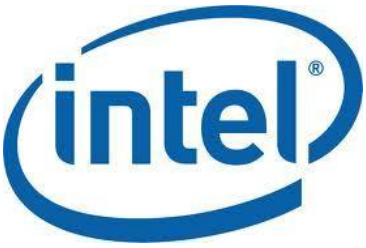


Graphics API Developer's Guide

For 6th Generation Intel® Core™ Processors

Abstract

Welcome to the Graphics API Developer's Guide for 6th generation Intel® Core™ processors, which provide developers and end users with cutting-edge CPU and graphics power enhancements, numerous added features and capabilities, and significant performance gains.

This guide is designed to help software developers optimize applications (apps) for use on systems with 6th gen Intel® Core™ processors, and best leverage the myriad upgrades and expansions that these processors introduce. Inside, you'll find an overview of these processors' next-generation graphics hardware, and how to best design software to take advantage of them. You'll also find guidance on structuring and optimizing Direct3D* 12 and OpenGL* 4.4 code, as well as hints, tutorials, step-by-step instructions, and references to key code samples that will help with the app development process.

This guide expands on previous guides and tutorials. For additional details, or further hints, tips, and expert insights on Intel® processors and prior graphics APIs, you can reference a full range of reference materials at Intel's online Processor Graphics document library.

Table of Contents

1. INTRODUCTION 5
2. ARCHITECTURAL OVERVIEW 6
   2.1. GPU Architecture 6
   2.2. Conservative Rasterization 8
   2.3. Memory 10
3. TOOLS 11
   3.1. Selecting the Most Effective Tool 11
   3.2. Intel® Graphics Performance Analyzers (Intel GPA) 12
      3.2.1. Intel GPA System Analyzer 12
      3.2.2. Intel GPA Graphics Frame Analyzer 14
      3.2.3. Intel GPA Platform Analyzer 16
   3.3. Intel® VTune™ Amplifier XE 17
      3.3.1. Hotspots Analysis 18
      3.3.2. Locks and Waits Analysis 18
      3.3.3. GPU and Platform Level Analysis 19
      3.3.4. Slow Frame Analysis 19
      3.3.5. User Task Level Analysis with Code Instrumentation 20
4. COMMON GRAPHIC OPTIMIZATIONS 21
   4.1. Optimizing Clear, Copy, and Update Operations 21
   4.2. Render Target Definition and Use 21
   4.3. Texture Sampling and Texture Definition 21
   4.4. Geometry Transformation 22
   4.5. General Shading Guidance 22
   4.6. Constants 23
   4.7. Anti-Aliasing 23
5. DIRECT3D 24
   5.1. Direct3D 12 Optimizations 24
      5.1.1. Clear Operations 24
      5.1.2. Pipeline State Objects 24
      5.1.3. Asynchronous Dispatch (3D + Compute) 24
      5.1.4. Root Constants 24
      5.1.5. Compiled Shader Caching 25
      5.1.6. Conservative Rasterization and Raster Order Views 25
      5.1.7. Root Signature 25

5.1.8. Command Lists and Barriers 25
5.1.9. Resource Creation 25
5.2. DirectCompute Optimizations 26
5.2.1. Dispatch and Thread Occupancy 26
5.2.2. Resource Access 27
5.2.3. Shared Local Memory 27
5.3. Evaluating Pipeline Hotspots 27
5.3.1. Selecting Ergs to Study 28
5.3.2. Performance Analysis with Hardware Metrics 28
5.3.3. GTI - Graphics Interface to Memory Hierarchy (LLC, EDRAM, DRAM) 30
5.3.4. Pixel Back-End – Color Write and Post-Pixel Shader (PS) Operations 30
5.3.5. Shader Execution FPU Pipe 0/1 31
5.3.6. Shader Thread EU Occupancy 31
5.3.7. 3D Thread Dispatch 31
5.3.8. Early Depth/Stencil 32
5.3.9. Rasterization 32
5.3.10. Geometry Transformation 32
5.3.11. Shader Execution Stalled 33
5.3.12. Unknown Shader Execution Hotspot 33
5.3.13. Graphics Cache (L3) 33
5.3.14. Sampler 33
5.3.15. Unknown Shader Stall 34
6. OPENGL 35
6.1. OpenGL Shaders 35
6.2. Textures 36
6.3. Images 36
6.4. Shader Storage Buffer Objects 36
6.5. Atomic Counter Buffers 37
6.6. Frame Buffer Object (FBO) 37
6.7. State Changes 37
6.8. Avoid CPU/GPU Synchronization 38
6.9. Anti-Aliasing Options 39
7. POWER 40
7.1. Idle and Active Power 40
7.2. Analysis Tips 40

7.3. Investigating Idle Power 41
7.4. Active Power and Speed Shift 41
7.5. When and How to Reduce Activity 42
7.5.1. Scale Settings to Match System Power Settings and Power Profile 42
7.5.2. Run as Slow as You Can, While Remaining Responsive 43
7.5.3. Manage Timers and Respect System Idle, Avoid Tight Polling Loops 43
7.5.4. Multithread Sensibly 44
8. SAMPLES AND REFERENCES 45
9. CONCLUSION 47

1. INTRODUCTION

With the release of 6th generation Intel® Core™ processors (codename: Skylake), which provide unparalleled computing and visual power, developers can now take graphics capabilities to the next level. These processors are readily capable of meeting the high-end computing and graphical performance needs of both mobile platforms and premium desktop gaming systems.

This guide introduces the graphics hardware architecture of 6th gen Intel Core processors and provides expert tips and best practices to consider when leveraging their features and capabilities. The document also provides guidance and direction on how to get better performance from the newest graphical APIs running on Intel® Processor Graphics.

In addition, you'll also find reference material for using the latest graphics APIs (Direct3D* 12 and OpenGL* 4.4), with occasional information on older APIs interwoven throughout. For full details on Direct3D 11, OpenGL 4.3, and earlier graphics API editions, as well as additional details on programming for earlier generations of Intel® processors, please see our online Processor Graphics document library.

2. ARCHITECTURAL OVERVIEW

The 6th gen Intel Core processors offer a number of all-new benefits over previous generations, and provide significant boosts to overall computing horsepower and visual performance.

Sample enhancements include a GPU in the 6th gen Intel Core processor that, coupled with the CPU's added computing muscle, provides up to 40 percent better graphics performance over prior Intel Processor Graphics, along with enhanced 4K video playback capabilities.

In several important respects, the 6th gen Intel Core processors' graphics hardware architecture is similar to that of 5th generation Intel® Core™ processors (please see previous guides for more information). However, 6th gen Intel Core processors have been redesigned to offer higher-fidelity visual output, higher-resolution video playback, and more seamless responsiveness for systems with lower power usage.

Because the high-level GPU architecture is similar to the previous generation of Intel® Core™ processors (codename: Broadwell), you'll find the same number of execution units (EUs) per slice (a group of EUs plus some shared logic), and the same EU-to-sampler ratio. (Please see the Gen8 compute architecture and Graphics API Developer's Guide for 5th generation Intel® Core™ processors documents for more information.)

As with 6th gen Intel Core processor CPUs, Intel Processor Graphics have evolved over time, too. The 9th generation (Gen9) GPUs used in these latest Intel processors now feature integrated graphics capabilities that provide more than a 50 percent performance gain over the previous 8th generation GPUs. This architecture supports GPUs with up to three slices (providing 72 EUs). This architecture also offers increased power gating and clock domain flexibility, which are well worth taking advantage of.

Below, you'll find additional information to help you optimize for the full range of capabilities found in the 6th gen Intel Core processors. For additional information on the compute capabilities of the 6th gen Intel Core processors, please see The Compute Architecture of Intel® Processor Graphics Gen9 whitepaper.

2.1. GPU Architecture

Memory access includes atomic min, max, and compare-and-exchange for 32-bit floating point values in either shared local memory or global memory. The new architecture also offers a performance improvement for back-to-back atomics to the same address. Tiled resources include support for large, partially resident (sparse) textures and buffers. Reading unmapped tiles returns zero, and writes to them are discarded. There are also new shader instructions for clamping LOD and obtaining operation status. There is now support for larger texture and buffer sizes. (For example, you can use up to 128k X 128k X 8B mipmapped 2D textures.) See Figure 2-1 to better understand the architecture.

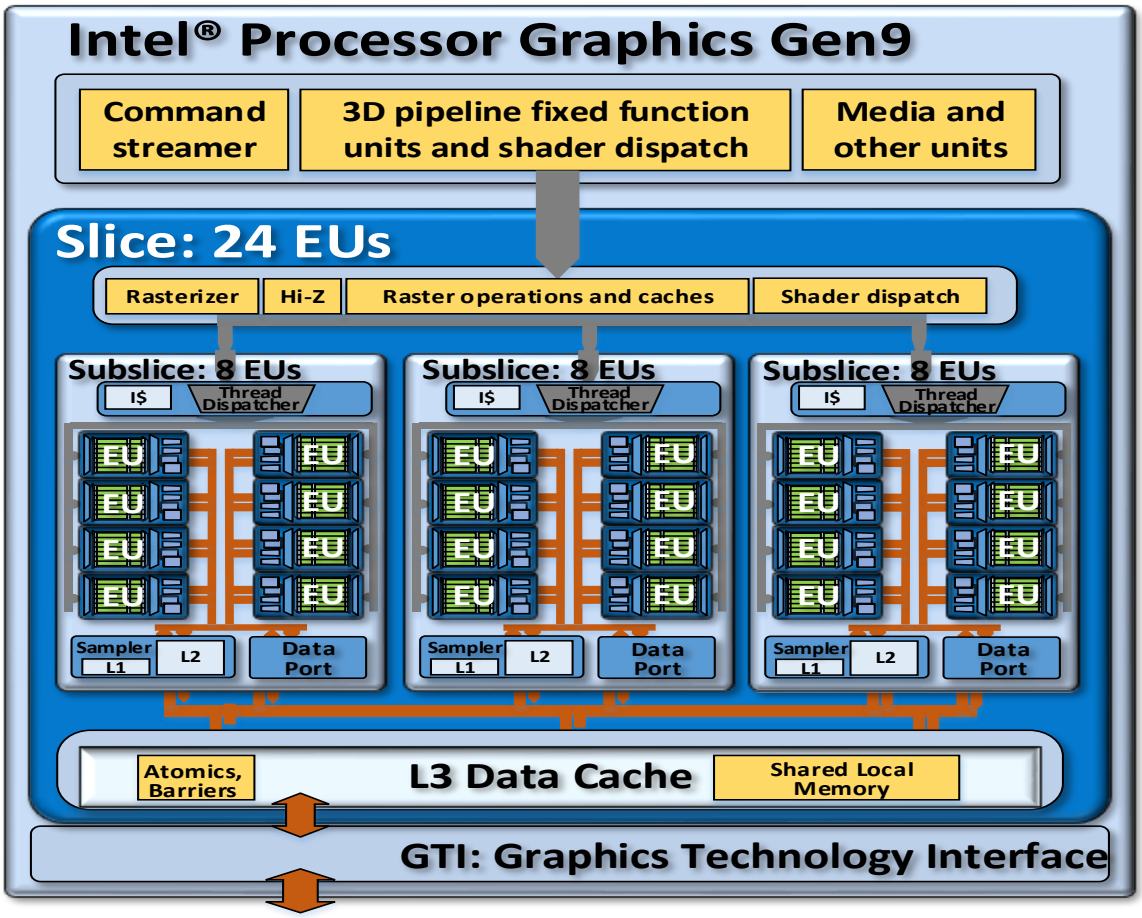


Figure 2-1: Gen9 graphics architecture

Bindless resources increase the number of dynamic resources a shader may use, from ~256 to ~2,000,000 when supported by the graphics API. This change reduces the overhead associated with updating binding tables, and provides more flexibility to programmers.

