インテル® プロセッサー・グラフィックス Gen11 API 向け開発者および最適化ガイド

この記事はインテル® デベロッパー・ゾーンに公開されている「Developer and Optimization Guide for Intel® Processor Graphics Gen11 API」の日本語参考訳です。

目次

概要	1
Gen11 アーキテクチャーの特徴	1
パフォーマンス解析ツール	3
インテル® プロセッサー・グラフィックス Gen11 向けのパフォーマンスに関する推奨事項	9
低電力向けの設計	19
インテル® デベロッパー・ゾーンとゲーム開発者ウェブサイトのリソース	24

概要

このドキュメントは、インテル® プロセッサー・グラフィックス Gen 11 のグラフィックス・ハードウェア・アーキテクチャー向けの開発者ガイドと最適化方法を示します。アーキテクチャーの能力とピーク・パフォーマンスを最も効率良く利用する、開発者向けのベスト・プラクティスを提供します。また、インテル® プロセッサー・グラフィックス Gen 11 で最新のグラフィックス API を使用する具体的な API ガイドも紹介します。

このガイドは、Gen 11 向けにインタラクティブ 3D レンダリング・アプリケーションを最適化しようとする開発者を対象としています。開発者は、Microsoft* DirectX* 12、Vulkan* (英語)、および Metal* 2 (英語) 向けのグラフィックス API パイプラインに関する基本的な知識を有することを前提としています。Gen11 は、DirectX* 11 (英語) と OpenGL* (英語) グラフィックス API もサポートしますが、DirectX* 12、Vulkan*、および Metal* 2 などの新しい下位レベル API を利用するアプリケーションは、パフォーマンス上の利点と低い CPU オーバーヘッドの恩恵が得られ、これらの API でしか利用できない新しいグラフィックス・アーキテクチャーの機能もあります。

Gen11 アーキテクチャーの特徴

Gen11 は、Gen9 よりも高いパフォーマンスと高効率を実現し、粗い粒度のピクセル・シェーディング、タイルベースのレンダリング、および新しいディスプレイ・コントローラーの機能を利用できます。さらに、Gen11 では次の改善が行われています。

- 最大テラフロップの計算処理能力
- 非常に低い共有ローカルメモリー (SLM) レイテンシー
- 大きな L3 キャッシュ
- 増加したメモリー帯域幅
- マルチサンプル・アンチエイリアシング (MSAA) のパフォーマンス向上

Gen11 のアーキテクチャーと新機能の詳細は、『インテル® プロセッサー・グラフィックス Gen11 アーキテクチャー』(英語) ガイドを参照してください。

タイルベースのレンダリング

Gen11 は、position only shading tile-based rendering (PTBR) として知られる、タイルベースのレンダリング・ソリューションを実装しています。タイルベースのレンダリングの利点は、タイルごとにデータへの複数のレンダリング・パスを効率良く管理することで、メモリー帯域幅を軽減できることです。タイルベースのレンダリングをサポートするため、Gen11 ではタイル・ビニング・エンジンとして動作する並列ジオメトリー・パイプラインが追加されています。これはレンダーパイプラインの前に、タイルごとの可視性ビニングの事前パスで使用されます。タイルごとにジオメトリーをループし、そのタイルの可視性ストリームを使用します。PTBR はタイルごとのデータを L3 キャッシュに保持し、外部メモリー帯域幅を軽減します。詳細については、アーキテクチャー・ガイドを参照するか、インテルのアプリケーション・エンジニアに問い合わせて、ワークロードに利点があるかを確認してください。

粗いピクセル・シェーディング

粗い粒度のピクセル・シェーディングは、DirectX* 12 では可変レート・シェーディングとして知られ、プログラマーは、レンダリング・ターゲットの解像度やラスタライズ・レートにかかわりなくシェーディング・レートを変更できます。このほかにも、この機能により、シェーディング・パラメーターがゆっくりと変化するコンテンツや、後続のレンダリング・パイプラインでぼかしが適応される可能性があるピクセルにおいて、ピクセルシェーダーの呼び出し回数を減らすことができます。この機能により、開発者はコンテンツの最も重要なピクセルにシェーダー操作を割り当てることができます。そして、深度とステンシルはフル・ピクセル・レートで維持されるため、低解像度でレンダリングしてから拡大するよりも優れたビジュアル・ソリューションが得られます。Gen11 ハードウェアは、DirectX* 12 可変レート・シェーディング (VRS) Tier 1 をサポートします。

高ダイナミック・レンジ・ディスプレイ

Gen11 では、ハイダイナミック・レンジ (HDR) ディスプレイのサポートが強化されました。この機能を利用するには、Microsoft* のドキュメント「ハイダイナミック・レンジと広色域 (Wide Color Gamut) の概要」(英語) を参照してください。

Adaptive Sync — 可変リフレッシュ・レート

Adaptive Sync (英語) は、可変リフレッシュ・レート・ディスプレイ向けの VESA 標準規格です。このディスプレイ・コントローラーとディスプレイ機能は、ティアリングとスタッタリングを回避することでより良いユーザー体験を可能にします。Adaptive Sync はまた、システム全体の消費電力を軽減します。Adaptive Sync の基本要件を以下に示します。

- ゲームや 3D アプリケーションによるフル・スクリーン・レンダリング
- 非同期バッファーフリップを確実にするアプリケーションのスワップチェーンの簡単な変更
- DisplayPort* 1.4 VESA Adaptive Sync をサポートする Gen11 以降のグラフィックス・デバイス
- DisplayPort* 1.4 VESA Adaptive Sync 対応のディスプレイ・パネル
- Windows® 10 RS5 以降

ゲームや 3D アプリケーションでは、レンダリング・スワップ・チェーンが非同期バッファーフリップを実装することを確認する必要があります。Adaptive Sync をサポートするディスプレイは、ディスプレイのリフレッシュが非同期スワップ・チェーン・フリップと動的に同期され、スムーズなインタラクティブ・レンダリングを可能にします。アプリケーションとプラットフォームが条件を満たす場合、Gen 11 ドライバーはデフォルトで Adaptive Sync を有効にします。これは、インテル® グラフィックス・ドライバーのコントロール・パネルを使用して、無効にすることもできます。

パフォーマンス解析ツール

インテルは、CPU とグラフィックス・プロセシング・ユニット (GPU) の両方でアプリケーションのパフォーマンスを向上するのに役立つツールを提供しています。インテル® VTune™ Amplifier とインテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA) は無料でダウンロードできます。RenderDoc* (英語)、Microsoft PIX (英語)、および Windows* Performance Analyzer などのツールは、インテル® プラットフォームで動作し、アプリケーションのパフォーマンスに関連する有用な情報を提供します。これらのツールの詳細は、リンク先のドキュメントを参照してください。

インテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA)

インテル® GPA には、ゲーム開発者が、インテル® Core™ プロセッサーやインテル® プロセッサー・グラフィックスなどのゲーム・プラットフォームのパフォーマンスを最大化するのに役立つ強力なアジャイルツールが含まれています。インテル® GPA ツールは、アプリケーションのパフォーマンス・データを視覚化し、システムレベルおよび個々のフレーム・パフォーマンスの問題を特定することを可能にします。これらのツールを使用すると、仮定を基にした検証を行い、最適化による潜在的なパフォーマンス向上を推測できます。

システム･アナライザー/ヘッドアップ･ディスプレイのオーバーレイ

インテル® GPA システム･アナライザーは、CPU、グラフィックス API、そして GPU のパフォーマンス･メトリックをリアルタイムに表示するツールです。システム･アナライザーは、重要なパフォーマンスの可能性とワークロードが CPU または GPU ボトルネックであるかを迅速に特定するのに役立ち、アプリケーションのパフォーマンスに最も影響する要素の最適化に集中できます。このツールは、コードを 1 行も変更することなく、ステート･オーバーライドを使用して、高速かつ高レベルでゲームの反復解析を実行できます。システム･アナライザーは DirectX* と OpenGL* アプリケーションをサポートします。

