第12世代インテル® Core™ プロセッサー (開発コード名 Alder Lake) パフォーマンス・ハイブリッド・アーキテクチャー向けゲーム開発者ガイド

ホワイトペーパー

2021 年 10 月

リビジョン 1.0

注意事項:

この日本語マニュアルは、インテル コーポレーションのウェブサイトで公開されている『Game Dev Guide for Alder Lake Performance Hybrid Architecture』の参考訳です。

著作権と商標について: 本資料には、開発の設計段階にある製品についての情報が含まれています。この情報は予告なく変更されることがあります。この情報だけに基づいて設計を最終的なものとしないでください。

性能は、使用状況、構成、その他の要因によって異なります。詳細については、www.Intel.com/PerformanceIndex (英語) を参照してください。

性能の測定結果はシステム構成の日付時点のテストに基づいています。また、現在公開中のすべてのセキュリティー・アップデートが適用されているとは限りません。詳細については、公開されている構成情報を参照してください。絶対的なセキュリティーを提供できる製品またはコンポーネントはありません。

すべての製品計画とロードマップは、予告なく変更される場合があります。

開発コード名は、開発中で一般に公開されていない製品、テクノロジー、またはサービスを識別するためにインテルによって使用されます。それらは、「商用的な」名称ではなく、商標としての機能を意図したものではありません。

実際の費用と結果は異なる場合があります。

インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、またはサービスの有効化が必要となる場合があります。

本資料に記載されているインテル製品に関する侵害行為または法的調査に関連して、本資料を使用または使用を促すことはできません。本資料を使用することにより、お客様は、インテルに対し、本資料で開示された内容を含む特許クレームで、その後に作成したものについて、非独占的かつロイヤルティー無料の実施権を許諾することに同意することになります。

本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスも許諾するものではありません。本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

インテルは、本資料で参照しているサードパーティーのベンチマーク・データまたはウェブサイトについて管理や監査を行っていません。本資料で参照しているウェブサイトにアクセスし、本資料で参照しているデータが正確かどうかを確認してください。

目次

改訂履歴

リビジョン番号	説明	日付
1.0	ドキュメントの最初のリリース。	2021 年 10 月

1. はじめに

第12世代インテル® Core™ プロセッサー (開発コード名 Alder Lake) は、2 つのコアタイプを結合した新しいパフォーマンス・ハイブリッド・アーキテクチャーです。Performance-core (P-core - 開発コード名 Golden Cove として知られています) と Efficient-core (E-core – 開発コード名 Gracemont として知られています)。このゲーム開発者ガイドは、ゲーム開発者を対象としており、開発コード名 Alder Lake パフォーマンス・ハイブリッド・アーキテクチャーを活用するゲーム最適化向けのアーキテクチャーの概要とベスト・プラクティスを提供します。

サンプル・アプリケーションとソースコードは、GitHub* (英語) で入手できます。

現代の CPU アーキテクチャーには、数十年にもわたる進化、経験、そして知識を包括しており、今も改善を続けています。CPU は、数万個のトランジスターで構成されるシンプルなシングルコア・シリコンから、数十億個のトランジスターを持つマルチコアの多目的システムへと進化してきました。このたび、インテルの最新 CPU アーキテクチャー開発コード名 Alder Lake (ADL) が登場しました。パーソナル・コンピューター向けの CPU では、「Performance-core」と「Efficient-core」を組み合わせたハイブリッド・コアが始めて採用されました。このハイブリッド・コアにより、開発者はハイパフォーマンスで電力効率の高いマルチコア CPU を、コンピューター・ゲームなどのワークロードで活用できるようになります。

携帯電話業界では、数年前から高性能な「Performance-core」(P-core) や電力効率の高い「Efficient-core」(E-core) のようなハイブリッド・アーキテクチャーが利用されており、開発者にはなじみ深いものとなっています。この利点をデスクトップ PC やノートブック PC でも得られるようになりましたが、携帯電話とは重要な違いがあります。E-core は P-core を補完することで、マルチスレッドのスループットを大幅に高め、許容される電力範囲内で最大の総合性能を発揮します。

今回アーキテクチャーが大幅に変更されたことで、開発者は、利用可能なすべての機能を活用する効率的なソフトウェアを作成するため、アーキテクチャーの詳細を理解し、新たな設計目標を取り入れ、ベスト・プラクティスに関する重要な決定を行う必要があります。ハイブリッド・プロセッサー上で動作するアプリケーションで最適なパフォーマンスと互換性を維持するため、CPU トポロジーを適切に検出しなければなりません。物理プロセッサーと論理プロセッサー間のハイパースレッディングに関する従来の想定が当てはまらなくなる可能性があります。これを解決しないとシステムに深刻な問題を与える可能性があります。開発者は、消費電力とパフォーマンスを最適化するため、システムで利用可能な論理プロセッサーを完全に理解する必要があります。

このドキュメントでは、アーキテクチャー、プログラミングのパラダイム、検出、および最適化の方針と例に関する情報を示しています。

2. アーキテクチャーの概要

このセクションでは、一部の 開発コード名 Alder Lake プロセッサーで採用されている、パフォーマンス・ハイブリッド・アーキテクチャーに関する情報を示します。

2.1. トポロジー

開発コード名 Alder Lake のパフォーマンス・ハイブリッド・アーキテクチャーは、ハイパフォーマンスの「Performance-core」(P-core、開発コード名 Golden Cove) と高効率の「Efficient-core」(E-core、開発コード名 Gracemont) を 1 つのシリコンチップに統合したものです。トポロジー的には、各 P-core は L2$ を内包し L3$ (LLC) に接続されます。各 E-core モジュールは 4 つの開発コード名 Gracemont コア (これをクラスターと呼びます) を持ち L2$ を共有します。E-core クラスターは、P-core と共有される L3$ に接続されます。すべてのコアは、それぞれ論理プロセッサーとして OS に公開されます。

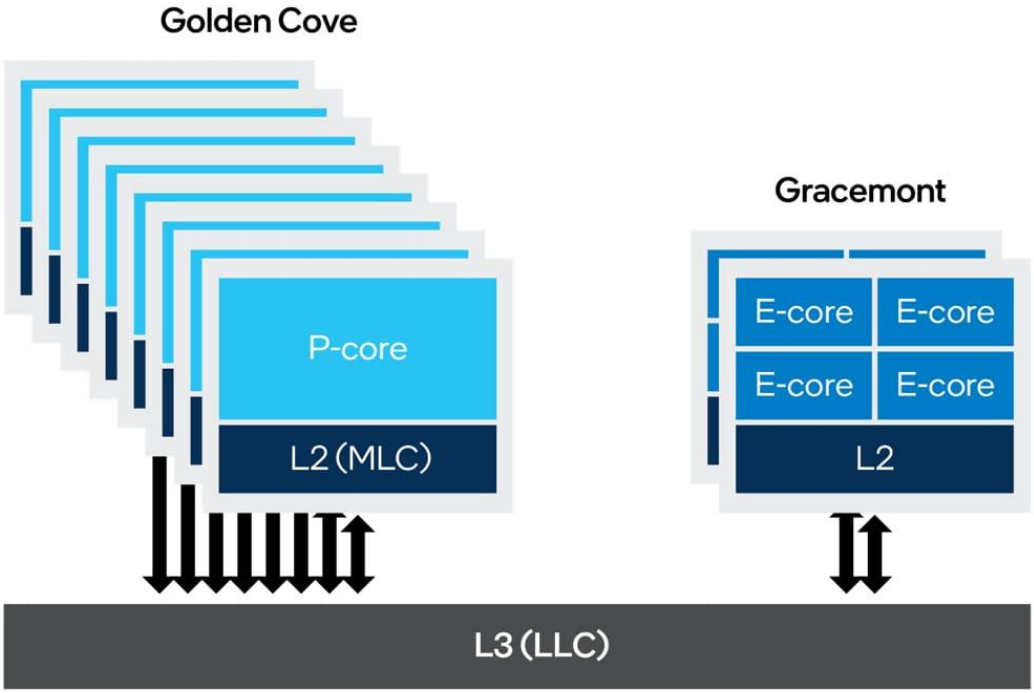


図 1. Performance-core (P-core、開発コード名 Golden Cove) と Efficient-core (E-core、開発コード名 Gracemont) のクラスターは、個別の L1 キャッシュと L2 キャッシュを搭載しているため、互いに独立して実行できます。L3 (LLC) キャッシュは共有されます。

2.2. 命令セット

プログラミング・モデルを単純にして柔軟性を持たせるため、命令セットレベルでは設計上次のような決定がなされました。

- すべてのコアタイプは同じ命令セットを持ちます。
- AVX512 は P-core では無効化されており、E-core では利用できません。

表 1. 両方のコアの拡張機能として、開発コード名 Golden Cove (P-core) 固有の機能が追加されました。

開発コード名 Golden Cove の機能	開発コード名 Alder Lake
インテル® AVX-512	P-core では無効化され、E-core では利用できません。
インテル® TSX	P-core と E-core の両方で無効化されています。
インテル® AVX-VNNI (ベクトル・ニューラル・ネットワーク命令)	対称 ISA を介して追加。
vAES (アドバンスト・エンクリプション・スタンダード)	対称 ISA を介して存在 (第 10 世代インテル® Core™ プロセッサーと第 11 世代インテル® Core™ プロセッサーにも存在します)。
vCLMUL (キャリーなし乗算)	対称 ISA を介して追加。
UMWAIT/TPAUSE	対称 ISA を介して追加。

2.3. インテル® スレッド・ディレクター (ITD) とオペレーティング・システム・ベンダー (OSV) によるパフォーマンス・ハイブリッド・アーキテクチャー向けの最適化

インテル® スレッド・ディレクター (ITD) は、インテル® ハイブリッド・テクノロジー搭載インテル® Core™ プロセッサー (開発コード名 Lakefield) のハードウェア・ガイドによるスケジューリング (HGS) のサポートの上位に構築されています。これは、オペレーティング・システムに命令セット・アーキテクチャー (ISA) を認識させ、Performance-core と Efficient-core 間のパフォーマンス差 (および電力効率の変化) を示します。これにより、インテル® スレッド・ディレクターはタスクに最も適したコアにスレッドをスケジュールできます。

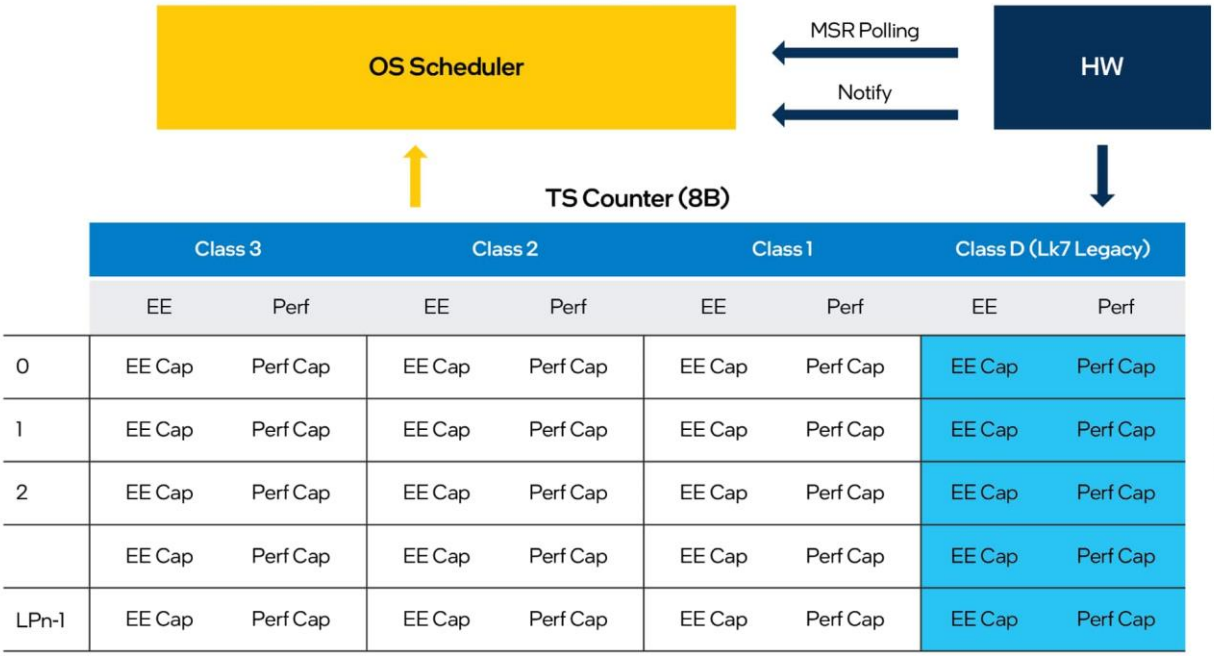


図 2. ハードウェア、オペレーティング・システム (OS) スケジューラー、
およびタスク・スケジューラー (TS) カウンター間のスケジュールの相互作用

注:

