント ZES_EVENT_TYPE_FLAG_FABRIC_PORT_HEALTH がトリガーされます。

API は、各ポートの現在の送信および受信ビットレートを提供します。また、各ポートを通過する受信および送信の帯域幅測定も可能です。これらのメトリックには、デバイスによって生成されるトラフィックに加えて、プロトコルのオーバーヘッドも含まれます。

ポートはデータを別のポートに直接渡すことができるため、ポートで測定された帯域幅は、2 つのポートに直接接続されたアクセラレーターの実際の帯域幅よりも高くなることがあります。そのため、各ポートの帯域幅メトリックは、ファブリック内の輻輳ポイントを特定することに関連しますが、2 つのアクセラレーター間で渡される合計帯域幅を測定することの関連性は高くありません。

ファブリック・ポートを管理するには、次の関数を使用できます:

関数	説明
zesDeviceEnumFabricPorts()	デバイス上のすべてのファブリック・ポートを列挙します。
zesFabricPortGetProperties()	ポートに関する静的プロパティー（モデル、ポート ID、最大受信/送信速度）を取得します。
zesFabricPortGetLinkType()	ポートに接続されている物理リンクの詳細を取得します。
zesFabricPortGetConfig()	ポートが UP に設定されているか、およびビーコンがオンかオフかを判断します。
zesFabricPortSetConfig()	ポートを UP または DOWN に設定し、ビーコンをオンまたはオフにします。
zesFabricPortGetState()	ポート接続の健全性、リンクの劣化または接続の問題の理由、現在の受信/送信、リモート・エンドポイントのポート ID を確認します。
zesFabricPortGetThroughput()	ポートの受信/送信カウンターと現在の受信/送信ポート速度を取得します。

デバイスがサブデバイスを持つ場合、ファブリック・ポートは通常、サブデバイス内にあります。デバイスハンドルが指定されると、zesDeviceEnumFabricPorts() には各サブデバイス上のポートが含まれます。この場合、zes_fabric_port_properties_t.onSubdevice は true に設定され、zes_fabric_port_properties_t.subdeviceId はそのポートが配置されているサブデバイス ID を指定します。

以下の疑似コードは、デバイスとサブデバイス内のすべてのファブリック・ポートの状態を取得する方法を示しています:

```
void ShowFabricPorts(zes_device_handle_t hSysmanDevice)
   uint32_t numPorts
   if ((zesDeviceEnumFabricPorts(hSysmanDevice, &numPorts, NULL) == ZE_RESULT_SUCCESS))
      zes_fabric_port_handle_t* phPorts =
         allocate_memory(numPorts * sizeof(zes_fabric_port_handle_t))
      if (zesDeviceEnumFabricPorts(hSysmanDevice, &numPorts, phPorts) == ZE_RESULT_SUCCESS)
         for (index = 0 .. numPorts-1)
            # 特定のポートに関する情報を表示
```

```
            output("    Port %u:n", index)
            call_function ShowFabricPortInfo(phPorts[index])
    free_memory(...)

function ShowFabricPortInfo(zes_fabric_port_handle_t hPort)
    zes_fabric_port_properties_t props {};
    props.stype = ZES_STRUCTURE_TYPE_FABRIC_PORT_PROPERTIES;
    if (zesFabricPortGetProperties(hPort, &props) == ZE_RESULT_SUCCESS)
        zes_fabric_port_state_t state
        if (zesFabricPortGetState(hPort, &state) == ZE_RESULT_SUCCESS)
            zes_fabric_link_type_t link
            if (zesFabricPortGetLinkType(hPort, &link) == ZE_RESULT_SUCCESS)
                zes_fabric_port_config_t config
                if (zesFabricPortGetConfig(hPort, &config) == ZE_RESULT_SUCCESS)
                    output("       Model:              %s", props.model)
                    if (props.onSubdevice)
                        output("       On sub-device:       %u", props.subdeviceId)
                    if (config.enabled)
                    {
                        var status
                        output("       Config:             UP")
                        switch (state.status)
                            case ZES_FABRIC_PORT_STATUS_HEALTHY:
                                status = "HEALTHY - The port is up and operating as expected"
                            case ZES_FABRIC_PORT_STATUS_DEGRADED:
                                status = "DEGRADED - The port is up but has quality and/or
bandwidth degradation"
                            case ZES_FABRIC_PORT_STATUS_FAILED:
                                status = "FAILED - Port connection instabilities"
                            case ZES_FABRIC_PORT_STATUS_DISABLED:
                                status = "DISABLED - The port is configured down"
                            default:
                                status = "UNKNOWN"
                        output("       Status:             %s", status)
                        output("       Link type:           %s", link.desc)
                        output(
                            "       Max speed (rx/tx):    %llu/%llu bytes/sec",
                            props.maxRxSpeed.bitRate * props.maxRxSpeed.width / 8,
                            props.maxTxSpeed.bitRate * props.maxTxSpeed.width / 8)
                        output(
                            "       Current speed (rx/tx): %llu/%llu bytes/sec",
```

```
                        state.rxSpeed.bitRate * state.rxSpeed.width / 8,
                        state.txSpeed.bitRate * state.txSpeed.width / 8)
                else
                    output("       Config:              DOWN")
```

zesFabricPortGetMultiPortThroughput 関数は、1 度の呼び出しで複数のポートのスループット値をまとめて収集するメカニズムを提供します。

次の疑似コードは、API を使用してスループットを収集する方法を示しています:

```
// ファブリック・ポートを列挙
uint32_t numPorts = 0;
zesDeviceEnumFabricPorts(hSysmanDevice, &numPorts, NULL);
zes_fabric_port_handle_t* phPorts =
    allocate_memory(numPorts * sizeof(zes_fabric_port_handle_t));
zesDeviceEnumFabricPorts(hSysmanDevice, &numPorts, phPorts);

// すべてのファブリック・ポートのスループットをまとめて収集
zes_fabric_port_throughput_t* pThroughput =
    allocate_memory(numPorts * sizeof(zes_fabric_port_throughput_t));
zesFabricPortGetMultiPortThroughput(hSysmanDevice, numPorts, phPorts, &pThroughput);
```

温度の照会

デバイスには、さまざまな場所に複数の温度センサーが設置されています。次の場所がサポートされています:

温度センサーの場所	説明
ZES_TEMP_SENSORS_GLOBAL	デバイス内のすべてのセンサーで測定された最大温度を返します。
ZES_TEMP_SENSORS_GPU	GPU アクセラレーター内のすべてのセンサーで測定された最大温度を返します。
ZES_TEMP_SENSORS_MEMORY	デバイスメモリー内のすべてのセンサーで測定された最大温度を返します。
ZES_TEMP_SENSORS_GLOBAL_MIN	デバイス内のすべてのセンサーで測定された最小温度を返します。
ZES_TEMP_SENSORS_GPU_MIN	GPU アクセラレーター内のすべてのセンサーで測定された最小温度を返します。
ZES_TEMP_SENSORS_MEMORY_MIN	デバイスメモリー内のすべてのセンサーで測定された最小温度を返します。

一部のセンサーでは、温度がしきい値を超えたときにイベントをトリガーすることを要求できます。これは、zesTemperatureGetConfig() 関数および zesTemperatureSetConfig() 関数を使用して実現されます。特定のイベントのサポートは、zesTemperatureGetProperties() を呼び出すことで実現されます。一般に、温度イベントは、ZES_TEMP_SENSORS_GLOBAL タイプの温度センサーでのみサポートされます。以下のリストは温度イベントのリストを示しています:

イベント	チェックポイント	説明
ZES_EVENT_TYPE_FLAG_TEMP_CRITICAL	zes_temp_properties_t.isCriticalTempSupported	このイベントは、温度が重大な周波数調整が行われる臨界領

イベント	チェックポイント	説明
		域を超えるとトリガーされます。
ZES_EVENT_TYPE_FLAG_TEMP_THRESHOLD1	zes_temp_properties_t .isThreshold1Supported	温度がカスタムしきい値 1 を超えると、イベントがトリガーされます。フラグを設定すると、高値から安値へ、または安値から高値へ交差するときにトリガーを制限できます。
ZES_EVENT_TYPE_FLAG_TEMP_THRESHOLD2	zes_temp_properties_t .isThreshold2Supported	温度がカスタムしきい値 2 を超えると、イベントがトリガーされます。フラグを設定すると、高値から安値へ、または安値から高値へ交差するときにトリガーを制限できます。

温度センサーを管理するには、次の関数を使用できます:

関数	説明
zesDeviceEnumTemperatureSensors()	デバイス上の温度センサーを列挙します。
zesTemperatureGetProperties()	温度センサーの静的プロパティーを取得します。特に、これはセンサーがデバイスのどの部分を測定するかを示します (zes_temp_sensors_t のいずれか)。
zesTemperatureGetConfig()	現在の温度しきい値に関する情報を取得します - 有効/しきい値/プロセス ID。
zesTemperatureSetConfig()	新しい温度しきい値を設定します。温度がこれらのしきい値を超えると、イベントがトリガーされます。
zesTemperatureGetState()	センサーの温度を読み取ります。

電源の操作

次の関数を使用して、デバイス上の各電源に関する情報にアクセスできます:

関数	説明
zesDeviceEnumPsus()	管理可能なデバイス上の電源を列挙します。
zesPsuGetProperties()	電源に関する静的な詳細を取得します。
zesPsuGetState()	電源の状態 (温度、電流、ファン) に関する情報を取得します。