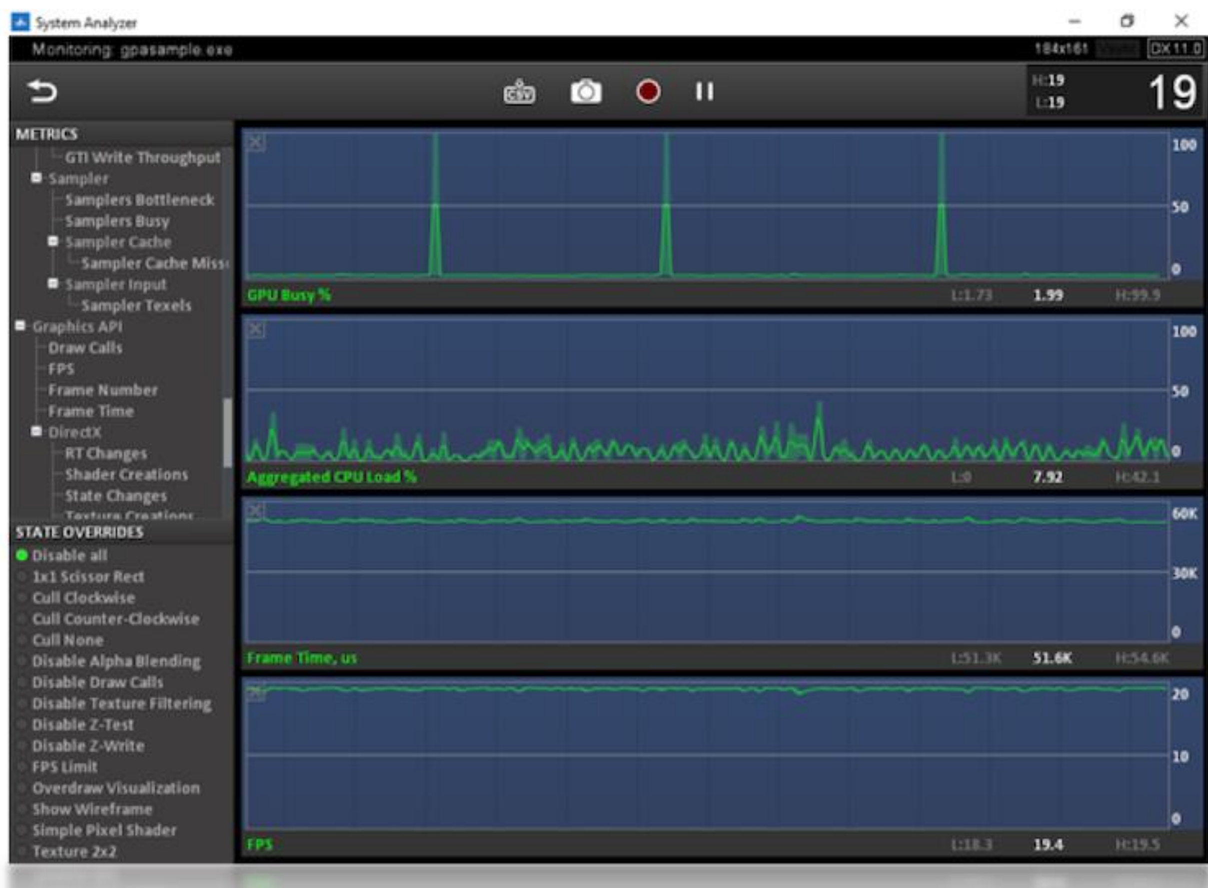

図 1. システム･アナライザーは、グラフィックス･アプリケーションの
リアルタイム･ステータスをシステムレベルで表示

システム･アナライザーは次の機能を提供します。

- 選択したアプリケーション全体のシステムメトリック、または特定のメトリックを表示します。
- CPU、GPU、グラフィックス･ドライバー、および API から各種メトリックを選択できます。
- 一般的なパフォーマンスのボトルネックを切り分けるため、オーバーライド･モードを使用してさまざまな仮定に基づく検証が可能です。
- フレームのキャプチャー、トレース、メトリック値のエクスポート、データ収集の停止/続行を行うことができます。
- 現在、最小、および最大のフレームレートを表示します。

図 2. ヘッドアップ･ディスプレイのオーバーレイはリアルタイムにメトリックを表示

詳細は、「システム･アナライザー導入ガイド」(英語) を参照してください。

グラフィックス･フレーム･アナライザー

インテル® GPA グラフィックス･フレーム･アナライザーは、主要なグラフィックス API ワークロード向けの強力で直観的なシングルフレームとマルチフレーム (DirectX* 11、DirectX* 12、および Valkan*) の解析および最適化を支援するツールです。シェーダー、レンダリング状態、ピクセル履歴、およびテクスチャーなど、描画呼び出しレベルまでの詳細なフレーム･パフォーマンス解析を行えます。仮定に基づく検証を通して、ソースコードを再コンパイルすることなく、変更がパフォーマンスとビジュアルに与える影響を確認できます。

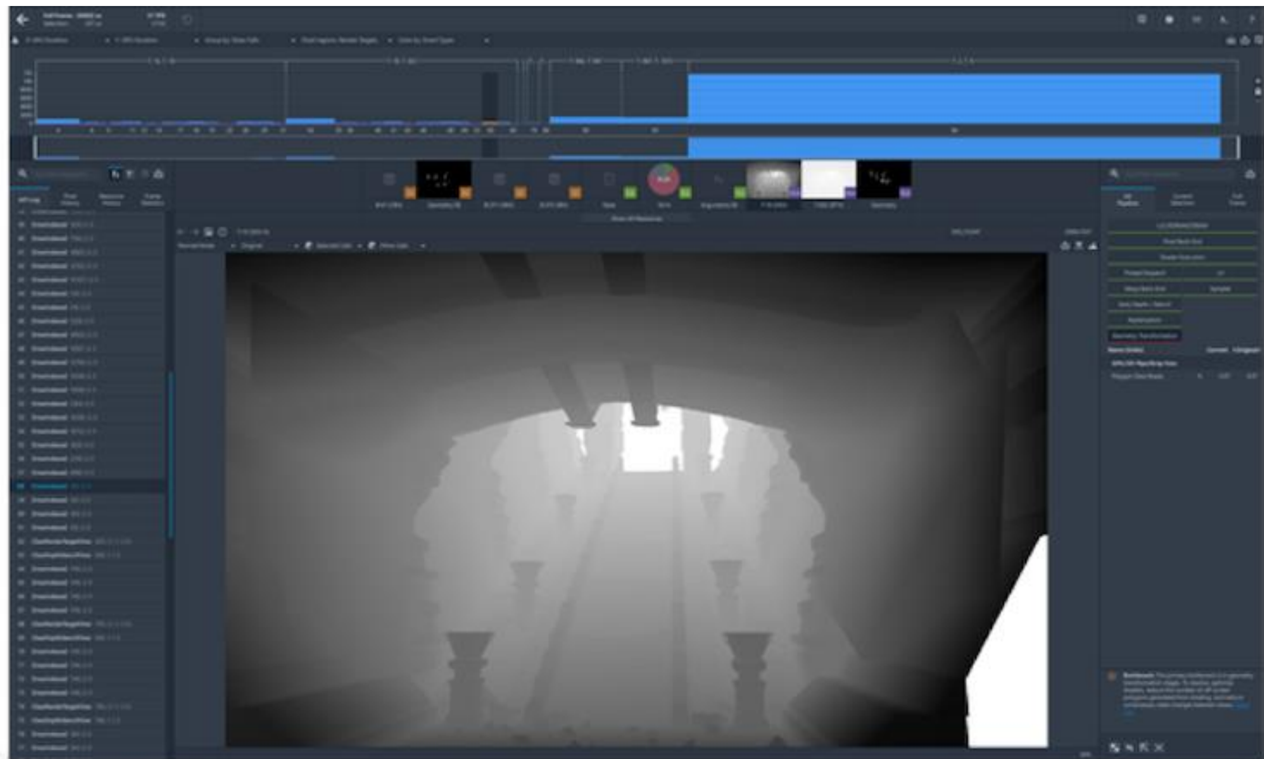

図 3. インテル® GPA グラフィックス・フレーム・アナライザー

インテル® グラフィックス・フレーム・アナライザーは次の機能を提供します。

- さまざまなグラフィックス・メトリックに基づく軸を使用したグラフ描画呼び出しが可能です。
- ピクセルの履歴を表示します。
- 階層ツリーで領域と描画呼び出しを選択できます。
- グラフィックス・パイプラインのリアルタイムの結果を実装および表示し、ボトルネックを特定して、不要なイベント、エフェクト、またはレンダリング・パスを分離します。
- ゲーム全体を変更することなく、シェーダーをインポートおよび変更して、単純または複雑なシェーダーのビジュアルおよびパフォーマンスへの影響を確認します。
- ジオメトリー、ワイヤーフレーム、および任意のフレームのオーバードロー表示を調べます。
- ハードウェア・メトリックを使用して、GPU パイプラインのボトルネックを特定します。
- ホットスポット・モードを使用して、ハードウェア・ボトルネックの描画呼び出しをグループ化および色分けします。
- Python* プラグイン・インターフェイスを使用して、ワークフローのプロファイルを多面的に自動化及び合理化します。事前にロードされたプラグインを調査するか独自のプラグインを作成します。