- クラス ID のパフォーマンス機能は、論理プロセッサー (LP Performance-core と Efficient-core) の相対的なパフォーマンス・レベルを示します。高い値はパフォーマンスが高いことを示しています。
- クラス ID ごとの EE 機能は LP の相対的なエネルギー効率のレベルを示します。この値が高いほど、EE が高いことを示し、エネルギー効率が必要なスレッドによって使用されます。
- OS は、電力ポリシーやバッテリースライダーなどのパラメーターに応じて、EE とパフォーマンスのいずれかを選択できます。

ここでのさまざまなクラスは、コア間のパフォーマンスの差を示しています。例えば、クラス 1 はインテル® AVX2-FP32 などの ISA で P-core が E-core よりもパフォーマンスが高いことを示し、クラス 2 はインテル® AVX-VNNI のパフォーマンス差が大きいことを示しています。UMWAIT/TPAUSE/PAUSE などの待機を追跡するクラスが導入され、実際のワークが Efficient-core に移行する間、Performance-core がアイドル状態になるのを防ぎます。

2.4. アーキテクチャーへの影響

プログラミングの観点からは、すべてのコアは機能的に同一であり、パフォーマンスと効率が異なるだけです。OS とハードウェアによるガイドの組み合わせにより、スレッドの分散はインテリジェントに行われますが、ハイブリッド・アーキテクチャー向けのコードを記述するには、いくつか注意すべきことがあります。

- CPUID 命令は実行するコアごとに異なる値を返すように設計されています。ハイブリッド・コアには多くの CPUID リーフがありますが、それらはさまざまなデータを保持します。つまり、暗号化のためハッシュを生成した

り、乱数生成に固有のシードを得るためコアを使用する場合、コアタイプによって変化しないデータを含むリーフのみを参照するように注意する必要があります。

- ビット操作 (AND、OR、XOR、NOT、ローテート) には、符号付きオーバーフローの概念がないため、定義される値はプロセッサー・アーキテクチャーによって異なり、無条件にビットをクリアするもの、変更しないもの、そして未定義の値を保持するものがあります。シフトと乗算は明確に定義された値を許可しますが、それは一貫して実装されるわけではありません。例えば、x86 命令セットでは、乗算と 1 ビット・シフトのオーバーフロー・フラグのみが定義されています。それ以外のシングル・ビット・シフトの動作は未定義であり、Performance-core と Efficient-core では異なる可能性があります。

2.5. ストック・キーピング・ユニット (SKU)

コアと実行ユニット (EU) を多様に組み合わせた複数のデスクトップおよびモバイル SKU が用意されています。

図 3. 開発コード名 Alder Lake 向けのノートブックとデスクトップの SKU

モバイル SKU には、最大 6 つの P-core と 8 つの E-core が搭載されます。すべてのモバイル SKU には E-core が含まれます。デスクトップ SKU には、最大 8 つの P-core と 8 つの E-core が搭載されます。特定のデスクトップ SKU では P-core のみが搭載されます。

2.6. CPU トポロジーの検出

開発コード名 Alder Lake のようなハイブリッド・プロセッサー上で動作するアプリケーションで最適なパフォーマンスと互換性を維持するには、CPU トポロジーを適切に検出する必要があります。これまで、開発者はシステム上の物理プロセッサー数をカウントすることで、パフォーマンスや消費電力の変化を容易に想定することができました。ハイパースレッドをサポートする CPU においては、物理プロセッサー数を 2 倍にして論理プロセッサー数を取得するなど、いくつかの想定が考えられます。ハイブリッド CPU では、従来の想定はもはや当てはまらず、開発者はシステム上で利用可能な論理プロセッサーを完全に把握し、各論理プロセッサーの電力とパフォーマンス特性を判断しなければなりません。

ターゲット CPU のトポロジーを決定するにはいくつかの方法があります。異なる方法論間には多くの重複があります。そのため、一般的なガイドラインとして、必要な情報を得るのに最も簡潔なソリューションとコードベース全体で一貫した方式を使用することです。クロスプラットフォームをサポートするため、CPUID 組込み関数はいくつかのトポロジー情報を提供しますが、Windows* API は GetLogicalProcessorInformation (英語) や GetSystemCPUSetInformation (英語) など、論理プロセッサーを完全に列挙するいくつかのメソッドをサポートしています。

さらに、クロスプラットフォームのサポート向けに、CPUID 組込み関数はいくつかのトポロジー情報を提供しますが、CPUID 組込み関数と GetLogicalProcessorInformation 関数を併用することで、ターゲットシステムの CPU に関するさらに詳しい情報を取得できます。

2.6.1. GetLogicalProcessorInformation (GLPI/GLPIEX)

Windows* API は、論理プロセッサーのトポロジーを決定するため、**GetLogicalProcessorInformation** (GLPI) や GetLogicalProcessorInformationEx (GLPIEX) (英語) などいくつかの方法を提供しています。GLPI/GLPIEX では、論理プロセッサー、キャッシュ、NUMA ノード、プロセッサー・グループ、パッケージ間のトポロジーの関係などを把握できます。この情報は、同じリソースを共有する論理プロセッサーを判断するのに役立ちます。

例えば、システムが Performance-core (P-core) プロセッサーの物理コアと SMT (同時スレッド) コアを持つ場合、GLPI は各ノードの **Relationship** フィールドと **ProcessorMask** フィールドを含む SYSTEM_LOGICAL_PROCESSOR_INFORMATION 構造体を返します。

SYSTEM_LOGICAL_PROCESSOR_INFORMATION の **Relationship** フィールドには、**RelationCache (2)** または **RelationProcessorCore (0)** の値が保持され、**RelationNumaNode (1)** と **RelationProcessorPackage (3)** には追加の値が格納されます。**Relationship** フィールドが **RelationProcessorCore (0)** に設定されていると、SYSTEM_LOGICAL_PROCESSOR_INFORMATION 構造体の **ProcessorCore** フィールドは **Flag** (プロセッサー・フラグ) を保持し、**RelationCache (2)** の **Cache** フィールドはキャッシュの特性を示す CACHE_DESCRIPTOR 構造体 (英語) を保持します。

ProcessorMask フィールドは、各論理プロセッサーとキャッシュに対する 64 ビット・マスクです。**ProcessorMask** フィールドは、1 つのグループのプロセッサーのみを表します。64 個以上の論理プロセッサーを持つ CPU では、GLPIEX を使用して各論理プロセッサーとキャッシュの **GroupMasks** を取得する必要があります。マスクの各ビットは 1 つの論理プロセッサーに対応し、少なくとも 1 つのビットが常に設定されています。ビット操作によってプロセッサー・マスクを比較することで、各マスクの関係を知ることができます。例えば、コアマスクがキャッシュマスクと関連するか調査するには、「&」オペレーターでプロセッサー・マスクを比較します。

次の疑似コードは、C++ でビット・マスクを比較する方法を示します。

```
// GLPI による物理プロセッサー・マスクの例
    // 00000000000000000000000000000000000000000000000000000000000000001
    unsigned long physical_processor_mask = 0x00000001;

    // GLPI による物理プロセッサー・マスクの例
    // 00000000000000000000000000000000000000000000000000000000000000010
    unsigned long logical_processor_mask = 0x00000002;

    // GLPI による物理プロセッサー・マスクの例
    // 00000000000000000000000000000000000000000000000000000000000000011
    unsigned long l1_cache_mask = 0x00000003;
    // 物理プロセッサー・マスクとキャッシュマスクに関係性があるか確認します
    if (physical_processor_mask & l1_cache_mask)
    {
        // 物理プロセッサーは L1 キャッシュマスクに関連しています
    }
    else
    {
```

```
        // 物理プロセッサーは L1 キャッシュに関連していません
    }
    if ((physical_processor_mask & l1_cache_mask) &&
        (logical_processor_mask & l1_cache_mask))
    {
        // 物理プロセッサーと論理プロセッサーは L1 キャッシュを共有します
    }
    else
    {
       // 物理プロセッサーと論理プロセッサーは L1 キャッシュを共有しません
    }
```

GLPI は 64 個までの論理プロセッサーをサポートします。しかし、SetThreadGroupAffinity (英語) を使用すると、64 個以上の論理プロセッサーを持つ CPU の特定グループのアクティブな 64 個のプロセッサーを検出できます。ターゲットシステムに 64 個以上の論理プロセッサーがある場合、代わりに GLPIEX や **GetSystemCPUSetInformation** を使用できます。

GetLogicalProcessorInformationEx (GLPIEX) は、アクティブなプロセッサー、キャッシュ、グループ、NUMA ノード、およびパッケージごとに SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX 構造体を返します。構造体の **Relationship** フィールドは、グループノードに対する追加の **RelationGroup (4)** を報告します。SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX の **Relationship** フィールドが **RelationGroup (4)** に設定されている場合、**Group** フィールドには GROUP_RELATIONSHIP 構造体 (英語) が入力されます。

GLPI とは異なり、GLPIEX から返される SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX 構造体は **ProcessorMask** を返しません。代わりに、PROCESSOR_RELATIONSHIP 構造体 (英語)、NUMA_NODE_RELATIONSHIP 構造体 (英語)、CACHE_RELATIONSHIP 構造体 (英語)、または GROUP_RELATIONSHIP 構造体のいずれかへの参照が含まれます。これらの構造体には、GROUP_AFFINITY タイプ (英語) の **GroupMask** が含まれます。**GroupMask** は、グループ ID および GLPI の **ProcessorMask** と同様に 64 ビットのビットフィールドである KAFFINITY マスクを含みます。

ハイブリッド CPU において、GLPIEX から返される **Relationship** が **RelationProcessorCore (0)** である場合、PROCESSOR_RELATIONSHIP 構造体は **EfficiencyClass** 値を返します。この値は、論理プロセッサーの電力とパフォーマンスの比率を示します。EfficiencyClass フィールドの **EfficiencyClass** 値が高いコアは、パフォーマンスは高くなりますが、電力効率は低くなります。GLPIEX が返す **EfficiencyClass** 値を使用して、論理プロセッサーを同種の CPU クラスターにグループ化できます。これは、特定の電力またはパフォーマンス要件でタスクをスケジュールする必要がある場合に役立ちます。

2.6.2. GetSystemCPUSetInformation

GetSystemCPUSetInformation を使用することには、いくつかの利点があります。1 つは、CPU ごとに 64 個を超える論理プロセッサーを照会できること、もう 1 つはグループに対応しグループ情報を報告できることです。また、GLPIEX で報告されない追加のフラグを表示できます。

GetSystemCPUSetInformation は、ターゲット CPU 上のアクティブな論理プロセッサーそれぞれのデータを含む SYSTEM_CPU_SET_INFORMATION 配列を返します。論理プロセッサーごとに繰り返して、CPU セット **ID**、**Group**、**LogicalProcessorIndex**、**EfficiencyClass**、および **SchedulingClass** に加え、割り当てとパーキングのフラグを取得できます。

```
// データ構造体内の要素の総数 （サイズ） を取得します
GetSystemCpuSetInformation(nullptr, 0, &size, curProc, 0);

// 最初の呼び出しで返されたサイズをベースにデータ構造体を割り当てます
std::unique_ptr<uint8_t[]> buffer(new uint8_t[size]);
PSYSTEM_CPU_SET_INFORMATION cpuSets =
      reinterpret_cast<PSYSTEM_CPU_SET_INFORMATION>(buffer.get());
PSYSTEM_CPU_SET_INFORMATION nextCPUSet;

// すべての CPUSet 要素を取得します
GetSystemCpuSetInformation(cpuSets, size, &size, curProc, 0);

nextCPUSet = cpuSets;

// 各論理プロセッサーで反復を処理します
for (DWORD offset = 0;
offset + sizeof(SYSTEM_CPU_SET_INFORMATION) <= size;
       offset += sizeof(SYSTEM_CPU_SET_INFORMATION), nextCPUSet++)
{
       // CPU セットのタイプが有効であることを確認します

       if (nextCPUSet->Type == CPU_SET_INFORMATION_TYPE::CpuSetInformation)
       {
              // 後で使用するため論理プロセッサー情報を保存します
              LOGICAL_PROCESSOR_INFO core;
              core.id                          = nextCPUSet->CpuSet.Id;
              core.group                       = nextCPUSet->CpuSet.Group;
              core.node                        = nextCPUSet->CpuSet.NumaNodeIndex;
              core.logicalProcessorIndex = nextCPUSet->CpuSet.LogicalProcessorIndex;
              core.coreIndex                   = nextCPUSet->CpuSet.CoreIndex;
              core.realTime              = nextCPUSet->CpuSet.RealTime;
              core.parked                = nextCPUSet->CpuSet.Parked;
              core.allocated                   = nextCPUSet->CpuSet.Allocated;
              core.allocatedToTargetProcess    = nextCPUSet-
>CpuSet.AllocatedToTargetProcess;
              core.allocationTag         = nextCPUSet->CpuSet.AllocationTag;
              core.efficiencyClass             = nextCPUSet->CpuSet.EfficiencyClass;
              core.schedulingClass             = nextCPUSet->CpuSet.SchedulingClass;
              procInfo.cores.push_back(core);
              procInfo.numLogicalCores++;

              // 効率クラスに基づいて論理プロセッサーを分類
              // ...
       }
}
```