ファンの操作

zesDeviceEnumFans() が 1 つ以上のファンハンドルを返す場合、それらの速度を管理できます。ハードウェア

には、ファンを固定速度（またはサイレント動作の場合は 0）で動作させるように指示することも、温度-速度ポイントのテーブルを提供するように指示することもできます。その場合、ハードウェアはチップの現在の温度に基づいてファン速度を動的に変更します。この構成情報は、zes_fan_config_t 構造体に記述されています。速度を指定する場合、1 分あたりの回転数（ZES_FAN_SPEED_UNITS_RPM）または最大 RPM のパーセンテージ（ZES_FAN_SPEED_UNITS_PERCENT）値を指定できます。

以下の関数が利用可能です:

関数	説明
zesDeviceEnumFans()	デバイス上のファンを列挙します。
zesFanGetProperties()	ファンの最大 RPM と、ファンの温度 - 速度テーブルで指定できるポイントの最大数を取得します。
zesFanGetConfig()	ファンの現在の構成（速度）を取得します。
zesFanSetDefaultMode()	ファン制御を工場出荷時のデフォルトに戻します。
zesFanSetFixedSpeedMode()	ファンが一定速度で回転するように設定します。
zesFanSetSpeedTableMode()	温度に応じてファン速度を設定します。
zesFanGetState()	ファンの現在の速度を取得します。

以下の疑似コードは、すべてのファンの速度を出力する方法を示しています:

```
function ShowFans(zes_device_handle_t hSysmanDevice)
   uint32_t numFans
   if (zesDeviceEnumFans(hSysmanDevice, &numFans, NULL) == ZE_RESULT_SUCCESS)
      zes_fan_handle_t* phFans =
         allocate_memory(numFans * sizeof(zes_fan_handle_t))
      if (zesDeviceEnumFans(hSysmanDevice, &numFans, phFans) == ZE_RESULT_SUCCESS)
         output("    Fans")
         for (fanIndex = 0 .. numFans-1)
            int32_t speed
            if (zesFanGetState(phFans[fanIndex], ZES_FAN_SPEED_UNITS_RPM, &speed)
               == ZE_RESULT_SUCCESS)
                  output("       Fan %u: %d RPM", fanIndex, speed)
   free_memory(...)
}
```

次の例は、制御が許可されている場合にのみ、すべてのファンの速度を RPM 単位で固定値に設定する方法を示しています:

```
function SetFanSpeed(zes_device_handle_t hSysmanDevice, uint32_t SpeedRpm)
{
   uint32_t numFans
   if (zesDeviceEnumFans(hSysmanDevice, &numFans, NULL) == ZE_RESULT_SUCCESS)
```

```
        zes_fan_handle_t* phFans =
            allocate_memory(numFans * sizeof(zes_fan_handle_t))
        if (zesDeviceEnumFans(hSysmanDevice, &numFans, phFans) == ZE_RESULT_SUCCESS)
            zes_fan_speed_t speedRequest
            speedRequest.speed = SpeedRpm
            speedRequest.speedUnits = ZES_FAN_SPEED_UNITS_RPM
            for (fanIndex = 0 .. numFans-1)
                zes_fan_properties_t fanprops {};
                fanprops.stype = ZES_STRUCTURE_TYPE_FAN_PROPERTIES;
                if (zesFanGetProperties(phFans[fanIndex], &fanprops) == ZE_RESULT_SUCCESS)
                    if (fanprops.canControl)
                        zesFanSetFixedSpeedMode(phFans[fanIndex], &speedRequest)
                    else
                        output("ERROR: Can't control fan %u.n", fanIndex)
    free_memory(...)
}
```

LED の操作

zesDeviceEnumLeds() が 1 つ以上の LED ハンドルを返す場合、デバイス上の LED を管理できます。これには、オン/オフの切り替えや、機能がある場合はリアルタイムで色を変更することが含まれます。

以下の関数が利用可能です:

関数	説明
zesDeviceEnumLeds()	管理可能なデバイス上の LED を列挙します。
zesLedGetProperties()	LED が色の変更をサポートしているか確認します。
zesLedGetState()	LED が現在オフ/オンになっているかどうか、および利用可能な色を確認します。
zesLedSetState()	機能が利用可能な場合、LED をオフ/オンにし、色を設定します。

RAS エラーの照会

RAS は、信頼性（Reliability）、可用性（Availability）、保守性（Serviceability）の略です。これは、ランダムなビットエラーを修正し、永続的な損傷が発生した場合に冗長性を提供する特定のデバイスの機能です。

デバイスが RAS をサポートしている場合、ハードウェアおよびソフトウェア・エラー・カウンターが維持されます。エラーには 2 つの種類があり、zes_ras_error_type_t で定義されています:

エラータイプ	説明
ZES_RAS_ERROR_TYPE_UNCORRECTABLE	ハードウェア・エラーが発生すると、データが失われたり、デバイスがハングする可能性が高くなります。エラーによってデバイスがロックアップした場合、そのエラーが報告される前にウォームブートが必要になります。

エラータイプ	説明
ZES_RAS_ERROR_TYPE_CORRECTABLE	これらはハードウェアによって修正されたエラーであり、データ破損はありません。

ソフトウェアは zesRasGetProperties() 関数を使用して、デバイスが RAS をサポートしているか、また有効であるかを確認できます。この情報は、zes_ras_properties_t 構造体で返されます。

zesDeviceEnumRasErrorSets() 関数は、使用可能な RAS エラーセットを列挙します。ハンドルが返されない場合、デバイスは RAS をサポートしていません。サブデバイスのないデバイスは、RAS がサポートされている場合、1 つのハンドルを返します。サブデバイスを持つデバイスは、各サブデバイスのハンドルを返します。

エラーが発生したか判断するには、ソフトウェアは zesRasGetState() 関数を使用します。これにより、発生した特定のタイプ（修正可能/修正不可能）エラーの合計数が返されます。

zesRasGetState() を呼び出すときに、ソフトウェアはエラーカウンターのクリアを要求できます。これを実行すると、指定されたタイプ（訂正可能/訂正不可能）のすべてのカウンターがゼロに設定され、この関数への後続の呼び出しでは新たに発生したエラーのみが表示されます。ソフトウェアがエラーをクリアする場合、それを実行する唯一のアプリケーションである必要があり、履歴解析のため適切なデータベースにカウンターを保存する必要があります。

zesRasGetState() は、zes_ras_state_t 構造体にカテゴリー別のエラーの内訳を返します。以下の表はカテゴリーについて説明しています:

エラーカテゴリー	ZES_RAS_ERROR_TYPE_CORRECTABLE	ZES_RAS_ERROR_TYPE_UNCORRECTABLE
ZES_RAS_ERROR_CAT_RESET	常にゼロ。	ドライバーが試みたアクセル・エンジン・リセットの回数。
ZES_RAS_ERROR_CAT_PROGRAMMING_ERRORS	常にゼロ。	ワークロードがハードウェアをプログラムする方法によって生成されたハードウェア例外の数。
ZES_RAS_ERROR_CAT_DRIVER_ERRORS	常にゼロ。	低レベルドライバーの通信エラー数。
ZES_RAS_ERROR_CAT_COMPUTE_ERRORS	アクセラレーター・ハードウェアで発生し修正されたエラーの数。	アクセラレーター・ハードウェアで発生し、修正されなかったエラーの数。これにより、ハードウェアがハングし、ドライバーがリセットされる可能性があります。
ZES_RAS_ERROR_CAT_NON_COMPUTE_ERRORS	固定機能アクセラレーター・ハードウェアで発生し、修正されたエラーの数。	固定機能アクセラレーター・ハードウェアで発生し、修正されなかったエラーの数。通常、これにより PCI バスリセットとドライバーリセットが発生します。

エラーカテゴリー	ZES_RAS_ERROR_TYPE_CORRECTABLE	ZES_RAS_ERROR_TYPE_UNCORRECTABLE
ZES_RAS_ERROR_CAT_CACHE_ERRORS	オンチップキャッシュ（キャッシュ/レジスターファイル/共有ローカルメモリー）で発生した ECC 訂正可能エラーの数。	オンチップキャッシュ（キャッシュ/レジスターファイル/共有ローカルメモリー）で発生した ECC 訂正不可能エラーの数。これにより、ハードウェアがハングし、ドライバーがリセットされる可能性があります。
ZES_RAS_ERROR_CAT_DISPLAY_ERRORS	ディスプレイで発生した ECC 訂正可能エラーの数。	ディスプレイで発生した ECC 訂正不可能エラーの数。

各 RAS エラータイプは、エラーカウンターがしきい値を超えるとイベントをトリガーできます。イベントを以下の表に示します。ソフトウェアは、zesRasGetConfig() 関数および zesRasSetConfig() 関数を使用して、各エラータイプのしきい値を取得および設定できます。デフォルトではすべてのしきい値が 0 に設定されており、イベントは生成されません。しきい値は、合計 RAS エラーカウンターまたは各詳細エラーカウンターに対して設定できます。

RAS エラータイプ	イベント
ZES_RAS_ERROR_TYPE_UNCORRECTABLE	ZES_EVENT_TYPE_FLAG_RAS_UNCORRECTABLE_ERRORS
ZES_RAS_ERROR_TYPE_CORRECTABLE	ZES_EVENT_TYPE_FLAG_RAS_CORRECTABLE_ERRORS

以下の表は、すべての RAS 管理関数をまとめたものです:

関数	説明
zesDeviceEnumRasErrorSets()	使用可能な RAS エラーグループへのハンドルを取得します。
zesRasGetProperties()	RAS エラーグループに関するプロパティー（RAS エラーのタイプと有効かどうか）を取得します。
zesRasGetConfig()	RAS グループ内の各カウンターの現在のしきい値のリストを取得します。しきい値を超えると、RAS エラーイベントが生成されます。
zesRasSetConfig()	RAS グループ内の各カウンターの現在のしきい値のリストを設定します。しきい値を超えると、RAS エラーイベントが生成されます。
zesRasGetState()	RAS エラーカウンターの現在の状態を取得します。カウンターはクリアすることもできます。