グラフィックス・トレース・アナライザー

インテル® GPA グラフィックス・トレース・アナライザーを使用して、アプリケーションが CPU と GPU で時間を費やしている場所を特定できます。これにより、ソフトウェアが現在のインテル® プラットフォームで利用できる処理能力を最大限に引き出せるようになります。

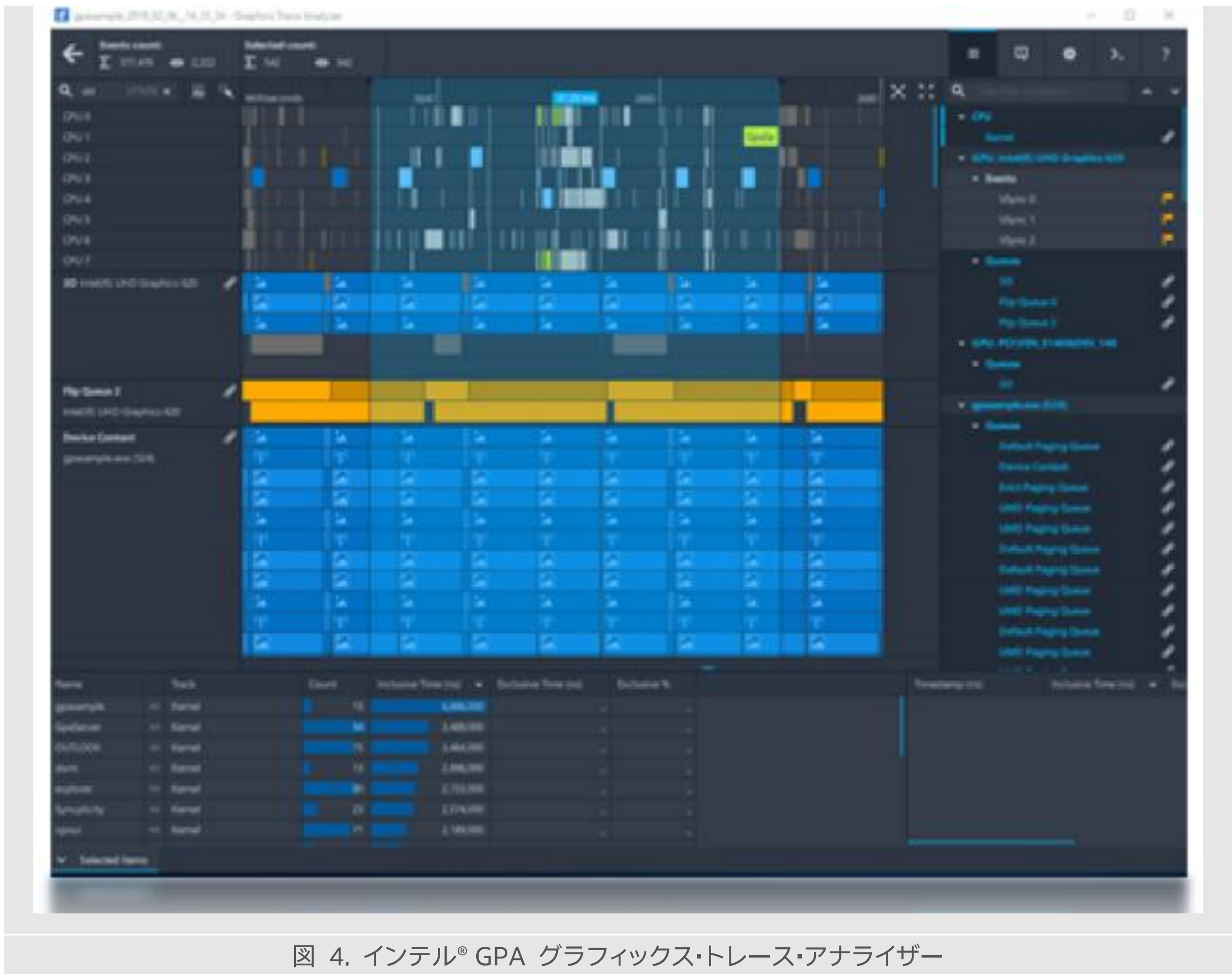

図 4. インテル® GPA グラフィックス・トレース・アナライザー

グラフィックス・トレース・アナライザーは、CPU と GPU のメトリックとワークロードのオフライン解析を提供します。タイムライン・ビューを使用して、コンテキスト中のタスク、スレッド、主要なグラフィックス API、および GPU でアクセラレートされたメディア・アプリケーションを解析できます。

グラフィックス・トレース・アナライザーは次の機能を提供します。

- 詳細なタイムラインでタスクデータを確認できます。
- CPU 依存または GPU 依存のプロセスを特定します。
- キューに投入された GPU タスクを調査します。
- CPU スレッドの利用率を調査して、使用される API に関連付けます。
- キャプチャーされたプラットフォームとハードウェア・メトリック・データに基づいて、CPU と GPU のアクティビティーを関連付けます。
- 特定の期間に注目するためタイムラインをフィルター処理し分離できます。
- システム全体およびアプリケーション・レベルでデータを取得します。

その他のインテル® GRA リソース

インテル® GPA に関する追加情報および最新情報については、製品ページをご覧ください。

インテル® VTune™ Amplifier

インテル® VTune™ Amplifier は、CPU や GPU のボトルネックの検出に役立つだけでなく、パフォーマンスのチューニング時に CPU で実行されるワークを最適化するのにも有用です。DirectX* アプリケーションのパフォーマンスをチューニングするため、インテル® VTune™ Amplifier は低速なフレームと DirectX* イベントを検出できます。インテル® VTune™ Amplifier はまた、フレームおよびイベント API を使用したトレースイベントのカスタマイズにも対応しています。インテル® VTune™ Amplifier のセットアップと使用方法の詳細については、『インテル® VTune™ Amplifier 2019 ユーザーガイド』をご覧ください。

このガイドは Gen11 向けのパフォーマンス最適化に限定されていますが、ゲーム・アプリケーションなどのグラフィカルなワークロードのパフォーマンス・チューニングに役立つ主要機能の説明も含まれています。CPU に関連するボトルネックや以下の機能の使用方法とトレーニングについては、『インテル® VTune™ Amplifier ユーザーガイド』を参照してください。

Windows* の DirectX* フレームの自動検出

インテル® VTune™ Amplifier は、低速なフレームと DirectX* イベントを検出する機能を備えています。この機能を使用して、低速なフレームを識別し、そのフレーム内のイベントをフィルター処理できます。

フレームとイベント API

インテル® VTune™ Amplifier のフレームとイベント API を使用すると、インテル® VTune™ Amplifier がプロファイルできるトレースイベントをカスタマイズできます。フレーム API では、これをフレームごとに行うことができます。例えば、ゲーム・アプリケーションでは、ゲームを開始イベントと終了イベントで囲み、フレームごとにプロファイルを生成するのが最適です。インテル® VTune™ Amplifier のフレーム API の詳細と使用例については、「フレーム API のドキュメント」を参照してください。

イベント API を使用すると、プロファイルするソフトウェア内のイベントを自由に区分することができます。例えば、ゲーム内のタスクを最適化するため、イベント API を使用してゲーム・アプリケーションでフレームを計算するのに必要なそれぞれのタスクを追跡できます。インテル® VTune™ Amplifier のイベント API の詳細と使用例については、「イベント API のドキュメント」を参照してください。