DirectX* standard swizzle support also enables the ability to directly author swizzled texture data and it offers more efficient sharing of textures across adapters with different architectures.

Adaptive scalable texture compression (ASTC) capabilities also offer higher compression ratios than other compression technologies. These upgrades provide significant enhancements for lower-power versions of 6th gen Intel Core processors as larger ASTC tiles provide better quality than reducing texture resolution. See Figure 2-2 to compare compression techniques.

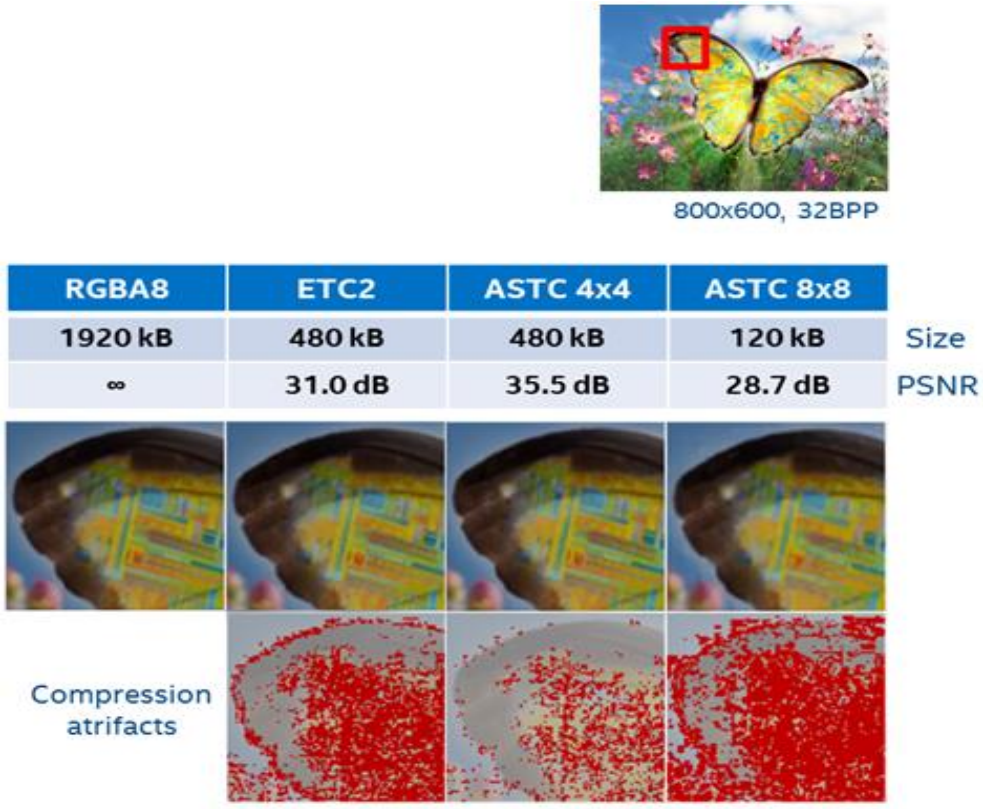


Figure 2-2: Different compression techniques and their compression artifacts

These enhancements additionally provide for lossless color compression with automatic compression on store to memory, and decompression on load from memory. There's a maximum peak compression ratio of 2:1.

The fixed function pipeline also offers improved geometry throughput and pixel back-end fill rates, with fast clear support for any clear color.

2.2. Conservative Rasterization

This GPU architecture adds hardware support for conservative rasterization, allowing for any pixels that are even partially covered to be rasterized (outer conservative). When you're using conservative rasterization, there's a flag in the shader to indicate whether the pixel is fully (inner conservative) or partially covered.

The implementation is truly conservative with respect to floating point inputs, and is at most $1/256^{th}$ of a pixel over-conservative. No covered pixels are missed or incorrectly flagged as fully-covered. Post-snapped degenerate triangles are not culled either. (See Figure 2-3 for details.)

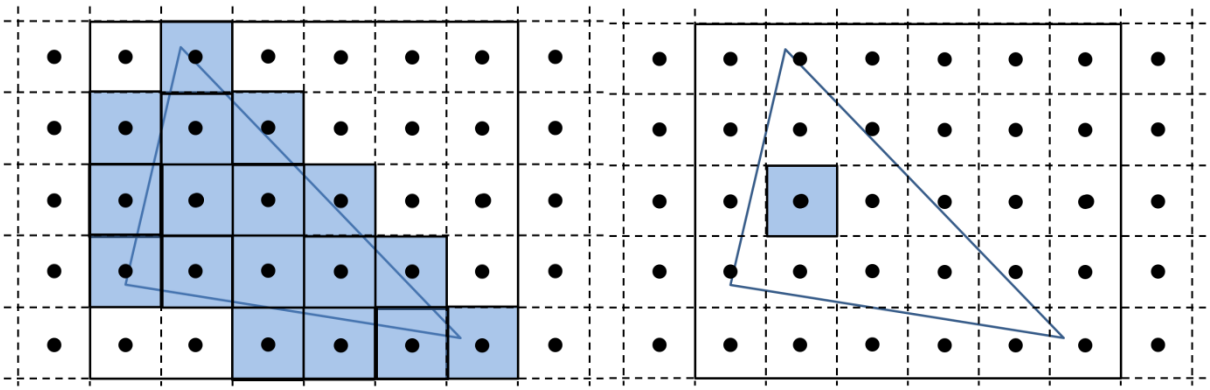


Figure 2-3: Triangles using outer-conservative (left) and inner-conservative (right) rasterization rules

A depth coverage flag also notes whether each sample was covered by rasterization and has also passed the early depth flag test or SV_DepthCoverage in Direct3D.

The new architecture further offers improved performance for 2x, 4x and 8x multi-sample anti-aliasing (MSAA), with added 16x MSAA support for all surface formats. The improved parallel copy engine can also copy and reorganize resources simultaneously to other compute or graphics work.

Execution units have improved native 16-bit floating point support as well. This enhanced floating point support leads to both power and performance benefits when using half precision.

Display features further offer multi-plane overlay options with hardware support to scale, convert, color correct, and composite multiple surfaces at display time. Surfaces can additionally come from separate swap chains using different update frequencies and resolutions (for example, full-resolution GUI elements composited on top of up-scaled, lower-resolution frame renders) to provide significant enhancements.

2.3. Memory

Graphics resources allocated from system memory with write-back caching will utilize the full cache hierarchy from the CPU: Level 1 (L1), Level 2 (L2), Last Level Cache (LLC), optional EDRAM, and finally DRAM. When accessed from the GPU, sharing can occur in LLC and further caches/memory. Additionally, the GPU architecture has numerous bandwidth improvements including increased LLC and ring bandwidth. Coupled with support for DDR4 RAM, these help hide latency.

This architecture supports products with up to 128 MB EDRAM. In these products, the EDRAM acts as memory-side cache for all DRAM access including I/O and display.

When detecting the amount of available video memory for your graphics device, remember that different rules may apply to different devices. For example: An integrated graphics part reports a small amount of dedicated video memory. Because the graphics hardware uses a portion of system memory, the reported system memory is a better indication of what your game may use. This should be taken into account when using the reported memory as guidance for enabling or disabling certain features. The GPU Detect Sample shows how to obtain this information in Windows*.

3. TOOLS

A number of helpful tools can help you understand how well your applications are running on Intel Processor Graphics, and how to optimize these applications to work most effectively with 6th gen Intel Core processors. Below, you'll find a number of tools that allow you to both locate and fix performance issues, and design code to operate more efficiently.

Because of the complex nature of games and graphics applications, bear in mind that no one single tool can provide all answers regarding possible performance problems. Therefore, if you find potential issues, it's best to use a collection of tools to understand overall performance behavior. By using each, you can gain deeper insights into possible performance challenges and bottlenecks.

Next, we'll take a closer look at several analysis tools from Intel. In addition, when reviewing applications' graphical performance, consider some other useful tools not described in this guide. For example:

- Microsoft* Visual Studio* Graphics Analyzers
- GPUView and WPA from the Windows Performance Toolkit
- RenderDoc*

Note: As graphics APIs evolve, these tools steadily change to support the latest API versions. Look for tools that support the most current editions of your favorite graphics APIs, and stay tuned for more exciting tool features to come in the months ahead.

3.1. Selecting the Most Effective Tool

One of the first steps when scanning for possible performance issues is to determine if the application is CPU- or GPU-bound.

A GPU-bound application has at least one CPU thread waiting for GPU work to complete. The usual waiting point is Present(), when the CPU is too far ahead of the GPU.

A CPU-bound application has the GPU wait on completion from work happening on the CPU. The GPU work gets queued in a command buffer. If the GPU finishes what is available in its queue before the CPU can provide more work, the GPU front-end starves. Remember that all graphics runtime and driver work is done on the CPU, even when its sole role is to drive the GPU.

Encountering problems with your software application's overall performance? The following steps will help you determine whether a target application is CPU- or GPU-bound.

- Make sure that the "Present" (or equivalent) call is not synchronized to the monitor refresh, that is, frames are being submitted as fast as possible.
- A quick-and-dirty method to determine if an application is CPU-bound is often to measure frame rate at high resolution (with MSAA on, and so on). Then, also do so at the lowest possible resolution with no MSAA enabled as well. If you discover that there is a very small delta between frame rates in these cases, the application is most likely CPU-bound.
- For a more accurate determination, use Intel® Graphics Performance Analyzers (Intel® GPA) System Analyzer (see Section 3.2) to measure the load on the GPU and CPU.
- For an even more accurate determination, including GPU-CPU synchronizations, use Intel GPA Platform Analyzer, Intel® VTune™ Amplifier XE or GPUView.

Once you've established where the application is bound, you can then drill down and either tune CPU performance using Intel VTune Amplifier XE (for more information, refer to section 3.3for more info), or tune GPU Performance using Intel® GPA.

3.2. Intel® Graphics Performance Analyzers (Intel GPA)

Intel GPA includes powerful, agile tools that enable game developers to utilize the full performance potential of their gaming platform, including (though not limited to) Intel® Core™ processors and Intel Processor Graphics, as well as Intel® processor-based tablets running the Android* platform. Intel GPA tools visualize performance data from your application, enabling you to understand system-level and individual frame performance issues. These tools also let you perform 'what-if' experiments to estimate potential performance gains from optimizations.