GLPIEX と同様に、**GetSystemCPUSetInformation** も各論理プロセッサーの **EfficiencyClass** 値を返します。**EfficiencyClass** は、パフォーマンスと消費電力の比率を表し、値が大きいほどプロセッサーのパフォーマンスが高く、電力効率が低いことを示します。**EfficiencyClass** を使用して論理プロセッサーをグループに分類することができます。

GetSystemCPUSetInformation は GLPI および GLPIEX に類似しています。しかし、これはプロセッサー情報にのみ当てはまり、最終レベルキャッシュのインデックス以外のキャッシュトポロジーは報告されません。CPU に関連する追加のトポロジー情報が必要な場合、**GetSystemCPUSetInformation** と GLPI または GLPIEX を組み合わせることもできます。

2.6.3. CPUID 組込み関数

2020 年 3 月の時点で、『インテル® アーキテクチャー命令セット拡張機能および将来の機能のプログラム・リファレンス』(英語) には、CPUID 組込み関数を使用してターゲットシステムのハイブリッド・トポロジーを決定する要件の詳細が説明されています。

2 つの新しいフラグが定義されています。

1. **ハイブリッド・フラグ**は、EAX レジスターに「07H」を設定して CPUID を呼び出し、EDX レジスターの 15 ビット目を読みだすことで取得できます。
2. **コア・タイプ・フラグ**は、EAX レジスターに「1AH」を設定して各論理プロセッサーで CPUID を呼び出すことで取得できます。これにより、EAX レジスターのビット 24-31 にそれぞれの論理プロセッサーのタイプが返されます。

コアタイプは、論理プロセッサーが Efficient-core (E-core、開発コード名 Gracemont) (20H) であるか、Performance-core (P-core、開発コード名 Golden Cove) (40H) であるかを識別するために使用します。ただし、**コアタイプ**の戻り値は、インテル® Core™ プロセッサーの物理コアと SMT コアを識別するものではありません。どちらもインテル® Core™ プロセッサー (40H) と示されます。E-core プロセッサーにはハイパースレッドの機能はなく、物理プロセッサーごとに論理コアを検出する必要はありません。また、**GetSystemCpuSetInformation** が返す「EfficiencyClass」を使用して、物理コア、SMT コア、および E-core プロセッサーを区別することができます。

表 2.ハイブリッド・フラグとコアタイプの CPUID 機能

CPUID ハイブリッド機能表						
名称	関数 (EAX)	リーフ (ECX)	レジスター	開始ビット	終了ビット	コメント
ハイブリッド・フラグ	07H	0	EDX	15	15	1 の場合、プロセッサーはハイブリッドとして認識されます。また、ハイブリッド (CPUID.07H.0H:EDX[15]=1) の場合、ソフトウェアはハイブリッド情報列挙リーフからネイティブモデル ID とコアタイプを参照する必要があります。
コアタイプ	1AH	0	EAX	24	31	ハイブリッド情報サブリーフ (EAX = 1AH, ECX = 0) EAX ビット 31-24: コアタイプ 10H: 予約済み 20H: Intel Atom® プロセッサー 30H: 予約済み 40H: インテル® Core™ プロセッサー

CPUID を使用してアプリケーションのハイブリッド・トポロジーを検出する際には、いくつかの注意点があります。

1. **ハイブリッド・フラグ**と**コアタイプ**は、従来のホモジニアス CPU ではサポートされません。Windows* の**アプリケーション互換モード**で実行されるアプリケーションでは、OS が CPUID 組込み関数呼び出しをインターセプトしないため、正しい値を取得できないことがありました。CPUID は結果を符号付き整数で返すため、符号なし整数にキャストするか、CPUID レジスターで返される 32 ビット値を読み取るため std::bitset<32> を使用する必要があります。
2. **コアタイプ**を確認する場合、対象の論理プロセッサーを一時的に「固定 (pin)」し、その論理プロセッサーの**コアタイプ**を取得する必要があります。このような理由から、トポロジーの検出には **GetLogicalProcessorInformationEx** や **GetSystemCpuSetInformation** などの方法を使用する必要がある場合があります。

2.6.4. 論理プロセッサーの分類

CPU のトポロジーを検出する場合、**EfficiencyClass** を共有する論理プロセッサーのグループや、共有キャッシュなどその他の特性からグループを作成できます。論理プロセッサーを効率良くグループ化することで、対象とするアーキテクチャーが必要な場合に、ワークをより簡単にセグメント化できます。プロセッサーをグループ化する方法は、アプリケーションで選択したトポロジーの検出方法と、論理プロセッサーのトポロジーを保存するデータ構造によって異なります。

CPUID を使用してトポロジーを検出する場合、ハイブリッド情報サブリーフから返されるコアタイプの値を利用できます。コアタイプを使用すると、P-core と E-core などの特定のタイプのプロセッサーに強く依存することになるため、クロスプラットフォームのサポートが複雑になります。そのため、論理プロセッサーを細粒度でグループ化する場合、**EfficientClass** を使用することを推奨します。

GLPI を利用する場合、ビット操作で **ProcessorMasks** を独自のカスタム 64 ビット・マスクにグループ化できます。ただし、GLPI は **EfficiencyClass** を返さないため、GLPI と CPUID または GLPIEX をペアにして、コアタイプまたは EfficiencyClass を GLPI のプロセッサーと関連付ける必要があります。このような理由から、GLPI よりも GLPIEX を利用するほうが望ましいとされています。

論理プロセッサーのグループ化には、GLPIEX または **GetSystemCPUSetInformation** から取得できる **EfficiencyClass** を使用するのが最適です。GLPIEX では、**GroupMasks** を使用する必要があります。これには、論理クラスターのカスタムマスクに分類 (ビニング) を行うためビット操作が必要です。**GetSystemCPUSetInformation** は、各論理プロセッサーを追跡する符号なし整数の識別子を取得できます。CPU セット ID のリストを格納するベクトルまたは配列を作成します。素早く論理クラスターを作成するには、最初のテンプレート・パラメーターに **EfficiencyClass** でグループ化された std::map を、2 番目のテンプレート・パラメーターには ULONG の std::vector を使用します。

```
#ifdef ENABLE_CPU_SETS
       // GetSystemCPUSetInformation から返された論理プロセッサーのマップを格納します。
       // unsigned = EffeciencyClass キー
       // std::vector<ULONG> = CPU セット ID のリスト1
       std::map<unsigned, std::vector<ULONG>> cpuSets;
#else
       // GLPI から返された論理プロセッサーのマップを格納します。
// short = コアタイプ
// ULONG64 = 64 ビットのプロセッサー・マスク
       std::map<short, ULONG64> coreMasks;
#endif
```

2.6.5. ハイブリッド・オペレーティング・システム (OS) の検出

Windows* 11 では、インテル® スレッド・ディレクター (ITD) のハードウェア・ヒントを利用して、OS がスレッドを自動的にスケジューリングできます。スレッドのスケジュールを OS や ITD に任せるには、最初にアプリケーションが動作している Windows* のバージョンを検出する必要があります。Windows* 11 のアップデートが適用されていないと ITD はサポートされません。ITD の一部の機能はバックポートされますが、サポートされている Windows* の最小バージョンを確認することはできません。VerifyVersionInfo API (英語) を使用すると、バージョン仕様にサービスパックのマイナー番号とビルド番号を含めることができます。

2.7. スレッドのスケジューリング

ハイブリッド・アーキテクチャーをサポートするプロセッサーで電力とパフォーマンスを最大限に活用するには、2 つのことを理解する必要があります。1 つは、OS がそれぞれのスレッドをどのようにスケジュールするか、そして 2 つ目は、追加のヒントを提供したり、必要に応じて完全に制御するために利用できる API を知ることです。Windows* では、開発コード名 Alder Lake 上のスケジューラーは以下を目標とします。

- シングルスレッドおよびマルチスレッドのパフォーマンスを最大化するため、最もパフォーマンスが高いコアを最初に使用します。
- スピルオーバーされたマルチスレッドのワークには、MT パフォーマンスが低いコアを使用します。
- パフォーマンスに影響を与える競合を避けるため SMT は最後に使用します。
- ITD を活用してスレッドのパフォーマンス差を利用し、適切なスレッドに適切なコアを選択するとともに、スレッドのコンテキスト・スイッチを最小限に抑えます。
- 最も効率の良いコアを利用することで電力の節約を期待します。
- MT パフォーマンスへの影響を軽減するため、バックグラウンドのワークが Efficient-core で実行されるようにします。
- 開発コード名 Alder Lake コアのパフォーマンスと効率の特性をより上手く活用するため、スレッドを状況に応じて特徴付ける API を提供します。

OS とハードウェアがプラットフォーム・レベルで連携して、パフォーマンスを最大化することを理解するのが重要です。特定のタスクを実行する適切な論理プロセッサーの選択は、環境によって異なる可能性があります。例えば、限られた電力で稼働するデバイスでは、特定クラスのプロセッサーを優先的に利用したり、一部のプロセッサーをパーキングして利用可能な電力を増やすことで利益を得られる場合があります。

システムは複雑であるため、従来のスレッド検出方法とは微妙に異なる点に注意が必要です。ハイブリッド・アーキテクチャーでは、スレッドのスケジュール先を探すボトムアップ方式のアプローチではなく、トップダウン方式のアプローチを採用するため、どのゲームスレッドがタイム・センシティブであるか、またはパフォーマンス・クリティカルであるかを特定するのに十分なコンテキスト情報を OS に提供することが効果的です。

2.8. スレッドの優先度

ゲーム開発者は、従来スレッドの優先度を 2 つの目的で利用してきました。

1. 例えば、オーディオを処理するスレッドが頻繁に実行され、スレッドが枯渇するとオーディオが途切れてしまうような場合には、高いスレッドの優先度は割り込み方式のワークを処理するために使用されます。この場合、ワーク自体はゲームで行われるオーバーワークの一部にすぎません。
2. また、スレッドの優先度は、レンダリング・スレッドなどゲームのフレームレートに重要なスレッドを示すために使用されます。割り込み型のワークとは異なり、これらのワークは長時間実行される可能性があります。優先度を高く設定する目的は、オーバー・サブスクリプションによりコンテキスト・スイッチが発生した場合に、即座にリソースを再割り当てできるようにするためです。

ハイブリッド・コアでは、スレッドの優先度にも特別の意味があります。優先度を高く設定されたスレッドは、当然ながらパフォーマンスが高いコアに割り当てられますが、これは副次的な考慮事項です。主な目的は、OS がスレッドのワークをスケジュールする頻度を示すことで、クリティカルなワークが可能な限り早く処理されるようにすることです。

これは、プレイヤーの視覚体験に対応するスレッドのワークについてスレッドの優先度を利用して重要度を示すという仮定で構築されているゲームエンジンでは、最適ではない動作につながる可能性があります。次のケーススタディーでは、考慮すべき課題に焦点を当てています。

- ゲームは、それぞれが論理プロセッサーと同じ数のスレッドを持つように 2 つのスレッドプールを作成します。
- 各スレッドプールには異なる優先度が割り当てられますが、一方は高とマークされ、他方は通常とマークされます。ファイルの入出力 (I/O) や手続き型コンテキストの生成などのバックグラウンド・タスクは、優先度の低いスレッドグループに送られ、レンダリング・ワークはすべて高い優先度のスレッドに送られます。
- ハイブリッド・システムではない環境では、優先度の高いタスクがスケジュールされるとそのスレッドがウェイクアップして、OS は優先度の低いスレッドの 1 つを強制的に停止します。
- ハイブリッド・システムでは同じような停止が発生しますが、低い優先度のスレッドが物理 Performance-core よりも多く、アクティブな低優先度のスレッドが論理プロセッサーの総数よりも少ない場合、OS は Efficient-core の 1 つをウェイクアップしてワークを実行します。
- 優先度の高いスレッドが Performance-core を利用できるよう、優先度の低いスレッドを Efficient-core に切り替えるか OS が決定するかどうかは、コンテキスト・スイッチのコストやスレッドの予測実行時間など内部ヒューリスティックによって決定されます。

上記のように、すべてのコアが同等であり、重要なワークが常にバックグラウンド・タスクに先行して処理されるように優先度だけで制御できるという現在の仮定は、ハイブリッド・システムでは十分ではないかもしれません。

2.9. スレッドのアフィニティー

ハイブリッド・アーキテクチャーをサポートするプロセッサーで電力とパフォーマンスを最大限に活用するには、前述のように、スレッドをどのコアで実行するか考慮することが重要です。例えば、クリティカル・パスのスレッドを処理する場合、論理 Performance-core で実行を優先させたいことがあります。しかし、バックグラウンドのワーカースレッド

を Efficient-core で実行するほうが適する場合もあります。以下の API リファレンスには、**SetThreadIdealProcessor()** や SetThreadPriority() (英語) のような弱いアフィニティー・ヒットによる OS レベルのガイドから、SetThreadInformation() (英語) や SetThreadSelectedCPUSets() (英語) のような強い制御、そして SetThreadAffinityMask() による最も強いアフィニティーの制御まで、多くの関数が説明されています。