インテル® プロセッサー・グラフィックス Gen11 向けのパフォーマンスに関する推奨事項

DirectX* 12、Metal*、Vulkan* などの最新のグラフィックス API を使用すると、以前はドライバー実装で処理されていた低レベルの選択をさらに細かく制御できます。それぞれの API は異なりますが、API に依存しないアプリケーション開発者向けの一般的な推奨事項があります。

グラフィックス・パイプライン状態の構成

パイプライン状態を構成する場合、次のことを考慮してください。

- パイプライン状態オブジェクト (PSO) を作成する場合、システムで利用可能なすべての CPU スレッドを利用してください。以前の API では、ドライバーがこれらのスレッドを生成していましたが、現在は開発者がスレッドを生成する必要があります。
- 同じスレッドで同様の PSO をコンパイルして、ドライバーとランタイムによる重複を排除します。
- 汎用シェーダーと専用シェーダーを組み合わせる代わりに、PSO 向けに最適化されたシェーダーを定義します。
- ステンシルを使用しない場合は、深さとステンシルの形式を定義しないで、D32 などの深さ形式のみを使用します。

リソースのバインド

最新のグラフィックス API を使用すると、DirectX* のルート・シグネチャーや Vulkan* のパイプライン・レイアウトなどのリソースバインドをより細かく制御できます。これらを使用してパフォーマンスを最大化するには特に注意が必要です。リソースバインドに対するアプリケーションの方針を設計するには、次のガイドに従ってください。

- ルート・シグネチャー・スロットまたはディスクリプターの数は、シェーダーが使用するもののみに抑えます。
- シェーダー全体で、ルート・シグネチャーまたはディスクリプターの再利用のバランスを考えてください。
- 描画間で変化しない複数の定数バッファーは、すべての定数バッファービューを 1 つのディスクリプター・テーブルにパックすることを検討してください。
- 連続したレジスターにまたがらず、描画間で変化しない複数の順序指定されていないアクセスビュー (UAV) またはシェーダー・リソース・ビュー (SRV) は、ディスクリプター・テーブルにパックするのが最適です。
- ディスクリプター・ヒープの変更を最小限にします。ディスクリプター・ヒープを変更すると、グラフィックス・パイプラインが大幅にストールします。理想としては、すべてのリソースに 1 つのディスクリプター・ヒープからビューを割り当てます。

- 使用されない不必要なディスクリプターを定義する汎用ルート・シグナチャーは避けてください。代わりに、必要な最小限のディスクリプター・テーブルに合わせてルート・シグネチャー定義を最適化します。
- ルート・ディスクリプターよりもルート定数を優先し、定数を使用する場合はディスクリプター・テーブルよりもルート・ディスクリプターを優先します。
    - ルート/プッシュ定数を使用して、定数バッファーデータへの高速アクセスを可能にします (これらはレジスターに事前にロードされています)。
    - ルート/プッシュ定数は、頻繁に変化する定数バッファーのデータに使用するのに適しています。
- 定数が高い頻度で変化する場合、ルート/プッシュ定数を使用します。
- D3D12_DESCRIPTOR_RANCE_FLAG_DATA_STATIC など、ドライバーが定数ベースの最適化を行えるようにヒントを使用してください。
- リソースが配置済みである場合、そのリソースにレンダリングする前に、クリア、コピー、または廃棄して初期化します。これにより、配置済みのリソースをアクティブな状態にして、適切に圧縮できるようになります。

レンダーターゲットとテクスチャー

汎用ガイド

次のガイドラインに従って、レンダーターゲットの帯域幅を効率良く利用できます。

- 必要最小限のレンダーターゲットを使用し、可能であればレンダーターゲットを組み合わせて、低レベルのキャッシュをより効率良く活用します。
- メモリー帯域幅の節約やキャッシュフェッチの最適化が必要ない場合、不要なチャネルや精度の高いデータ形式を定義しないでください。
- 同じメモリー・オブジェクトから複数のリソースを作成します。
- Vulkan* 固有の最適なデバイスアクセス:
    - GPU アクセスには常に VK_IMAGE_LAYOUT_{}_OPTIMAL を使用します。
    - VK_IMAGE_LAYOUT_GENERAL は、本当に必要な場合にのみ使用します。
    - VK_IMAGE_CREATE_MULTIPLE_FORMAT_BIT は、本当に必要な場合にのみ使用します。

UAV と SSBO

UAV やシェーダー・ストレージ・バッファー・オブジェクト (SSBO) など、シェーダーがリードとライトの両方のアクセスを行うリソースを扱う場合、次のことを考慮します。

- これらのリソースタイプは、Gen11 の 64 バイト・キャッシュラインで非効率な部分書き込みを引き起こすことがあります。キャッシュ階層全体で最大限の帯域幅を得るには、部分書き込みは避けなければなりません。これは、4x2 のピクセルグループでシェーダーを実行する単一のスレッドが、出力に連続した 64 バイトを書き込むことで達成できます。

- リード専用データのアクセスは、リード/ライトデータよりもはるかに効率的です。これらのリソースは、適切なオプションがない場合に注意して使用する必要があります。
- リソースが UAV としてバインドされない場合、リソース設定で UAV バインドフラグを使用してはなりません。このプログラミング手法によってリソース圧縮が無効になることがあります。

アンチエイリアシング

マルチサンプル・アンチエイリアシングで最高のパフォーマンスを得るには、以下の推奨に従ってください。

- MSAA が有効な場合、ステンシルやブレンドの使用を最小限にします。
- 結果がすぐに利用されるか、ループ反復間で複製されるループや分岐内からリソース情報を照会することは避けてください。
- サンプルごとの操作を最小限にします。サンプルごとにシェーディングを行う場合、ピクセルのキル操作によりサーフェスが圧縮される数を最大化します。

「Conservative Morphological Anti-Aliasing 2.0」(英語) など、最適化された計算シェーダーの後処理アンチエイリアシングを使用することを推奨します。

リソースバリア

一般に、各リソースバリアはキャッシュのフラッシュや GPU のストール操作を引き起こし、パフォーマンスに影響します。そのため、次のガイドラインが推奨されます。

- パイプライン・バリアをまとめ、レンダーパスを使用して、バリアを適切にバッチ処理し、ドライバーがバリアを延期およびホイストして、パスの端をレンダリングできるようにします。
- バリアのみを含むコマンドバッファーは避けます。必要であれば、バリアはコマンドキューの最後に配置します。
- 可能であれば、暗黙のレンダー・パス・バリアを使用します。
- バッチ処理によりリソースの遷移を制限し、ディスパッチ/レンダーパスとのインターリーブを避けます。
- 通常、表示、コンテキストの共有、または CPU アクセスに必要な場合を除き、レンダーターゲットの変更以外のバリアと D3D12_RESOURCE_STATE_COMMON などの状態を避けます。
- 可能であれば、バリアにリソースを提供します。特に、エイリアシング・バッファーでは、適切な GPU キャッシュフラッシュが可能になります。
- また、分割バリアを使用して、同期イベントを最大限に予測できるようにします。
- リソース状態を移行する場合、使用されていなければ上書きすることは避けてください。上書きすると過度のキャッシュフラッシュにつながります。

コマンド送信

コマンドキューとコマンドバッファーを使用する場合、次のことを推奨します。

- 可能であれば、GPU が枯渇しない範囲で DirectX* 12 の ExecuteCommandLists でコマンドリストをバッチ送信します。これにより、CPU と GPU を効率良く利用できます。
- コマンドバッファーやコマンドキューを埋める際に、可能であれば複数の CPU コアを使用します。これにより、アプリケーションのシングルコア CPU のボトルネックが減少します。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) の利用も有効です。
- Gen11 アーキテクチャーは、3D と計算 (DirectX* では非同期計算と呼ばれます) の同時実行をサポートしていません。そのため、これらを個別の非同期キューに送信しても Gen11 のパフォーマンスは向上しません。適切な場合、計算ワークはまとめてバッチ処理し、3D ワークとの混在を避けます。これによりワークの送信が最適化され、計算ワークと 3D ワークの切り替えによるペナルティーがなくなります。
- 再利用されるコマンドバッファーは注意して使用します。
- DirectX* 12 固有:
  - CPU と GPU のオーバーヘッドが生じる可能性があるため、バンドルの過剰使用は避けてください。
- Vulkan* 固有:
  - 内部バッチバッファーの使用法によりパフォーマンスが向上するため、可能な限りプライマリー・コマンド・バッファーを使用します。
  - プライマリー・コマンド・バッファーでは、USAGE_ONE_TIME_SUBMIT_BIT を使用します。
  - プライマリー・コマンド・バッファーでは、USAGE_SIMULTANEOUS_USE_BIT は使用しません。
  - セカンダリー・コマンド・バッファーは、プライマリー・コマンド・バッファーよりも効率が劣り、深度クリアの効率が良くないため、使用を最小限に留めます。