3.2.1. Intel GPA System Analyzer

Intel GPA System Analyzer is a real-time tool that displays CPU, Graphics API, GPU performance and power metrics. It can help you quickly identify key performance opportunities and identify whether your workload is CPU- or GPU-bound so you know where to focus your efforts. With the tool, you can use graphics pipeline state override experiments to conduct a fast, high-level iterative analysis of your game, all without changing a single line of code.

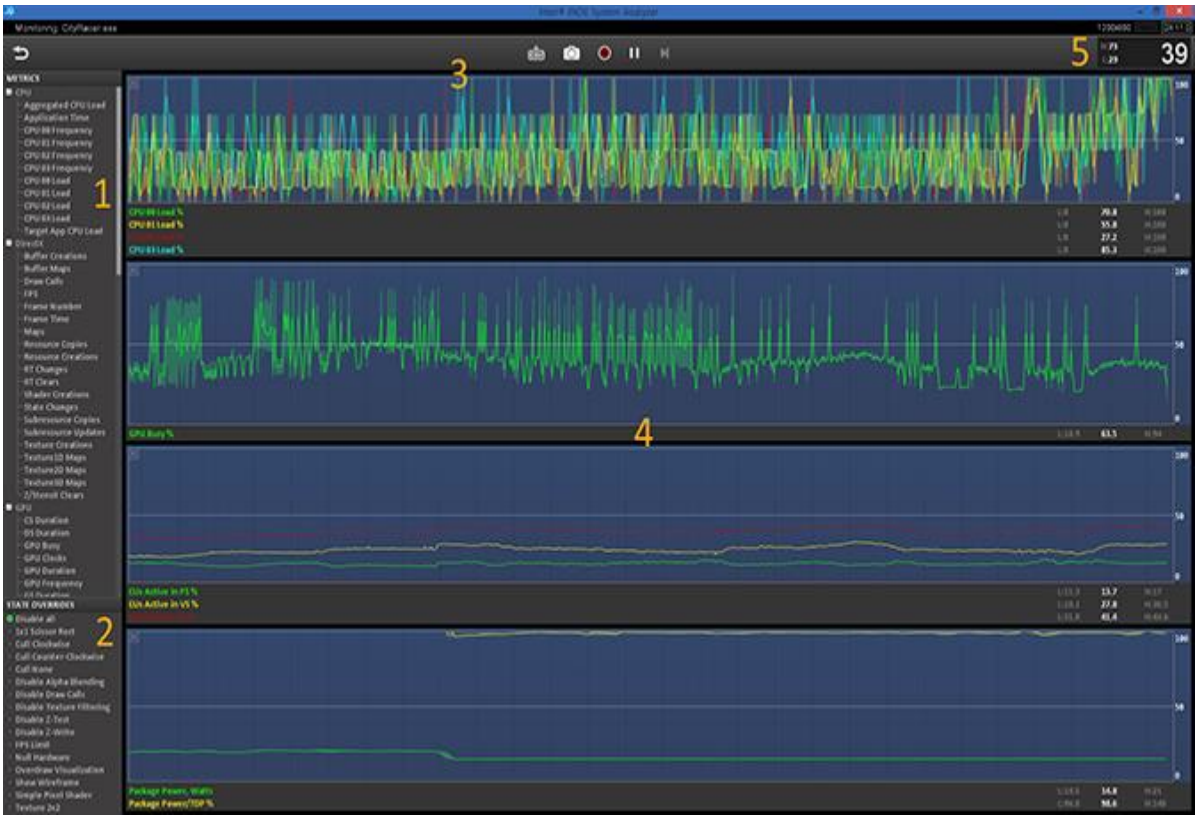


Figure 3-1: System Analyzer shows real-time status for DirectX or OpenGL ES games

With Intel GPA System Analyzer you can:

1. Display the overall system metrics, or specific metrics, for a selected application.
2. Select from a range of metrics from the CPU, GPU, graphics driver, and either DirectX or OpenGL ES.
3. Perform various “what if” experiments using override modes to quickly isolate many common performance bottlenecks.
4. Capture frames, traces, export metric values, and pause/continue data collection.
5. Show the current, minimum, and maximum frame rate.

Figure 3-2: The Heads-Up Display (HUD) for Microsoft DirectX* workloads shows realtime metrics

3.2.2. Intel GPA Graphics Frame Analyzer

Intel GPA Graphics Frame Analyzer is a powerful, intuitive, single-frame analysis and optimization tool for Microsoft DirectX game workloads. It provides deep frame performance analysis down to the draw call level, including shaders, render states, pixel history, and textures. You can conduct “what if” experiments to see how changes iteratively impact performance and visuals without having to recompile your source code.

There are two versions of Intel GPA Graphics Frame Analyzer. For DirectX 9, 10, and 11 games, open your frame with Intel GPA Graphics Frame Analyzer (DirectX 9, 10, 11). For all other supported frame types (including Direct3D 12 and OpenGL ES), open the frame with Intel GPA Graphics Frame Analyzer. The features and navigation between the two analyzers vary slightly, but each supports the same types of analysis.

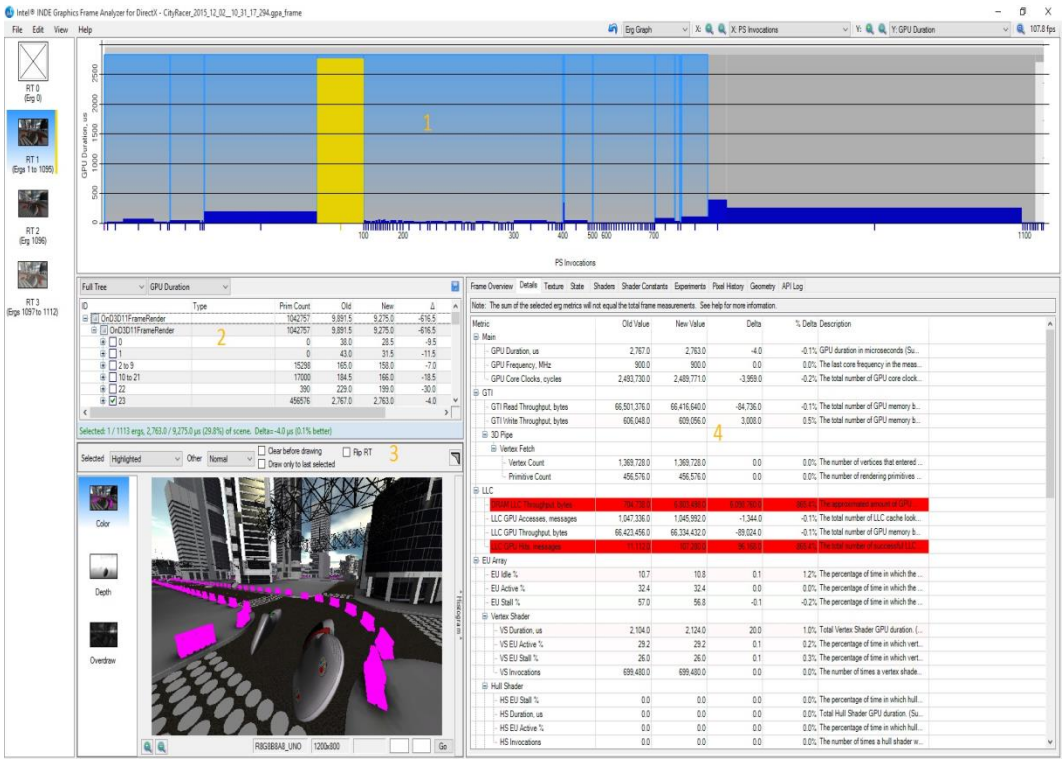


Figure 3-3: Intel GPA Graphics Frame Analyzer (DirectX 9, 10, 11)

With Intel GPA Graphics Frame Analyzer (DirectX 9, 10, 11), you can:

1. Graph draw calls with axes based on a variety of graphics metrics.
2. Select regions and draw calls in a hierarchical tree.
3. View the impact of selections and real-time experiments on render targets and the final frame buffer.
4. View detailed metrics and frame data, perform live graphics pipeline bottleneck experiments, as well as edit graphics state, shaders, and more.
5. Study geometry.

3.2.3. Intel GPA Platform Analyzer

Intel GPA Platform Analyzer lets you see where your application is spending time across the CPU and GPU. This will help ensure that your software takes full advantage of the processing power available from today's Intel® platforms.

Intel GPA Platform Analyzer provides offline analysis of CPU and GPU metrics and workloads with a timeline view for analysis of tasks, threads, Microsoft DirectX, OpenGL ES, and GPU-accelerated media applications in context.

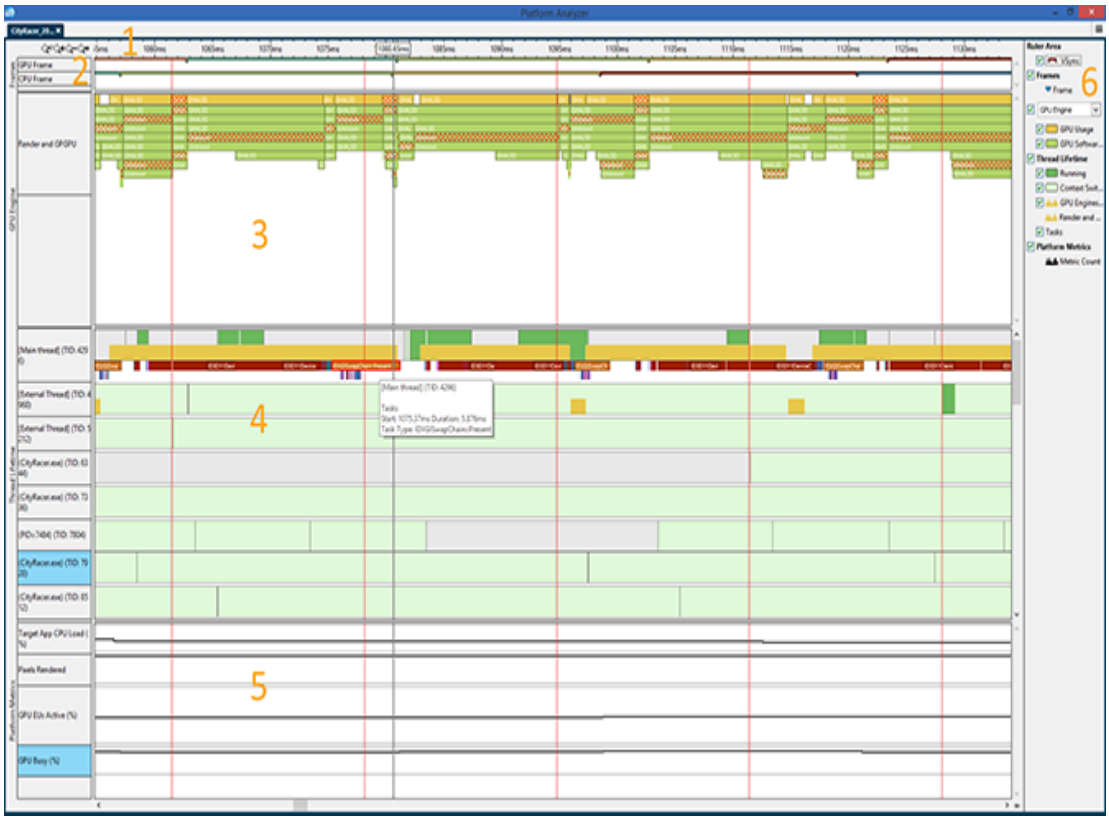


Figure 3-4: Intel GPA Platform Analyzer shows a timeline of activity and metrics in the CPU and GPU

With Intel GPA Platform Analyzer, you can:

1. View task data in a detailed timeline.
2. Identify CPU and GPU frame boundaries.
3. Explore queued GPU tasks.
4. Explore CPU thread utilization and correlate to DirectX API use if applicable.
5. Correlate CPU and GPU activity based on captured platform and hardware metric data.
6. Filter and isolate the timeline to focus on a specific duration in time.

