Qualcomm Technologies, Inc.

Game Developer Guide

80-78185-2 AH

December 9, 2024

© Qualcomm Technologies, Inc. and/or its subsidiaries. All rights reserved.

Contents

0.1 Introduction 3
0.2 Guides 3
0.3 Components 133
0.4 Tutorials 142

0.1 Introduction

0.2 Guides

How to optimize for Snapdragon platforms:

Qualcomm® Adreno™ GPU

Adreno GPUs are integrated in the all-in-one design of Qualcomm® Snapdragon™ processors to provide sophisticated rendering capabilities to enable the latest games, user interfaces, and web technologies present in mobile devices today. They have been designed purposely for mobile APIs and device constraints, with an emphasis on performance and efficient power use.

This guide outlines the various technologies and subsystems provided by the Adreno GPU to support the graphics developer. Best practices, along with articles going into some technical detail, can be found in the Adreno GPU on Mobile: Best Practices section.

This document strives to cover a breadth of topics relevant to various Adreno GPUs. As the architecture evolves, new functionalities are added to the GPU. Some sections of this guide will only be relevant to the Adreno GPUs that support the given features.

When relevant, the Adreno GPU series in which a feature is present is described. If not specified, it can be assumed such functionality is present on most or all Adreno GPUs.

Overview

Introduction to Snapdragon Adreno™

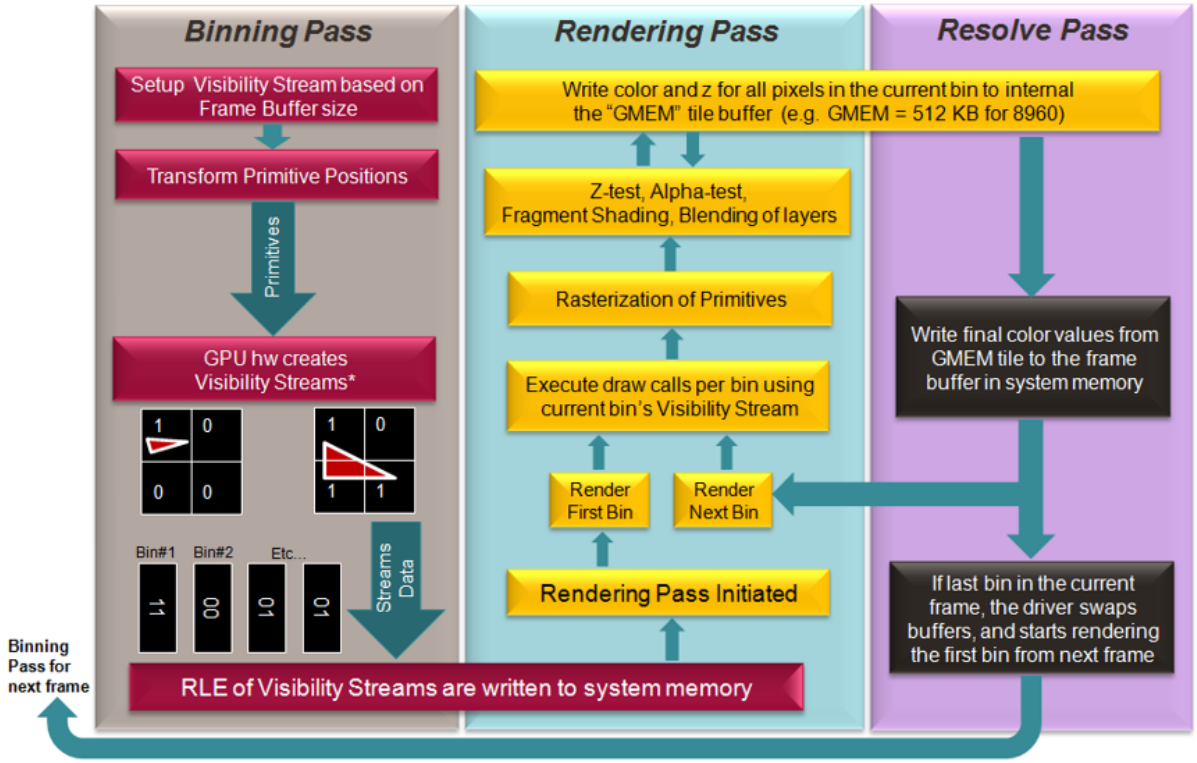
The 3D rendering process is a compute-intensive activity. Screen resolutions are growing larger, with some about to reach Ultra HD resolution. This means that GPUs need to rasterize more fragments within the same fixed time period. Assuming a target frame rate of 30 fps, a game must not spend more than 33.3 ms on a single frame. If it does, then the number of screen updates per second will drop, and it will become more difficult for users to immerse themselves fully into the game.

Tile-based Rendering

To optimize rendering for low-power and memory-bandwidth-limited devices, Adreno GPUs use a tiled-based rendering architecture. This rendering mechanism breaks the scene frame buffer into small rectangular regions for rendering. Region sizes are automatically determined (although the developer can influence the number of bins – which also defines bin size) so that they are optimally rendered using local, low-latency memory on the GPU (referred to as GMEM), rather than using a bandwidth-limited bus to system memory.