クリア、コピー、および更新操作の最適化

クリア、コピー、および更新操作で最大のパフォーマンスを達成するには、次のガイドラインに従ってください。

- API が提供する機能を使用して、クリア、コピー、および更新操作を行い、独自実装の利用を控えます。ドライバーは、これらの操作が最大のパフォーマンスを発揮できるように最適化および調整されています。
- API ごとに定義されているハードウェア 'fast clear' 値を有効にします。
  - DirectX* 12 では、リソース作成時に ID3D12Device::CreateCommittedResource の引数として値が定義されています。
  - Vulkan* では、VK_ATTACHMENT_LOAD_OP_CLEAR を使用して vkCmdClearColorImage の使用は避けます。
  - そのほかの API では、(0,0,0,0) または (1,1,1,1) を使用します。
  - 水平方向は 128b、垂直方向は 64b でアライメントします。

- 深度とステンシルサーフェスは、両方を無条件にコピーするのではなく、必要な場合にのみコピーします。これらは Gen11 では個別に保存されます。
- 転送およびコピー操作をバッチ処理します。

ジオメトリー変換

次のガイドラインに従って、バーテックスおよびジオメトリー・シェーダー関数が適切に動作することを確認します。

- バーテックス・フェッチのスループットは、クロックごとに 6 属性です (およびクロックあたり最大 2 つのバーテックス)。バインドされた属性がすべて使用されていることを確認してください。ジオメトリー・ワークが描画のボトルネックである場合、バーテックスごとの属性数を減らすとパフォーマンスが向上します。
- 詳細レベルごとにモデルのバーテックス数を調整できる、モデル精度の柔軟性を備えた詳細レベルのシステムを実装します。
- プリミティブとピクセルの比率を解析します。この比率が高い場合、追加のバーテックス・シェーダー・スレッドは、最終レンダーターゲットにわずかな値しか追加しません。この比率を 1:4 以下に保ちます。
- 隠れたジオメトリーを送信しないように、効率良い CPU オクルージョン・カリングを実装します。この手法により、CPU (描画を送信) と GPU (レンダリング) の両方で時間を節約できます。これには、高度に最適化されたマスク・ソフトウェア・オクルージョン・カリングを使用することを推奨します。最終的には、組み合わせにより、より細やかな GPU カリングを使用できます。
- 入力ジオメトリーをバーテックス・バッファーの配列構造体として定義します。タイルベースのレンダリング向けのタイル・ビニング・エンジンをアシストするため、位置情報のバーテックスデータを個別の入力スロットでグループ化します。
- Gen 11 のバーテックス・キャッシュは、インスタンス化された属性をキャッシュしません。インスタンス化された呼び出しでは、バーテックス・シェーダーで属性を明示的にロードすることを検討してください。
- (バーテックスからジオメトリーへの) 変換シェーダーを最適化して、後続のパイプライン・ステージで使用される属性だけを出力します。例えば、ピクセルシェーダーで使用されない、バーテックス・シェーダーからの不要な出力を定義しないようにします。これにより、L3 キャッシュの帯域幅と空間を適切に使用できます。
- Metal* 2 固有 — テッセレーション係数は計算エンジンで計算され、レンダリングのため 3D エンジンに返されます。複数の描画に対し連続してテッセレーション係数を計算することで、3D ワークロードと計算ワークロード間のコンテキスト・スイッチを減らします。

タイルベースのレンダリング

Gen11 PTBR ハードウェアを最も効率良く使用するには、帯域幅が制限されるパスに対し次のガイドラインに従います。

- トライリストまたはトライストリップ・トポロジーのみを使用します。
- DirectX* 12 では、D3D12_RENDER_PASS_ENDING_ACCESS_TYPE_DISCARD で ID3D12GraphicsCommandList4::EndRenderPass を使用します。
- Vulkan* では、VK_ATTACHMENT_STORE_OP_DONT_CARE で VkRenderPass/VkSubpass を使用します。
- テッセレーション、ジオメトリーおよび計算シェーダーを避けます。テッセレーションとジオメトリー・シェーダーを使用するパスは、PTBR ハードウェアの利点を得られません。
- ライトハザード後のレンダー内パスのリードを避けます。
- 位置の計算に必要な属性を個別のバーテックス・バッファーに分離します。

シェーダー最適化

汎用シェーダーガイド

シェーダーを作成する場合、次の最適化の可能性を考えてください。

- シェーダーを構造化して、サンプリングやメモリーフェッチなどの高いレイテンシー操作の依存関係を排除します。
- サンプリング操作の結果をベースとするシェーダー制御フローを避けます。
- 不均一な変数を使用するフロー制御を回避することで、シェーダーの均一な実行を目指します。
- アルゴリズムの出力を事前に決定したり、完全なアルゴリズムよりも低コストで計算できる場合、シェーダーで早期リターンを実装します。
- シェーダーのセマンティクスを使用してフラット化、分岐、ループ、およびアンロールを行います。多くの場合、アンロールは、シェーダー・コンパイラーが判断するよりも、開発者が明示的に指示する方が適切です。
- 分岐は、分岐コストを上回る十分な命令サイクルが確保できる場合に使用します。
- 高度な数学およびサンプリング操作は多くのサイクル数を必要とするため、分岐の利点があるかもしれません (発行レートについては図 6 を参照)。

命令	単精度 (ops/EU/クロック)	理論上のサイクルカウント
FMAD	8	1
FMUL	8	1
FADD	8	1
MIN、MAX	8	1
CMP	8	1
INV	2	4
SQRT	2	4
RSQRT	2	4
LOG	2	4
EXP	2	4
POW	1	8
IDIV	1 – 6	1.33 – 8
TRIG	2	4
FDIV	1	8

図 1. Gen11 EU 命令発行レート

- コード内の短い分岐は、フラット化することでパフォーマンスが向上する場合があります。
- アンロールは慎重に行います。多くの場合、短いループをアンロールするとパフォーマンスが向上しますが、ループをアンロールするとシェーダー命令数が増加します。反復回数が多い長いループをアンロールすると、命令キャッシュ内のシェーダーの常駐に影響する可能性があり、パフォーマンスに影響します。
- サンプラー操作が後にゼロ乗算される可能性がある場合、余分なサンプラー操作を避けます。例えば、2 つのサンプルを補間する場合、補間が 0 または 1 である可能性が高いときは、分岐を追加して通常のケースを高速化して、必要な場合にのみロードを行います。

- ランタイム時にリソースを照会しないでください。例えば、ハイレベル・シェーディング言語 (HLSL) の GetDimensions 呼び出しで制御フローを決定したり、不必要にリソース情報をアルゴリズムに組込まないでください。
- 属性をピクセルシェーダーに渡す場合、プリミティブ内のバーテックス間で変化しない属性は、定数としてマークします。
- 深度テストが無効化されているシェーダーでは、出力がレンダーターゲットの最終色に関連しない場合破棄します (またはほかの強制終了操作を行います)。アルゴリズム出力のアルファチャネル値がゼロの場合、または出力を無効化するゼロシェーダーに入力を追加する場合は、ブレンドをスキップできます。