3.3. Intel® VTune™ Amplifier XE

Intel® VTune™ Amplifier XE provides a rich set of insights into CPU and GPU performance, threading performance, scalability, bandwidth, caching and much more. You can use the Intel VTune Amplifier XE's powerful analysis tools to sort, filter and visualize results on the timeline and on your source. For more on Intel VTune Amplifier XE and key features, please see the product webpage. You can also find full instructions and support for using Intel VTune Amplifier XE online at Intel's website.

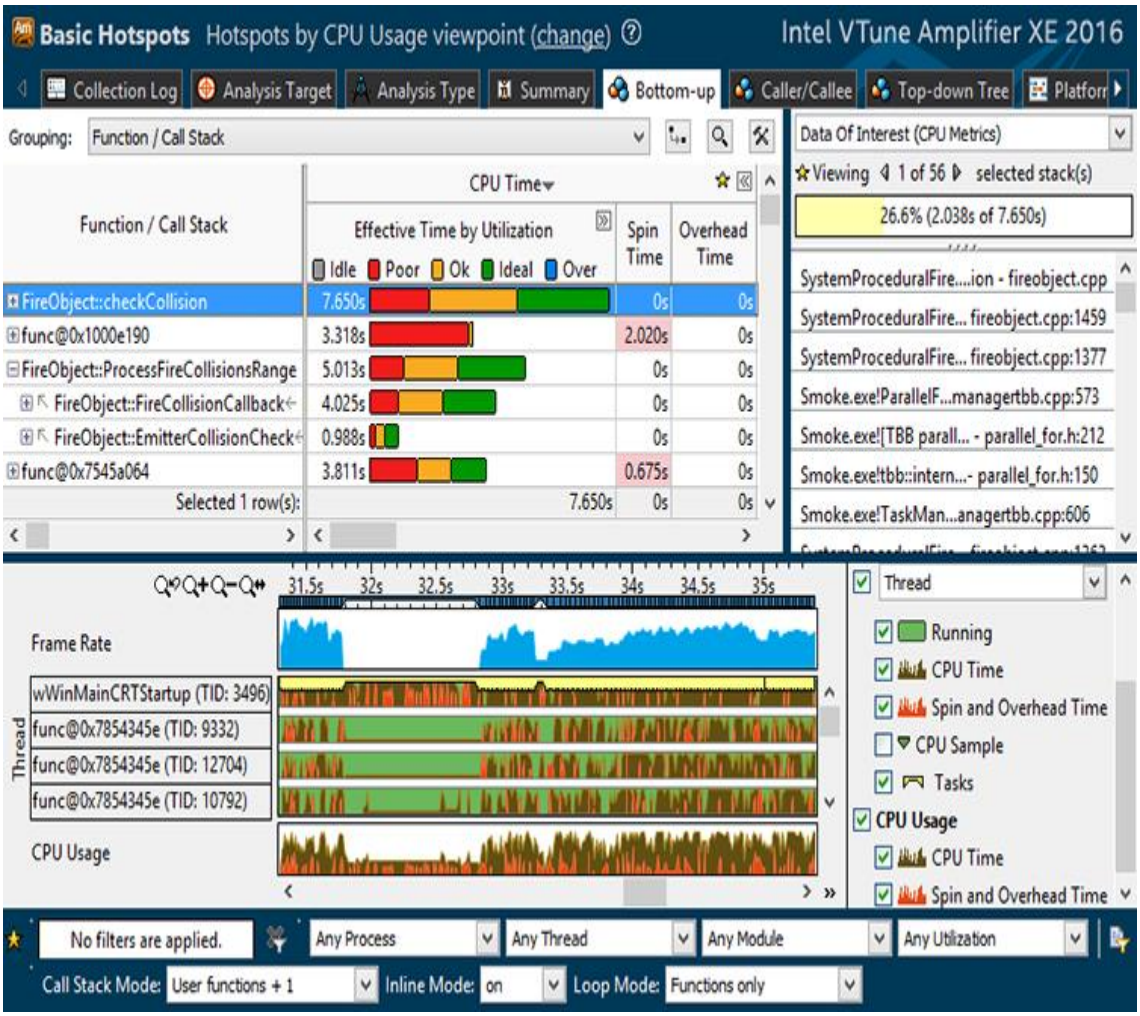


Figure 3-5: Sort, filter and visualize data from the GUI or automate with the command-line interface

3.3.1. Hotspots Analysis

Intel VTune Amplifier XE quickly locates code that is taking up a lot of time. Hotspots analysis features provide a sorted list of the functions that are using considerable CPU time - an area where fine-tuning can provide significant gains, especially if your game is CPU-bound. If you have symbols and sources, you can easily drill down to find the costliest functions, and Intel VTune Amplifier XE will provide profiling data on your source to indicate hotspots within the function of interest.

3.3.2. Locks and Waits Analysis

With Intel VTune Amplifier XE, you can also locate slow, threaded code more effectively. You can use locks and waits analysis functions (see Figure 3-6) to resolve issues such as having to wait too long on a lock while cores are underutilized. You can also use them to address challenges where wait is a common cause of slow performance in parallel programs.

Visualization on the timeline (as depicted below) can additionally help you maximize software performance by letting you easily monitor and spot lock contention, load imbalance, and inadvertent serialization. These are all common causes of poor parallel performance.

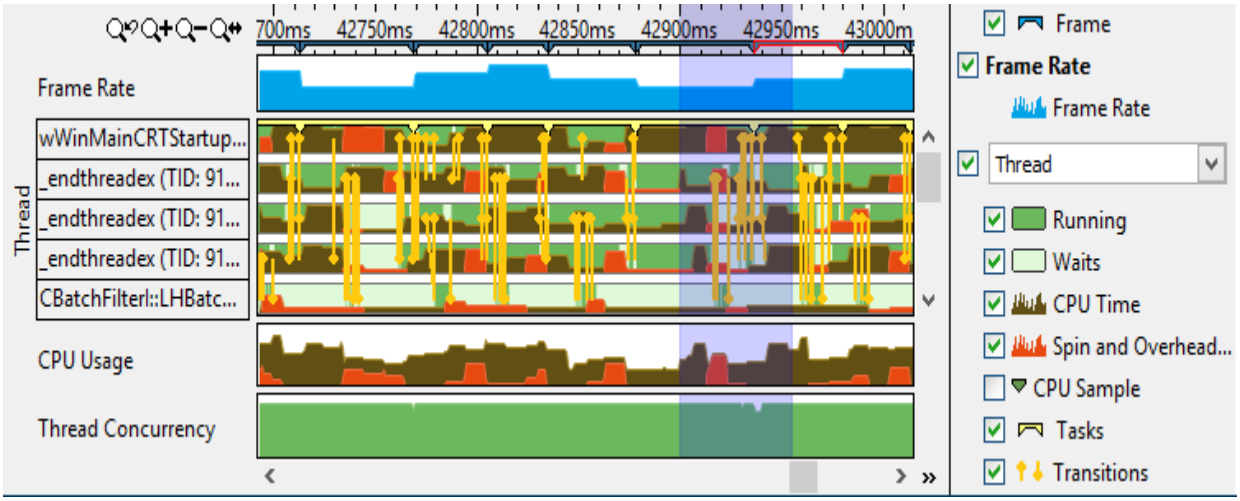


Figure 3-6: Timeline filtering helps you find issues with CPU time and parallel performance.

In Figure 3-6, the yellow lines depict transitions, where a high density of transitions may indicate lock contention and poor parallel performance. For ease of use, you can also turn off CPU time marking to diagnose issues with spin locks to see precisely when threads are running or waiting. Doing so will let you quickly spot inadvertent serialization.

3.3.3. GPU and Platform Level Analysis

On newer Intel Core processors, you can collect GPU and platform data, and correlate GPU and CPU activities. Configure Intel VTune Amplifier XE to explore GPU busyness over time and understand whether your application is CPU- or GPU-bound.

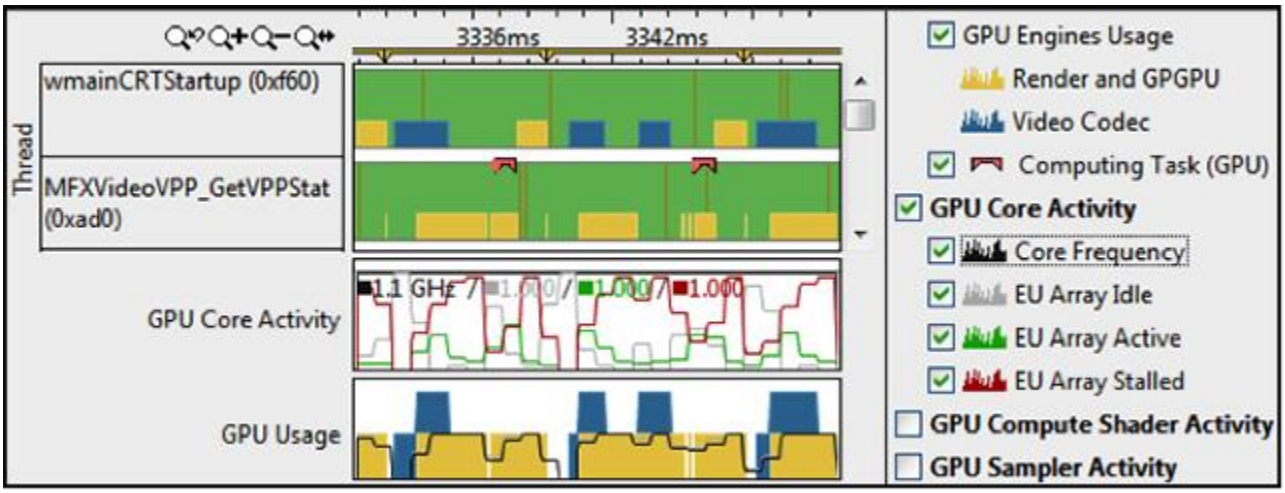


Figure 3-7: GPU and CPU activities can be compared and correlated.

A sample rule of thumb: If the Timeline Pane in the Graphics window shows that the GPU is busy most of the time with small idle gaps between busy intervals, and a GPU software queue that rarely decreases to zero, your application is GPU-bound. However, if the gaps between busy intervals are big and the CPU is busy during these gaps, your application is most likely CPU-bound.