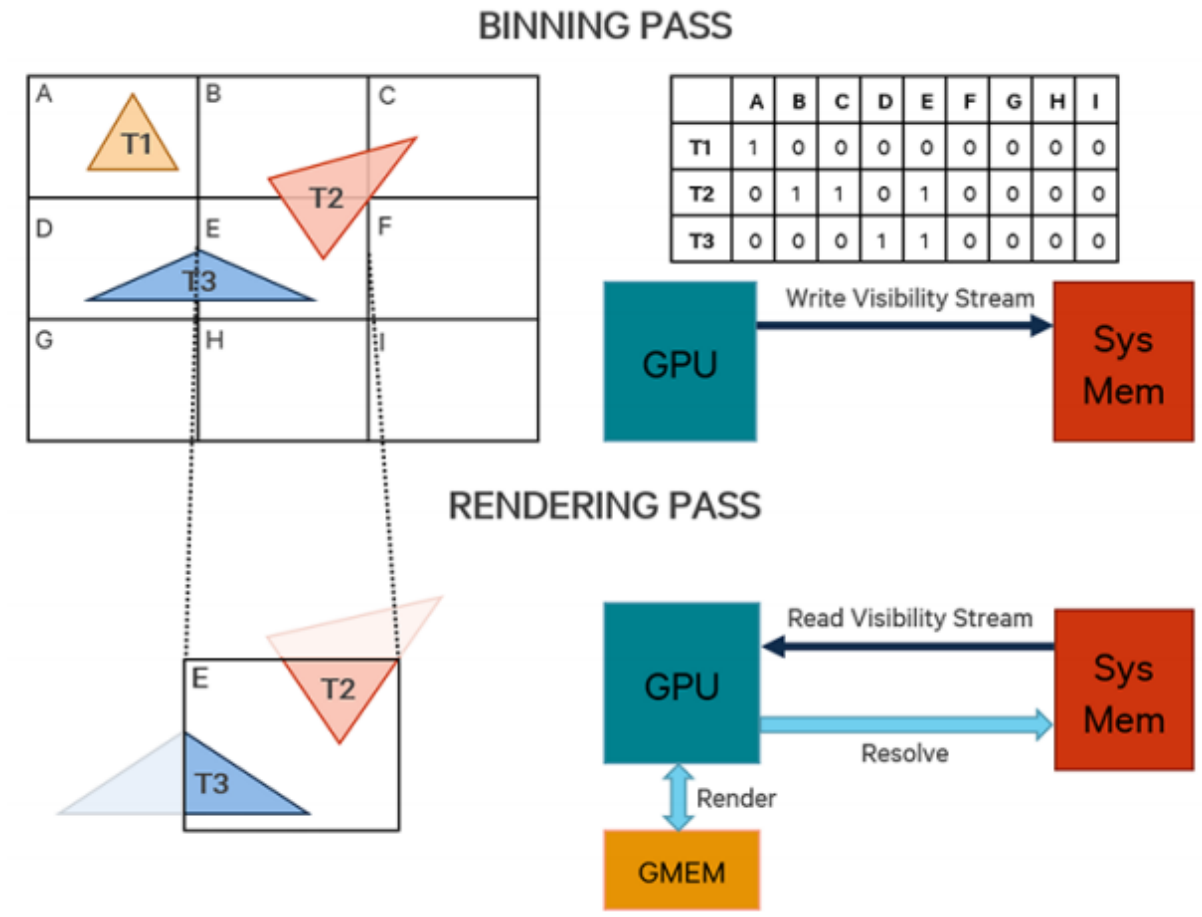
The Adreno GPU divides a frame into bins and renders them one at a time. During rendering it uses on-chip high performance Graphics Memory (GMEM) to avoid the cost of going to system memory.

In the image below, you see the two passes that are performed over the graphic primitives (Binning and Rendering). In this example there are three triangles that will be rendered in the frame buffer. The Binning Pass marks which bins a triangle is visible in a visibility stream. This stream is stored to system memory.



In the Rendering Pass, only the visible primitives for each tile to be rendered are processed by reading the visibility stream. Using GMEM as a local color and Z-buffer, the primitives are rendered. Once the rendering is complete for the tile, the GMEM color contents are sent back (resolved) to system memory. This process is repeated for all bins.

> Vulkan's Renderpass feature is advantageous for tiling architectures like Adreno, because multiple rendering passes can be done in GMEM. This ultimately minimizes costly resolve operations.



FlexRender™ technology (Hybrid Deferred and Direct Rendering mode)

FlexRender allows Adreno GPUs to switch between tile-based binned rendering and direct rendering to a frame buffer – since depending on workload, direct or binned rendering may provide superior performance. The driver and GPU analyze the rendering parameters for a given render target and selects the mode automatically.

The driver heuristics that determine which mode are not exposed to the developer, but generally these scenarios trigger direct mode.

Concurrent Binning

Concurrent binning was introduced by the Snapdragon A7x series. For each render pass, if no dependencies prohibit it, concurrent binning allows the binning pass to run asynchronously before vertex shader execution.

Position-Only Vertex Shader

As with (traditional) synchronous binning, the compiler generates a position-only vertex shader (a version of the vertex shader that the compiler attempts to simplify to only the instructions that affect vertex positions) that will be used to determine which bin (or bins, in the event a triangle overlaps multiple bins) to place each triangle.

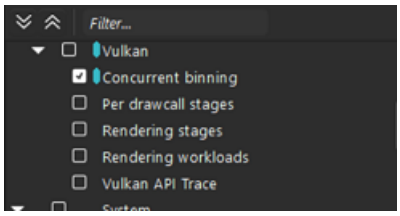
With concurrent binning, the execution of the position-only vertex shader happens asynchronously with other operations.

How to use it

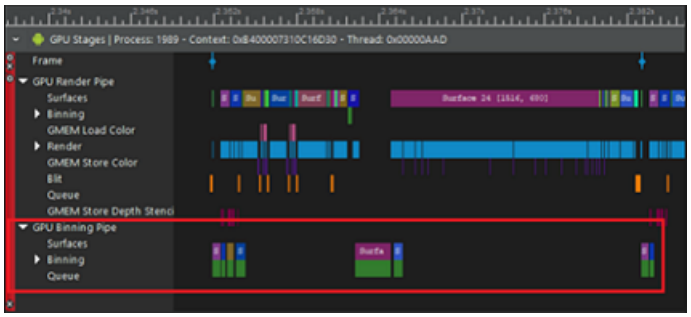
No additional steps are required to activate concurrent binning – but care should be taken not to prevent it.

Profiling

The Snapdragon Profiler can help you determine when concurrent binning is active. When capturing a trace, include the concurrent binning property:



Concurrent binning information will be visible under your Render Pipe as Binning Pipe:



Note that synchronous binning is located on the Render Pipe, and that on some occasions the driver will attempt to start concurrent binning when GPR usage is low – even with sequential dependencies.