テクスチャーのサンプリング

テクスチャーとテクスチャー操作から最大のパフォーマンスを得るには、次のことを考慮してください。

- レンダーターゲットからサンプリングを行う場合、sample_l/sample_b などの命令を使用して、サーフェスのミップレベルでのサンプリングは避けるようにしてください。
- API 定義やアーキテクチャーでサポートされている圧縮形式 (BC1 - BC7) を大きなテクスチャーに使用して、サンプリング操作を行う際のメモリー帯域幅利用率とメモリーの局所性を改善します。
- サンプル命令間で依存関係があるテクスチャー・サンプルを避けます。例えば、後続のサンプル操作の UV 座標が、直前のサンプル操作の結果に依存しないようにします。依存関係があると、シェーダー・コンパイラーは命令の最適化または並べ替えを行うことができず、サンプラーのボトルネックとなる可能性があります。
- シェーダーコード内で冗長で重複するサンプラー状態を避け、可能であれば静的/不変なサンプラーを使用します。
- サンプル操作とフィルターモードに適切なリソースタイプを定義します。ボリュームサーフェスを 2D 配列として使用しないでください。
- 配列サーフェスからフェッチする場合、すべての SIMD (単一命令、複数データ) レーンでインデックスが均一であることを確認してください。
- グラデーションなど、シェーダーで計算できるテクスチャーで定数データを定義しないでください。
- sRBG テクスチャーで異方性フィルター処理は避けてください。
- sample_d はピクセルごとの勾配を提供し、スループットは 4 分の 1 に低下します。異方性フィルターが必要な場合を除き、sample_l を優先します。
- VRS を使用すると描画ピクセルが粗くなるため、異方性フィルター処理は必要ありません。非異方性フィルター処理では、サンプラーのスループットが向上します。

定数

シェーダー定数を定義する場合、次のガイドラインがパフォーマンス向上に役立ちます。

- パフォーマンスを改善するため、メモリーアクセスがすべて同じキャッシュラインで行われるようにするには、定数バッファーを構造化してキャッシュの局所性を高めます。

- 実行時にオフセットを計算する間接アクセスではなく、コンパイル時にオフセットが判明している直接アクセスを使用する定数アクセスを優先します。これは、フロー制御やサンプリングなど高いレイテンシーの操作に有効です。
- キャッシュ利用率を高めるため使用頻度の高い定数をグループ化し、バッファーの先頭に移動します。
- 更新の頻度によって定数をまとめ、値が更新された時にのみアップロードします。
- バッファーや構造化されたバッファーからデータをロードする場合、キャッシュライン全体または大部分が使用されるようにデータアクセスを整理します。例えば、構造化バッファーに 10 個の属性があり、そのうち 1 つのみがリード/ライトに使用される場合、その属性のみを別の構造化バッファーに分割することを推奨します。
- 連続したデータをロードする場合は、型付きのバッファーからデータをロードする代わりに、ByteAddressBuffers の使用を検討してください。これらは、シェーダー・コンパイラーによって最適化できます。

一時レジスター変数の使用

実行ユニット上のそれぞれのスレッドには、値を格納するため固有のレジスターセットがあります。レジスター-レジスター操作で実行できるワークが増えると、メモリーアクセスのペナルティーを減らすことができます。しかし、レジスターよりも一時変数の方が多いと、一部の変数はメモリーに保存する必要があり、リードとライトにはレイテンシー・コストが生じます。このレジスターのスピルを回避するとパフォーマンスの向上につながります。

シェーダーを作成する際に、レジスタースピルを減らしてパフォーマンスを高めるには、次のガイドラインに従います。

- 一時変数はシェーダーごとに 16 個以下にします。これにより、高いレイテンシー・コストのメモリーとレジスター間の転送回数を抑えることができます。命令セットのアセンブリー・コードを出力して、レジスタースピルの回数を調査します。スピルは、高いレイテンシーのメモリー操作に関連する命令を減らす最適化の機会をもたらします。これは、インテル® GPA でシェーダーを選択して、コンパイラーが生成したマシンコードを調査することで確認できます。
- 可能であれば、変数の宣言と割り当てを参照される場所の近辺に移動します。
- 同じ空間に多くの値を保存できるよう、変数の完全精度と部分精度を比較検討します。同じ命令で部分精度と完全精度を混在させると、冗長な型変換につながる可能性があることに注意してください。
- 分岐間で共通する冗長コードを分岐から移動します。これにより、冗長な変数の重複を排除できます。
- 定数バッファー/バッファーデータへの不均一なアクセスを避けてください。不均一なアクセスでは、SIMD レーンごとにデータを格納するため、多くの一時レジスターが必要になります。
- 定数バッファーデータから制御フローを判断することは避けてください。これは、コンパイラーが最適でないマシンコードを生成する原因になります。代わりに、特殊化定数を使用するか、複数の特殊シェーダー置換を生成します。

計算シェーダーに関する考察

計算シェーダーを開発する場合、スレッド・グループ・サイズを選択する際に最適なパフォーマンスを実現するため、次のガイドラインが役立ちます。

- ワークロードのメモリー・アクセス・パターンの性質に合わせて、スレッドグループのサイズと次元を決定します。例えば、アプリケーションが線形にメモリーをアクセスする場合、64 x 1 x 1 などの線形な次元のスレッド・グループ・サイズを選択します。
- 2 次元スレッドグループでは、通常、スレッド・グループ・サイズが小さいほどパフォーマンスが向上し、実行ユニットのスレッド占有率が高まります。
- 一般に、8 x 8 のスレッド・グループ・サイズは Gen 11 でうまく機能します。状況によっては、メモリー・アクセス・パターンやキャッシュの局所性の観点からこれが最適でない場合があります。この場合、16 x 16 以上の次元数をテストして、テスト結果のパフォーマンスに応じて次元数を選択してください。
- 256 スレッド以上のスレッド・グループ・サイズは、スレッド占有の問題を引き起こす可能性があります。

SLM を使用する計算シェーダーを開発する場合、次のことを考慮してください。

- リードとライトの数を最小限にします。例えば、float4 データの配列は、float 配列の 4 つのバンクではなく、float4 型の 1 つのバンクにロードおよびストアする必要があります。
- メモリーアクセスのペナルティーを回避するため、SLM ではなくレジスターに変数を保持するようにします。
- 連続してアクセスされるデータ要素は、メモリー上で連続するようにデータをロードおよびストアします。これにより、リードとライトアクセスを結合して、メモリー帯域幅を効率良く利用できます。
- ユーザー定義操作で同じ操作を行うため SLM との間でデータを移動する代わりに、HLSL インターロック機能を使用して、min、max、またはその他のリダクションを実行します。コンパイラーは、HLSL 関数をハードウェア実装にマッピングできます。

ウェーブ組み込み関数

Gen 11 では、3D および計算ワークロードの両方でウェーブ組込み関数をサポートしています。これらを使用して効率良いリダクションを記述し、レジスターがレーン間通信でグローバルまたはローカルメモリーへアクセスする回数を減らすことができます。これにより、スレッドグループ内のスレッドは、バリアを使用することなく情報を共有でき、同じウェーブ内のスレッドに対してほかのレーン間操作が可能になります。Gen11 でウェーブ組込み関数を使用する場合、次のことに注意してください。

- Gen アーキテクチャーでは、ウェーブ幅がシェーダーによって異なり、SIMD8、SIMD16、または SIMD32 からシェーダー・コンパイラーによって選択されます。そのため、ウェーブサイズに依存するアルゴリズムでは、WaveGetLaneCount() などの命令を使用します。

- メモリー結果をストアして再ロードする代わりに、ほかのスレッドによってすでにレジスターに格納されているデータにアクセスできるようにすることで、ウェーブ操作のメモリー帯域幅を軽減できます。これは、テクスチャー・ミップマップ生成などの最適化に適しています。

フレーム表示

フレームを表示する場合、可能な限り全画面モードで行うことを推奨します。ウィンドウやその他のモードでは、追加のコンテキスト・スイッチが必要になります。

低電力向けの設計

モバイルやウルトラ・モバイル・コンピューティングは至るところで行われます。これらのプラットフォームでは、CPU と GPU で電力が共有されるため、CPU を最適化すると GPU のパフォーマンスが向上することがよくあります。その結果、バッテリー寿命、デバイス温度、および省電力パフォーマンスが重要になってきました。製造プロセスの細微化が進むと、CPU とプロセッサー・グラフィックスのワットあたりのパフォーマンスは向上します。ソフトウェアがモバイルデバイスの電力使用量を軽減し、電力効率を高める方法は多数あります。以降のセクションでは、それらのパフォーマンス向上をうまく認識する方法と推奨事項を示します。