一般には、強いアフィニティーはアプリケーションと OS 間の決まり事であるため、避けるように指示されています。強いアフィニティーを使用すると、プラットフォームの最適化ができなくなり、OS はインテル® スレッド・ディレクターからのアドバイスを無視することになります。強いアフィニティーは予期しない問題を引き起こす可能性があるため、ミドルウェアが強いスレッド・アフィニティーを使用しているか確認する必要があります。これは、強いスレッド・アフィニティーがアプリケーションによるハードウェアへのアクセスに直接影響するためです。強いアフィニティーの問題は、低消費電力デバイスなど Performance-core よりも Efficient-core が多いシステムでは、強いアフィニティーによって OS のスケジュールが制限されるため、特に問題となります。

アプリケーションに適したアフィニティー・レベルを決定することは、電力とパフォーマンスの要件を満たすには重要なことです。オペレーティング・システムと ITD にスレッド・スケジュールの大部分を任せることを選択する開発者は、「弱い」アフィニティーを好むかもしれません。「より強力な」アフィニティーによりスレッドのスケジュールを最大限に制御できますが、アプリケーションはさまざまな特性を持つハードウェアで実行されるため、注意する必要があります。アプリケーションから OS の動作を制御するよりも、スレッド間の動的な負荷分散と各種ハードウェアのパフォーマンス特性を考慮して、与えられたスレッド・アルゴリズムを注意深く設計することが望まれます。

アフィニティーの方針を選択する場合、実行時にスレッドのアフィニティーを変更することによって発生する可能性があるスレッド・コンテキスト・スイッチと、キャッシュフラッシュの頻度を考慮しなければなりません。**SetThreadAffinityMask** などの強いアフィニティー API 呼び出しの多くは、スレッドがアフィニティー・マスクで指定されたプロセッサーに存在しない場合、直ちにコンテキスト・スイッチされる可能性があります。**SetThreadPriority** などの弱いアフィニティー関数は、コンテキスト・スイッチをすぐに強制しないこともありますが、スレッドが実行されているクラスターやプロセッサーに固定する保証は低くなります。いずれの方針を選択する場合でも、スレッドの起動時または初期化時にスレッド・アフィニティーを設定することを推奨します。フレームごとに何度もスレッド・アフィニティーを設定することを避け、1 フレーム中のコンテキスト・スイッチは可能な限り少なくします。

2.9.1. SetThreadIdealProcessor

SetThreadIdealProcessor() は、シングル・プロセッサーでスレッドに弱いアフィニティーを設定できます。ただし、システムはスレッドが理想的なプロセッサーにスケジュールされることを保証するものではなく、可能な限り理想的なプロセッサーにスケジュールすることのみを保証します。つまり、スレッドは理想的なプロセッサーに頻繁にスケジュールされる可能性は高くなりますが、パフォーマンスが重要である場合は、より強力なアフィニティーの設定を選択する必要があります。また、SetThreadIdealProcessorEx() (英語) 関数を使用して、理想的なプロセッサーを設定したり、以前に割り当てられたスレッドの理想的なプロセッサーを取得することもできます。ハイパフォーマンス・コンピューティングのシナリオでは、**SetThreadIdealProcessor** の使用は避けるべきです。

2.9.2. SetThreadPriority

SetThreadPriority() (英語)は、スレッドに「弱い」アフィニティーを設定する方法の 1 つですが、スレッドが特定のクラスターまたはプロセッサーにスケジュールされることを保証するものではありません。これにより、OS によってスレッドがスケジュールされる頻度と、実行時に割り当てられるタイムスライスの長さを制御できます。デフォルトでは、スレッドが最初に生成されるときに **THREAD_PRIORITY_NORMAL** が割り当てられます。GetThreadPriority() 関数を使用して、アプリケーションの任意のスレッドの優先度を確認できます。

バックグラウンド・ワーカーを生成する場合、優先度 **THREAD_MODE_BACKGROUND_BEGIN** と **THREAD_MODE_BACKGROUND_END** を使用して、スレッドをバックグラウンド・モードにしたり、バックグラウンド・モードから移行することができます。または、クリティカル・パスのスレッドを **THREAD_PRIORITY_ABOVE_NORMAL**、**THREAD_PRIORITY_HIGHEST**、**THREAD_PRIORITY_TIME_CRITICAL** などの高い優先度に設定することもできます。スレッドの優先度を極度に高く設定すると、スレッドが CPU で利用可能なすべてのリソースを消費する可能性があることから、CPU の過度な利用を減らすため、優先度のバランスが適切であることを確認してください。さらに、**REALTIME_PRIORITY_CLASS** では、ディスクキャッシュのフラッシュが妨げられたり、キーボードやマウス入力への応答が停止するなどの原因となる可能性があります。有害な影響がないことを確認できる場合にのみ、スレッドの優先度を最高値に設定してください。

スレッドに動的な優先度クラスが割り当てられると、オペレーティング・システムが優先度を上げることがあります。スレッドの優先度を動的に引き上げる必要がある場合は、SetThreadPriorityBoost() (英語) 関数を使用してアプリケーションの必要性に応じて優先度の引き上げを有効または無効にできます。GetThreadPriorityBoost() (英語) を使用して、実行時に特定のスレッドの優先度の引き上げ状態を確認します。

2.9.3. SetThreadInformation

SetThreadInformation() (英語) を使用すると、スレッドの電力スロットリングやメモリーの優先度を制御できます。これは、「弱いアフィニティー」API 関数のクラスに分類されますが、オペレーティング・システムがスレッドを扱う方法を制御するには、いくつかの操作モードがあります。最初のモードは、**SetThreadInformation** により特定のスレッドの電力スロットリングを制御する方法です。これにより、スレッドの電力/パフォーマンスの比率を制御できます。電力スロットリングには、有効、無効、そして自動の 3 つのモードがあります。電力スロットリングが有効になっている場合、OS は論理プロセッサーのパフォーマンスに上限を設定することで電力効率を高め、Efficient-core などより電力効率の高い論理プロセッサーを選択することができます。電力スロットリングを有効にするには、**THREAD_POWER_THROTTLING_STATE** 構造体を作成して、**THREAD_POWER_THROTTLING_EXECUTION_SPEED** を **ControlMask** と **StateMask** に設定します。

```
inline bool EnablePowerThrottling(HANDLE threadHandle)
{
    THREAD_POWER_THROTTLING_STATE throttlingState;
    RtlZeroMemory(&throttlingState, sizeof(throttlingState));

    throttlingState.Version = THREAD_POWER_THROTTLING_CURRENT_VERSION;
    throttlingState.ControlMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;
    throttlingState.StateMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;

    return SetThreadInformation(threadHandle, ThreadPowerThrottling,
                                &throttlingState, sizeof(throttlingState));
}
```

重要な注意点:

独自のスケジューラーを使用している場合、スレッドが実行される場所を完全に把握して、スレッドが最大スピードで実行されることに集中します。より高いパフォーマンスを必要とし、電力効率を心配する必要がない場合は、電力スロットリングを無効にすることもできます。電力スロットリングを無効にするには、**ControlMask** を **THREAD_POWER_THROTTLING_EXECUTION_SPEED** に設定し、**StateMask** をゼロに設定します。

```
bool DisablePowerThrottling(HANDLE threadHandle)
{
    THREAD_POWER_THROTTLING_STATE throttlingState;
    RtlZeroMemory(&throttlingState, sizeof(throttlingState));

    throttlingState.Version = THREAD_POWER_THROTTLING_CURRENT_VERSION;
    throttlingState.ControlMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;
    throttlingState.StateMask = 0;

    return SetThreadInformation(threadHandle, ThreadPowerThrottling,
                          &throttlingState, sizeof(throttlingState));
}
```

デフォルトでは、電源スロットリングを有効または無効にすることを明確に選択しないかぎり、システムは電源スロットリングを決定する独自の方針を取ります。スレッドをデフォルトの「自動」電力スロットリングに戻すには、**ControlMask** と **StateMask** の両方をゼロに設定します。

```
bool AutoPowerThrottling(HANDLE threadHandle)
{
    THREAD_POWER_THROTTLING_STATE throttlingState;
    RtlZeroMemory(&throttlingState, sizeof(throttlingState));

    throttlingState.Version = THREAD_POWER_THROTTLING_CURRENT_VERSION;
    throttlingState.ControlMask = 0;
    throttlingState.StateMask = 0;

    return SetThreadInformation(threadHandle, ThreadPowerThrottling,
                                &throttlingState, sizeof(throttlingState));
}
```

SetThreadInformation は、特定のスレッドのメモリー優先度を制御します。メモリー優先度を使用すると、オペレーティング・システムはトリミングの前に、ページをワーキングセットに保持する時間を制御できます。優先度の低いページは、優先度の高いページの前にトリミングされます。ファイルやデータに頻繁にアクセスしないバックグラウンド・ワーカースレッド、テクスチャー・ストリーミング、設定やゲームファイルをディスクから読み取るスレッドなど、メモリーの優先度を下げた方が良い結果をもたらす場合があります。

メモリー優先度は、開発者がスレッド化のロジックを完全に制御するのを望むもう 1 つの領域です。例えば、AI システムはかなりの帯域幅を利用する傾向がありますが、早期にキャッシュをダンプすることが多く、一方、バーテックスを介して動作するスレッドは、より長い時間スレッドを保持することがあります。これらの操作を微調整することでゲーム性を大幅に向上できます。

```
bool SetMemoryPriority(HANDLE threadHandle, UINT memoryPriority)
{
    MEMORY_PRIORITY_INFORMATION memoryPriorityInfo;
    ZeroMemory(&memoryPriorityInfo, sizeof(memoryPriorityInfo));

    memoryPriorityInfo.MemoryPriority = memoryPriority;

    return SetThreadInformation(threadHandle, ThreadMemoryPriority,
                                        &memoryPriorityInfo,
sizeof(memoryPriorityInfo));
}
```

2.9.4. SetThreadSelectedCPUSets

CPU セットは、OS の電力管理と互換性のある「ソフトな」方法でアプリケーションのスレッド・アフィニティーを制御する API を提供します (ThreadAffinityMask API とは異なります)。さらにこの API は、プロセス内の OS スレッドとの干渉を避けるため、**プロセス・デフォルト**・メカニズムを利用して、プロセス内のすべてのバックグラウンド・スレッドをプロセッサーのサブセットに再初期化する機能を提供します。SetThreadSelectedCPUSets() (英語)は、unsigned long に格納された CPU ID 番号のリストまたは配列を受け取ります。各論理プロセッサーの CPU ID は、GetSystemCpuSetInformation() (英語) で取得できます。**SetThreadSelectedCPUSets** は、**SetThreadAffinityMask** で定義される 64 個の論理プロセッサーの制限を超える可能性があります。

```
{
        unsigned long size;
        // 現在のプロセスハンドルを取得します
        HANDLE curProc = GetCurrentProcess();

        // データ構造体内の要素の総数 (サイズ) を取得します。
        GetSystemCpuSetInformation(nullptr, 0, &size, curProc, 0);

        // 最初の呼び出しで返されたサイズをベースにデータ構造体を割り当てます。
        std::unique_ptr<uint8_t[]> buffer(new uint8_t[size]);
        PSYSTEM_CPU_SET_INFORMATION cpuSets =
reinterpret_cast<PSYSTEM_CPU_SET_INFORMATION>(buffer.get());
        PSYSTEM_CPU_SET_INFORMATION nextCPUSet;

        // すべての CPUSet 要素を取得します
        GetSystemCpuSetInformation(cpuSets, size, &size, curProc, 0);

        nextCPUSet = cpuSets;

        // 各論理プロセッサーで反復を処理します。
        for (DWORD offset = 0;
        offset + sizeof(SYSTEM_CPU_SET_INFORMATION) <= size;
        offset += sizeof(SYSTEM_CPU_SET_INFORMATION), nextCPUSet++)
        {
            // CPU セットのタイプが有効であることを確認します
            if (nextCPUSet->Type == CPU_SET_INFORMATION_TYPE::CpuSetInformation)
            {
                // 後で使用するため論理プロセッサー情報を保存します。
                LOGICAL_PROCESSOR_INFO core;

                // GetSystemCpuSetInformation から CPUID を読み取ります
                core.id = nextCPUSet->CpuSet.Id;
                // ... ここで追加の CPU セット・プロパティーを読み取ります ...
            }
        }

        return true;

    }
```