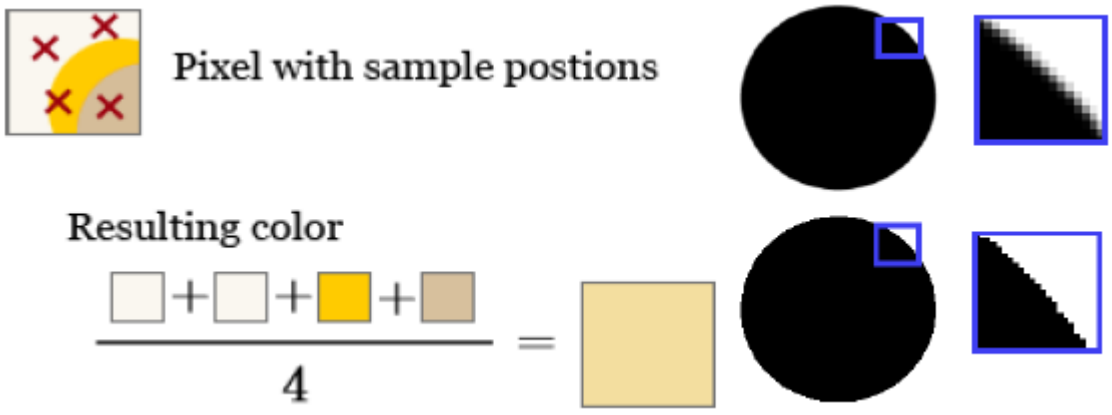
SDP will display concurrent binning parameters on a different section (underneath GPU Stages – which is attained after capturing a Trace with Vulkan|OpenGL ES->Rendering Stages checked) for each captured option on your trace capture (which can include things like vertex shader usage, cache misses, texture fetches, etc).

MSAA: Multisample anti-aliasing

Anti-aliasing is an important technique for improving the quality of generated images.

It reduces the visual artifacts of rendering into discrete pixels. Among the various techniques for reducing aliasing effects, multisample anti-aliasing (MSAA) is efficiently supported by Adreno GPUs.

As shown in the following image, multisampling divides every pixel into a set of samples, each of which is treated like a “mini-pixel” during rasterization. Each sample has its own color, depth, and stencil value. Those values are preserved until the image is ready for display. When it is time to compose the final image, the samples are resolved into the final pixel color.



For small-pixel form factors (like mobile phones), some content may get little to no visual benefit from anti-aliasing.

Snapdragon hardware can perform MSAA efficiently by performing it in tile memory – with no unnecessary system memory or blit operations: OpenGL ES MSAA Sample Code

Render Surfaces

sRGB textures and render targets

sRGB is a standard RGB color space created cooperatively by Hewlett-Packard and Microsoft in 1996 for use on monitors, printers, and the Internet. Smartphone and tablet displays today also assume sRGB (nonlinear) color space. sRGB provides the best viewing experience with correct colors, and ensures that the color space for render targets and textures match the color space for the display.

API support varies, as does the efficiency of different pixel formats.

For more information, see Qualcomm TrueHDR.

Universal bandwidth compression

Universal bandwidth compression (UBWC) is supported by all GPUs since A5x. UBWC is a unique predictive bandwith compression scheme that improves effective throughput to system memory. By minimizing the bandwidth of data, significant power savings can be achieved.

UBWC works across many components in Snapdragon processors including GPU, Display, Video, and Camera. The compression supports YUV and RGB formats, and reduces memory bottlenecks. Snapdragon Profiler typically shows surfaces as being encoded as “Optimal” (UBWC) or “Linear” (much less performant, but laid out like a C-array rather than with our proprietary compression scheme).

Graphics APIs must be used correctly to maximize the use of UBWC – for example, in Vulkan VK_IMAGE_TILING_LINEAR and VK_IMAGE_TILING_OPTIMAL generally map to “Linear” and “Optimal” as expected.

Percentage Closer Filtering for depth textures

Adreno GPUs have hardware support for the OpenGL ES 3.0 and Vulkan Percentage Closer Filtering (PCF) feature.

A hardware bilinear sample is fetched from the shadow map texture, which alleviates the aliasing problems that can be seen with real-time shadow mapping.

LRZ, Early-Z and Fast-Z

Low Resolution Z pass

A Low Resolution Z (LRZ) pass was added with Adreno 5X (A5X). This pass is also referred to as draw order independent depth rejection.

During the binning pass, a low resolution Z-buffer is constructed, and can reject LRZ-tile wide contributions to boost binning performance. This LRZ is then used during the rendering pass to reject pixels efficiently before testing against the full resolution Z-buffer.

LRZ costs negligible CPU resources. The binning process is performed in the GPU and the visibility streams generated (which dictate which drawcalls affect which bins) are placed in system memory for the GPU to be consumed in the render pass.

This feature reduces memory access and the number of rendered primitives, lessening an application's need to draw front to back – and thereby usually improving frame rate.

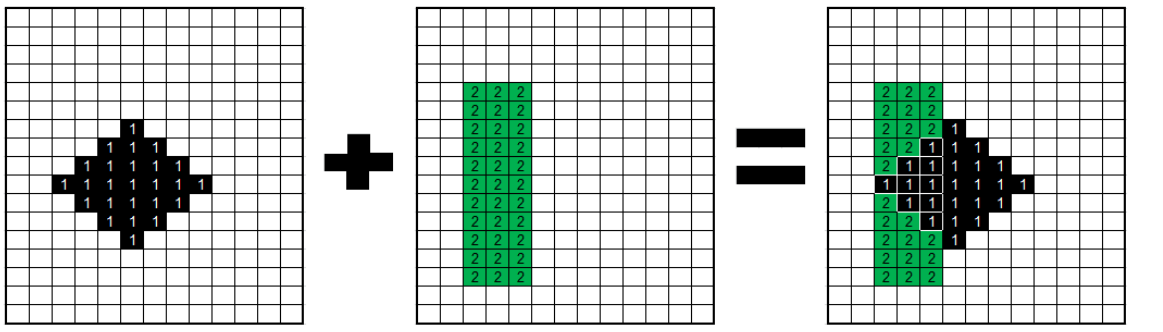
LRZ cannot be directly used by the developer – not through Vulkan, DirectX, OpenGL ES, or any other public API – but the developer can influence the likelihood of LRZ being executed.

Early Z rejection

For pixels that pass the LRZ test (if active), Early Z rejection mode provides a fast occlusion method with the rejection of unwanted render passes for objects that are not visible (hidden) from the view position.

The image below shows a color buffer represented as a grid, and each block represented as a pixel. The rendered pixel area on this grid is colored black. The Z-buffer value for these rendered black pixels is 1.