アイドルとアクティブ電力

プロセッサーは、P ステートや C ステートなどさまざまな電力状態で実行されます。C ステートは本質的にアイドル状態であり、プロセッサーを積極的にシャットダウンすることで消費電力を抑えます。P ステートは、プロセッサーが積極的に電力を消費して、高い周波数でさらに高速に動作するパフォーマンス状態です。

これらの電力状態は、プロセッサーがスリープしている時間とアクティブ時に利用可能な電力を割り当てる方法を定義します。電力状態は急激に変化する可能性があるため、スリープ状態は、リアルタイム・アプリケーションなど、利用可能なすべての電力を消費しないほとんどのアプリケーションに関係します。

アプリケーションを最適化する場合、異なる 2 つの方法で電力を節約してください。

- 必要な場所でアプリケーションが使用するアイドル時間を増やします。
- 全体の電力消費量を改善し、アクティブ時のバランスを取ります。

それぞれの状態の時間を測定することで、アプリケーションの電力状態の動作を判断できます。各状態の電力消費量は異なるため、アプリケーション全体の電力消費を経時的に示す図を取得します。

解析の秘訣

最初に、複数のケースおよび異なる負荷でアプリケーションのベースラインとなる電力使用量を測定します。

- ビデオ再生中のユーザー・インターフェイス (UI) はほぼアイドル状態です。
- 平均的な効果のシーンでの負荷は平均的です。

最悪の状況における負荷は、予測する場所で発生しない可能性があります。あるアプリケーションでは、カットシーンのビデオ再生中に高いフレームレート (1000 フレーム/秒 (FPS)) を確認しました。これにより、GPU と CPU が必要以上の電力を消費する可能性があります。アプリケーションについて調査するには、次のヒントを試してください。

- アプリケーションがバッテリー駆動できる (平均) 時間を測定し、そのパフォーマンスを別のアプリケーションと比較します。消費電力を定期的に測定することで、アプリケーションを変更した場合に消費電力が増加したかどうかを判断できます。
- バッテリー寿命アナライザー (BLA) は、この作業に適したツールです (Windows* のみ)。詳細については、BLA の紹介記事と、高レベルでデータを収集してアプリケーションの電力使用量を解析する方法をご覧ください。BLA のデータが、誤った C ステートに長時間存続する問題を示す場合、さらに詳しく調査します。
- アプリケーションに問題があると報告された場合、または予期しないウェイクアップがある場合は、電力の最適化を行います。これには、Windows* Performance Analyzer (WPA) ツールの使用します。このツールは、CPU 解析に WPA を使用するワークフローを可能にします。
- インテル® VTune™ Amplifier は、ウェイクアップの原因を特定できるため、電力コールスタックを取得するのにも役立ちます。

このような手法で取得されたデータを基に、ウェイクアップを削減または統合して、低電力状態をさらに長く維持します。

アイドル電力の調査

アイドルに近い状態を調査する場合、高いフレームレートに注意してください。

アイドルに近い電力 (カットシーン、メニュー、またはその他の GPU 負荷が低い部分) でのアプリケーションのフレームレートが高い場合、表示間隔を 60Hz のリフレッシュ・レートに固定すると (または、フレームレートを 30FPS に下げると)、アプリケーションでこれらの部分が問題なく表示されます。

ゲームのメニュー、画面の読み込み、およびその他の GPU 負荷の低い部分の動作に注目し、必要に応じてスケーリングし消費電力を最小限に抑えます。これにより、必要に応じてインテル® ターボ・ブースト・テクノロジーを起動できるため、CPU 集約型のロード時間を改善できます。

アクティブ電力とスピードシフト

アクティブ状態では、プロセッサーとオペレーティング・システムの両方が各種システム (CPU、GPU、メモリーリングなど) の動作周波数を決定します。現世代のインテル® Core™ プロセッサーは、オペレーティング・システムとの相互運用性を強化し、電力需要の変化に効率良くかつ迅速に対応します (このプロセスは、インテル® スピード・シフト・テクノロジーと呼ばれます)。

システムは、アクティビティーに基づいて動作周波数のバランスを取り、最も要求の高い場所の周波数 (および電力) を増やします。その結果、アクティブなワークロードは、消費電力に基づいて GPU と CPU のバランスを維持する周波数で動作することがあります。

CPU で動作するワークを減らすと、GPU に電力が回され、その逆もあります。これにより、一方がパフォーマンスのボトルネックであったとしても、全体のパフォーマンスは向上することがあります。

インテル® Power Gadget (英語) などのツールは、主要コンポーネントの動作周波数をリアルタイムで確認するのに役立ちます。ツールを実行して、ターゲットデバイス上の各種サブシステムの周波数を監視できます。

パフォーマンスのボトルネックが高い周波数で実行されておらず、消費電力が利用可能な上限に達した場合、アプリケーションの電力配分バランスが取れていることが分かります。

アクティビティーを減らす場所と方法

ユーザーがパフォーマンスと引き換えにバッテリー寿命を要求する場合、そのような要求を満たすためできることがあります。また、アプリケーションにはわずかな結果しかもたらさないにもかかわらず電力を消費するパターンや、全体の電力使用量を制御するためより効率良く対処できるパターンがあります。以降のセクションでは、全体の消費電力を削減する際に注意すべきいくつかの問題を確認します。

システムの電力設定と電力プロファイルに合わせた設定

以前は電力設定とプロファイル (例えば、GetSystemPowerStatus()) をポーリングする必要がありましたが、Windows Vista* 以降、Windows* では非同期電源通知 API をサポートしています。

- RegisterPowerSettingNotification() と適切なグローバル一意識別子 (GUID) を使用して変更を追跡します。
- 電力プロファイルとデバイスが電源に接続されているかによって、アプリケーションの設定と動作をスケールします。解像度を調整し、最大フレームレートを上限まで下げ、表示品質を下げます。
- フレームレートを制限する場合、V-Sync を使用できます。また、フレームレートと解像度も自身で管理できます。ダイナミック解像度レンダリング (DRR) のサンプル (英語) では、フレームレートを維持するためフレーム解像度を調整する方法を示します。

応答性を維持しつつ可能な限り遅く実行

可能な限り遅く実行 (ただし応答性は維持) すると、電力を節約してバッテリー寿命を延ばすことができます。

- 電力管理モードであることを検出し、フレームレートを制限します。これにより、バッテリー駆動時間が延び、システムを低い温度で動作させることができます。60Hz に代わって 30Hz で実行することで、電力を大幅に節約できます。
- ベンチマーク向けにフレームレートの制限を無効にする手段を提供します。バッテリー消費が早いことをプレイヤーに警告します。プレイヤーにフレームレートの上限を制御する手段を提供する必要もあります。
- オフスクリーン・バッファーを使用して、ゲーム内のユーザー・インターフェイス (状態、パワーアップなどの表示用の小さなパネルに限定されることが多い) のスマート合成を行います。ユーザー・インターフェイスの更新は、通常、ゲーム内のシーンよりはるかに低速であるため、ゲームフレームと同じ頻度で行う必要はありません。ここでも、DDR は UI レンダリングをメインシーンのレンダリングから分離するのに有効な場合があります。

タイマー管理とシステムアイドルの優先、タイトなポーリングループの回避

ほかにも注意すべきことがあります。

- 高解像度の定期的なタイマーへのアプリケーションの依存度を減らします。
- 小さなループでは Sleep() 呼び出しを避けます。代わりに Wait*() AP を使用します。Sleep() またはその他のビジー待機 API により、オペレーティング・システムがマシンをアイドル状態にするのを防ぐことができます。インテルのモバイル・プラットフォーム・アイドル最適化のプレゼンテーション (英語) では、使用する API と回避すべき API の概要を説明しています。
- タイトなポーリングループを回避します。タイトなループでポーリングを行う場合、イベント駆動型に変更します。ポーリングする必要がある場合、可能な限りポーリング間隔を広げてください。
- ビジー待機呼び出しを避けます。これにより、余分な電力消費が生じる可能性があります。オペレーティング・システムや電力管理ハードウェアが、コードが有用でないことを検出する方法はありません。