But such obvious situations are often rare. You may need to undertake a detailed analysis to understand all dependencies. For example, an application may be mistakenly considered GPU-bound when GPU engine usage is serialized. (Example: when GPU engines responsible for video processing and for rendering are loaded in turns.) In this case, an ineffective scheduling on the GPU results from the application code running on the CPU instead.

If you find that the GPU is intensely busy over time, you can look deeper and understand whether it is being used effectively. Intel VTune Amplifier XE can collect metrics from the Render engine of Intel GPUs to help this analysis.

3.3.4. Slow Frame Analysis

Got a slow spot in your Windows gameplay? You don't just want to know where you are spending a lot of time. You also want to know where the frame rate is slow. Intel VTune Amplifier XE can automatically detect DirectX frames and filter results to show you what's happening in slow frames.

3.3.5. User Task Level Analysis with Code Instrumentation

Intel VTune Amplifier XE also provides a task annotation API that you can use to annotate your source. Then, when you study your results in Intel VTune Amplifier XE, it can display which tasks are executing. For instance, if you label the stages of your pipeline, they will be marked in the timeline, and hovering over them will reveal further details. This makes profiling data much easier to understand.

4. COMMON GRAPHIC OPTIMIZATIONS

There are many similarities between the different APIs that run on Intel Processor Graphics. In this section we'll take a closer look at how to maximize performance when designing and engineering applications for use with Intel Core processors.

Optimizations covered in this section apply to all APIs interacting with Intel Processor Graphics. Below, you'll find a number of hints and tips for working with these interfaces when designing applications for use with Intel Core processors. Use them to ensure optimum performance.

4.1. Optimizing Clear, Copy, and Update Operations

To achieve the best performance when performing clear, copy, or update operations on resources, please follow these guidelines:

- Use the provided API functions for clear, copy, and update needs. Do not implement your own version of the API calls in the 3D pipeline.
- Enable hardware fast clear as specified, and utilize 0 or 1 for each channel in the clear color for color or depth.
- Copy depth and stencil surfaces separately instead of trying to copy both at the same time.

4.2. Render Target Definition and Use

Use the following guidelines to achieve the best performance when rendering to a render target:

- Use as few render targets as you can. Combine render targets when possible.
- Define the appropriate format for rendering, that is, avoid defining unnecessary channels or higher-precision formats when not needed.
- Avoid using SRGB formats where unnecessary.

4.3. Texture Sampling and Texture Definition

Sampling is a common shader operation and it is important to optimize both the definition of surfaces sampled along with the sampling operations. Follow these guidelines to achieve the best performance when sampling:

- When sampling from a render target, avoid sampling across levels in the surface with instructions like sample_l when possible.
- Make use of API-defined compression formats (BC1-BC7) to reduce memory bandwidth and improve locality of memory accesses when sampling.

- If you're creating textures that are not CPU-accessible, define them so they match the render target requirements. This will enable lossless compression when possible.
- Avoid read hazards between sample instructions. For example, the UV coordinate of the next sample instruction is dependent upon the results of the previous sample operation.
- Define appropriate resource types for sampling operation and filtering mode. Example: Do not use volumetric surface options when texture 2D or 2D array would have been appropriate.
- Avoid defining constant data in textures that could be procedurally computed in the shader such as gradients.
- Use constant channel width formats (for example, R8B8G8A8) rather than variable channel width formats (R10G10B10A2) for operations that may be sampler bottlenecked. This includes those surfaces that are read many times with little other work in a shader.
- Use non-volumetric surfaces for operations that may be sampler bottlenecked.

4.4. Geometry Transformation

Follow these guidelines to ensure maximum performance during geometry transformation:

- Define input geometry in structure of arrays (SOA) instead of array of structures (AOS) layouts for vertex buffers, for example, by providing multiple streams (one for each attribute) versus a single stream containing all attributes.
- Do not duplicate shared vertices in mesh to enable better vertex cache reuse, that is, merge edge data to avoid duplicating the vertices.
- Optimize transformation shaders (VS->GS) to only output attributes consumed by downstream shader stages. Example: Avoid defining unnecessary outputs from a vertex shader that are not consumed by pixel shader.
- Avoid spatial overlap of geometry within a single draw when stencil operations are enabled. Pre-sort geometry to minimize overlap, commonly seen when performing functions such as foliage rendering.

4.5. General Shading Guidance

To achieve optimal performance when shading follow these guidelines:

- Avoid creating shaders where the number of temporaries (example: dcl_temps for D3D) is≥ 16. Optimize to reduce the number of temporaries in the shader assembly.
- Structure code to avoid unnecessary dependencies (example: dependencies on high latency operations like sample).

- Avoid flow control decisions on high-latency operations. Structure your code to hide the latency of the operation that drives control flow.
- Avoid flow control decisions using non-uniform variables, including loops. Try to ensure uniform execution among the shader threads.
- Avoid querying resource information at runtime to make decisions on control flow, or unnecessarily incorporating resource information into algorithms. For example: disabling a feature by binding an arbitrarily-sized (1x1) surface.
- Implement fast paths in shaders to early-out when possible in algorithms that deal with common values where the output of the algorithm can be predetermined or computed at a lower cost of the full algorithm.
- Use discard (or other kill pixel operations) where output will not contribute to the final color in the render target. For example: Blending where the output of the algorithm is an alpha of 0, or adding inputs to shaders that are 0s that negate output.

4.6. Constants

Follow these guidelines when defining and using constants to achieve the best possible results from software applications:

- Structure constant buffers to improve cache locality, that is, so accesses all occur on the same cache line.
- Accesses have two modes, direct (offset known at compile time) and indirect (offset computed at runtime). Avoid defining algorithms that rely on indirect accesses, especially with control flow or tight loops. Make use of direct accesses for high-latency operations like control flow and sampling.

4.7. Anti-Aliasing

For the best anti-aliasing performance, follow these guidelines:

- For improved performance over MSAAx4, use CMAA (see this CMAA article for more information)
- Avoid querying resource information from within a loop or branch where the result is immediately consumed or duplicated across loop iterations.
- Minimize per-sample operations and when shading in per-sample maximize the number of cases where any kill pixel operation is used (example: discard) to get the best surface compression.
- Minimize the use of stencil or blend when MSAA is enabled.

5. DIRECT3D

This section describes optimizations for Direct3D 12 and DirectCompute. It also shows tips for measuring hotspots in the pipeline. For older versions of Direct3D, older processors and for more information on DirectCompute, see the Intel Processor Graphics page.

5.1. Direct3D 12 Optimizations

In addition to general optimization guidelines, here are some additional guidelines for Direct3D 12. By following them, you can ensure maximum performance from applications, and ensure optimum visual fidelity.

5.1.1. Clear Operations

To get maximum performance of depth and render target clears, provide the optimized clear color when creating the resource and use only that clear color when clearing any of the RTVs/DSVs associated with the surface.

5.1.2. Pipeline State Objects

Introduced in Direct3D 12 as a collection of shaders and some states, follow these guidelines when defining Pipeline State Objects (PSOs) to optimize output:

- Avoid creating duplicate PSOs and do not rely on the driver to cache PSOs. This includes duplicate definitions of incorporated state like depth/stencil, blend, and so on.
- Define optimized shaders for PSOs instead of using combinations of generic shaders mixed with specialized shaders. For example: don't define a PSO with a vertex shader generating unnecessary output not consumed by pixel shader.
- Match depth-stencil format to operations needed for PSOs, for example, don't define a depth + stencil format if stencil will not be enabled.

5.1.3. Asynchronous Dispatch (3D + Compute)

While it is defined by the API to allow for asynchronous dispatch of 3D and compute, it is not recommended to structure algorithms with the expectation of latency hiding by executing 3D and compute functions simultaneously. Instead, structure algorithms to allow for minimal latency/dependency between 3D and compute capabilities.

5.1.4. Root Constants

Use root constants for cases where the constants are changing at a high frequency. Favor root constants over root descriptors, and favor root descriptors over descriptor tables when working with constants.

5.1.5. Compiled Shader Caching

Use the features provided in Direct3D 12 to natively cache compiled shaders to reduce CPU overhead upon startup.

5.1.6. Conservative Rasterization and Raster Order Views

Use hardware conservative rasterization for full-speed rasterization and use raster order views (ROVs) only when you need synchronization. Also favor ROVs over atomics.

5.1.7. Root Signature

Follow these guidelines to achieve maximum performance:

- Limit visibility flags to shader stages that will be bound.
- Actively use DENY flags where resources will not be referenced by a shader stage.
- Avoid generic root signature definitions where unnecessary descriptors are defined and not leveraged. Instead, optimize root signature definitions to the minimal set of descriptor tables needed.

5.1.8. Command Lists and Barriers

Bundle together as many command lists as possible when calling execute command lists as long as bundling does not starve the GPU. If the cost to create command lists is greater than the cost to execute the previous command list the GPU goes idle.

Try to group together barriers into a single call instead of many separate calls, and avoid the use of unnecessary barriers.

5.1.9. Resource Creation

Prefer committed resources. They have less padding, so the memory layout is more optimal.

5.2. DirectCompute Optimizations

This section covers additional compute-specific optimization guidelines to follow in addition to applicable guidelines from 3D. For more info on General Purpose GPU programming, please see The Compute Architecture of Intel® Processor Graphics Gen9.

5.2.1. Dispatch and Thread Occupancy

To achieve optimal dispatch and thread occupancy of EUs, there are three factors to balance when optimizing for dispatch and occupancy: thread group dimensions, Single Instruction Multiple Data (SIMD) width of execution, and shared local memory (SLM) allocation per thread group.

The combination of thread group dimensions and SIMD width of execution will control the dispatch of thread groups on sub-slices.

- All 6th gen Intel Core processors have 56 hardware threads per sub-slice. A few other processors that use Gen9 graphics have slightly fewer.
- Hardware thread utilization per thread group is determined by taking thread group dimensions (that is, dim x * dim y * dim z) and dividing by SIMD width of shader (SIMD16 when number of temporaries are ≤ 16, otherwise SIMD8).
    - Example: 16x16x1 thread group = 256 threads / 16 (SIMD width) = 16 hardware threads for each thread group. 56 total threads / 16 hardware threads per group allows for 3 thread groups to fully execute per sub-slice with one possible partial (depending on barriers defined).
    - SIMD32 exception overrides SIMD16 compilation when thread group dimension is too large to fit on a sub-slice. Example: 32x32x1 thread group = 1024 threads / 16 (SIMD width) = 64 hardware threads. As there are 56 hardware threads per sub-slice, you need to compile as SIMD32 to fit.

SLM allocation per sub-slice is 64 kb for up to the number of hardware threads supported by your sub-slice (see above). SLM allocation per thread group can limit thread group dispatch on a sub-slice. Example: A thread group defined with 32kb of SLM would limit to only two thread groups executing per sub-slice.

    - Example: A 16x16x1 thread group consuming 16 hardware threads (SIMD16) with 32 kb of SLM would only fill 32 of 56 hardware threads, leaving the sub-slice ~43 percent idle during execution.

5.2.2. Resource Access

When reading from or writing to surfaces in compute, avoid partial writes so that you can get maximum bandwidth through the graphics cache hierarchy. Partial writes are any accesses where the data written doesn't fully cover the 64 byte cache line. UAV writes commonly cause partial writes.