If you are trying to render a new primitive onto the same pixels of the existing color buffer that has the Z-buffer value of 2 (second grid with green blocks), the conflicting pixels in this new primitive will be rejected (third grid representing the final color buffer).



Color buffer with existing rendered pixels with Z value 1

New pixels to render with Z value 2

Highlighted conflicting Pixels are rejected by Early Z rejection feature saving efforts to coloring conflicting pixels

To get the maximum benefit of this feature, ensure Early Z rejection is executed by the driver. Also, we recommend drawing a scene with primitives sorted out from front-to-back; i.e., near-to-far – this ensures that the Z-reject rate is higher for the far primitives, which typically optimizes applications with high-depth complexity.

Fast-Z

The GPU has a special mode that writes Z-only pixels at twice the normal rate – this can improve performance when, for example, an application renders to a shadow map, or a z-prepass. The developer can influence the driver's likelihood of activating Fast-Z.

Texture Features

Texture compression

Texture compression can significantly improve the performance and load time of graphics applications because it reduces texture memory and bus bandwidth use.

Important compression texture formats supported by Adreno GPUs include (beginning with, generally, the best choice):

- **ASTC** - Texture compression format supported in OpenGL ES (3.0 and later) and Vulkan that allows compression to use a variable block size, and supports sRGB texture data more efficiently than the alternatives
- **ETC2** - Standard OpenGL ES 3.0 and Vulkan texture compression format supporting R, RG, RGB, and RGBA component layouts, as well as sRGB texture data
- **ATC** - Proprietary Adreno texture compression format (for RGB and RGBA)
- **ETC** - Standard OpenGL ES 2.0 texture compression format (for RGB only)
- **DXT** - DirectX Texture Compression
- **BC** - Block Compression (for DirectX)

Adreno GPUs support both HDR and LDR profiles for ASTC.

Unlike desktop OpenGL, Vulkan does not provide the necessary infrastructure for the application to compress textures at runtime, so texture data needs to be authored off-line.

Here are several methods for converting textures to Adreno hardware:

- On-device conversion: a time-consuming one-time conversion of texture assets that occurs when the game starts up
- Prepackaging: the most optimized solution, but requires alternative versions of the APK for each GPU

- Downloading: requires GPU detection and an internet connection, but allows for even more control on the exact texture format for each GPU

In any case, the first step is to create the compressed textures. The effectiveness of supported compression formats is described here.

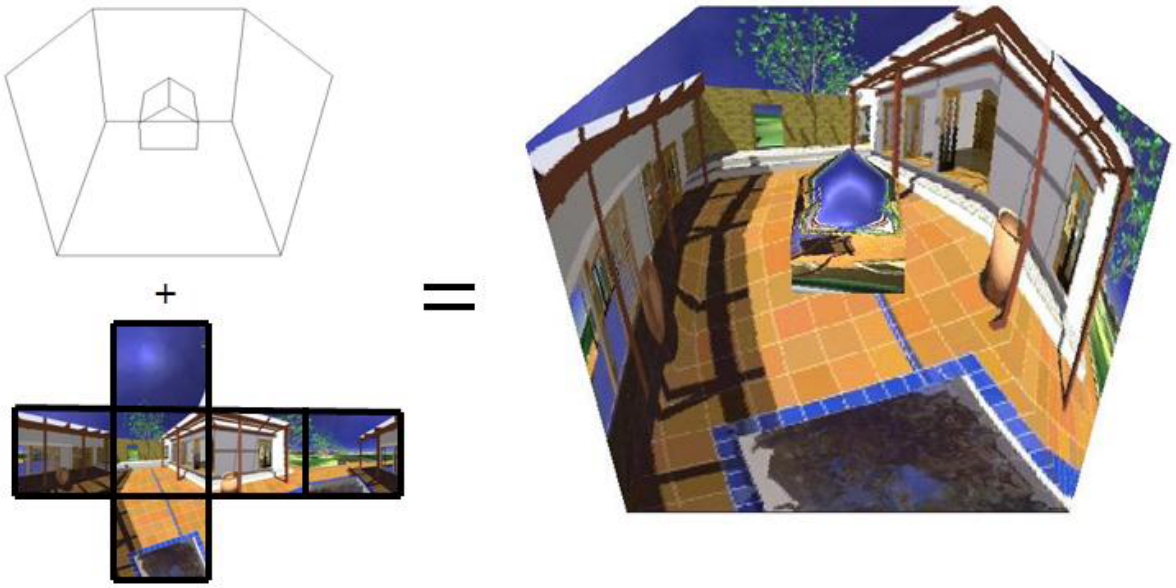
Floating point textures

Adreno GPUs support floating point texturing features.

Cube mapping with seamless edges

Cube mapping is a fast and inexpensive way to create graphic effects such as environment mapping. Cube mapping takes a three-dimensional texture coordinate and returns a texel from a given cube map (similar to a sky box).

Adreno GPUs support seamless edge support for cube map texture sampling.



Video textures

Adreno GPUs support video textures, which consist of moving images that are streamed in real-time from a video file.

There is no dedicated video memory which the application developer can control – video textures are allocated from system memory by the graphics driver like other textures.