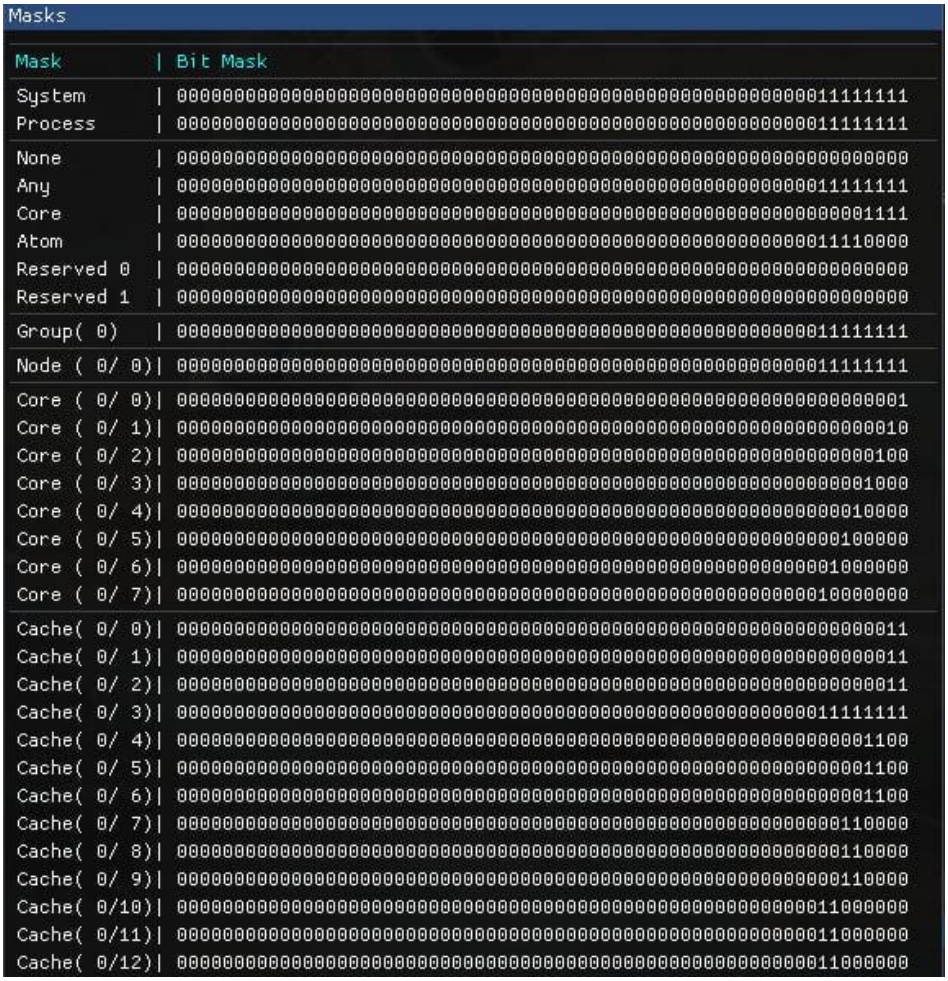
論理クラスターの作成は **SetThreadAffinityMask** と同様に機能しますが、64 ビット・マスクを保存する代わりに、アプリケーションがターゲットとする論理クラスターを表す CPU ID の配列またはベクトルを保持する点が異なります。

2.9.5. SetThreadAffinityMask

SetThreadAffinityMask() (英語) は、Windows* API 関数の「強い」アフィニティー・クラスに属します。特定のスレッドを最大 64 個の論理プロセッサーのいずれで実行するか制御するために、64 ビット・マスクを使用します。64 個の論理プロセッサーを超える場合、**GroupMask** を使用できます。しかし、CPU セットのような別の選択肢が望ましいと思われます。**SetThreadAffinityMask** は、基本的にオペレーティング・システム間との取り決めであり、ビット・マスクで指定された論理プロセッサー上でのみスレッドが実行されることを保証します。スレッドを実行できる論理プロセッサー数を減らすことで、スレッドの総合的なプロセッサー時間を短縮できます。**SetThreadAffinityMask** で強いアフィニティーを設定する場合、スレッドのスケジュールを完全に制御するケースを除き注意が必要です。**SetThreadAffinityMask** を適切に使用しないと、OS がスレッドをスケジュールする際に制約が多くなり、OS スケジューラーと競合してパフォーマンスが低下する可能性があります。

SetThreadAffinityMask を使用して、スレッドシステムを論理プロセッサーのクラスターにセグメント化できます。**SetThreadAffinityMask** により、論理 P-core クラスターと論理 E-core クラスターで構成される 2 つの論理スレッドプールを持つスレッド・スケジューラーを作成することもできます。オペレーティング・システムは、指定されたクラスターマスク内でスケジュールを行いますが、ITD による電力やパフォーマンス最適化の恩恵は得られません。また、スレッドをシングル・プロセッサーに固定する目的で利用できます。ピニング (固定) は理想的な方法ではなく、アトミックな操作を実行する場合や、論理プロセッサーの CPUID を読み取る場合など、特殊なケースでのみ使用してください。アフィニティー・マスクを実行時に設定すると、即時にコンテキスト・スイッチが発生する可能性があるため、可能な限り利用を避けるべきです。可能であれば、実行時ではなく初期化時にのみアフィニティー・マスクを設定します。実行時にアフィニティー・マスクを入れ替える必要がある場合、頻度を少なくしてコンテキスト・スイッチの発生を減らすようにします。

論理プロセッサー・クラスターは、同じタイプのプロセッサーに限定されません。目的に応じて複数のアフィニティー・マスクを作成することもできます。これらには、キャッシュ・マップ・マスク、システム/プロセスマスク、クラスターマスク、および特殊ユースケース向けの新たなマスクが含まれます。次の図は、アフィニティー・マスクを使用した一般的なビットのマスクマップを示しています。



Masks

Mask	Bit Mask
System	00000000000000000000000000000000000000000000000000000000000011111111
Process	00000000000000000000000000000000000000000000000000000000000011111111
None	00000000000000000000000000000000000000000000000000000000000000000000
Any	00000000000000000000000000000000000000000000000000000000000011111111
Core	00000000000000000000000000000000000000000000000000000000000000001111
Atom	00000000000000000000000000000000000000000000000000000000000011110000
Reserved 0	00000000000000000000000000000000000000000000000000000000000000000000
Reserved 1	00000000000000000000000000000000000000000000000000000000000000000000
Group(0)	00000000000000000000000000000000000000000000000000000000000011111111
Node (0/ 0)	00000000000000000000000000000000000000000000000000000000000011111111
Core (0/ 0)	00000000000000000000000000000000000000000000000000000000000000000001
Core (0/ 1)	00000000000000000000000000000000000000000000000000000000000000000010
Core (0/ 2)	00000000000000000000000000000000000000000000000000000000000000000100
Core (0/ 3)	00000000000000000000000000000000000000000000000000000000000000001000
Core (0/ 4)	00000000000000000000000000000000000000000000000000000000000000010000
Core (0/ 5)	00000000000000000000000000000000000000000000000000000000000000100000
Core (0/ 6)	00000000000000000000000000000000000000000000000000000000000001000000
Core (0/ 7)	00000000000000000000000000000000000000000000000000000000000010000000
Cache(0/ 0)	00000000000000000000000000000000000000000000000000000000000000000011
Cache(0/ 1)	00000000000000000000000000000000000000000000000000000000000000000011
Cache(0/ 2)	00000000000000000000000000000000000000000000000000000000000000000011
Cache(0/ 3)	00000000000000000000000000000000000000000000000000000000000011111111
Cache(0/ 4)	00000000000000000000000000000000000000000000000000000000000000001100
Cache(0/ 5)	00000000000000000000000000000000000000000000000000000000000000001100
Cache(0/ 6)	00000000000000000000000000000000000000000000000000000000000000001100
Cache(0/ 7)	00000000000000000000000000000000000000000000000000000000000000110000
Cache(0/ 8)	00000000000000000000000000000000000000000000000000000000000000110000
Cache(0/ 9)	00000000000000000000000000000000000000000000000000000000000000110000
Cache(0/10)	00000000000000000000000000000000000000000000000000000000000011000000
Cache(0/11)	00000000000000000000000000000000000000000000000000000000000011000000
Cache(0/12)	00000000000000000000000000000000000000000000000000000000000011000000

図 4.コアとキャッシュのビットのマスクマップ

2.9.6. 論理トポロジー

ハイブリッド・アーキテクチャーをサポートするプロセッサー向けのスレッドシステムを設計する場合、論理プロセッサーをいずれのタイプの論理トポロジーに分割するか検討する必要があります。開発コード名 Alder Lake プロセッサーでは、P-core プロセッサーを E-core プロセッサーから分離するスプリットトポロジーを採用するのが明確な選択です。しかし、特定のアプリケーションの要求に応じて、別の論理トポロジーを検討することもできます。代替案として、P-core プロセッサーのみ、E-core プロセッサーのみ、または P-core プロセッサーと E-core プロセッサーを 1 つの論理クラスターにまとめて使用することができます。さらに、P-core、E-core、SMT コアを区別する必要がある場合、スプリットトポロジーを 3 つのトポロジーに拡張できます。まれに、特殊用途のため新しい論理トポロジーを作成することもあります。

キャッシュ・コヒーレンシーが不可欠なアプリケーションでは、プロセッサーの EfficiencyClass に変わってキャッシュトポロジーに基づいて論理クラスターを作成できます。この設計方式は「キャッシュマッピング」と呼ばれ、一般に関連付けられているキャッシュに従ってプロセッサーをグループ化することで、キャッシュ・コヒーレンスのコストを削減することができます。開発コード名 Alder Lake では、論理 E-core プロセッサーのレベル 2 キャッシュは、4 つの論理プロセッサー・バンクで共有されています。E-core プロセッサーの L2 キャッシュ周辺のクラスターのリングは、LLC よ

りも L2 キャッシュのコヒーレンスを向上させる可能性があります。論理 P-core プロセッサーと論理 E-core プロセッサー間でキャッシュを共有する場合、最終レベルのキャッシュ・コヒーレンスが制限されます。

論理クラスターをターゲットとするスレッドを生成する場合、スレッドの利用目的に応じて、弱いアフィニティーと強いアフィニティーの混在を検討してください。例えば、バックグラウンドのスレッドは、低いメモリー優先度と電力スロットリングの恩恵を受け、E-core プロセッサーでの実行に適しています。また、リアルタイム・スレッドは、リアルタイムの優先度、より高いメモリー優先度、電力スロットリングの無効化を必要とし、P-core プロセッサーでの実行に適します。アプリケーションの設計時にスレッドを分類するため、次のような簡単なアフィニティーのリストを作成できます。

表 3.サンプルのスレッド・アフィニティーの表

スレッド	SetThreadPriority	SetThreadInformation (メモリー優先度)	SetThreadInformation (電力スロットリング)	SetThreadSelectedCPUSets
メインスレッド	THREAD_PRIORITY_NORMAL	MEMORY_PRIORITY_NORMAL	無効化	コアのスレッドプール
オーディオスレッド	THREAD_PRIORITY_BELOW_NORMAL	MEMORY_PRIORITY_BELOW_NORMAL	自動	すべてのスレッドプール
レンダースレッド	THREAD_PRIORITY_TIME_CRITICAL	MEMORY_PRIORITY_NORMAL	無効化	コアのスレッドプール
ワーカースレッド	THREAD_PRIORITY_NORMAL	MEMORY_PRIORITY_MEDIUM	自動	すべてのスレッドプール
バックグラウンド・スレッド	THREAD_MODE_BACKGROUND_BEGIN	MEMORY_PRIORITY_LOW	有効化	E-core のスレッドプール

3. パフォーマンス最適化

開発コード名 Alder Lake は CPU アーキテクチャーが大幅に変更されたため、両方のコアタイプを最大限に活用するには、アプリケーションを微調整する必要があるかもしれません。これは、複雑なジョブ管理を行うマルチスレッド・アプリケーションや、複数の用途や環境で使用されるミドルウェアでは特に重要です。それ以外では、開発コード名 Alder Lake のパフォーマンス・ハイブリッド・アーキテクチャーを考慮して OS スケジューラーが最適化されており、その影響を最小限に抑えることができます。

ハイブリッド・アーキテクチャー上でゲームを解析したところ、大半のゲームは良好に動作し、古いゲームや負荷の低いゲームでは Performance-core が適していました。高度にマルチスレッド化され、2 桁のコア数までスケールするゲームでは、ハイブリッド・アーキテクチャーによってスループットの向上が得られることが分かりました。しかし、マルチスレッド・ゲームの設計に不備があったり、OS のスケジュールが上手く機能しなかったり、スレッドのオーバーヘッドが増加することで、パフォーマンスが逆転することもあります。このような問題には、インテル® スレッド・ディレクター・テクノロジー、OS スケジューラーへのヒント (Microsoft との協業)、およびそれぞれのスレッド化ライブラリーを導入することで対応しています。開発コード名 Alder Lake をはじめとするハイブリッド・アーキテクチャーで最高のパフォーマンスを発揮するには、OS、ツール、ライブラリーを定期的に更新し、開発環境を最新の状態に維持する必要があります。

4. ハイブリッドを有効にする一般的なガイドライン

開発コード名 Alder Lake のパフォーマンス・ハイブリッド・アーキテクチャーは、ソフトウェアを開発する過程にユニークな可能性と課題をもたらします。以下は、新しい CPU ファミリーのパフォーマンスを最大限に発揮するためのガイドラインと可能性です。