以下の疑似コードは、RAS がサポートされているかどうか、および RAS エラーの現在の状態を確認する方法を示しています:

```
void ShowRasErrors(zes_device_handle_t hSysmanDevice)
    uint32_t numRasErrorSets
    if ((zesDeviceEnumRasErrorSets(hSysmanDevice, &numRasErrorSets, NULL) ==
ZE_RESULT_SUCCESS))
```

```
      zes_ras_handle_t* phRasErrorSets =
         allocate_memory(numRasErrorSets * sizeof(zes_ras_handle_t))
      if (zesDeviceEnumRasErrorSets(hSysmanDevice, &numRasErrorSets, phRasErrorSets) ==
ZE_RESULT_SUCCESS)
         for (rasIndex = 0 .. numRasErrorSets)
            zes_ras_properties_t props {};
            props.stype = ZES_STRUCTURE_TYPE_RAS_PROPERTIES;
            if (zesRasGetProperties(phRasErrorSets[rasIndex], &props) ==
ZE_RESULT_SUCCESS)
               var pErrorType
               switch (props.type)
                  case ZES_RAS_ERROR_TYPE_CORRECTABLE:
                     pErrorType = "Correctable"
                  case ZES_RAS_ERROR_TYPE_UNCORRECTABLE:
                     pErrorType = "Uncorrectable"
                  default:
                     pErrorType = "Unknown"
               output("RAS %s errors", pErrorType)
               if (props.onSubdevice)
                  output("    On sub-device: %u", props.subdeviceId)
               output("    RAS supported: %s", props.supported ? "yes" : "no")
               output("    RAS enabled: %s", props.enabled ? "yes" : "no")
               if (props.supported and props.enabled)
                  zes_ras_state_t errorDetails
                  if (zesRasGetState(phRasErrorSets[rasIndex], 1, &errorDetails)
                     == ZE_RESULT_SUCCESS)
                        uint64_t numErrors = 0
                        for (int i = 0; i < ZES_RAS_ERROR_CAT_MAX; i++)
                           numErrors += errorDetails.category[i];
                        output("    Number new errors: %llun", (long long unsigned
int)numErrors);
                        if (numErrors)
                           call_function OutputRasDetails(&errorDetails)
   free_memory(...)

function OutputRasDetails(zes_ras_state_t* pDetails)
   output("       Number new resets:              %llu", pDetails-
>category[ZES_RAS_ERROR_CAT_RESET])
   output("       Number new programming errors:    %llu", pDetails-
>category[ZES_RAS_ERROR_CAT_PROGRAMMING_ERRORS])
```

```
    output("       Number new driver errors:        %llu", pDetails-
>category[ZES_RAS_ERROR_CAT_DRIVER_ERRORS])
    output("       Number new compute errors:       %llu", pDetails-
>category[ZES_RAS_ERROR_CAT_COMPUTE_ERRORS])
    output("       Number new non-compute errors:    %llu", pDetails-
>category[ZES_RAS_ERROR_CAT_NON_COMPUTE_ERRORS])
    output("       Number new cache errors:         %llu", pDetails-
>category[ZES_RAS_ERROR_CAT_CACHE_ERRORS])
    output("       Number new display errors:       %llu", pDetails-
>category[ZES_RAS_ERROR_CAT_DISPLAY_ERRORS])
```

診断の実行

診断は、ハードウェアに自己チェックと修復を実行するように要求するプロセスです。

警告: 診断を実行すると、現在のデバイスの状態が破棄される可能性があります。開始する前にすべてのワークロードを停止することが重要です。

これは `zesDiagnosticsRunTests()` 関数で実現されます。関数から戻ると、ソフトウェアは診断リターンコード（`zes_diag_result_t`）を使用して新しいアクションを決定できます:

1. `ZES_DIAG_RESULT_NO_ERRORS` - エラーは検出されず、ワークロードはハードウェアへの送信を再開できます。
2. `ZES_DIAG_RESULT_ABORT` - ハードウェアで診断テストの実行に問題が発生しました。
3. `ZES_DIAG_RESULT_FAIL_CANT_REPAIR` - ハードウェアの修復設定に問題がありました。カードをシステムから削除する必要があります。
4. `ZES_DIAG_RESULT_REBOOT_FOR_REPAIR` - ハードウェアは修復の準備が整っており、再起動が必要です。その後、ワークロードの送信を再開できます。

`zesDeviceGetState()` 関数を使用して、デバイスが修復されたか判断できます。

実行できる診断テストスイートは複数あります。 `zesDeviceEnumDiagnosticTestSuites()` 関数は利用可能な各テスト スイートを列挙し、`zesDiagnosticsGetProperties()` 関数を使用して各テストスイートの名前（`zes_diag_properties_t.name`）を判別できます。

各テストスイートには 1 つ以上の診断テストが含まれています。一部のシステムでは、テストのサブセットのみを実行できます。この機能が使用可能であるか判断するには、`zesDiagnosticsGetProperties()` 関数を使用し、`zes_diag_properties_t.haveTests` が `true` であることを確認します。`True` の場合、`zesDiagnosticsGetTests()` 関数を呼び出して、実行できる個々のテストのリストを取得できます。

`zesDiagnosticsRunTests()` を使用してテストスイートの診断を実行する場合、スイート内のテストの開始とインデックスを指定できます。`ZES_DIAG_FIRST_TEST_INDEX` および `ZES_DIAG_LAST_TEST_INDEX` に設定すると、スイート内のすべてのテストが実行されます。テストのサブセットを実行できる場合、テスト開始とテスト終了のインデックスを指定します。この範囲のインデックスを持つすべてのテストが実行されます。

以下の表は、すべての診断管理関数をまとめたものです:

関数	説明
zesDeviceEnumDiagnosticTestSuites()	実行可能な診断テストスイートへのハンドルを取得します。
zesDiagnosticsGetProperties()	テストスイートに関する情報（タイプ、名前、場所、個々のテストを実行できるか）を取得します。
zesDiagnosticsGetTests()	実行できる個々の診断テストのリストを取得します。
zesDiagnosticsRunTests()	すべての診断テストまたは個別の診断テストを実行します。

以下の疑似コードは、すべてのテストスイートと各スイート内のテストを検出する方法を示しています:

```
function ListDiagnosticTests(zes_device_handle_t hSysmanDevice)
{
   uint32_t numTestSuites
   if ((zesDeviceEnumDiagnosticTestSuites(hSysmanDevice, &numTestSuites, NULL) ==
ZE_RESULT_SUCCESS))
      zes_diag_handle_t* phTestSuites =
         allocate_memory(numTestSuites * sizeof(zes_diag_handle_t))
      if (zesDeviceEnumDiagnosticTestSuites(hSysmanDevice, &numTestSuites, phTestSuites) ==
ZE_RESULT_SUCCESS)
         for (suiteIndex = 0 .. numTestSuites-1)
            uint32_t numTests = 0
            zes_diag_test_t* pTests
            zes_diag_properties_t suiteProps {};
            suiteProps.stype = ZES_STRUCTURE_TYPE_DIAG_PROPERTIES;
            if (zesDiagnosticsGetProperties(phTestSuites[suiteIndex], &suiteProps) !=
ZE_RESULT_SUCCESS)
               next_loop(suiteIndex)
            output("Diagnostic test suite %s:", suiteProps.name)
            if (!suiteProps.haveTests)
               output("    There are no individual tests that can be selected.")
               next_loop(suiteIndex)
            if (zesDiagnosticsGetTests(phTestSuites[suiteIndex], &numTests, NULL) !=
ZE_RESULT_SUCCESS)
               output("    Problem getting list of individual tests.")
               next_loop(suiteIndex)
            pTests = allocate_memory(numTests * sizeof(zes_diag_test_t*))
            if (zesDiagnosticsGetTests(phTestSuites[suiteIndex], &numTests, pTests) !=
ZE_RESULT_SUCCESS)
               output("    Problem getting list of individual tests.")
               next_loop(suiteIndex)
            for (i = 0 .. numTests-1)
```

```
                output("    Test %u: %s", pTests[i].index, pTests[i].name)
    free_memory(...)
```

イベント

イベントは、デバイス上で何らかの変更が発生したかどうかを判断する手段です（例：新しい RAS エラーの発生）。アプリケーションは、通知を受信したいイベントを登録し、通知を受信するためにクエリーを実行します。クエリーはブロッキング待機を要求できます。これにより、新しい通知が受信されるまで、呼び出し元のアプリケーション・スレッドがスリープ状態になります。

アプリケーションがイベントを受信するデバイスごとに、次のアクションを実行する必要があります:

1. `zesDeviceEventRegister()` を使用して、監視するイベントを指定します。
2. 各イベントについて、必要に応じてデバイス・コンポーネント関数を呼び出して、イベントをトリガーする条件を設定します。

最後に、アプリケーションは、イベントを監視するデバイスハンドルのリストを使用して `zesDriverEventListen()` を呼び出します。待機タイムアウトは、非ブロッキング操作（タイムアウト = 0）またはブロッキング操作（タイムアウト = `UINT32_MAX`）を要求するため、またはイベントが受信されなくても指定された時間後にリターンするために使用されます。

イベントなしで `zesDeviceEventRegister` を呼び出すと（引数イベントを“0”に設定）、監視されているすべてのイベントが登録解除されることに注意してください。アプリケーションに関数 `zesDriverEventListen()` でブロックされたスレッドがあり、監視するイベントが場合、関数はブロックを解除し、イベントカウント 0 でアプリケーション・スレッドに制御を戻します。

イベントが受信されると、`zesDriverEventListen()` 関数の呼び出しが完了するとイベントが返されます。これにより、どのデバイスがイベントを生成したか、および各デバイスのイベントタイプのリストが表示されます。その後、関連するデバイス・コンポーネント関数を使用して、変更された状態を判断するのはアプリケーションの役割です。たとえば、デバイスに対して RAS エラーイベントがトリガーされた場合、`zesRasGetState()` 関数を使用して RAS エラーカウンターのリストを取得します。