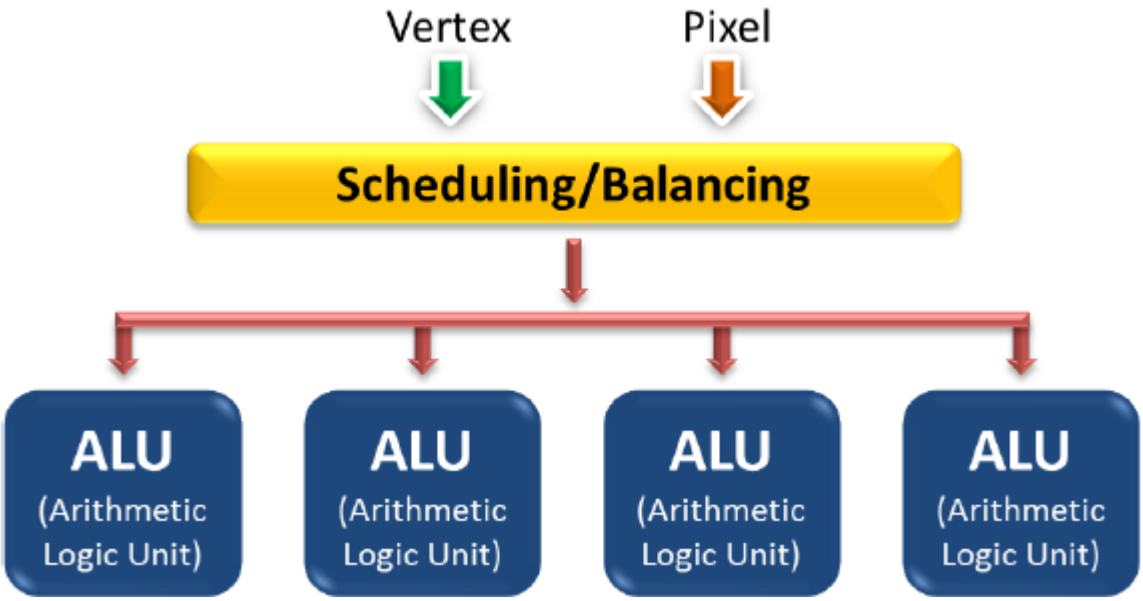
Shaders

Unified shader architecture

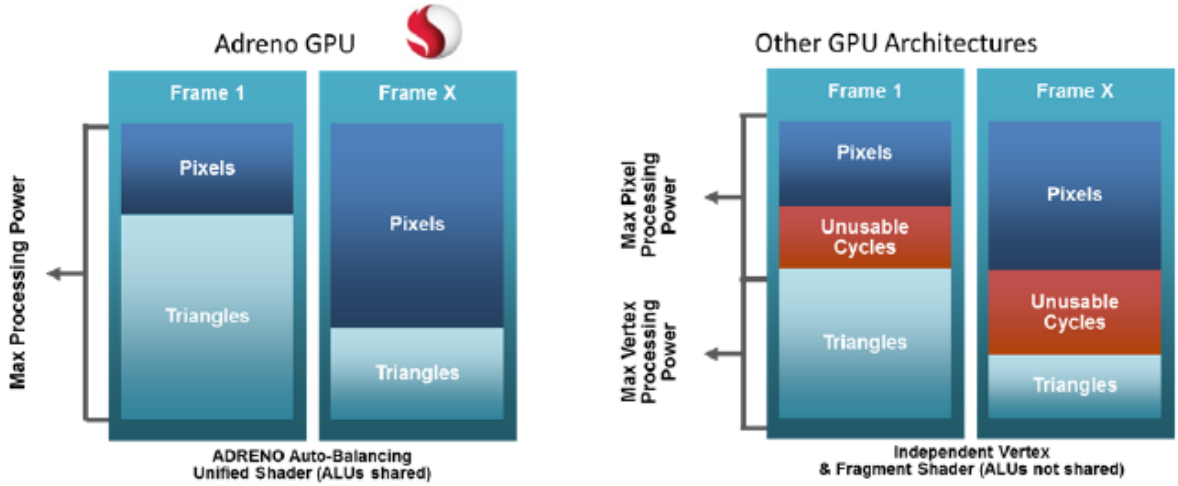
All Adreno GPUs support the Unified Shader Model, which uses a consistent instruction set across all shader types.

In hardware terms, Adreno GPUs have computational units (e.g., Algorithmic Logic Unit, or ALUs) and fetch resources that support vertex, fragment and compute shaders – so much of the following advice applies to most or all of these shaders.

Shader processing is done within the unified shader architecture, as shown in the following image. This image shows that vertices and pixels are processed in groups of four as a vector, or a thread. When a thread stalls, the shader ALUs can be reassigned.



In unified shader architecture, there is no separate hardware for the vertex and fragment shaders, as shown in the image below. This allows for greater flexibility of pixel and vertex load balances.



The Adreno shader architecture is also multithreaded. If a fragment shader execution stalls due to a texture fetch, another shader is scheduled for execution. Multiple shaders can be “ready to efficiently schedule” as long as there is room in the hardware (primarily GPRs).

No special steps are required to use the unified shader architecture. The Adreno GPU intelligently makes the most efficient use of shader resources depending on scene composition.

Scalar architecture

Adreno GPUs have a scalar component architecture. The smallest component they support natively is a scalar component. This results in more efficient hardware resource use for processing scalar components, as it does not waste a full vector component to process the scalar.

There are numerous ways to make efficient use of the hardware.

GPU-Driven Rendering

Geometry instancing

Geometry instancing renders multiple copies of the same mesh or geometry in a scene at once. This technique is used primarily for objects like trees, grass, or buildings that can be represented as repeated geometry without appearing unduly repetitive. However, geometry instancing can also be used for characters.

While vertex data is identical across all instanced meshes, each instance can have other differentiating parameters changed to reduce visual repetition – for example: color, transforms, or lighting.

As shown in the image below, all barrels in the scene could use a single set of vertex data that is instanced multiple times instead of using unique geometry for each one.

Geometry instancing usually saves memory: it allows the GPU to draw the same geometry multiple times in a single draw call with different positions, while storing only a single copy of the vertex data, a single copy of the index data, and an additional stream containing one copy of a transform for each instance. Without instancing, the GPU would have to store multiple copies of the vertex and index data.

Indirect draw calls

Introduced in OpenGL ES 3.1, indirect draw calls move the draw call overhead from the CPU to the GPU. This can provide a significant performance benefit over regular draw calls.

Low Priority Asychronous Compute (LPAC)

LPAC (Low Priority Asynchronous Compute) is intended to perform less latency-sensitive compute tasks concurrently with other processing on the Graphics Pipe – in other words, somewhat longer-lived (on the scale of multiple milliseconds) compute tasks are often best issued with LPAC, and have somewhat different performance characteristics.

Qualcomm True HDR

Introduction

HDR displays have been around for a long time in the PC and TV space, and mobile devices featuring OLED screens were introduced in 2018 to support a higher dynamic range and wider color gamut.

To make better use of OLED screens' wide color gamut and high dynamic range, Qualcomm launched True HDR gaming as part of Snapdragon Elite Gaming – the first end-to-end HDR solution on mobile.

True HDR gaming takes full advantage of Snapdragon's GPU and Display Process Unit (DPU) capabilities: HDR content (R10G10B10A2 format/BT2020 gamut) directly. Color Volume Mapping is performed by the DPU.