5.2.3. Shared Local Memory

Memory is shared across threads in the thread group. Follow these guidelines for maximum performance:

- Layout elements in structured buffers as SOA instead of AOS to improve caching and reduce unnecessary memory fetch of unused/unreferenced elements. Additionally, structure data in SOA where element granularity in the SOA is 4 bytes between elements and the total size of the array is not a multiple of 16.
    - Example: Split RGBA definition in a single structure into R, G, B, A where array length of each R, G, B, A is padded out to a non-multiple of 16.
    - SOA removes bank collisions when accessing a single channel (example: R). Padding out to a non-multiple of 16 removes bank collisions when accessing multiple channels (example: RGBA).
- If AOS is required, pad structure size to a prime number for structure sizes greater than 16 bytes.

5.3. Evaluating Pipeline Hotspots

You often need to understand hotspots in your graphics pipeline. In this section, you'll learn how to do this efficiently with Intel GPA Graphics Frame Analyzer (DirectX 9, 10, 11).

As of release 16.1, Intel GPA Graphics Frame Analyzer contains alpha support for Direct3D 12. Since the Direct3D 12 tool support is in alpha, the Direct3D 12 tools omit some hardware metrics in this version. Later releases of Intel GPA will show all metrics exposed by the platform and driver for all supported API versions.

The available hardware metrics may vary between hardware platforms, graphics drivers, and tool versions. The metrics shown here are available across all 6th gen Intel Core processors. The metrics in this section are generally exposed through the latest graphics drivers. As always, study performance with the latest driver and version of Intel GPA.

To get started with your analysis, capture a sample frame from your game, running on a 6th gen Intel Core processor. Capture the frame with the Intel GPA Heads-Up Display (HUD) or Intel GPA System Analyzer. The HUD has very low performance impact. If you prefer to use the Intel GPA System Analyzer, run it from a separate analysis system to avoid any

performance impact on your game. Open your frame with the Intel GPA Graphics Frame Analyzer version for your version of Direct3D.

Before doing a low-level analysis, look for large, obvious problems and eliminate them. Once that's done, you're ready to proceed.

5.3.1. Selecting Ergs to Study

To properly observe the graphics architecture, select enough ergs from the frame so their total execution time (check the GPU Core Clocks, cycles metric) is long enough to overcome any warm-up conditions. Select one or more ergs that take at least 200,000 cycles to ensure the metrics are as accurate as possible. If you want to study short draw calls where the cycle count is <200,000, select multiple back-to-back ergs as long as the following conditions are met:

- Total cycle count of all selected ergs is ≥ 200,000
- There are no state changes between the ergs (that is, shader changes, pipeline state, and so on). Texture and constant changes are exempt from this rule, unless the texture is a dynamically-generated surface.
- Ergs share the same render, depth, and stencil surface.

5.3.2. Performance Analysis with Hardware Metrics

Now that the frame analysis tools of Intel GPA expose hardware metrics, you can accurately triage the hotspots associated with major graphics architectural blocks. This section will guide you through studying your performance and give you implementation tips to avoid future performance problems.

Intel Processor Graphics perform deeply pipelined parallel execution of front-end (geometry transformation, rasterization, early depth/stencil, and so on) and back-end (pixel shading, sampling, color write, blend, and late depth/stencil) work within a single erg. Because of the deeply pipelined execution, hotspots from downstream architectural blocks will bubble up and stall upstream blocks. This can make it difficult to find the actual hotspot. To use metrics to find the primary hotspot, you must walk the pipeline in reverse order.

The metrics for your selected erg(s) appear in the Details tab (and metric names are underlined here, for clarity). With those metrics, follow these workflows to find your primary hotspot. For each step, see the following sections to discover if that metric shows a hotspot. If so, optimize that hotspot with the tips in that section; if not, continue through the workflow. Because they're a bit different, there are separate workflows for 3D (Figure 5-1) and GPGPU (Figure 5-2) workloads.

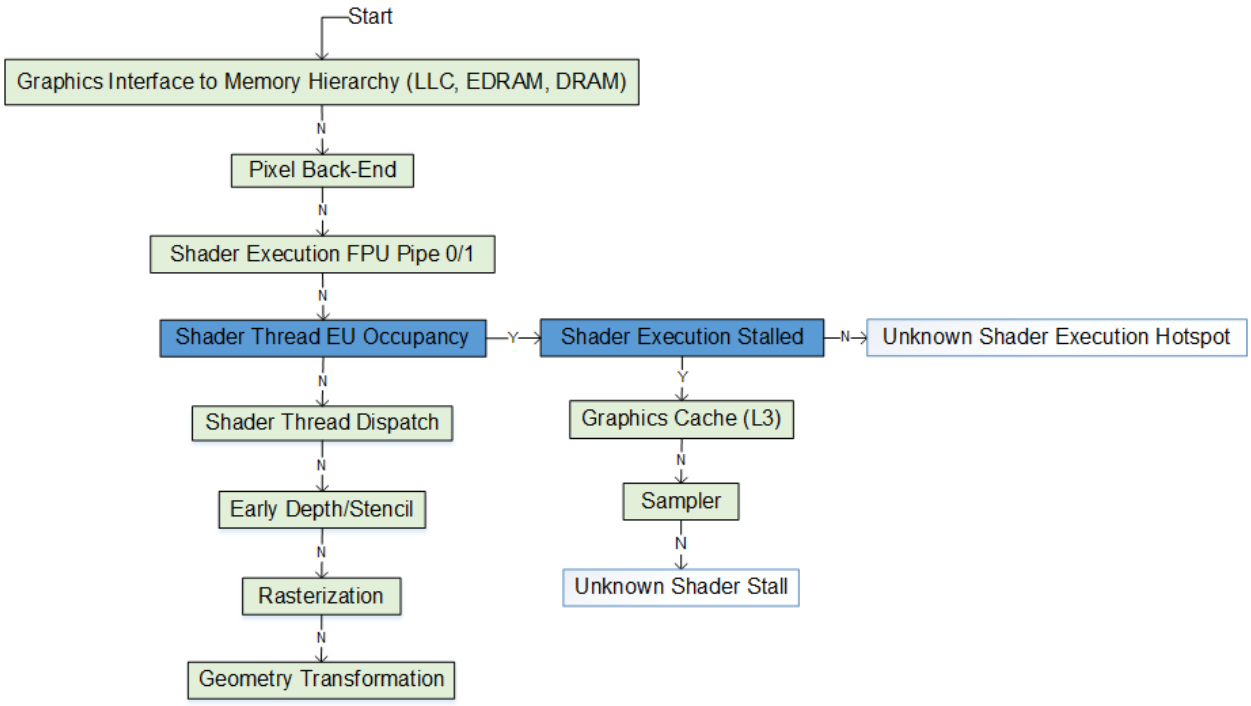


Figure 5-1: For 3D workloads, check each metric in order – is it the hotspot?

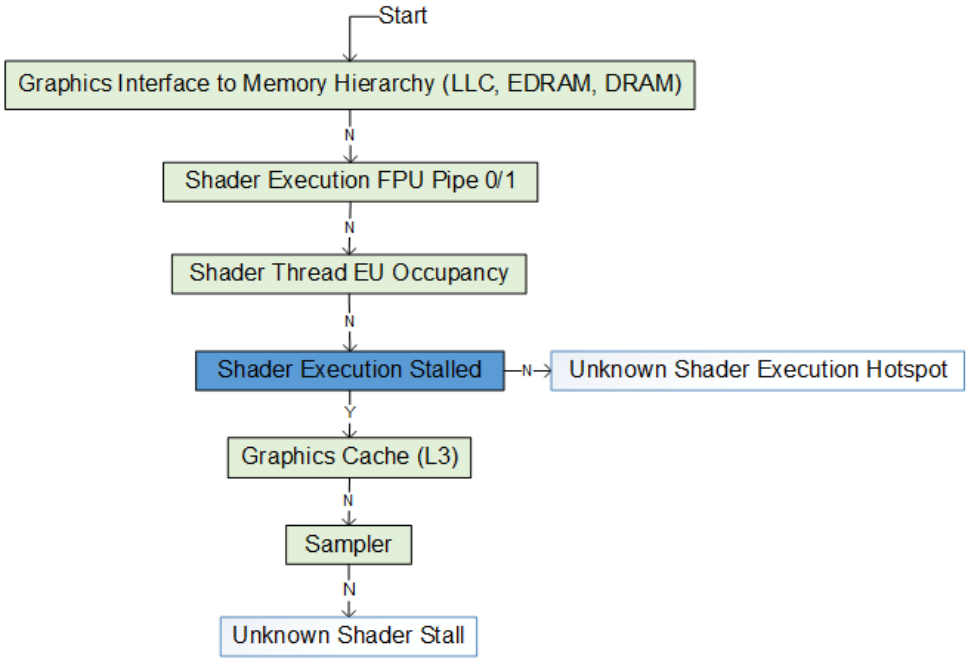


Figure 5-2: For GPGPU workloads, check each metric in order – is it the hotspot?

For more information on compute design and troubleshooting, see the Gen9 compute reference guide.

5.3.3. GTI - Graphics Interface to Memory Hierarchy (LLC, EDRAM, DRAM)

Metric Name	Description
GTI: SQ is full	Percentage of time that the graphics-to-memory interface is fully saturated for the erg(s) due to internal cache misses.

When GTI: SQ is full more than 80 percent of the time, this is probably a primary hotspot. Improve the memory access pattern of the erg(s) to reduce cache misses. Even if this isn't a primary hotspot, memory latency can make this a minor hotspot any time this is above 30 percent.

5.3.4. Pixel Back-End – Color Write and Post-Pixel Shader (PS) Operations

Metric Name	Description
GPU / 3D Pipe: Slice <N> PS Output Available	Percentage of time that color data is ready from pixel shading to be processed by the pixel back-end for slice 'N'.
GPU / 3D Pipe: Slice <N> Pixel Values Ready	Percentage of time that pixel data is ready in the pixel back-end (following post-PS operations) for color write.

There are two stages in the pixel back-end: Post-PS operations and color write. Post-PS operations occur after the color data is ready from pixel shading to the back-end, and can include blending, late depth/stencil, and so on. Following post-PS operations, the final color data is ready for write-out to the render target.

If the GPU / 3D Pipe: Slice <N> PS Output Available is greater than 80 percent, the pixel back-end is probably the primary hotspot, either in post-PS operations or color write. To check, compare Output Available with Pixel Values Ready. If Output Available is >10 percent more than Pixel Values Ready, the post-PS operations are the primary hotspot. Otherwise the primary hotspot is color write.

If there's a post-PS hotspot, adjust the algorithm or optimize the post-PS operations. To improve color write, improve the locality of writes (that is, geometry ordering, and so on) by using other render target formats, dimensions, and optimizations.

5.3.5. Shader Execution FPU Pipe 0/1

Metric Name	Description
EU Array / Pipes: EU FPU0 Pipe Active	Percentage of time the FPU pipe is active executing instructions.
EU Array / Pipes: EU FPU1 Pipe Active	Percentage of time the Extended Math (EM) pipe is active executing instructions.