マルチスレッドの感度

バランスの良いスレッド処理は、パフォーマンスの利点をもたらしますが、バランスの悪いスレッド処理はパフォーマンスと電力効率を低下させる可能性があるため、GPU 連携して動作させる方法を検討する必要があります。オペレーティング・システムがスレッドを直接スケジュールできるように、スレッドのアフィニティー制御を避けます。必要な場合、SetIdealProcessor() を使用してヒントを与えます。

SIMD の使用

インテル® SPMD プログラム・コンパイラー (インテル® ISPC) (英語) や組込み関数を使用して SIMD 命令を生成すると、電力効率とパフォーマンスを大幅に高めることができます。最新の命令セットを使用するとさらに効果が大きくなります。

ただし、インテル® Core™ プロセッサーでは、SIMD 命令を実行する際に SIMD アーキテクチャー・ブロックに電力を供給するため電圧を上げる必要があります。インテル® Core™ プロセッサーは、消費電力の増加を避けるため低い周波数で動作します。スカラー・ワークロードが主体でわずかな SIMD 命令を実行する場合、パフォーマンスが低下する可能性があります。そのため、散発的な SIMD 命令の実行は避けてください。

電力とフレームレート

最新のグラフィックス API (DirectX* 12、Vulkan*、Metal* 2 など) は、CPU のオーバーヘッドを大幅に軽減し、図 7 の左に示すように、固定フレームレート (33FPS) でさらに低い CPU 消費電力となります。フレームレートの制限がない場合、合計消費電力は変化しませんが、GPU 利用率の増加によりパフォーマンスが大幅に向上します。詳細は、Asteroids* と DirectX* 12 のホワイトペーパーをご覧ください。

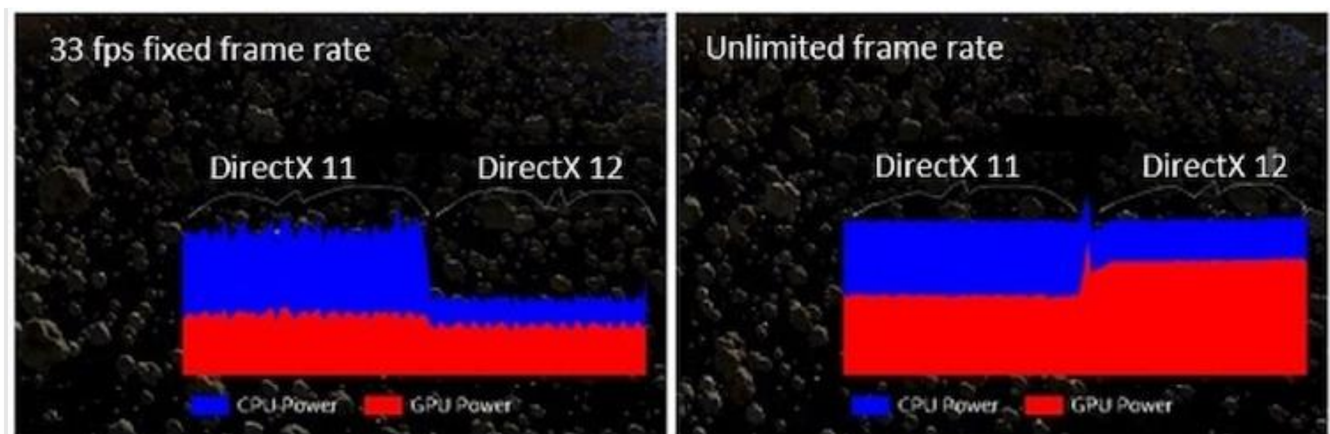

図 5. Asteroids* デモ — 電力とフレームレート

インテル® デベロッパー・ゾーンとゲーム開発者ウェブサイトのリソース

インテル® デベロッパー・ゾーンとゲーム開発者ウェブサイト

インテルは、開発者コミュニティー向けにさまざまなトピックのサンプルコードを定期的に公開しています。最新のサンプルコードについては、次のリンクを参照してください。

- インテル® デベロッパー・ゾーンのコードサンプル (英語)
- GitHub* のインテル・リポジトリー (英語)

以下は、現在のインテル® システムをターゲットとする開発者向けの説明とサンプルへのリンクです。

ダイナミック解像度のレンダリング

DRR (英語) は、表示するレンダーターゲットの固定解像度を維持することでゲームのパフォーマンスを向上し、スムーズにすることを目的とするアルゴリズムですが、エンジン・シェーディングを駆動する解像度を動的に変化させます。

DDR の導入を妨げる主な問題は、後処理パイプラインの変更が必要なことです。DirectX* 12 と配置されたリソースの導入により、追加のダイナミック解像度レンダリング・ターゲット・バッファーを使用してメモリー使用量を増やすことと引き換えに、ほとんどの後処理パイプラインを変更しなくても済むように更新されたアルゴリズムを導入できます。

チェッカーボード・レンダリング

一部のグラフィックス最適化は、ジオメトリーの詳細レベルの削減を重視しますが、チェッカーボード・レンダリング (CBR) では感知できないシェーディングの量を削減します。この技術は、最新の後処理技術と互換性のあるフル解像度のピクセルを生成し、前方と遅延レンダリングの両方に実装できます。さらに詳しい情報、実装の詳細、およびサンプルコードは、「インテル® インテグレーテッド・グラフィックス上でリアルタイム・アップスケーリングを実現するチェッカーボード・レンダリング」をご覧ください。

Conservative Morphological Anti-Aliasing 2.0

Conservative Morphological Anti-Aliasing 2.0 (CMAA-2) は、画像ベースの保守的な形態学的アンチエイリアシング・アルゴリズムの更新です。以前の実装と比較して、アンチエイリアシングの品質とパフォーマンスが向上しています。詳細については、「Conservative Morphological Anti-Aliasing 2.0」(英語) を参照してください。

アダプティブ・スクリーン・スペース・アンビエント・オクルージョン

スクリーン・スペース・アンビエント・オクルージョン (SSAO) は、リアルタイム・レンダリングで使用される一般的なエフェクトで、小規模なアンビエント・エフェクトとコンタクト・シャドウ・エフェクトを生成します。多くの最新ゲームエンジンで使用され、通常フレーム GPU 時間の 5 ~ 10% を使用します。多くのパブリック実装が存在しますが、すべてがオープンソースもしくは無料で利用できるわけではありません。また、低電力モバイルやデスクトップ・デバイスの両方で求められるパフォーマンス・スケーリングを提供するわけではありません。このギャップを、アダプティブ・スクリーン・スペース・アンビエント・オクルージョン (ASSAO) が埋めます。ASSAO は、低電力デバイスとシナリオから高解像度のハイエンド・デスクトップまで、業界標準と同等の外観、設定、および品質を備えた 1 つの実装で拡張できるように設計されています。詳細については、「アダプティブ・スクリーン・スペース・アンビエント・オクルージョン」(英語) を参照してください。

GPU の検出

GPU 検出サンプル (英語) は、GPU からベンダーと ID を取得する方法を示します。インテル® プロセッサー・グラフィックス向けに、デフォルト・グラフィックス品質のプリセット (低、中、高)、DirectX* 9 および DirectX* 11 拡張のサポート、およびビデオメモリーの量を照会する方法を示します。ハードウェアとドライバーでサポートされている場合、最小周波数と最大周波数を照会する方法も示します。

このサンプルでは、ベンダー ID およびデバイス ID ごとにインテル® プロセッサー・グラフィックスをリストした構成ファイルと、各デバイスで推奨されるグラフィックス品質レベルを使用します。パフォーマンスを最大限に高めるには、アプリケーションでいくつかのデバイスをテストし、それぞれに適した品質レベルを決定しなければなりません。プラットフォームのパフォーマンスは利用可能な電力にも大きく依存するため、デバイス ID だけで判断しないでください。デバイスメーカーは、利用可能な電力を最適な熱設計ポイントよりも低い値に設定できます。

高速 ISPC テクスチャー圧縮

高速 ISPC テクスチャー比較の例 (英語) は、インテル® ISPC を使用して CPU で SIMD 命令セットを活用し、高品質な BC7、BC6H、ETC1、および ASTC 圧縮を行います。

関連情報

Direct3D* ウェブサイト (英語) – DirectX* 12 とほかの DirectX* のリソース
Vulkan* (英語) – Khronos* の追加リソースを提供するサイト
Metal* 2 (英語) – Apple* の Metal* 2 向け開発者サイト

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