This guide is meant as an overview of HDR in general, as well as the Qualcomm True HDR gaming solution.

High Dynamic Range (HDR)

In the following image, the world's luminance level is [0, infinite) candela per square meter(cd/m ^{2}). Through the human eye's rods and cones, we perceive a luminance of around [10 ^{6}, 10 ^{8}] cd/m ^{2}.

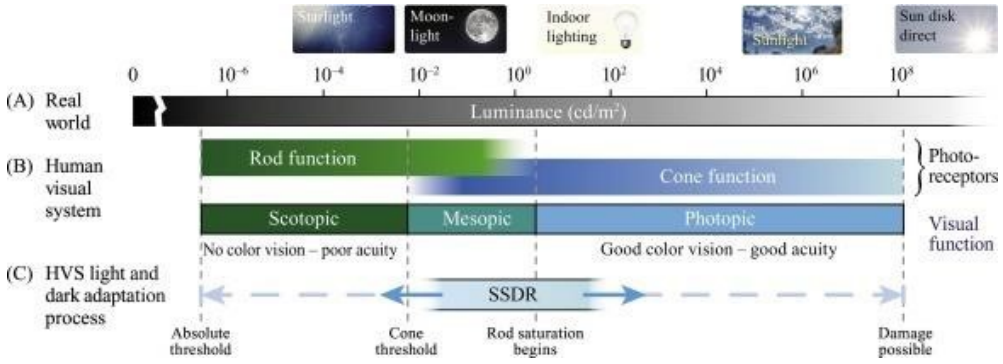


Figure1 Real-world luminance levels and the high-level functionality of the Human Visual system – image taken from Perceptual Design for High Dynamic Range Systems. p. 18, 1

Standard Dynamic Range (SDR) and High Dynamic Range (HDR) support different luminance levels:

- SDR support luminance in [0.0002, 100] cd/m ^{2}
- HDR support luminance in [0.00005,1000 or larger] cd/m ^{2}

The change of dynamic range and color gamut in the SDR and HDR pipeline are seen here:

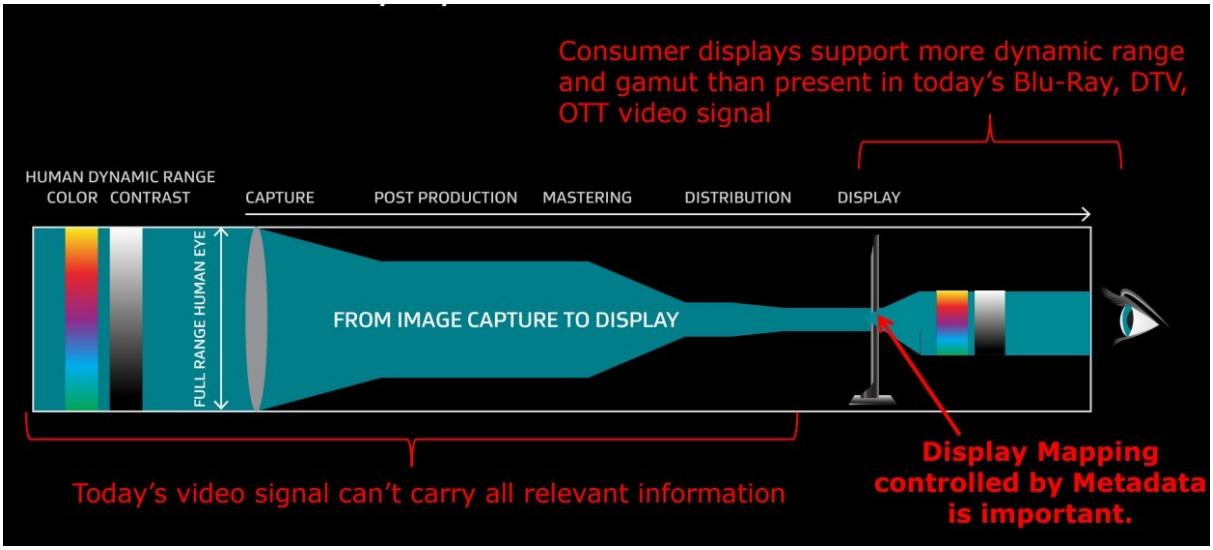


Figure2 Content delivery in SDR. Image Credit: Dolby Laboratories, Inc.

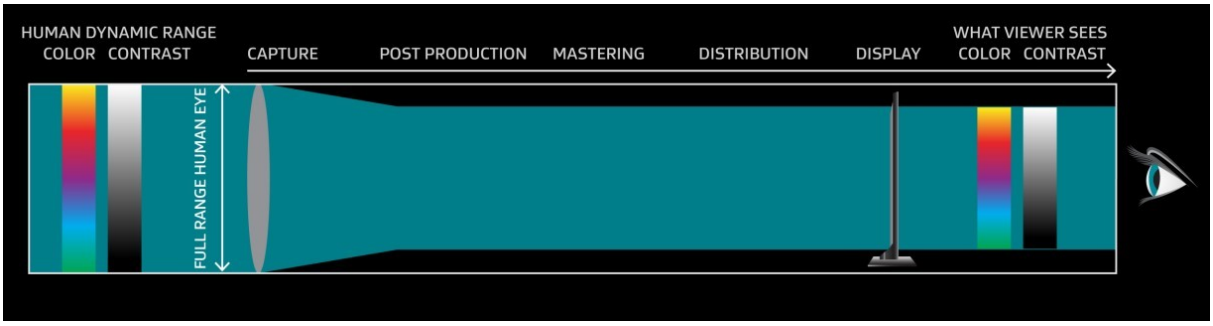


Figure3 Content delivery in HDR. Image Credit: Dolby Laboratories, Inc.

[1] Kunkel, T & Daly, Scott & Miller, S & Froehlich, Jan. (2016). Perceptual Design for High Dynamic Range Systems. 10.1016/B978-0-08-100412-8.00015-2.

Wide Color Gamut (WCG)

The human eye has three types of color sensors that respond to different ranges of wavelengths. The color consists of two parts: brightness and chromaticity. For example, white and grey coors have the same chromaticity but different brightness.