イベントのリストは以下の表に記載されています。イベントごとに、対応する構成と状態関数が表示されます。構成関数が表示されない場合、イベントは自動的に生成されます。構成関数が表示されていると、イベントを有効にしたり、しきい値条件を提供するため、その関数を呼び出す必要があります。

イベント	トリガー	設定関数	状態関数
`ZES_EVENT_TYPE_FLAG_DEVICE_DETACH`	デバイスはドライバーによってリセットされようとしています		
`ZES_EVENT_TYPE_FLAG_DEVICE_ATTACH`	デバイスはドライバーによってリセットされました		
`ZES_EVENT_TYPE_FLAG_DEVICE_SLEEP_STATE_ENTER`	デバイスはディープスリープ状態に入ります		
`ZES_EVENT_TYPE_FLAG_DEVICE_SLEEP_STATE_EXIT`	デバイスはディープスリープ状態を終了しています		

イベント	トリガー	設定関数	状態関数
ZES_EVENT_TYPE_FLAG_FREQ_THROTTLED	周波数が制限され始めます		zesFrequencyGetState()
ZES_EVENT_TYPE_FLAG_ENERGY_THRESHOLD_CROSSED	エネルギー消費しきい値に達しました	zesPowerSetEnergyThreshold()	
ZES_EVENT_TYPE_FLAG_TEMP_CRITICAL	臨界温度に達しました	zesTemperatureSetConfig()	zesTemperatureGetState()
ZES_EVENT_TYPE_FLAG_TEMP_THRESHOLD1	温度がしきい値 1 を超えました	zesTemperatureSetConfig()	zesTemperatureGetState()
ZES_EVENT_TYPE_FLAG_TEMP_THRESHOLD2	温度がしきい値 2 を超えました	zesTemperatureSetConfig()	zesTemperatureGetState()
ZES_EVENT_TYPE_FLAG_MEM_HEALTH	デバイスメモリーの健全性が変化しました		zesMemoryGetState()
ZES_EVENT_TYPE_FLAG_FABRIC_PORT_HEALTH	ファブリック・ポートの健全性が変化しました		zesFabricPortGetState()
ZES_EVENT_TYPE_FLAG_RAS_CORRECTABLE_ERRORS	RAS 訂正可能エラーがしきい値を超えました	zesRasSetConfig()	zesRasGetState()
ZES_EVENT_TYPE_FLAG_RAS_UNCORRECTABLE_ERRORS	RAS 訂正不能エラーがしきい値を超えました	zesRasSetConfig()	zesRasGetState()
ZES_EVENT_TYPE_FLAG_DEVICE_RESET_REQUIRED	Driver has determined that an immediate reset is required		zesDeviceGetState()

zesDriverEventListen() の呼び出しには、ドライバーハンドルとデバイスハンドルのリストが必要です。デバイスハンドルは、そのドライバーから列挙されている必要があります。そうでない場合はエラーが返されます。アプリケーションが複数のドライバーでデバイスを管理している場合、ドライバーごとにこの関数を個別に呼び出す必要があります。

以下の表は、すべてのイベント管理関数をまとめたものです:

関数	説明
zesDeviceEventRegister()	特定のイベントハンドルに登録するイベントを設定します。
zesDriverEventListen()	指定されたデバイスのリストにイベントが到着するのを待機します。

以下の疑似コードは、温度が指定されたしきい値を超えたとき、または臨界温度に達したときにイベントをトリガーするようにすべての温度センサーを構成する方法を示しています。

```
function WaitForExcessTemperatureEvent(zes_driver_handle_t hDriver, double tempLimit)
{
    # これにはイベントを監視するデバイスの数が含まれます
    var numListenDevices = 0

    # このドライバ下にあるすべてのデバイスのリストを取得
    uint32_t deviceCount = 0
    zeDeviceGet(hDriver, &deviceCount, nullptr)
    # すべてのデバイスハンドルにメモリーを割り当て
    ze_device_handle_t* phDevices =
        allocate_memory(deviceCount * sizeof(ze_device_handle_t))

    # 温度イベントを監視するデバイスにメモリーを割り当て
    zes_device_handle_t* phListenDevices =
        allocate_memory(deviceCount * sizeof(zes_device_handle_t))
    # phListenDevices 内の各デバイスから受信したイベントにメモリーを割り当て
    zes_event_type_flags_t* pDeviceEvents =
        allocate_memory(deviceCount * sizeof(zes_event_type_flags_t))
    # phListenDevices のデバイスハンドルをデバイス・インデックスにマッピングできるようにメモリーを割り当て
    uint32_t* pListenDeviceIndex = allocate_memory(deviceCount * sizeof(uint32_t))

    # すべてのデバイスハンドルを取得
    zeDeviceGet(hDriver, &deviceCount, phDevices)
    for(devIndex = 0 .. deviceCount-1)
        # デバイスの Sysman ハンドルを取得
        zes_device_handle_t hSysmanDevice = (zes_device_handle_t)phDevices[devIndex]

        # すべての温度センサーのハンドルを取得
        uint32_t numTempSensors = 0
        if (zesDeviceEnumTemperatureSensors(hSysmanDevice, &numTempSensors, NULL) !=
ZE_RESULT_SUCCESS)
            next_loop(devIndex)
        zes_temp_handle_t* allTempSensors
            allocate_memory(deviceCount * sizeof(zes_temp_handle_t))
        if (zesDeviceEnumTemperatureSensors(hSysmanDevice, &numTempSensors, allTempSensors)
== ZE_RESULT_SUCCESS)
            # 各温度センサーを設定して、重大なイベントとしきい値 1 イベントをトリガー
            var numConfiguredTempSensors = 0
            for (tempIndex = 0 .. numTempSensors-1)
                if (zesTemperatureGetConfig(allTempSensors[tempIndex], &config) !=
ZE_RESULT_SUCCESS)
```

```
                next_loop(tempIndex)
            zes_temp_config_t config
            config.enableCritical = true
            config.threshold1.enableHighToLow = false
            config.threshold1.enableLowToHigh = true
            config.threshold1.threshold = tempLimit
            config.threshold2.enableHighToLow = false
            config.threshold2.enableLowToHigh = false
            if (zesTemperatureSetConfig(allTempSensors[tempIndex], &config) ==
ZE_RESULT_SUCCESS)
                numConfiguredTempSensors++

        # イベントを生成するようにセンサーを設定した場合、このデバイスで受信するように登録します
        if (numConfiguredTempSensors)
            if (zesDeviceEventRegister(phDevices[devIndex],
                ZES_EVENT_TYPE_FLAG_TEMP_CRITICAL | ZES_EVENT_TYPE_FLAG_TEMP_THRESHOLD1)
                == ZE_RESULT_SUCCESS)
            phListenDevices[numListenDevices] = hSysmanDevice
            pListenDeviceIndex[numListenDevices] = devIndex
            numListenDevices++

    # どのデバイスでもイベントを受信するように登録した場合は、今すぐリスニングき始めます
    if (numListenDevices)
        # イベントを受信するまでブロック
        uint32_t numEvents
        if (zesDriverEventListen(hDriver, UINT32_MAX, numListenDevices, phListenDevices,
&numEvents, pDeviceEvents)
            == ZE_RESULT_SUCCESS)
            if (numEvents)
            for (evtIndex .. numListenDevices)
                if (pDeviceEvents[evtIndex] & ZES_EVENT_TYPE_FLAG_TEMP_CRITICAL)
                    output("Device %u: Went above the critical temperature.",
                        pListenDeviceIndex[evtIndex])
                else if (pDeviceEvents[evtIndex] & ZES_EVENT_TYPE_FLAG_TEMP_THRESHOLD1)
                    output("Device %u: Went above the temperature threshold %f.",
                        pListenDeviceIndex[evtIndex], tempLimit)

    free_memory(...)
```

生存モード

生存モードは、重大な障害が発生した場合にドライバーがファームウェアのアップグレードが可能な状態になるよう

に設計されています。これにより、デバイスの起動に失敗した状態であってもシステムの回復が可能です。

レベルゼロ Sysman API を使用して生存モードを検出し、回復するフローは以下のとおりです。

```
zesInit(0);
uint32_t driversCount = 1;
zes_driver_handle_t driver;
zesDriverGet(&driversCount, &driver);

uint32_t deviceCount = 0;
zesDeviceGet(driver, &deviceCount, nullptr)
zes_device_handle_t* hSysmanDevices = allocate_memory(deviceCount *
sizeof(zes_device_handle_t))
zesDeviceGet(driver, &deviceCount, hSysmanDevices);

# 生存モード検出
for(devIndex = 0 .. deviceCount-1){
    ze_device_properties_t device_properties {};
    device_properties.stype = ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;

    result = zeDeviceGetProperties(hSysmanDevices[devIndex], &device_properties);
    if(result == ZE_RESULT_ERROR_SURVIVABILITY_MODE_DETECTED){
            # デバイスは生存モードです。デバイスを回復するにはファームウェア・イメージをフラッシュします
            # zesDeviceEnumFirmwares()、zesFirmwareFlash() API を使用したリカバリー
    }
}
free_memory(...)
```

セキュリティー

Linux

アクセラレーター・ドライバーによって提供されるデフォルトのセキュリティーでは、UNIX ユーザー **root** に対してシステムリソースの照会と制御を許可し、UNIX グループ **root** のメンバーであるユーザーのみを照会し、他のユーザーはアクセスできないようにします。一部のクエリーはどのユーザーでも許可されます（例: 現在の周波数の要求、スタンバイ状態の確認）。