When EU Array / Pipes: EU FPU0 Pipe Active or EU Array / Pipes: EU FPU1 Pipe Active are above 80 percent, it can indicate that the primary hotspot is due to the number of instructions per clock (IPC). If so, adjust shader algorithms to reduce unnecessary instructions or implement using more efficient instructions to improve IPC. For IPC-limited pixel shaders, ensure maximum throughput by limiting shader temporary registers to ≤ 16.

5.3.6. Shader Thread EU Occupancy

Metric Name	Description
EU Array: EU Thread Occupancy	Percentage of time that all EU threads were occupied with shader threads.

For GPGPU cases, when EU Array: EU Thread Occupancy is below 80 percent, it can indicate a dispatch issue (see GPGPU section “Dispatch and Thread Occupancy” for tips on improving dispatch), and the kernel may need to be adjusted for more optimal dispatch.

For 3D cases, low occupancy means the EUs are starved of shader threads by a unit upstream of thread dispatch.

5.3.7. 3D Thread Dispatch

Metric Name	Description
GPU / Rasterizer / Early Depth Test: Slice <N> Post-Early Z Pixel Data Ready	Percentage of time that early depth/stencil had pixel data ready for dispatch.

When GPU / Rasterizer / Early Depth Test: Slice <N> Post-Early Z Pixel Data Ready is greater than EU Array: EU Thread Occupancy, the bottleneck is in the shader dispatch logic. Reduce the shader’s thread payload (e.g. register usage and so on) to improve thread dispatch.

5.3.8. Early Depth/Stencil

Metric Name	Description
GPU / Rasterizer: Slice <N> Rasterizer Output Ready	Percentage of time that input was available for early depth/stencil evaluation from rasterization unit.

When GPU / Rasterizer: Slice <N> Rasterizer Output Ready is above 80 percent, it indicates that early depth/stencil is sufficiently fed from rasterization. A drop between GPU / Rasterizer: Slice <N> Rasterizer Output Ready and GPU / Rasterizer / Early Depth Test: Slice <N> Post-Early Z Pixel Data Ready of > 10 percent indicates an early depth/stencil could be the primary hotspot. Evaluating other stencil operations, improving geometry (that is, reducing spatial overlap), and improving memory locality can all improve performance.

5.3.9. Rasterization

Metric Name	Description
GPU / Rasterizer : Slice <N> Rasterizer Input Available	Percentage of time that input was available to the rasterizer from geometry transformation (VS-GS + clip/setup).

If GPU / Rasterizer: Slice <N> Rasterizer Input Available is below 80 percent, the rasterizer does not have enough data from transformation. If Input Available is >10 percent more than Output Ready, simplify or reduce the amount of geometry that must be rasterized. (Example: Fewer vertices, better clipping/culling, and so on).

5.3.10. Geometry Transformation

Reaching this point in the flow indicates that geometry transformation is taking up a significant amount of execution time, so further optimization is needed to reduce the cost of shading, clip, and setup as indicated by the low output from rasterization.

Possible optimizations are any shader optimizations for VS->GS, reducing the number of off-screen polygons generated from shading, and reducing unnecessary state changes between draws.

5.3.11. Shader Execution Stalled

Metric Name	Description
EU Array : EU Stall	Percentage of time that the shader threads were stalled.

When EU Array: EU Stall is above 10 percent, the stall could come from internal dependencies or from memory accesses initiated by the shader and the L3 and Sampler need to be evaluated. Otherwise the execution hotspot is unknown and needs further debugging.

5.3.12. Unknown Shader Execution Hotspot

When you hit this point and the stall is low, but the occupancy is high, it indicates that there is some EU execution inefficiency associated with the workload. Optimize the shader code itself to improve IPC.

5.3.13. Graphics Cache (L3)

Metric Name	Description
GTI / L3 : Slice <N> L3 Bank <M> Active	Percentage of time that L3 bank 'M' on slice 'N' is servicing memory requests.
GTI / L3 : Slice <N> L3 Bank <M> Stalled	Percentage of time that L3 bank 'M' on slice 'N' has a memory request but cannot service.

When GTI / L3: Slice <N> L3 Bank <M> Active is above 80 percent in conjunction with a GTI / L3: Slice <N> L3 Bank <M> Stalled value greater than 5 percent, it can indicate a hotspot on L3. Several clients interface with L3 for memory requests, and when a hotspot is seen here, improve memory access patterns to SLM, UAVs, texture, and constants.

5.3.14. Sampler

Metric Name	Description
GPU / Sampler : Slice <N>Subslice<M> Sampler Input Available	Percentage of time there is input from the EUs on slice 'N' and subslice 'M' to the sampler.
GPU / Sampler : Slice <N>Subslice<M> Sampler Output Ready	Percentage of time there is output from the sampler to EUs on slice 'N' and subslice 'M'.

When Input Available is >5 percent greater than Output Ready, the sampler is not returning data back to the EUs as fast as it is being requested. The sampler is probably the hotspot. This comparison only indicates a significant hotspot when the samplers are relatively busy.

The sampler has multiple paths that can hotspot internally. Adjust the sampling access pattern, filtering mode, surface type/format, and the number of surfaces sampled to speed up the sampler.

5.3.15. Unknown Shader Stall

Indicates that while a stall was seen during shader execution, the root cause is not clear. Further debugging will be required to determine it. Reduce internal dependencies within the shader code and improve memory access pattern for all memory operations accessed during execution.

6. OPENGL

6th gen Intel Core processors support OpenGL version 4.4, the most widely adopted 2D and 3D graphics API in the industry. This section will lead you through some of the main performance points for OpenGL games.

6.1. OpenGL Shaders

Key points to keep in mind when working with OpenGL shaders include:

- When writing GLSL shaders, use and pass only what is needed, and declare only the resources that will be used. This includes samplers, uniforms, UBOs, SSBOs, and images. Used attributes should only be passed between stages.

- Use built-in shader functions rather than other equivalents. Use compile-time constants to help the compiler generate of optimal code, including loop iterations, indexing buffers, inputs and outputs.

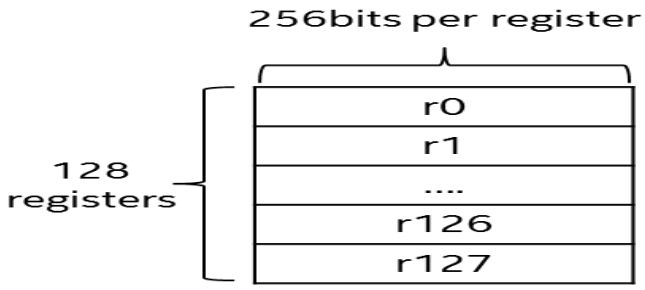
- Use medium precision for OpenGL ES contexts, for improved performance.

- Each thread in the Gen9 architecture has 128 registers and each register is 8x32 bits. For shaders other than fragment shaders, keep in mind that each register represents a single channel of vec4 variable for 8 data streams.

- Fragment shaders can be launched by the driver in SIMD8, SIMD16 or SIMD32 modes. Depending on the mode, a single channel of vec4 variable can occupy a single register, or two or four registers for 8, 16 or 32 pixels.

- Registers are used for the delivery of initial hardware-dependent payload as well as uniforms, which can consume up to 64 registers. A bigger number of variables and uniforms in shaders will increase register pressure which can lead to saving/restoring registers from memory. This, in turn, can have a negative impact on the performance of your shaders.

  Limit the number of uniforms in the default block, to minimize register pressure, it's good to limit the number of uniforms in the default block. However, the default uniform block is still the best method of constant data delivery over UBOs for smaller and more frequently changed constants.



6-1 Textures provide a better read than images

- Don't use compute shader group sizes larger than 256 (and recall that larger groups will require a wider SIMD). They result in higher register pressure and may lead to EU underutilization

6.2. Textures

Optimizing texture accesses is an important aspect of OpenGL applications.

- Always use mipmaps, to minimize the memory bandwidth used for texturing.
- Textures with power-of-two dimensions will have better performance in general.
- When uploading textures, provide textures in a format that is the same as the internal format, to avoid implicit conversions in the graphics driver.

6.3. Images

When using OpenGL images, remember that textures usually provide better read performance than images. Some of the image formats required by OpenGL specifications are not natively supported by hardware devices, and will therefore need to be unpacked in the shaders.

6.4. Shader Storage Buffer Objects

Shader Storage Buffer Objects provide a universal mechanism for providing input/output both to and from shaders. Since they are flexible, they can also be used to fetch vertex data based on gl_VertexId. Use Vertex Arrays where possible, as they usually offer better performance.

6.5. Atomic Counter Buffers

Atomic Counter Buffers and atomic counters are internally implemented as Shader Storage Buffer Objects atomic operations. Therefore there are no real performance benefits to be utilized from using Atomic Counter Buffers.

6.6. Frame Buffer Object (FBO)

For FBOs, consider a few important things:

- When switching color/depth/stencil attachments, try to use dedicated framebuffer objects for each set in use. Switching the entire frame buffer object is more efficient than switching individual attachments one at a time.
- Don't clear buffers that are never used.
- Skip color buffer clears if all pixels are to be rendered. In many cases, clearing just the depth buffer should be sufficient.
- Limit functions that switch color/depth/stencil attachments between rendering and sampling. They are expensive operations.

6.7. State Changes

State changes can reduce your performance.

- Minimize state changes. Group similar draw calls together.
- For texturing, use texture arrays. They can provide a much more efficient way of switching textures compared to reconfiguration of texture units.
- By using instancing and differentiating, rendering based on gl_InstanceID can also provide a high-performance alternative to rendering similar objects when compared to reconfiguring the pipeline for each object individually.
- Use the default uniform block rather than uniform buffer objects for small constant data that changes frequently.
- Limit functions that switch frame buffer objects and GLSL programs. They are the most expensive driver operations.
- Avoid redundant state changes.

- In particular, do not bind a state to "default" values between drawcalls, as not all of these state changes can be optimized in the driver:

  glBindTexture(GL_TEXTURE_2D, tex1);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 2);
  ~~glBindTexture(GL_TEXTURE_2D, 0);~~

  glBindTesxture(GL_TEXTURE_2D, tex2);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 2);
  ~~glBindTexture(GL_TEXTURE_2D, 0);~~

6.8. Avoid CPU/GPU Synchronization

Synchronization between the CPU and GPU can cause stalls.

- Avoid calls that synchronize between the CPU and GPU, for example, glReadPixels or glFinish.

- Use glFlush with caution. Use sync objects to achieve synchronization between contexts.

- Avoid updating resources that are used by the GPU. Static resources should be created at the startup of the application and not modified later. Whenever possible, create vertex buffer objects as static (GL_STATIC_DRAW).

- Avoid updating resources that are used by the GPU. For example: Don't call glBufferSubData/glTexImage if there are queued commands that access a given VBO/texture. Limit the chances of simultaneous read/write access to resources.

- For creation of buffers and textures, use immutable versions of API calls: glBufferStorage() and glTexStorage*().