4.1. コア数の増加によるスケーラビリティーの欠如

検証の結果、一部のレガシー・ソフトウェアでは、コア数を増やすとパフォーマンスが逆転することが判明しました。これは 開発コード名 Alder Lake に限ったことではなく、スケーラビリティーの問題であるといえます。スレッド数が増加すると、それを管理するオーバーヘッドも大きくなります。スレッドシステムに十分なワークが供給されない場合、オーバーヘッドがワークの分散化の利益を上回る可能性があります。開発者は、システムで利用可能なコア数に合わせて安易にスケールするのではなく、設定するしきい値よりも利点が下回るポイントに合わせてスケールすべきです。これにより、OS はハードウェアのリソースを管理し、未使用のコアをパーキングしたり、電力を必要とするシステムに供給することで、周波数が向上する可能性があります。将来的には、アプリケーションがゲームのスレッド化コード内に独自のフィードバック・システムを備え、ワークがゲームのクリティカル・パスのタイミングに影響するかどうかに応じて、スレッド化の範囲を調整することが可能になるでしょう。

4.2. 特定の命令セット・アーキテクチャー (ISA) をターゲットにする

これは一般には推奨されません。ISA の違いによって性能差があっても、ITD はそれぞれのコアのランタイム状態を OS に提供できるため、OS は与えられた電力/熱の制約内で適切なコアを選択し、その ISA を最も効率良く実行できます。

4.3. 動的な負荷分散/ワークスチール

独自のスレッドプールを作成する場合、一部のコアがほかのコアよりも早くタスクを終える可能性がある開発コード名 Alder Lake 世代の CPU のハイブリッド特性を考慮して最適化を行う必要があります。その場合、2 つのコアタイプにはパフォーマンスの差があるため、動的な負荷分散とワークスチールによってさらに利点が得られる可能性があります。

4.4. アクティブスピンを UMWAIT や TPAUSE に置き換える

アクティブスピンは、多くのクライアント・アプリケーション (クリエーター/生産性およびゲームセグメント) で利用されます。消費電力とパフォーマンスの両方を考慮すると、アクティブスピンを MWAIT (UMWAIT) やスレッドポーズ (TPAUSE) 命令に置き換える必要があります。これらの命令の使用方法は、『インテル® アーキテクチャー命令セット拡張機能および将来の機能のプログラム・リファレンス』(英語) に記載されています。

UMWAIT と TPAUSE 命令は、コアを低消費電力のアクティブな C-ステート (C0.1 や C0.2) にすることで、電力を節約してクライアント上でスピンを必要としません (例えば、PAUSE 命令はクライアントで 140 コアサイクルを消費します)。UMWAIT/TPAUSE および PAUSE 命令は、C0.1 または C0.2 サイクル増加させまが、これは ITD によって検出されます。

ハイブリッド・アーキテクチャーでは、コア数が増えるとスピン時間が増加します。これは、スレッド化ライブラリーが **parallel_for** やその他の並列構造でスピンを行うためです。アクティブスピンは次のような状況で発生します。

1. ハードウェア・リソースを待機する場合
2. あるスレッドがほかのスレッドを待機する場合 (生産/消費モデルの場合)
3. 競合によりロックをバックオフする場合
4. スレッドプールの動作終了時に、システムがスレッドプールのスレッド・インバランスを検出した場合

parallel_for を閉じるなど、並列処理で生成されたスレッドプールは、別の構造を予想してスピンすることがあります。

4.5. ゲームへ影響するハイブリッド・アーキテクチャー

ゲームは開発コード名 Alder Lake-S (ADL-S) にとって重要なセグメントであり、パフォーマンスの低下は許容されません。多くのゲームは GPU に依存しますが、ハイエンドのディスクリート GPU で動作する上位のタイトルは、CPU に依存することもあります。ゲームは通常、1 または 2 つの主要スレッドが CPU 依存であり、論理プロセッサー数または物理プロセッサー数に列挙されたタスクシステムを持っています。クリティカルなゲームスレッドからワークを取り出すことは困難であるため、アムダールの法則によりコア数が 6 - 8 コアを超えるスケーリングが妨げられることがありますが、これはゲームが 6 - 8 以上のスレッド/コアを使用しないことを意味するものではありません。

解析の結果では、マルチスレッドの問題、OS のスケジュール、またはスレッドのオーバーヘッド増加に起因するパフォーマンスの逆転が示されています。

4.6. クリティカル・パス

拡張クリティカル・パスは、プログラムで実行されたすべてのコードセグメントとして定義され、スモール ∑ でレデュースすると、特定のプロセッサー数での完了時間が短縮できます。

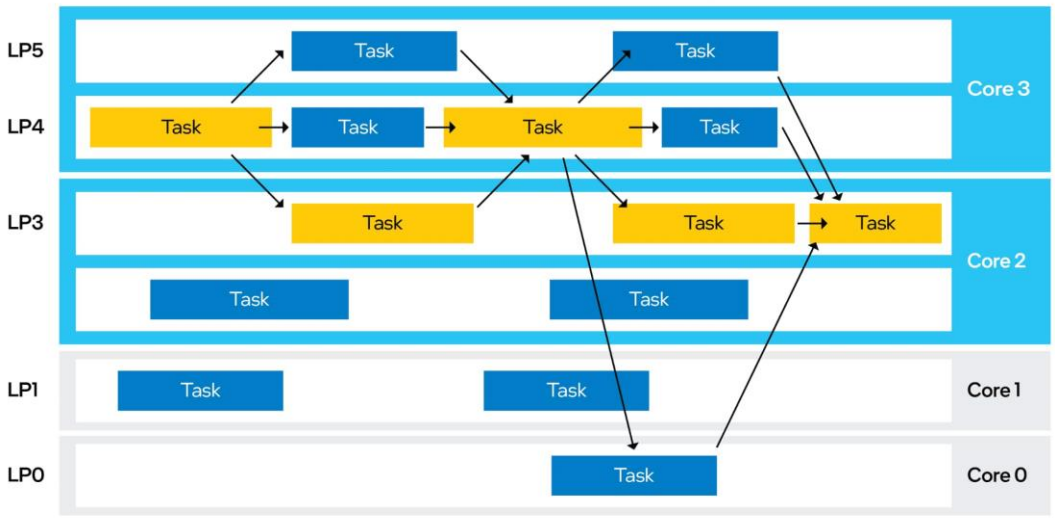


図 5. 4 コアのクリティカル・パスの図。P-core は青色、E-core は灰色で示され、
クリティカル・パスは黄色、非クリティカル・パスは紺色で示されています。

物理コア上の共有リソースは、論理プロセッサー (LP) の IPC に影響します。関数呼び出しの時間は、スケジュールされるコアタイプに影響を受けます。スレッドプール内のロングテール・タスクは、Efficient-core にスケジュールするとパフォーマンスが低下します。

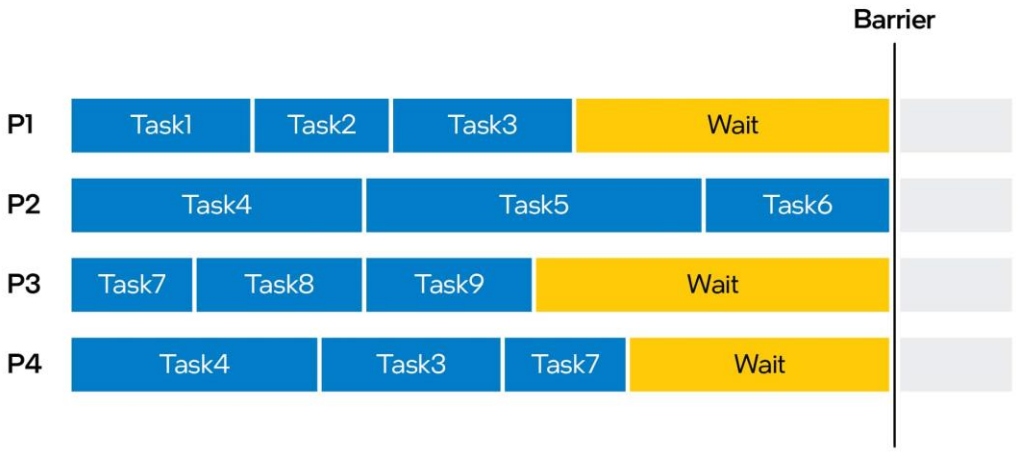


図 6. 待機時間のあるロングテール・タスクはスレッドのパフォーマンスを低下させます。

4.7. 最適化の方針を選択 - デスクトップ

デスクトップ・システムの最適化方針を決定する際は、次のシナリオを考慮してください。

4.7.1. 最適化なし

ITD からのフィードバックをもとに、OS スケジューラーがスレッドをインテリジェントにスケジューリングし、ワークロードを動的に配置します。これにより、ソフトウェアでスケジューリングを行う開発者の労力が解消されます。アプリケーションが最適化されていない場合、ITD はそのアルゴリズムに基づいてワークロードの配置を試みます。この配置は通常、パフォーマンスの向上につながりますが、状況によっては、一部の非クリティカル・タスクが Performance-core に割り当てられ、一部のクリティカル・パスのタスクが Efficient-core に割り当てられる可能性があります。特に、アプリケーションが複数のミドルウェアのコンポーネントを使用する場合、競合の可能性を無視して独自のスレッドシステムを作成するとその可能性が高くなります。開発者は、アプリケーションのスレッド化アルゴリズムを確認し、次のシナリオのいずれかを選択してください。

4.7.2. 「良い」シナリオ

「良い」シナリオには、アプリケーションがハイブリッドを認識する最小限のステップが含まれており、タスクシステムの複雑な書き換えは必要ありません。

- プライマリー・ワークロードは、Performance-core をターゲットにする必要があります。
    - Performance-core の数、またはワークロードに必要なスレッドの最大数に基づいて、タスクシステムを列挙します。
    - タスクシステムが負荷分散できることを確認します。

- スレッド優先度/サービス品質 (QoS) API を使用して、適切なワークを適切なコアに割り当てます。

    - ゲーム/レンダースレッドの優先度を通常よりも高く設定し、Performance-core をターゲットにします。
    - タスクシステムのスレッドを通常の優先度に設定し、Performance-core への配置を促します。
    - 必要に応じて、バックグラウンド・スレッドの優先度を通常よりも低く設定し、Efficient-core をターゲットにします。

4.7.3. 「最良の」シナリオ

最高のパフォーマンスを維持しながら、完全にハイブリッドを考慮したタスクシステムを構築するには、2 つのスレッドプールを作成することが望まれます。

- プライマリーのスレッドプールは Performance-core をターゲットにします。Performance-core にのみ配置する、または優先させるジョブを実行します。
- セカンダリーのスレッドプールは Efficient-core をターゲットにします。このプールは、次のような Efficient-core に適したジョブを実行します。

    - シェーダーのコンパイル
    - オーディオのミキシング
    - アセットのストリーミング
    - デコンプレッション
    - その他のクリティカルではないワーク

システムをさらに最適化して負荷分散と利用率を高めるには、Performance-core が過負荷のときに Efficient-core に余裕があるならば、タスクをプライマリーからセカンダリーのスレッドプールに「オフロード」するタスク・スチール・アルゴリズムを実装する必要があります。これは、Performance-core の数が制限されているノートブック PC の SKU では特に重要です。

4.8. 最適化の方針を選択 – ノートブック

モバイル SKU は通常、熱設計電力 (TDP) が制限されており、ノートブック・コンピューターで適切なパフォーマンスを達成するには、電力に特化した最適化を検討する必要があります。これは特に、インテル® グラフィックスをプライマリー GPU とする Ultrabook™ デバイスやコンバーチブル・デザインに当てはまります。

5. ツールと互換性

開発コード名 Alder Lake プラットフォームは、Microsoft をはじめとする OS やシステム・ソフトウェアの開発者、および主要なゲームエンジンによってサポートされます。次のリストは増え続けています。

ランタイム

- .NET
- JS
- MSVC++

パフォーマンス・ツール

- MSVC++
- インテル® VTune™ プロファイラー
- インテルの SEP/EMON
- インテル® SoC Watch
- インテル® SPMD プログラム・コンパイラー

UMWAIT をサポートするスレッド化ライブラリー

- インテル® スレッディング・ビルディング・ブロック (インテル® TBB)
- MSVC++
- OpenMP*

注: インテル® TBB ユーザーが開発コード名 Alder Lake を完全にサポートするには、最新のインテル® oneAPI スレッディング・ビルディング・ブロック (インテル® oneTBB) API に移行する必要があります。

5.1. インテル® VTune™ プロファイラー

インテル® VTune™ プロファイラーは、開発コード名 Alder Lake を完全にサポートし、イベントやメトリックをコアタイプ別にグループ化するコア・グルーピング機能を備えています。事前定義されたグループのセットが用意されていますが、カスタムセットを作成することもできます。