The International Commission on Illumination (CIE) created the CIE 1931 RGB color space to model how we see color. The horseshoe-shaped region represents what a “typical” human can see in response to different light wavelength.

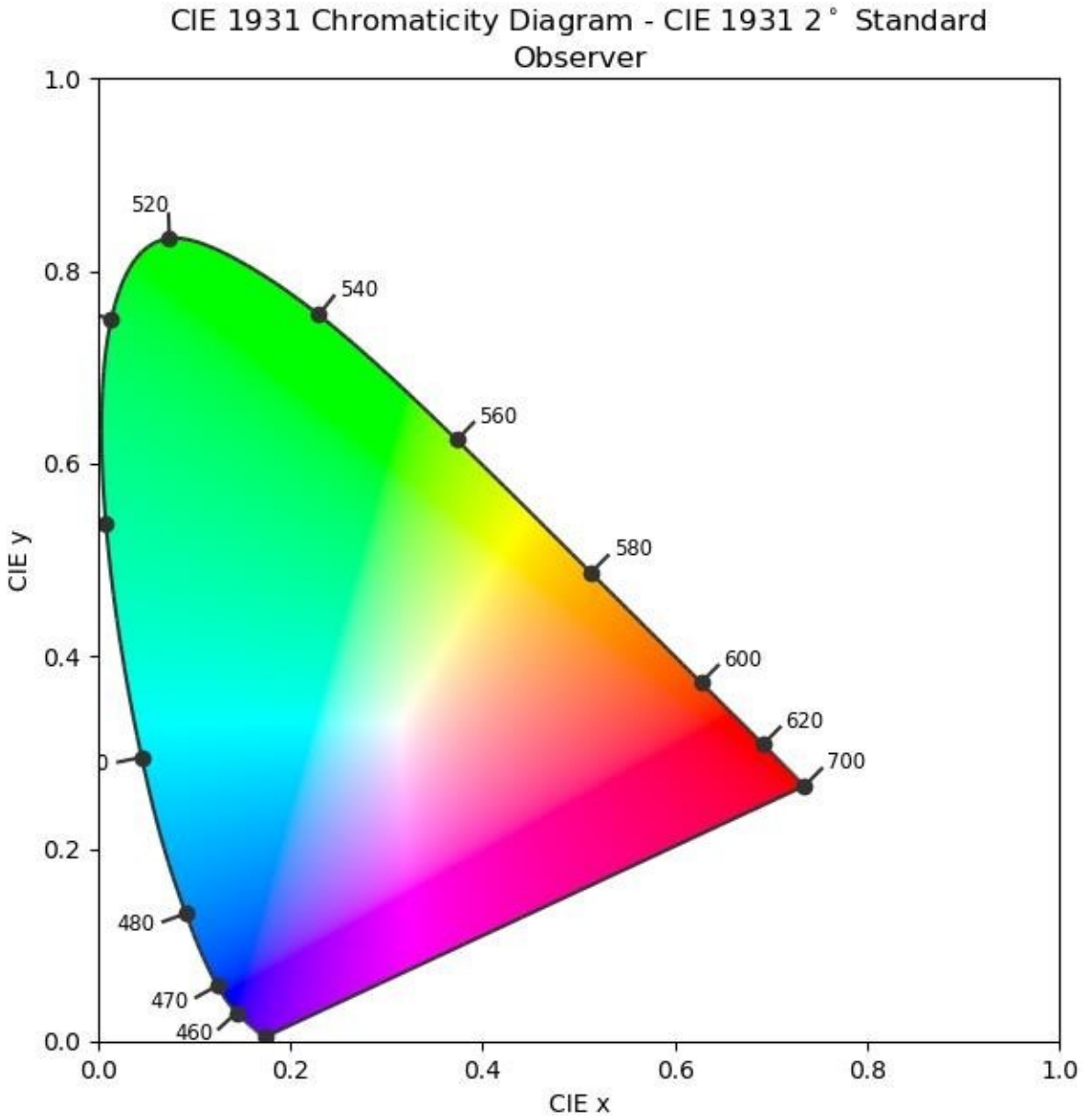


Figure4 CIE 1931 color space.

The color gamut is a specific, complete subset of colors. It is often referred to as the entire range of colors available on a display device. The following image shows various standards' color gamuts.