これらの権限を緩和または厳しくするのは、Linux ディストリビューションまたはシステム管理者の責任です。これは通常、udev デーモンルールを追加することによって行われます。たとえば、多くの Linux ディストリビューションには次のようなルールがあります:

```
root    video   /dev/dri/card0
```

これにより、UNIX グループ **video** のすべてのユーザーがシステムリソースに関する情報を照会できるようにな

ります。ビデオグループのユーザーに制御アクセスを開放するには、関連する制御ごとに udev ルールを追加する必要があります。たとえば、ビデオグループ内の誰かがスタンバイを無効にできるようにするには、次の udev デーモンルールが必要になります:

```
chmod g+w /sys/class/drm/card0/rc6_enable
```

API で使用される `sysfs` ファイルの完全なリストを以下に示します。各ファイルについて、影響を受ける API 関数のリストが示されます。

sysfs ファイル	説明	関数
/sys/class/drm/card0/rc6_enable	スタンバイを有効/無効にするために使用されます。	zesDeviceEnumStandbyDomains() zesStandbyGetProperties() zesStandbyGetMode() zesStandbySetMode()
TBD	開発中	TBD

Windows

Windows ドライバーは、管理者権限を持つユーザーからのテレメトリー要求のみを許可します。
LocalServiceSid 権限を持つシステムサービスの制御のみを許可します。

仮想化

仮想化環境では、API 機能にアクセスできるのはホストのみです。仮想マシンで API を使用すると失敗します。

SPIR-V プログラミング・ガイド

はじめに

SPIR-V（英語）は、並列計算カーネルを表現できるオープンでロイヤリティフリーの標準中間言語です。SPIR-V は複数の実行環境に適応できます。SPIR-V モジュールは、クライアント API で指定された実行環境によって使用されます。このドキュメントでは、‘oneAPI’ レベルゼロ API の SPIR-V 実行環境について説明します。SPIR-V 実行環境では、一部の SPIR-V 機能に必要なサポート、一部の SPIR-V 命令の追加セマンティクス、および SPIR-V バイナリーモジュールが有効と見なされるため、遵守する必要がある追加の検証ルールについて説明します。

このドキュメントは、‘oneAPI’ レベルゼロ API で使用することを目的とした SPIR-V モジュールを生成するコンパイラー開発者、‘oneAPI’ レベルゼロ API の実装者、および ‘oneAPI’ レベルゼロ API で SPIR-V モジュールを使用するソフトウェア開発者向けに記述されています。

共通のプロパティー

このセクションでは、SPIR-V モジュールを使用するすべての ‘oneAPI’ レベルゼロ 環境に共通するプロパティーについて説明します。

SPIR-V モジュールは、ホストのエンディアンで 32 ビットワードの連続として解釈され、リテラル文字列は SPIR-V 仕様書に記載されている方法でパッキングされます。SPIR-V モジュールの最初の数ワードは、SPIR-V 仕様書に記載されている通り、マジックナンバーと SPIR-V バージョン番号でなければなりません。

サポートされている SPIR-V バージョン

デバイスでサポートされる最大の SPIR-V バージョンは、`ze_device_module_properties_t.spirvVersionSupported` で記述されます。

拡張命令セット

OpenCL の拡張命令セット（英語）である **OpenCL.std** がサポートされています。

ソース言語のエンコード

ソース言語バージョンは純粋に情報提供のみを目的としており、意味はありません。

数値タイプの形式

浮動小数点タイプは、IEEE-754（英語）セマンティクスを使用して表現および保存されます。すべての整数形式は 2 の補数形式を使用して表現および保存されます。

サポートされるタイプ

以下のタイプがサポートされています。一部のタイプでは追加の機能が必要になる場合があり、すべての環境でサポートされない場合があることに注意してください。

基本的なスカラータイプとベクトルタイプ

OpTypeVoid がサポートされます。

次のスカラータイプがサポートされています:

- **OpTypeBool**
- **OpTypeInt**、*幅*は 8、16、32、または 64、*符号*は 0 で、符号セマンティクスがないことを示します。
- **OpTypeFloat**、*幅*は 16、32、または 64 です。

OpTypeVector ベクトルタイプがサポートされています。ベクトル・*コンポーネント・タイプ*は、上記のいずれかのスカラータイプになります。サポートされるベクトル・*コンポーネント数*は 2、3、4、8、または 16 です。

`OpTypeArray` 配列タイプ、`OpTypeStruct` 構造体タイプ、`OpTypeFunction` 関数、および `OpTypePointer` ポインタータイプがサポートされています。

イメージ関連のデータタイプ

次の表は、サポートされている **OpTypeImage** イメージのタイプを示しています:

次元	深度	配列	説明
1D	0	0	1D イメージ。
1D	0	1	1D イメージ配列。
2D	0	0	2D イメージ。
2D	1	0	2D 深度イメージ。
2D	0	1	2D イメージ配列。
2D	1	1	2D 深度イメージ配列。
3D	0	0	3D イメージ。
バッファー	0	0	1D バッファーイメージ。

OpTypeSampler サンプラータイプがサポートされています。

カーネル

OpEntryPoint で識別される SPIR-V モジュール内の **OpFunction** は、ホスト API インターフェイスで起動できるカーネルを定義します。

カーネルの戻り値タイプ

`OpEntryPoint` で識別される `OpFunction` の*結果タイプ*は `OpTypeVoid` である必要があります。

カーネル引数

OpEntryPoint で識別される **OpFunction** の **OpFunctionParameter** は、カーネル引数を定義します。カーネル引数に許可されるタイプは次のとおりです:

- **OpTypeInt**
- **OpTypeFloat**

- **OpTypeStruct**
- **OpTypeVector**
- **OpTypePointer**
- **OpTypeSampler**
- **OpTypeImage**

OpTypeInt パラメーターの場合、サポートされる*幅*は 8、16、32、および 64 であり、符号セマンティクスを持ちません。

OpTypeFloat パラメーターでサポートされる*幅*は 16 と 32 です。

OpTypeStruct パラメーターでサポートされる構造体の*メンバータイプ*は次のとおりです:

- **OpTypeInt**
- **OpTypeFloat**
- **OpTypeStruct**
- **OpTypeVector**
- **OpTypePointer**

OpTypePointer パラメーターでサポートされている*ストレージクラス*は次のとおりです:

- **CrossWorkgroup**
- **Workgroup**
- **UniformConstant**

拡張機能またはオプション機能をサポートする環境では、エントリーポイントのパラメーター・リストに追加タイプが許可される場合があります。

要件

SPIR-V 1.0

SPIR-V 1.0 をサポートする環境は、次の機能を宣言する SPIR-V 1.0 モジュールをサポートする必要があります:

- **Addresses**
- **Float16Buffer**
- **Int64**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Vector16**
- **GenericPointer**
- **Groups**
- **ImageBasic**（ze_device_image_properties_t.supported をサポートするデバイスの場合）
- **Float16**（ZE_DEVICE_MODULE_FLAG_FP16 をサポートするデバイスの場合）

- **Float64**（ZE_DEVICE_MODULE_FLAG_FP64 をサポートするデバイスの場合）
- **Int64Atomics**（ZE_DEVICE_MODULE_FLAG_INT64_ATOMICS をサポートするデバイスの場合）

‘oneAPI’ 環境が **ImageBasic** 機能をサポートしている場合、次の機能もサポートする必要があります:

- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**
- **ImageReadWrite**

SPIR-V 1.1

SPIR-V 1.1 をサポートする環境は、上記の SPIR-V 1.0 モジュールに必要な機能を宣言する SPIR-V 1.1 モジュールをサポートする必要があります。

SPIR-V 1.1 では、新たに必須となる機能は追加されていません。

SPIR-V 1.2

SPIR-V 1.2 をサポートする環境は、上記の SPIR-V 1.1 モジュールに必要な機能を宣言する SPIR-V 1.2 モジュールをサポートする必要があります。

SPIR-V 1.2 では、新たに必須となる機能は追加されていません。

検証ルール

以下は、すべての ‘oneAPI’ レベルゼロ環境で実行される SPIR-V モジュールに適用される検証ルールのリストです:

OpEntryPoint で宣言される*実行モデル*は **Kernel** である必要があります。

OpMemoryModel で宣言される*アドレス指定モデル*は、デバイスポインターが 64 ビットであることを示す **Physical64** である必要があります。

OpMemoryModel で宣言される*メモリーモデル*は **OpenCL** である必要があります。

すべての **OpTypeInt** 整数タイプ宣言命令について:

- *Signedness* は 0 でなければならず、これは符号付きセマンティクスがないことを示します。

すべての **OpTypeImage** タイプ宣言命令について: * *Sampled Type* は **OpTypeVoid** でなければなりません。* *Sampled* は 0 でなければなりません。これは、イメージの使用方法がコンパイル時ではなく実行時に判明することを示します。* *MS* は 0 でなければなりません。これは単一サンプリングされたコンテンツを示します。* *Arrayed* は、*Dim* が **1D** または **2D** に設定されている場合にのみ 1 に設定でき、コンテンツの配列を示します。* *Image Format* は **Unknown** である必要があります。これは、イメージに指定された形式がないことを示します。* オプションのイメージ*アクセス修飾子*は存在する必要があります。

イメージ書き込み命令 **OpImageWrite** には、オプションの*イメージオペランド*を含めることはできません。

イメージ読み取り命令 **OpImageRead** および **OpImageSampleExplicitLod** には、オプションの*イメージオペランド* **ConstOffset** を含めることはできません。

すべてのアトミック命令について:

- *結果タイプ*および/または*値*タイプでは、32 ビットの整数タイプがサポートされています。ZE_DEVICE_MODULE_FLAG_INT64_ATOMICS をサポートするデバイスでは、*結果タイプ*や*値*タイプとして 64 ビット整数タイプがオプションでサポートされます。
- *ポインター*オペランドは、**関数**、**ワークグループ**、**クロスワークグループ**、または**汎用***ストレージクラス*へのポインターである必要があります。