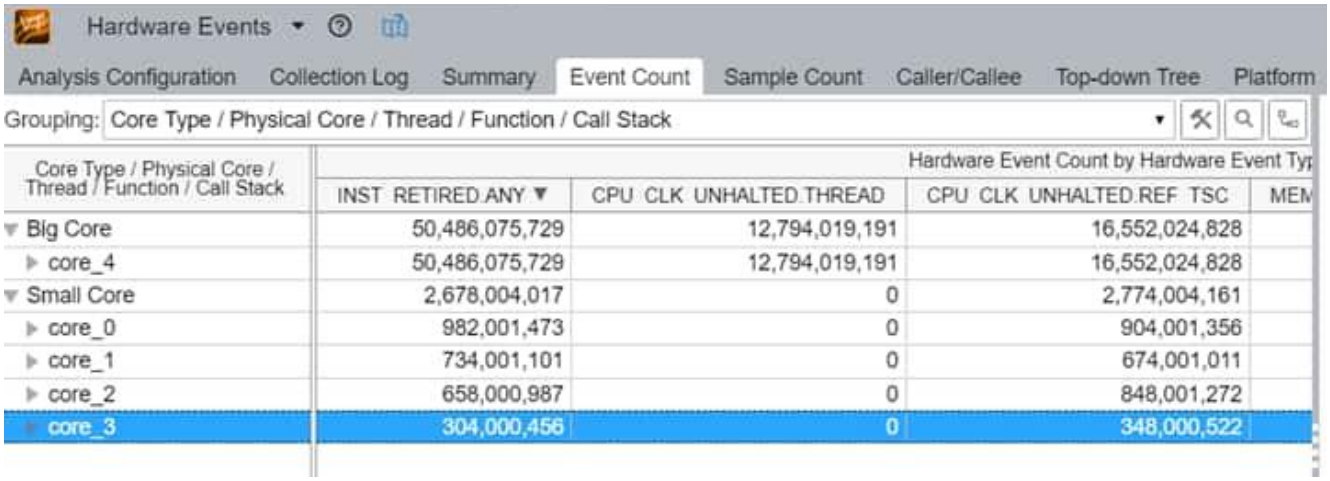


図 7. インテル® VTune™ プロファイラーと事前定義されたグループ

6. ゲームエンジン・サブシステムにおける具体的な最適化とシナリオ

ゲーム開発者にとって開発コード名 Alder Lake プラットフォームの最大の利点は、ハイパフォーマンスなコアが高いクロックで動作することで、最適化されたシステムがクリティカル・パスのタスクを可能な限り高速に実行できることです。タスクの優先度を決定することは、クリティカル・パスのタスクを処理するためアーキテクチャーを最大限に活用する上で重要です。

メインのゲームスレッドは、ほかのスレッドを開始し、ほかのタスクを管理します。効率良いパフォーマンスのため、開発者はメインスレッドを可能な限り軽量にする必要があります。開発コード名 Alder Lake アーキテクチャーで効率良いパフォーマンスを可能にする最初の作業として、並列化可能なワークロードはすべて「ワーカースレッド」に移行します。これは全く新しい概念ではなく、ほとんどの開発者は並列実行の管理として経験していることでしょう。開発コード名 Alder Lake ハイブリッド・アーキテクチャーでは、メインスレッド (通常はジョブ管理を含む) が開発コード名 Alder Lake アーキテクチャーを完全に識別し、タスクを Performance-core または Efficient-core のクラスターに適切に配置する必要があります。

6.1. レンダースレッド

レンダースレッドは、P-core で動作するときに最大のパフォーマンスを発揮し、レンダリングのため GPU に情報を送信するロジックを処理します。これは、レンダリング・パスごとにオブジェクトのリストを解析し、GPU が処理できるようデータをセットアップする非常に負荷が高いワークです。大部分のレンダリング・エンジンは、これらのタスクを可能な限り早い段階で処理しようとします。例えば、光源の強度がオブジェクトによってブロックされていたり、または光の強度がゼロに近いとエンジンが判断すると、GPU 自体がパイプラインの早い段階でそれを拒否できる場合でも、エンジンはシーンに何も影響しないと判断してその光を拒否しようとします。レンダリング・スレッドが早い段階で拒否することで、その光のデータを GPU のデータ構造に格納する CPU の処理時間を大幅に削減できます。

6.2. CPU によるオクルージョン・カリングを含むシーンの可視性

シーンの可視化を理解することは、効率良いレンダリングと高いパフォーマンスには非常に重要です。これらのタスクは、エンジンがレンダリング・ワークを開始し、そのワークを GPU に送信するためフレームの最初にできるだけ早く行う必要があり、タスクを高いパフォーマンスのコアで実行する必要があります。シーンの可視化には 2 つのタイプがあります。

1. BSP、ポータル、AABB ツリー、シーンクラスターなど、通常はエンジンのアーキテクチャーと深く関連する各種アルゴリズムに基づく階層的なエンジンシーンの可視性。
2. MOC (マスクされたオクルージョン・カリング) などの CPU でのオクルージョン・カリング。これは、シーンの可視化の第 2 段階であり、2 番目のパスを使用して、第 1 段階で除外されたものの不可視になる可能性があるため、別のビューのレンダリングから除外する必要があるオブジェクトの数を最小化します。

6.3. オクルージョン・カリング

オクルージョン・カリング (OC) では、完全に覆われたオブジェクトを GPU に送信して不要なレンダリングを行わないようにする「微調整」を行います。最新の OC は、次のフレーム、または同一フレームでのオクルージョン・テストの結果を使用して、GPU と CPU で実行できます。通常、OC アルゴリズムには、非常に高速なラスタライズ・アルゴリズムが含まれており、少ない解像度でテストされたジオメトリーのピクセルが深度テストを通過し、スクリーン上にレンダリングされるかどうか確認します。テストに合格したピクセルがゼロである場合、そのオブジェクトは「覆われている」と

定義され、メインのレンダリングから除外できます。マスクされたオクルージョン・カリング (MOC) などの CPU アルゴリズムでは、ラスタライズは高度にベクトル化されたコードにより CPU 上で実行されます。これにより GPU の時間が節約され、インテル® アーキテクチャーにも十分に最適化されます。

6.4. 階層型エンジンシーンの可視性

ゲームでは、すべてのフレームで、カメラの視点から見えるオブジェクトのリストを決定する必要があります。ゲームワールドのオブジェクトは通常、検索を容易にするため何らかのデータ構造で編成されていますが、これらのデータ構造はシーン内を移動するオブジェクトへの対応が難しい場合があります。動的なオブジェクトが多数ある場合、これらのデータ構造を維持するのは CPU にとって負担になります。さらに、ゲーム内のすべてのカメラビューに対する保守が必要になる場合があります。

フレーム中で複数のカメラビューによるシーンを構成することがあります。メインプレイヤーの視点はメインカメラですが、ゲームではスパイゲームのセキュリティー映像や影や照明に、ほかのカメラを使用したりします。

環境マップではライティングがよりリアルになる傾向があります。通常環境は、ある世界の特定の場所からその世界を立方体の側面に 6 回レンダリングして作成されます。つまり 6 つのカメラビューがあります。多くのライティング・モデルでは、正確なライティングのため、1 つのエリアに数十枚の環境マップを使用します。例えば、Rockstar Games* は、「GTA オンライン: ロスサントス・チューナー」で、1 つのエリアに 16 枚の環境マップを使用する環境マップ・ライティング・システムを開発しました。これだけ多くの場所のシーンを可視化する計算を CPU で行うにはコストがかかるため、1 フレームでは 2 つの環境マップしか更新できませんでした。この問題の解決策は、16 枚の環境マップをラウンドロビン方式で更新することで、すべての環境を更新するのに 8 フレームが必要でした。Rockstar では、異なる環境マップ間で表示されるアイテムを共有したり、プレイヤーが見ている方向に近い環境マップや、カットシーンからカメラが切り替わったときに優先的に更新されるようなロジックを追加しています。全体的な要点は、すべての環境マップをフレームごとに更新するには CPU パワーが足りませんが、ある仮定を行うことでコストを処理できたということです。

6.5. Efficient-core

E-core は低い周波数で動作するため、実行する命令のパフォーマンスは低くなりますが、電力効率は高くなります。モバイル SKU では、E-core が MOC の主力となる可能性があります。そのため、ほとんどの開発者は非クリティカルな並列ワークロードを E-core に移行することを考慮するでしょう。

それらのワークロードについては以降で説明しますが、通常は次の基準を満たしています。

要件:

- 非同期
- クリティカル・パス外部

オプション:

- Performance-core に 100% の負荷をかけない。
- ベクトル命令セットを使いすぎない。

6.5.1. AI

AI の計算は非同期で実行でき、フレームレートにも影響しないため、E-core への移行には適しています。

Assassin's Creed* の初期バージョンでは、多数の非プレイヤー・キャラクター (NPC) が相互に影響しながら歩き回るため、CPU にストレスを与えていました。さらに、AI データがランダムにアクセスされることで、リソースの過度な利用やキャッシュシステムの競合などのストレスが発生します。プレイヤーがこれらの群衆に接近すると、各 NPC の行動は設計者が設定した規則に基づいてモデル化されなければなりません。NPC は恐怖を感じるているか、好奇心を持っているか、または興味を持っているか? どのようにしてその状態になり、そしてどのような行動をとるのか? NPC は、警戒、攻撃、探索、冷静など、何を引き金に次の状態に遷移するのか? その状態で相互に交流はあるのか? ゲーム中これらに対処するため AI が使用されます。多数の NPC 向けの処理は、その数にもよりますが CPU にとっては重いタスクです。AI の計算を E-core に移行することで、CPU の負荷を軽減できます。

6.5.2. キャラクター・アニメーション

ほとんどのゲームにはキャラクターのアニメーションがありますが、そのほとんどは GPU を利用したスキニングで処理されます。しかし、多くのゲームでは、よりリアルなアニメーションを実現するため CPU が補間処理を行います。

例えば、ゲームエンジン Frostbite* では、大容量の高品質なアニメーション・データを CPU が展開しています。多くのゲームスタジオでは、ゲーム・キャラクターの高品質なアニメーションを作成する際にモーション・キャプチャーを使用しますが、このデータを効率良く圧縮できます。その後、パイプラインを通して GPU に送信される前に、データを展開しなければなりません。Frostbite* は、このプロセスを高品質のアニメーションで処理していますが、画面上の多くのキャラクターが存在してそれらが異なる動きをする場合、かなりの時間を要します。

Frostbite* や CRYENGINE* などのゲームエンジンは、アニメーション・データの地形や衝突を調整し、「逆運動学」と組み合わせて武器を正確に表現します。武器の照準は、一般に上、下、左、右の 4 つのアニメーションがブレンドされます。イベント向けの高品質なモーション・キャプチャー・データは、通常、平面など 1 つの環境で行われますが、ゲームでは多様な環境でキャラクターが動いています。そのため、エンジンが正しく動作するようにアニメーションを調整する必要があります。また、地面に足がついているか、どのサーフェス角度でも正しく体が配置されているか、なども チェックしなければなりません。クリエーターは、キャラクターが武器を直接ターゲットに向け、ほかのオブジェクトと衝突したときはリアルな反応をすることを望んでいます。これらのすべては、通常、CPU で実行されます。また、これらの調整は非同期で行うことができるため、このタスクは E-core アレイで実行する有力な候補となります。

6.5.3. 物理演算

物理演算は通常、非同期で固有の頻度で実行されるため、E-core アレイへ切り替える最適な候補です。物理演算のシミュレーションは、あらゆる 3D ゲームにおいて重要な要素です。没入感のある体験は、オブジェクト、環境、キャラクター、エフェクト間の複雑な物理演算ベースの相互作用に依存します。これには、衝突検出やオブジェクトの移動が含まれます。

ゲームエンジンは、物理演算のシミュレーションを独自に実装することも、Havok* Physics、Bullet などのサードパーティーの物理演算エンジンを使用することもできます。Havok などの物理演算エンジンは、独自のジョブ管理を使用してマルチスレッドの物理演算シミュレーションを行うため、ミドルウェア開発者は自身のエンジンが開発コード名 Alder Lake アーキテクチャーを完全に認識していることを確認する必要があります。

物理演算では、バランスを取る必要がある方程式のシステムを使用します。一般に、物理演算エンジンは自身の頻度、またはワールド/サーバー頻度で動作し、レンダリング・フレームレートとは関連ありません。そのため、物理演算は E-core への移行に適しています。

物理演算エンジンの中核は「ソルバー」と呼ばれます。ソルバーは、オブジェクトの物理的状態を表す方程式のバランスを取ろうとします。そして、方程式のバランスがとれると、物理オブジェクトは「休息状態 (resting state)」になります。ソルバーがこれを達成できないと、物理オブジェクトがガタついて見えます。

多くの場合、これらの方程式を直接解くことはできませんが、理論的な解決に向かって少しずつ前進し、「十分に近い」ところまで達したら完了とすることができます。「十分に近い」状態とは実際の解の 0.001% 以内であり、動きの速いゲームでは、プレイヤーはその違いに気づくことはないため、この状態は適切です。しかし、システムが正解に接近できず、ゲームの動作にガタつきが生じて没入感を阻害することもあります。特定のオブジェクトがほかのオブジェクトを移動または相互作用するのに制限を加えることで、この問題を回避できます。このような制限がガタつきの問題を回避できる例として、関節を大きく曲げられないラグドールがあります。

電車やトラック、飛行機、またはボートなどの内部でゲームをプレイする場合、その内装のワークロードは CPU の負荷となる可能性があります。そのような場合、すべてのゲームプレイのオブジェクトは自動的に物理的なオブジェクトとなり、物理演算エンジンはフレームごとにオブジェクトを追跡して解決しなければなりません。