- One way you can update buffers and avoid GPU/CPU synchronization issues is to create a pool of bigger buffers with glBufferStorage() and permanently map them with the glMapBuffer() function call. The application can then iterate over individual buffers with increasing offsets, providing new chunks of data.

- For uniform buffer objects, use glBindBufferRange()to bind new chunks of data at the current offset. For vertex buffer objects, access newly copied chunks of data with firstIndex (for glDrawArrays) or indices/baseVertex parameters (for glDrawElements/BaseVertex). Increase the initial number of pools if the oldest buffer submitted for GPU consumption is still in use. Monitor the progress of the GPU with accessing the data from the buffers with sync objects.

6.9. Anti-Aliasing Options

OpenGL drivers and 6th gen Intel Core processors support standard MSAA functionality. As described above, you can also use Conservative Morphological Anti-Aliasing (CMAA), an Intel-provided alternative to MSAA.

- With CMAA, image quality will prove equal or better when compared to an FXAA/MLAA solution. When performance counts, consider CMAA a more efficient replacement of MSAAx4.

- The OpenGL extension INTEL_framebuffer_CMAA is simple to use.

7. POWER

Mobile and ultra-mobile computing are ubiquitous. As a result, battery life, device temperature, and power-limited performance have become significant issues. As manufacturing processes continue to shrink and improve, we see improved performance per-watt characteristics of CPUs and processor graphics. However, there are many ways that software can reduce power use on mobile devices, as well as improve power efficiency.

In the following sections, you'll find a number of key insights and recommendations illustrating how to best recognize these performance gains.

7.1. Idle and Active Power

Processors execute in different power states, known as P-states and C-states. C-states are essentially idle states that minimize power draw by progressively shutting down more and more of the processor. P-states are performance states where the processor will consume progressively more power and run faster at a higher frequency.

These power states define how much the processor is sleeping, and how it distributes available power when active. Power states can change very quickly, so sleep states are relevant to most applications that do not consume all the power available, including real-time applications.

When you optimize applications, try to save power in two different ways:

- Increase the amount of idle time your application uses where it makes sense.
- Improve overall power usage and balance under active use.

You can determine the power state behavior of your application by measuring how much time it spends in each state. Since each state consumes a different amount of power, you'll get a picture over time of your app's overall power use.

7.2. Analysis Tips

To start, begin by measuring your app's baseline power usage in multiple cases, for example:

- At near idle, for example, in the UI during videos
- Under an average load, for example, during an average scene with average effects

The worst-case load may not occur where you think. For example, we have seen very high frame rates (1000 FPS) during cut scene playback in certain apps, a situation that can cause the GPU and CPU to use unnecessary power.

As you study your application, try a few of these tips.

- Measure how long (on average) your application can run on battery power, and compare its performance with other, similar apps. Measuring power consumption regularly will inform you if any recent changes caused your app to utilize more power.

- Intel's Battery Life Analyzer (BLA) is a good (Windows-only) tool for this work. For more information, please see this article showcasing BLA and how it can collect high-level data and analyze an app's power use. If the data BLA provides shows that you have issues residing in the wrong C-states for too long, it's time to look deeper.

- If your app is reported as deficient or there are unexpected wakeups, start optimizing for power. To do so, you'll want to look at the Windows Performance Analyzer (WPA) tool, which showcases workflow using WPA for CPU analysis.

- Intel VTune Amplifier XE is also useful to get power call stacks, since it can identify the cause of the wake-up.

Use the data gained through these methods to reduce or consolidate wake-ups, thus remaining in a lower power state longer.

7.3. Investigating Idle Power

As you study power at near idle, watch for very high frame rates.

If your app has high frame rates at near idle power (for example, during cut scenes, menus, or other low-GPU-intensive parts), remember that these parts of your app will look fine if you lock the Present interval to a 60Hz display refresh rate (or clamp your frame rate lower, to 30 FPS).

Watch for these behaviors in menus, loading screens, and other low-GPU-intensive parts of games and scale accordingly to minimize power consumption.

7.4. Active Power and Speed Shift

While in active states, the processor and the OS jointly decide frequencies for various parts of the system (CPUs, GPU, and memory ring, in particular). The 6th gen Intel Core processors add more interaction between the OS and the processor(s) to respond more efficiently and quickly to changes in power demand - a process referred to as Speed Shift.

The way these frequencies are chosen is beyond the scope of this document. The system will balance the frequencies based on activity and will increase frequency (and thus consumed power) where it is needed most. As a result, a mostly active workload may have its GPU and CPU balance frequencies based on power consumption.

- Reduce the amount of work done on one side (for example, CPU) to free power for the other side. This can result in better overall performance, even when the other side was the primary performance bottleneck. See slides 30-40 of this presentation for additional details on how to achieve these performance gains.

Tools like Intel Power Gadget can also help you see the frequencies of each clock domain in real-time. By running this tool, you can monitor the frequencies of different sub-systems on target devices.

- You can tell that your app's power distribution is getting balanced when the primary performance bottleneck is not running at full frequency but power consumption is reaching the maximum limits available.

7.5. When and How to Reduce Activity

There are times when the user explicitly requests trading performance for battery life, and there are things you can do to more effectively meet these demands. There are also patterns in application usage that always consume extra power for little return, patterns which you can more effectively address to handle overall power usage.

In the next sections, you'll see some issues to watch for when trying to reduce power consumption.

7.5.1. Scale Settings to Match System Power Settings and Power Profile

It was once necessary to poll for power settings and profile (for example, using GetSystemPowerStatus()). Since the launch of Windows Vista*, Windows supports asynchronous power notification APIs.

- Use RegisterPowerSettingNotification() with the appropriate GUID to track changes.

- Scale your app's settings and behavior based on the power profile and whether your device is plugged in to power. Scale the resolution, reduce the max frame rate to a cap and/or reduce quality settings.

- If you cap the frame rate, you can use the V-Sync mechanism in DirectX or OpenGL. You can also manage the frame rate and resolution yourself as well. The dynamic resolution rendering sample shows how to adjust frame resolution to maintain a frame rate.

7.5.2. Run as Slow as You Can, While Remaining Responsive

If you run as slow as you can (but still remain responsive) then you can save power and extend battery life.

- Detect when you are in a power-managed mode and limit frame rate. This will prolong battery life and also allow your system to run cooler. Running at 30 Hz instead of 60 Hz can save significant power.

- Provide a way to disable the frame rate limit, for benchmarking. Warn players that they will use their battery quickly. You should also want to let the player control the frame rate cap.

- Use off-screen buffers and do smart compositing for in-game user interfaces (which are often limited to small panels for displays like health, power-ups, and so on). Since user interfaces usually change much more slowly than in-game scenes, there's no need to change them at the same rate as the game frame. Here again, Dynamic Resolution Rendering (DRR) may be useful, in helping you decouple UI rendering from main scene rendering.

7.5.3. Manage Timers and Respect System Idle, Avoid Tight Polling Loops

There are several other related points to watch.

- Reduce your app's reliance on high-resolution periodic timers.

- Avoid Sleep() calls in tight loops. Use Wait*() APIs instead. Sleep() or any other busy-wait API can cause the OS to keep the machine from being in the Idle state. Intel's Mobile Platform Idle Optimization presentation offers an extensive rundown of which APIs to use and avoid.

- Avoid tight polling loops. If you have a polling architecture that uses a tight loop, convert it to an event-driven architecture. If you must poll, use the largest polling interval possible.

- Avoid busy-wait calls. In Figure 7-1, the Direct3D query will be called repeatedly, causing unnecessary power use. There's no way for the OS or the power management hardware to detect that the code does nothing useful. You can avoid this with some basic planning and design tweaks.

```
HRESULT res;
IDirect3DQuery9 *pQuery;
// create a query
res = pDevice->CreateQuery(.., &pQuery);
...
// busy-wait for query data
while ( (res = pQuery->GetData(... , 0)) == S_FALSE);
```

Figure 7-1: Busy wait loop – don't do this!

7.5.4. Multithread Sensibly

Balanced threading offers performance benefits, but you need to consider how it operates alongside the GPU, as imbalanced threading can also result in lower performance and reduced power efficiency. Avoid affinitizing threads, so the OS can schedule threads directly. If you must, provide hints using SetIdealProcessor().

8. SAMPLES AND REFERENCES

Intel regularly releases to the developer community code samples covering a variety of topics. For the most up-to-date samples and links, please see the following resources:

- Intel Developer Zone
- GitHub Intel Repository

Below are samples that may also be of interest to developers targeting current Intel systems. Click on any subheading to access the desired information.

GPU Detect

This DirectX sample demonstrates how to get the vendor and ID from the GPU. For Intel Processor Graphics, the sample also demonstrates a default graphics quality preset (low, medium, or high), support for DirectX 9 and DirectX 11 extensions, and the recommended method for querying the amount of video memory. If supported by the hardware and driver, it also shows the recommended method for querying the minimum and maximum frequencies.

The sample uses a config file that lists many of Intel's GPUs by vendor ID and device ID, along with a suggested graphics quality level for each device. To maximize performance, you should test some representative devices with your application and decide which quality level is right for each.

Fast ISPC Texture Compression

This sample performs high-quality BC7, BC6H, ETC1 and ASTC compression on the CPU using the Intel® SPMD Program Compiler (Intel® SPC) to exploit SIMD instruction sets.

Asteroids and DirectX 12

An example of how to use the DirectX 12 graphics API to achieve performance and power benefits over previous APIs.

Multi-Adapter Support with DirectX 12

This sample shows how to implement an explicit multi-adapter application using DirectX 12. Intel's integrated GPU and a discrete NVIDIA GPU are used to share the workload of ray-tracing a scene. The parallel use of both GPUs allows for an increase in performance and for more complex workloads.

Early-Z Rejection

This sample demonstrates two ways to take advantage of early Z rejection: Front to back rendering and z pre-pass. With a z pre-pass the first pass populates the Z buffer with depth values from all opaque geometry. A null pixel shader is used, and the color buffer is not updated. For the second pass, the geometry is resubmitted with Z writes disabled, but Z testing on, and full vertex and pixel shading is performed.

Other code samples that you may find useful include the following:

- Sparse Procedural Volumetric Rendering
- Dynamic Resolution Rendering
- Software Occlusion Culling
- Sample Distribution Shadow Maps
- Programmable Blend with Pixel Shader Ordering
- Adaptive Volumetric Shadow Maps
- Adaptive Transparency Paper
- CPU Texture Compositing with InstantAccess
- Dynamic Resolution Rendering
- Conservative Morphological Anti-Aliasing Article and Sample
- OpenGL Fragment Shader Ordering Extension
- OpenGL Map Texture Extension

9. CONCLUSION

The 6th gen Intel Core processors provide superior CPU and GPU performance, and a significant number of features and enhancements over previous editions that enable even higher-quality visual and computing output. Follow the hints, tips, and instructions outlined in this guide, and you'll quickly and easily be able to optimize applications to take full advantage of these features and capabilities.

We hope that the information and best practices outlined here will help you build and optimize your Direct3D and OpenGL games more easily and effectively. To learn more, or just stop by and let us know what you think, we invite you to visit the Intel Game Development site. Now go out and build something awesome - we can't wait to see all your amazing creations!

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.