再帰はサポートされていません。エントリーポイントの静的関数呼び出しグラフにはサイクルを含めることはできません。

還元不可能な制御フローが合法かどうかは実装によって定義されます。

OpGroupAsyncCopy および OpGroupWaitEvents 命令の場合、*実行スコープ*は次のようになります:

- **Workgroup**

その他のすべての命令の*実行スコープ*は次のいずれかである必要があります:

- **Workgroup**
- **Subgroup**

*メモリースコープ*は次のいずれかである必要があります:

- **CrossDevice**
- **Device**
- **Workgroup**
- **Invocation**
- **Subgroup**

拡張機能

インテル・サブグループ

‘oneAPI’ レベルゼロ API 環境は、**OpExtension** を介して SPV_INTEL_subgroups 拡張機能を宣言する SPIR-V モジュールを受け入れる必要があります。

OpExtension を介してモジュールで SPV_INTEL_subgroups 拡張機能が宣言されている場合、環境は次の SPIR-V 機能を宣言するモジュールを受け入れる必要があります:

- **SubgroupShuffleINTEL**
- **SubgroupBufferBlockIOINTEL**
- **SubgroupImageBlockIOINTEL**

環境は、**SubgroupShuffleINTEL** 命令の*データ*として次のタイプを受け入れる必要があります:

- *次のコンポーネント・タイプ* の 2、4、8、または 16 個の*コンポーネント・カウント*を持つスカラーおよび **OpTypeVectors**:

    - *幅* 32 ビットの **OpTypeFloat**（float）
    - *幅* 8 ビット、*符号* 0 の **OpTypeInt**（char および uchar）
    - *幅* 16 ビット、*符号* 0 の **OpTypeInt**（short および ushort）
    - *幅* 32 ビット、*符号* 0 の **OpTypeInt**（int および uint）
- *幅* 64 ビット、*符号* 0 の **OpTypeInt** のスカラー（long および ulong）

さらに、**Float16** 機能が宣言されサポートされている場合:

- *幅* 16 ビットの **OpTypeFloat** のスカラー（half）

さらに、**Float64** 機能が宣言されサポートされている場合:

- 幅 64 ビットの **OpTypeFloat** のスカラー（double）

環境は、SubgroupBufferBlockIOINTEL および SubgroupImageBlockIOINTEL 命令の*結果*および*データ*に対して次のタイプを受け入れる必要があります:

- *次のコンポーネント・タイプ* の 2、4 または 8 個の*コンポーネント・カウント*を持つスカラーおよび **OpTypeVectors**:
    - *幅* 32 ビット、*符号* 0 の **OpTypeInt**（int および uint）
    - *幅* 16 ビット、*符号* 0 の **OpTypeInt**（short および ushort）

Ptr の場合、有効な*ストレージクラス*は次のとおりです:

- **CrossWorkGroup**（global）

*イメージ*の場合:

- *Dim* は *2D* である必要があります
- *Depth* は 0 である必要があります（深度イメージではありません）
- *Arrayed* は 0 である必要があります（配列化されていないコンテンツ）
- *MS* は 0 でなければなりません（単一サンプルコンテンツ）。

Coordinate では、次のタイプがサポートされています:

- *幅*が 32 ビット、*符号*が 0（int2）の*コンポーネント・タイプ* **OpTypeInt** の 2 つの*コンポーネント・カウント*を持つ **OpTypeVectors**

注意事項と制限事項

SubgroupShuffleINTEL 命令は、非均一制御フロー内に配置できるため、サブグループ内のすべての呼び出しで必ずしも実行される必要はありませんが、*データ*のシャッフルは **SubgroupShuffleINTEL** 命令を実行する呼び出し間でのみ行われれます。**SubgroupShuffleINTEL** 命令を実行しない呼び出しからの*データ*のシャッフルは、未定義の結果となります。

SubgroupShuffleINTEL 命令では、範囲外のシャッフル・インデックスの動作は定義されていません。

SubgroupBufferBlockIOINTEL 命令と **SubgroupImageBlockIOINTEL** 命令は、サブグループ内で均一に制御フロー内に配置された場合にのみ、正しく動作することが保証されます。これにより、いずれかの呼び出しが

これを実行すれば、すべての呼び出しがこれを実行することが保証されます。他の場所に配置した場合、動作は未定義になります。

`SubgroupBufferBlockIOINTEL` 命令では、範囲外の動作は定義されていません。

`SubgroupImageBlockIOINTEL` 命令は境界チェックをサポートしていますが、イメージ要素単位ではなく、`uints` 単位でイメージの幅に対し境界チェックを行います。これは次のことを意味します:

- イメージの*イメージ・フォーマット*・サイズが、`uint` のサイズ（4 バイト、たとえば **`Rgba8`**）に等しい場合、イメージは正しく境界チェックされます。この場合、範囲外の読み取りではエッジイメージ要素（**`ClampToEdge`** と同等）が返され、範囲外の書き込みは無視されます。
- イメージの*イメージ・フォーマット*・サイズが `uint` のサイズ（**R8** など）より小さい場合、イメージ全体をアドレス指定できますが、境界チェックは遅延して実行されます。そのため、範囲外の読み取りは無効なデータを返す可能性があり、範囲外の書き込みは他のイメージやバッファーを予期せず破損させる可能性があるため、範囲外の読み取りや書き込みを避けるように注意を払う必要があります。

`SubgroupBufferBlockIOINTEL` 命令には次の制限が適用されます:

- ポインター *`Ptr`* は、読み取りの場合は 32 ビット（4 バイト）にアライメントする必要があり、書き込みの場合は 128 ビット（16 バイト）にアライメントする必要があります。

`SubgroupImageBlockIOINTEL` 命令には次の制限が適用されます:

- 要素サイズが 4 バイトを超えるイメージ（**`Rgba32f`** など）の場合、**`SubgroupImageBlockIOINTEL`** 命令の動作は未定義です。

`OpSubgroupImageBlockWriteINTEL` 命令には次の制限が適用されます:

- 任意のバイトオフセットから読み取り可能なイメージブロック読み取り命令とは異なり、イメージ ブロック書き込み命令のバイト座標の x コンポーネントは 4 の倍数である必要があります。つまり、書き込みは 32 ビット境界から始まる必要があります。座標の y 成分には制限はありません。

浮動小数点アトミック

拡張機能 **`ZE_extension_float_atomics`** をサポートする‘oneAPI’レベルゼロ API 環境は、追加のアトミック命令、機能、およびタイプをサポートする必要があります。

アトミックロード、ストア、交換

‘oneAPI’レベルゼロ API 環境が拡張機能 **`ZE_extension_float_atomics`** をサポートし、`ze_device_fp_atomic_ext_flags_t`.`fp16Flags` に `ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_LOAD_STORE` または `ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_LOAD_STORE` が含まれている場合、**アトミック命令 `OpAtomicLoad`、`OpAtomicStore`、**および **`OpAtomicExchange`** は次のようになります:

- *結果タイプ*および/または*値*タイプでは、16 ビットの浮動小数点タイプがサポートされています。
- `ze_device_fp_atomic_ext_flags_t`.`fp16Flags` に `ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_LOAD_STORE` が含まれる場合、*ポインター*オペランドは **`CrossWorkGroup`** *ストレージクラス*へのポインターである可能性があります。

- ze_device_fp_atomic_ext_flags_t.fp16Flags に ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_LOAD_STORE が含まれる場合、*ポインター*オペランドは **Workgroup** *ストレージクラス*へのポインターである可能性があります。

アトミック加算と減算

'oneAPI' レベルゼロ API 環境が拡張機能 **ZE_extension_float_atomics** をサポートし、ze_device_fp_atomic_ext_flags_t.fp16Flags、ze_device_fp_atomic_ext_flags_t.fp32Flags、または ze_device_fp_atomic_ext_flags_t.fp64Flags に ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_ADD または ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_ADD が含まれる場合、環境は拡張機能 SPV_EXT_shader_atomic_float_add および SPV_EXT_shader_atomic_float16_add の使用を宣言するモジュールを受け入れる必要があります。

追加:

- ze_device_fp_atomic_ext_flags_t.fp16Flags に ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_ADD または ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_ADD が含まれている場合、**AtomicFloat16AddEXT** 機能がサポートされている必要があります。
- ze_device_fp_atomic_ext_flags_t.fp32Flags に ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_ADD または ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_ADD が含まれている場合、**AtomicFloat32AddEXT** 機能がサポートされている必要があります。
- ze_device_fp_atomic_ext_flags_t.fp64Flags に ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_ADD または ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_ADD が含まれている場合、**AtomicFloat64AddEXT** 機能がサポートされている必要があります。
- これらの拡張機能によって追加された**アトミック命令** **OpAtomicFAddEXT** の場合:
  - ze_device_fp_atomic_ext_flags_t.fp32Flags、ze_device_fp_atomic_ext_flags_t.fp64Flags、または ze_device_fp_atomic_ext_flags_t.fp16Flags に ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_ADD が含まれる場合、ポインターオペランドは **CrossWorkGroup** *ストレージクラス*へのポインターである可能性があります。
  - ze_device_fp_atomic_ext_flags_t.fp32Flags、ze_device_fp_atomic_ext_flags_t.fp64Flags、または ze_device_fp_atomic_ext_flags_t.fp16Flags に ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_LOAD_STORE が含まれる場合、*ポインター*オペランドは **Workgroup** *ストレージクラス*へのポインターである可能性があります。