外部 (エクステリア) 環境では、シミュレーションする動的オブジェクトがより広い空間で過多になる可能性があります。どちらの状況でも、パフォーマンスを維持するにはオブジェクト数のバランスを取ることが重要です。どうしても必要になるまでインタラクションを停止したり、一定時間経過したらインタラクションを停止することで、ショートカットできるようになります。

6.5.4. 経路探索

経路探索は、数百または数千の AI ユニットが登場する戦略型ゲームでは特に重要なアルゴリズムですが、そのほかのゲームでも利用されます。これは、2 点間の最短ルートを求めるもので、ある地点を起点に目的のノードに到達するまで隣接するノードを探索し、最も安価なルートを検出することを目的とします。経路探索の主な問題は、グラフ上の 2 つのノード間のパスを検出すること、および最適な最短パスを見つけることです。幅優先探索や深さ優先探索などの基本的アルゴリズムは、最初の問題に対してすべての可能性を示すことで対処します。つまり、与えられたノードから出発して、目的のノードに到達するすべてのパスを反復します。計算は反復的で短く、通常はフレームではなく「ワールド」の頻度で再計算されるため、経路探索は E-core への移行に適した候補です。

6.5.5. ラグドール物理を含む衝突

物理演算エンジンの重要な機能の 1 つに、ゲーム内のオブジェクト間の動き追跡、衝突の検出、および衝突の解決があります。ラグドール物理は、ビデオゲームにおける従来の静的な死体アニメーション (static death animation) に代わる手続き型のアニメーションの一種であり、適切に実行されると没入感に貢献します。これは、E-core への移行に適した操作といえます。

6.5.6. 破壊

破壊エンジンは、ソリッド・オブジェクトの破壊をシミュレートするもので、ゲーム体験のリアルさを左右する重要な入力をもたらします。破壊アルゴリズムの複雑なオーバーヘッドは、破壊によってその場で新しいオブジェクトが作成されるため、非常に複雑な入れ子になったシミュレーションが行われ、明示的なスレッドのスケジュールが必要になる

ことに由来します。この複雑性を克服するには、「事前に破壊された」オブジェクトを使用して、インスタンスを壊れたオブジェクトと入れ替え、適切なタイミングで物理演算の制御下に配置します。地形などの環境変化では、変形する高度マップやほかの同様な構成を実装できます。環境変化でさらに複雑なことは、ナビゲーション・メッシュの再構築です。

6.5.7. 流体力学

流体力学は一般に複雑な処理ですが、ほとんどの物理演算エンジンは、水、雲、天候、火などの特定のエフェクトに対するカスタム・ソリューションを実装することで、問題を単純化します。このようなカスタム・ソリューションを使用するかどうかは、ゲームの種類、没入型の流体力学を必要とするか、およびセットアップがどの程度重要であるかによって異なります。水のシミュレーションがゲームにおいて重要な場合、リアルな気象シミュレーションに割り当てる CPU 時間の優先度を下げることもできます。同様に、タイトルによっては、正確な火災シミュレーションが必要なく、シンプルな火災アニメーションで十分なこともあります。

6.5.8. パーティクルの物理演算

パーティクルの物理演算をスムーズに実装するのは、ゲームやゲームエンジンでは不可欠です。パーティクル・システムは基本的なものである可能性があり、その場合 GPU はそれらを効率良く実行できます。ステートレス・パーティクル・システムは、GPU でも効率良く動作します。しかし、環境やオブジェクトと複雑な衝突を伴うエフェクトがある場合、そのパーティクル・システムを管理するには CPU が適しています。パーティクル・システムの中には、負荷に応じて CPU と GPU を切り替えるものがあります。アニメーションもこの方法で処理できます。

6.5.9. サウンド

サウンドシステムは、Efficient-core (E-core) アレイに移動できる非同期ワークロードの良い例です。サウンド・ワークロードは通常、品質と利用可能な性能に対してスケーラブルであり、さまざまなフレームで実行できます。

ゲームでは通常、ミドルウェアを利用してさまざまな効果音、音楽、およびほかの機能を有効にします。ミドルウェアの提供元が開発コード名 Alder Lake への対応をどのように実装しているか確認する必要があります。サウンド・ミドルウェアが独自のスレッド/コア管理を行い、高い Performance-core に排他的にピニングすることで、サウンド・ミドルウェアが開発コード名 Alder Lake の問題となるケースを回避してください。

7. まとめ

開発者は、電力とパフォーマンスを最適化するため、システムで利用可能な論理プロセッサーを完全に理解する必要があります。パフォーマンス・ハイブリッド・アーキテクチャー (開発コード名 Alder Lake など) を考慮して、ソフトウェアを設計する必要があります。パフォーマンス・ハイブリッド・アーキテクチャーをサポートするには、次の手順でコードを準備します。

1. ハイブリッド・トポロジーの検出を有効にします。
2. パフォーマンスを最適化するためアーキテクチャーを選択します。
3. スレッドのスケジューリングは事前に計画します。

Performance-core と Efficient-core の両方を活用するハイブリッド・アーキテクチャーを最適化する際に直面するコーディング上の問題に慣れるため、インテルではサンプル・アプリケーションを作成し、ソースコードを公開しています。このドキュメントの冒頭にあるリンクから、アプリケーションをダウンロードしてください。

8. よくある問い合わせ (FAQ)

以下は、ゲーム開発者が開発コード名 Alder Lake アーキテクチャー向けの最適化を行う際に、問い合わせがある質問のリストです。

Q1: 開発コード名 Alder Lake プラットフォームはいつ市場に投入されますか? (英語)

A1: 開発コード名 Alder Lake プラットフォームは、2021 年第 4 四半期と 2022 年第 1 四半期に発売される予定です。

Q2: 開発コード名 Alder Lake はデスクトップ PC でのみ利用できますか? それともノートブック PC やその他のフォームファクターでも利用できますか? (英語)

A2: 開発コード名 Alder Lake プラットフォームは、エントリー・ワークステーション、デスクトップ PC、ノートブック PC、Ultrabook™ デバイス、およびさまざまな SKU を備えたコンバーチブル・フォームファクターで利用できます。

Q3: E-core (SONY* PS2* で採用されていたような) にアクセスする特別な API を使用する必要はありますか? (英語)

A3: いいえ。すべてのコアは API を使用することなくアプリケーションで利用できます。必要に応じて、開発者は Microsoft API を使用して、利用可能なコア数とタイプを検出し、それぞれの構成を最大限に活用してアプリケーションを最適化することが可能です (このドキュメントではその例を紹介しています)。

Q4: 開発コード名 Alder Lake プラットフォームは、Unreal Engine*、Unity*、またはその他のゲームエンジンとミドルウェアでサポートされますか? (英語)

A4: インテルは主要なミドルウェア企業と協力して、将来のリリースでこのプラットフォームをサポートすることを計画しています。

Q5: 図には L1 キャッシュが示されていません。存在するのでしょうか? (英語)

A5: はい。ハイパースレッドの機能を備える Performance-core は L1 キャッシュを共有し、Efficient-core はそれぞれ個別の L1 キャッシュを持ちます。

Q6: 共有 L2 を利用するため、タスクを E-core クラスターにグループ化する最良の方法はありますか? (英語)

A6: キャッシュマップから始めます。キャッシュとコアの関連付けは、GroupID/Mask (別名グループマスク) を使用して行われます。これは、GetLogicalProcessor/GetLogicalProcessorEx を使用した HybridDetect サンプルで実証されています。次のスクリーンショットを参照してください。

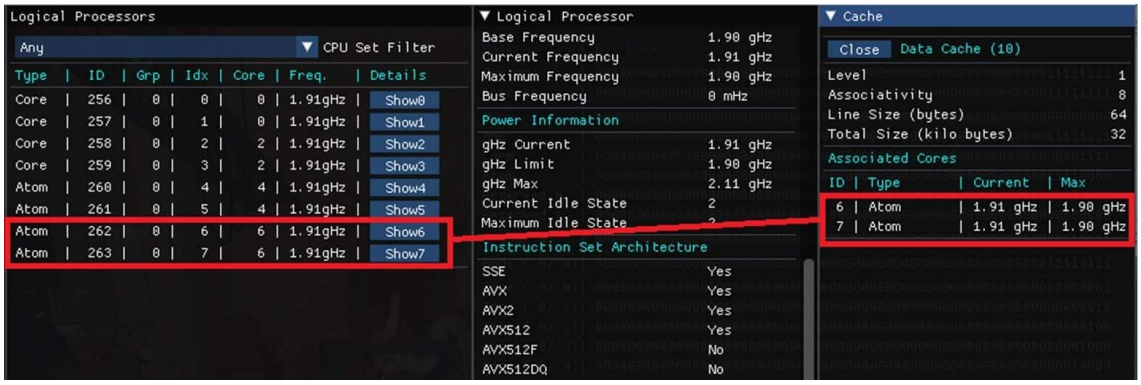


図 8. グループマスクと呼ばれる GroupID/Mask を使用して、キャッシュとコアを関連付けます。

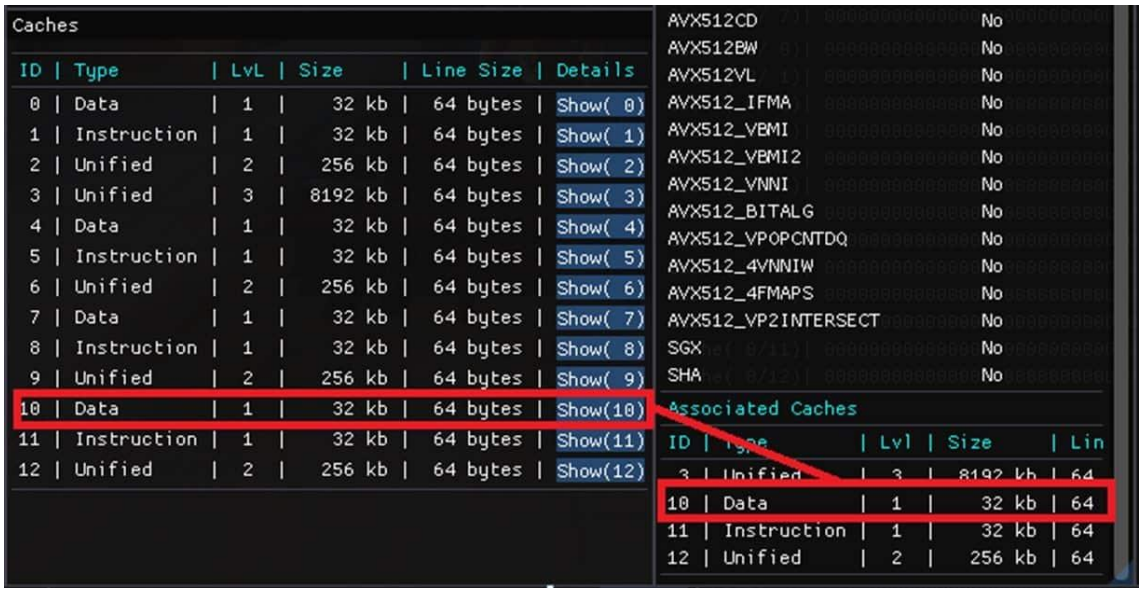


図 9. データキャッシュを論理プロセッサー上のキャッシュに関連付けます。

Q7: データの関係管理 (DRM) ミドルウェアとの潜在的な互換性の問題はありますか? (英語)

A7: DRM ミドルウェアを使用するゲームでは、ミドルウェアの開発元に問い合わせ、そのミドルウェアが通常のハイブリッド・アーキテクチャーをサポートしているか (特に開発コード名 Alder Lake プラットフォームをサポートしているか) 確認すると良いでしょう。最新の DRM アルゴリズムの性質上、CPU 検出を使用する可能性があり、今後のハイブリッド・プラットフォームでは注意が必要です。インテルは、Denuvo* のような主要な DRM プロバイダーと協力して、彼らのソリューションが新しいプラットフォームをサポートできるようにしています。

9. 関連情報

Intel Atom® プロセッサー・ファミリー (英語)
Intel Atom® プロセッサー仕様 (英語)
新しいハイパフォーマンス・グラフィックス向けのインテルのプレスリリース (英語)
インテル® VTune™ プロファイラーのパフォーマンス解析クックブック
インテル® Parallel Studio XE のドキュメント (英語)
インテル® アーキテクチャー命令セット拡張および将来の機能のプログラミング・リファレンス (英語)
インテル® 64 および IA-32 アーキテクチャー最適化リファレンス・マニュアル
インテル® oneAPI レンダリング・ツールキット (英語)
CES 2021 開発コード名 Alder Lake の発表を含むインテルのプレスリリース (英語)
Unreal Engine* 4.19 を使用した CPU 機能の検出に関する記事 (英語)
キャッシュマップの作成方法を説明した Microsoft のドキュメント (英語)
World of Tanks* 1.0+: CPU に最適化されたグラフィックスと物理演算でユーザー体験を向上させる方法 (英語)

製品とパフォーマンス情報

[1]パフォーマンスは、利用、構成、およびその他の要因によって異なります。詳細については、www.Intel.com/PerformanceIndex (英語) をご覧ください。