アトミック最小値と最大値

'oneAPI' レベルゼロ API 環境が拡張機能 **ZE_extension_float_atomics** をサポートし、ze_device_fp_atomic_ext_flags_t.fp32Flags、

ze_device_fp_atomic_ext_flags_t.fp64Flags、または
ze_device_fp_atomic_ext_flags_t.fp16Flags ビットフィールドに
ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_MIN_MAX または
ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_MIN_MAX が含まれる場合、環境は拡張機能
SPV_EXT_shader_atomic_float_min_max の使用を宣言するモジュールを受け入れる必要があります。

追加:

- ze_device_fp_atomic_ext_flags_t.fp32Flags に
  ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_MIN_MAX または
  ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_MIN_MAX が含まれている場合、
  AtomicFloat32MinMaxEXT 機能がサポートされている必要があります。
- ze_device_fp_atomic_ext_flags_t.fp64Flags に
  ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_MIN_MAX または
  ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_MIN_MAX が含まれている場合、
  AtomicFloat64MinMaxEXT 機能がサポートされている必要があります。
- ze_device_fp_atomic_ext_flags_t.fp16Flags に
  ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_MIN_MAX または
  ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_MIN_MAX が含まれている場合、
  AtomicFloat16MinMaxEXT 機能がサポートされている必要があります。
- これらの拡張機能によって追加された**アトミック命令** **OpAtomicFMinEXT** および **OpAtomicFMaxEXT** の場合:
  - ze_device_fp_atomic_ext_flags_t.fp16Flags、
    ze_device_fp_atomic_ext_flags_t.fp32Flags、または
    ze_device_fp_atomic_ext_flags_t.fp64Flags に
    ZE_DEVICE_FP_ATOMIC_EXT_FLAG_GLOBAL_MIN_MAX が含まれる場合、*ポインター*オペランドは
    CrossWorkGroup *ストレージクラス*へのポインターである可能性があります。
  - ze_device_fp_atomic_ext_flags_t.fp16Flags、
    ze_device_fp_atomic_ext_flags_t.fp32Flags、または
    ze_device_fp_atomic_ext_flags_t.fp64Flags に
    ZE_DEVICE_FP_ATOMIC_EXT_FLAG_LOCAL_MIN_MAX が含まれる場合、*ポインター*オペランドは
    Workgroup *ストレージクラス*へのポインターである可能性があります。

拡張サブグループ

拡張機能 **ZE_extension_subgroups** をサポートする‘oneAPI’レベルゼロ API 環境は、追加のサブグループ命令、機能、およびタイプをサポートする必要があります。

拡張タイプ

次のグループ命令は、**Subgroup** に等しい*実行スコープ*でサポートされる必要があります:

- **OpGroupBroadcast**
- **OpGroupIAdd**、**OpGroupFAdd**
- **OpGroupSMin**、**OpGroupUMin**、**OpGroupFMin**

- **OpGroupSMax**、**OpGroupUMax**、**OpGroupFMax**

これらの命令について、*値*の有効なタイプは次のとおりです:

- サポートされるタイプのスカラー:
  - **OpTypeInt**（char、uchar、short、ushort、int、uint、long、ulong と等価）
  - **OpTypeFloat**（half、float、double と等価）

さらに、**OpGroupBroadcast** の場合、*値*の有効なタイプは次のとおりです:

- サポートされているタイプの 2、3、4、8、または 16 個のコンポーネント・カウントを持つ **OpTypeVectors**:
  - **OpTypeInt**（charn、ucharn、shortn、ushortn、intn、uintn、ongn、ulongn と等価）
  - **OpTypeFloat**（halfn、floatn、doublen と等価）

Vote

次の機能をサポートする必要があります:

- **GroupNonUniform**
- **GroupNonUniformVote**

これらの機能を必要とする命令の*実行スコープ*は次のようになります:

- **Subgroup**

OpGroupNonUniformAllEqual 命令について、*値*の有効なタイプは次のとおりです:

- サポートされるタイプのスカラー:
  - **OpTypeInt**（char、uchar、short、ushort、int、uint、long、ulong と等価）
  - **OpTypeFloat**（half、float、double と等価）

Ballot

次の機能をサポートする必要があります:

- **GroupNonUniformBallot**

これらの機能を必要とする命令の*実行スコープ*は次のようになります:

- **Subgroup**

非均一ブロードキャスト **OpGroupNonUniformBroadcast** 命令について、*値*の有効なタイプは次のとおりです:

- サポートされるタイプのスカラー:
  - **OpTypeInt**（char、uchar、short、ushort、int、uint、long、ulong と等価）
  - **OpTypeFloat**（half、float、double と等価）
- サポートされているタイプの 2、3、4、8、または 16 個のコンポーネント・カウントを持つ **OpTypeVectors**:
  - **OpTypeInt**（charn、ucharn、shortn、ushortn、intn、uintn、longn、ulongn と等価）
  - **OpTypeFloat**（halfn、floatn、doublen と等価）

`OpGroupNonUniformBroadcastFirst` 命令について、*値*の有効なタイプは次のとおりです:

- サポートされるタイプのスカラー:
  - **`OpTypeInt`** (char、uchar、short、ushort、int、uint、long、ulong と等価)
  - **`OpTypeFloat`** (half、float、double と等価)

`OpGroupNonUniformBallot` 命令の場合、有効な結果タイプは、**`OpTypeInt`** の 4 つのコンポーネント・カウントを持ち、*幅*が 32、*符号*が 0 (`uint4` に相当) に等しい `OpTypeVector` です。

`OpGroupNonUniformInverseBallot`、`OpGroupNonUniformBallotBitExtract`、`OpGroupNonUniformBallotBitCount`、`OpGroupNonUniformBallotFindLSB`、および `OpGroupNonUniformBallotFindMSB` 命令の場合、*値*の有効なタイプは、`OpTypeInt` の 4 つの*コンポーネント・カウント*を持ち、*幅*が 32、*符号* が 0 (uint4 に相当) である `OpTypeVector` です。

`SubgroupEqMask`、**`SubgroupGeMask`**、**`SubgroupGtMask`**、**`SubgroupLeMask`**、または SubgroupLtMask で装飾されたビルトイン変数の場合、**値**の有効なタイプは、**`OpTypeInt`** の 4 つの*コンポーネント・カウント*を持ち、*幅*が 32、*符号* が 0 (`uint4` に相当) である `OpTypeVector` です。

非均一算術演算

次の機能をサポートする必要があります:

- **`GroupNonUniformArithmetic`**

これらの機能を必要とする命令の*実行スコープ*は次のようになります:

- **`Subgroup`**

`OpGroupNonUniformLogicalAnd`、`OpGroupNonUniformLogicalOr`、および `OpGroupNonUniformLogicalXor` 命令の場合、*値*の有効タイプは `OpTypeBool` です。

それ以外の場合、**`GroupNonUniformArithmetic`** スキャンおよびリダクション命令の場合、*値*の有効なタイプは次のとおりです:

- サポートされるタイプのスカラー:
  - **`OpTypeInt`** (char、uchar、short、ushort、int、uint、long、ulong と等価)
  - **`OpTypeFloat`** (half、float、double と等価)

`GroupNonUniformArithmetic` スキャンおよびリダクション命令の場合、オプションの *`ClusterSize`* オペランドは必要ありません。

シャッフル

次の機能をサポートする必要があります:

- **`GroupNonUniformShuffle`**

これらの機能を必要とする命令の*実行スコープ*は次のようになります:

- **`Subgroup`**

これらの機能を必要とする命令 `OpGroupNonUniformShuffle` および `OpGroupNonUniformShuffleXor`

の場合、有効な*値*のタイプは次のとおりです:

- サポートされるタイプのスカラー:
  - **OpTypeInt**（char、uchar、short、ushort、int、uint、long、ulong と等価）
  - **OpTypeFloat**（half、float、double と等価）

相対シャッフル

次の機能をサポートする必要があります:

- **GroupNonUniformShuffleRelative**

これらの機能を必要とする命令の*実行スコープ*は次のようになります:

- **Subgroup**

GroupNonUniformShuffleRelative 命令について、*値*の有効なタイプは次のとおりです:

- サポートされるタイプのスカラー:
  - **OpTypeInt**（char、uchar、short、ushort、int、uint、long、ulong と等価）
  - **OpTypeFloat**（half、float、double と等価）

クラスター化されたリダクション

次の機能をサポートする必要があります:

- **GroupNonUniformClustered**

これらの機能を必要とする命令の*実行スコープ*は次のようになります:

- **Subgroup**

GroupNonUniformClustered 機能が宣言されている場合、**GroupNonUniformArithmetic** スキャンおよびリダクション命令にはオプションの *ClusterSize* オペランドを含めることができます。

Linkonce ODR

拡張機能 **ZE_extension_linkonce_odr** をサポートする‘oneAPI’レベルゼロ API 環境は、**OpExtension** を介して SPV_KHR_linkonce_odr 拡張機能を宣言する SPIR-V モジュールを受け入れる必要があります。

OpExtension を介してモジュールで SPV_KHR_linkonce_odr 拡張機能が宣言されている場合、環境は **LinkOnceODR** リンケージタイプを含むモジュールを受け入れる必要があります。

Bfloat16 変換

拡張機能 **ZE_extension_bfloat16_conversions** をサポートする‘oneAPI’レベルゼロ API 環境は、**OpExtension** を介して SPV_INTEL_bloat16_conversion 拡張機能を宣言する SPIR-V モジュールを受け入れる必要があります。

OpExtension を介してモジュールで SPV_INTEL_bloat16_conversion 拡張機能が宣言されている場合、環境は **Bfloat16ConversionINTEL** 機能を宣言するモジュールを受け入れる必要があります。

拡張機能によって追加された命令 OpConvertFToBF16INTEL および OpConvertBF16ToFINTEL の場合:

- *結果タイプ*、*浮動小数点値*、および *Bfloat16 値*の有効なタイプは、2、3、4、8、または 16 個の*コンポーネント・カウント*を持つスカラーと **`OpTypeVectors`** です

数値コンプライアンス

‘oneAPI’ レベルゼロ環境は、OpenCL SPIR-V 環境仕様で定義された数値コンプライアンス要件を満たすか、それを上回ります。参照: 数値コンプライアンス（英語）。

イメージのアドレス指定とフィルター処理

‘oneAPI’ レベルゼロ環境でのイメージのアドレス指定およびフィルター処理は、OpenCL SPIR-V 環境仕様で定義された動作と互換性があります。参照: イメージのアドレス指定とフィルター処理（英語）。

拡張機能

目的

デバイスまたはベンダー固有の機能は拡張機能として公開します。

この仕様では、次の 2 つのタイプの拡張機能が定義されています:

1. **標準** - 仕様の現在のバージョンおよび将来のすべてのバージョンに組み込むために承認された拡張機能。
2. **実験的** - 拡張機能は承認前に実験とアプリケーション・ベンダーからのフィードバックを必要とするため、アプリケーションは運用環境で実験的な拡張機能に依存しないでください。

上記に加えて、実装は、この仕様で定義されていない非標準の拡張機能を提供することを選択できます。

非標準の拡張機能を定義して文書化し、それらの拡張機能が標準のコア API または sysman API の機能や拡張機能と競合したり干渉したりしないことを保証するのは実装の責任です。

次の表は、さまざまな種類の拡張機能の検出に使用される API をまとめたものです。この仕様の開始時から利用可能な zeDriverGetExtensionProperties の他に、他の API がその後追加されました。API が追加された仕様のバージョンも次の表に記載されています。

API カテゴリー	追加された API バージョン	拡張タイプ	拡張機能検出 API
コア	NA	標準	zeDriverGetExtensionProperties
コア	NA	実験的	zeDriverGetExtensionProperties
コア	1.1	実装非標準	zeDriverGetExtensionFunctionAddress
Sysman	1.8	標準	zesDriverGetExtensionProperties
Sysman	1.8	実験的	zesDriverGetExtensionProperties
Sysman	1.8	実装非標準	zesDriverGetExtensionFunctionAddress

要件

- 拡張機能では、マクロ、列挙型、構造体、関数にグローバルな一意の名前を指定する必要があります
- 拡張機能には、zeDriverGetExtensionProperties、zesDriverGetExtensionProperties から報告されるグローバルで一意の拡張機能名が必要です
- すべての拡張機能はこの仕様で定義する必要があります
- 拡張機能は、この仕様で定義されている標準 API の下位互換性を損なってはなりません
- 標準拡張バージョンは以前のバージョンと下位互換性がなければなりません

命名規則

標準拡張機能の場合、次の命名規則に従う必要があります:

- すべての拡張関数には *Ext* を接尾辞として付ける必要があります。
- すべてのマクロは、適切なプリフィクスを付けてすべて大文字にする必要があります。コアマクロは *ZE_NAME_EXT* を使用し、Sysman マクロは *ZES_NAME_EXT* を使用します。
- すべての構造体、列挙体、およびその他のタイプは、適切なプリフィクス付きのスネークケース規則に従う必要があります。コア構造体と列挙タイプは *ze_name_ext_t* を使用し、Sysman は *zes_name_ext_t* を使用します。
- すべての列挙子の値は、適切なプリフィクスを付けてすべて大文字にする必要があります。コア列挙子の値は *ZE_ENUM_EXT_ETOR_NAME* を使用し、Sysman は *ZE_ENUM_EXT_ETOR_NAME* を使用します。
- すべてのハンドルタイプは *ext_handle_t* で終わる必要があります
- すべての記述子構造体は *ext_desc_t* で終わる必要があります
- すべてのプロパティー構造体は *ext_properties_t* で終わる必要があります
- すべてのフラグ列挙子は *ext_flags_t* で終わる必要があります

実験的な拡張機能では、次の命名規則に従う必要があります:

- 実験的な拡張機能は、いつでもドライバーに追加または削除される可能性があります。
- 実験的な拡張機能は、バージョン間での前方または後方互換性が保証されません。
- 実験的な拡張機能は、製品版のドライバーリリースでサポートされる保証はなく、リリースごとに追加されたり、削除される可能性があります。
- すべての拡張関数には *Exp* を接尾辞として付ける必要があります。ベンダー名は、*ze* または *zes* プリフィクスに続き、キャメルケース規則に従う必要があります。
- すべてのマクロは、適切なプリフィクスを付けてすべて大文字にする必要があります。コアマクロは *ZE_NAME_EXP* を使用し、Sysman マクロは *ZES_NAME_EXP* を使用します。ベンダー名は *ZE* または *ZES* プリフィクスの後に続く必要があります。
- すべての構造体、列挙体、およびその他のタイプは、適切なプリフィクス付きのスネークケース規則に従う必要があります。コア構造体と列挙タイプは *ze_name_exp_t* を使用し、Sysman は *zes_name_exp_t* を使用します。ベンダー名は *ze* または *ZES* プリフィクスの後に続く必要があります。
- すべての列挙子の値は、適切なプリフィクスを付けてすべて大文字にする必要があります。コア列挙子の値は *ZE_ENUM_EXP_ETOR_NAME* を使用し、Sysman は *ZES_ENUM_EXP_ETOR_NAME* を使用します。ベンダー名は *ZE* または *ZES* プリフィクスの後に続く必要があります。
- すべてのハンドルタイプは *exp_handle_t* で終わる必要があります
- すべての記述子構造体は *exp_desc_t* で終わる必要があります
- すべてのプロパティー構造体は *exp_properties_t* で終わる必要があります
- すべてのフラグ列挙子は *exp_flags_t* で終わる必要があります

列挙の拡張

既存の列挙は、`etor` を追加することで拡張できます。Etors は拡張子命名規則を引き続き使用し、将来の互換性問題を回避するため値を割り当てる必要があります。

構造体の拡張

基本記述子または基本プロパティー構造タイプから派生した構造体は、構造体チェーンを使用して拡張できます。

他の方法を使用することもできますが、既存の構造体を拡張するにはこれが必須です。

構造体チェーンには、任意の順序で複数の拡張構造体を含めることができます。したがって、拡張機能は他の拡張機能に対する順序に依存すべきではなく、実装は順序に依存してはなりません。さらに、実装ではサポートされていない拡張構造体は無視されます。

拡張機能では、構造体チェーンを使用して拡張できる特定の構造体と関数を文書化する必要があります。

標準コア拡張機能のリスト

- “ZE_extension_device_cache_line_size”
- “ZE_extension_eu_count”
- “ZE_extension_pci_properties”
- “ZE_extension_rtas”
- “ZE_extension_srgb”
- “ZE_extension_bfloat16_conversions”
- “ZE_extension_cache_reservation”
- “ZE_extension_device_luid”
- “ZE_extension_device_vector_sizes”
- “ZE_extension_device_ip_version”
- “ZE_extension_driver_ddi_handles”
- “ZE_extension_event_query_kernel_timestamps”
- “ZE_extension_external_memmap_sysmem”
- “ZE_extension_float_atomics”
- “ZE_extension_image_copy”
- “ZE_extension_image_query_alloc_properties”
- “ZE_extension_image_view”
- “ZE_extension_image_view_planar”
- “ZE_extension_kernel_max_group_size_properties”
- “ZE_extension_linkage_inspection”
- “ZE_extension_linkonce_odr”
- “ZE_extension_memory_compression_hints”
- “ZE_extension_memory_free_policies”
- “ZE_extension_device_memory_properties”
- “ZE_extension_raytracing”
- “ZE_extension_subgroups”
- “ZES_extension_device_ecc_default_properties”
- “ZES_extension_engine_activity”

実験的拡張機能のリスト

- “ZE_experimental_rtas_builder”
- “ZE_experimental_bandwidth_properties”
- “ZE_experimental_bindless_image”
- “ZE_experimental_command_list_clone”

- “ZE_experimental_event_pool_counter_based”
- “ZE_experimental_event_query_timestamps”
- “ZE_experimental_fabric”
- “ZE_experimental_global_offset”
- “ZE_experimental_image_memory_properties”
- “ZE_experimental_image_view”
- “ZE_experimental_image_view_planar”
- “ZE_experimental_immediate_command_list_append”
- “ZE_experimental_kernel_allocation_properties”
- “ZE_extension_kernel_binary_exp”
- “ZE_experimental_scheduling_hints”
- “ZE_experimental_mutable_command_list”
- “ZE_experimental_power_saving_hint”
- “ZE_experimental_module_program”
- “ZE_experimental_relaxed_allocation_limits”
- “ZE_experimental_sub_allocations”
- “ZET_experimental_global_metric_timestamps”
- “ZET_experimental_concurrent_metric_groups”
- “ZET_experimental_metric_export_data”
- “ZET_experimental_metric_group_marker”
- “ZET_experimental_programmable_metrics”
- “ZET_experimental_metrics_runtime_enable_disable”
- “ZET_experimental_metric_tracer”
- “ZET_experimental_calculate_multiple_metrics”
- “ZET_experimental_api_tracing”
- “ZES_experimental_firmware_security_version”
- “ZES_extension_mem_state”
- “ZES_extension_power_domain_properties”
- “ZES_extension_ras_state”
- “ZES_experimental_sysman_device_mapping”
- “ZES_experimental_virtual_function_management”

標準 Sysman 拡張機能のリスト

- “ZES_extension_power_limits”

API ドキュメント

コア API（英語）

ツール API（英語）

Sysman API（英語）

API ドキュメントの詳細は、オリジナルの英文ドキュメント（英語）を参照してください。

バージョン

https://oneapi-src.github.io/level-zero-spec/releases/index.html#level-zero（英語）