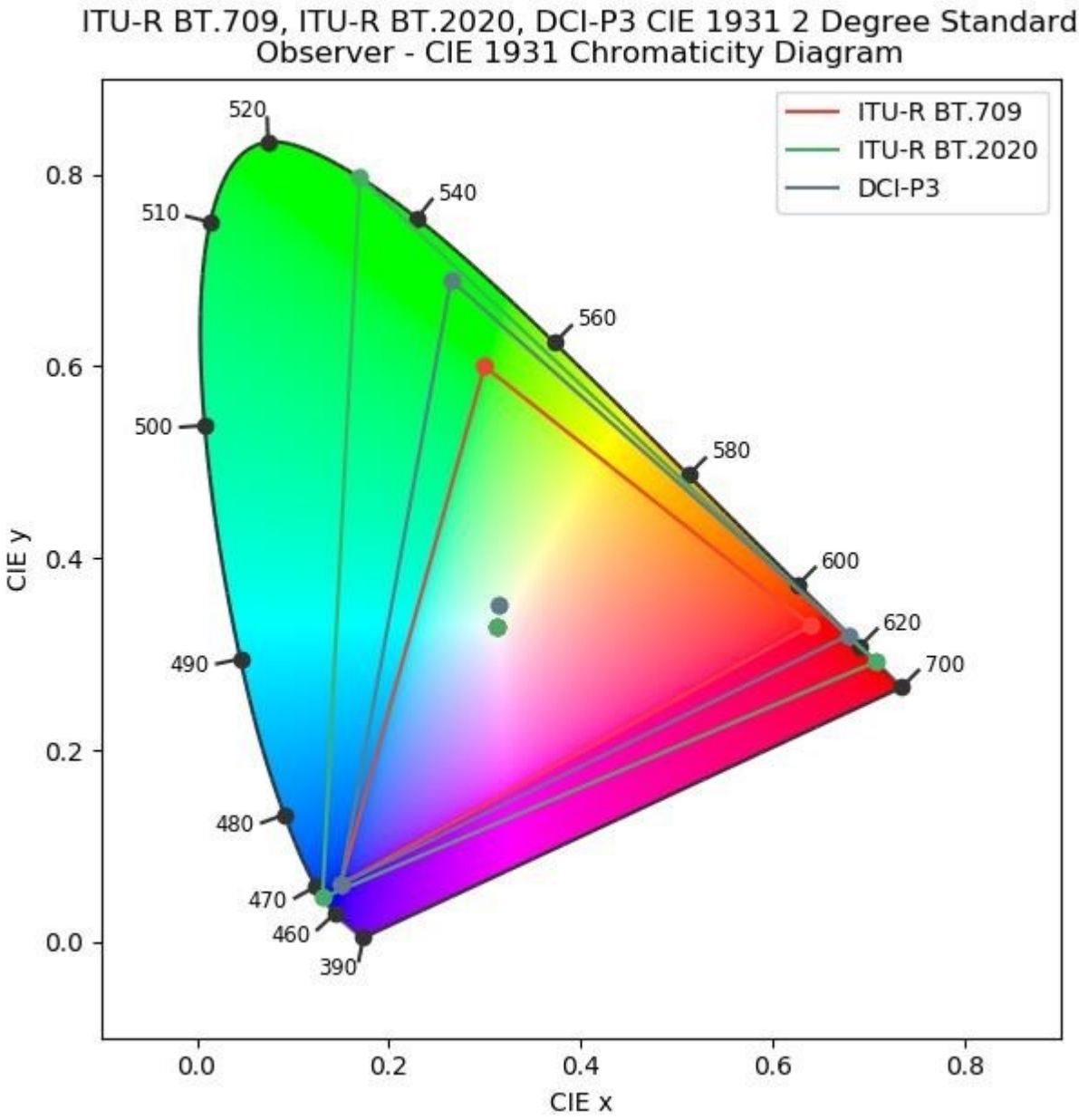


Figure5 BT.709, DCI-P3, BT.2020.

- BT.709, high definition television (HDTV) standard
- DCI-P3, used for cinema theater presentations and an intermediate gamut for UHD content
- BT.2020, ultra-high-definition television (UHDTV) standard

Color gamut with a larger range than BT.709, such as BT.2020, is often referred as Wide Color Gamut (WCG).

Color volume

Brightness and color gamut define color volume, so a higher dynamic range and wider color gamut means a larger color volume.

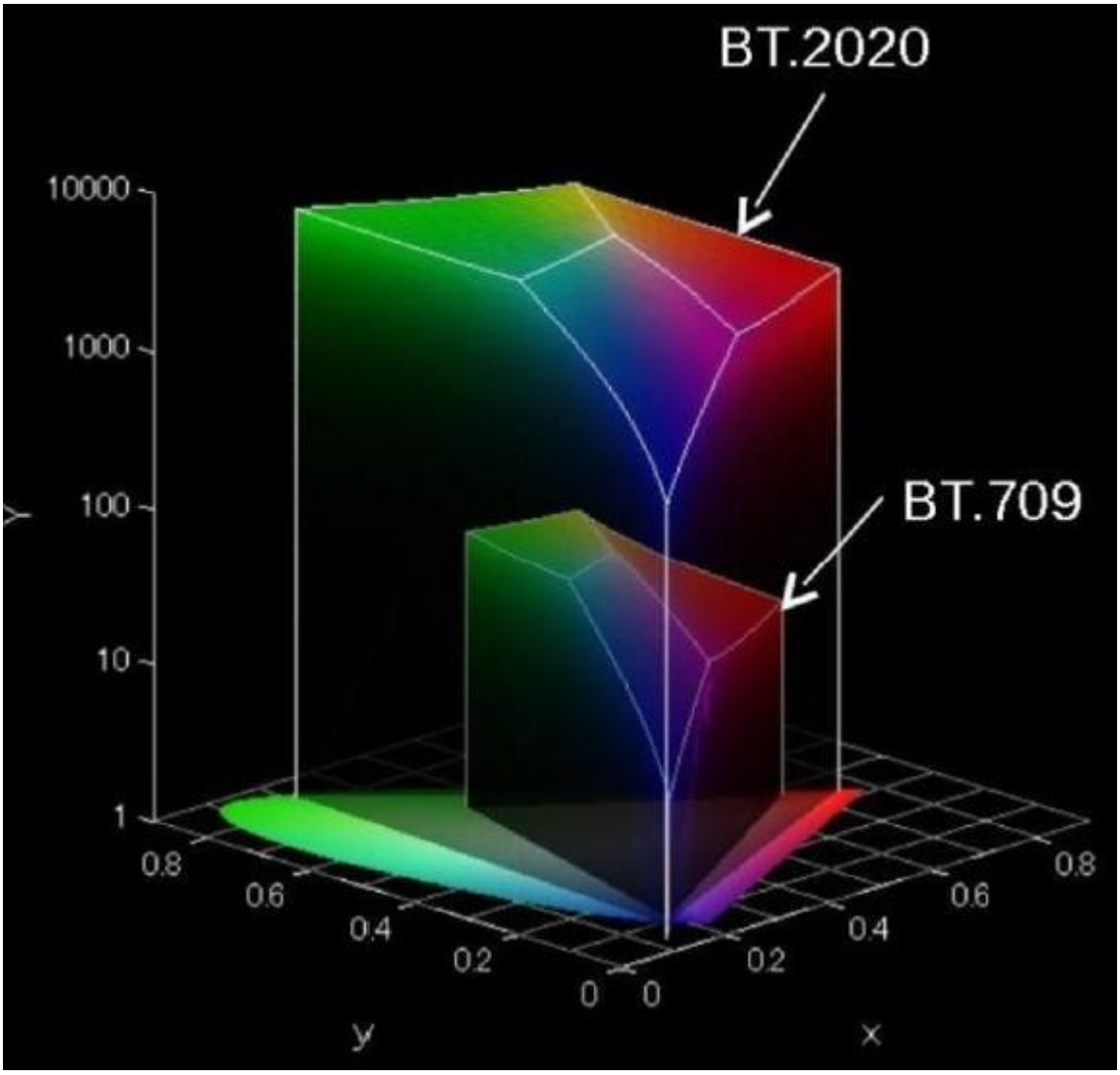


Figure6 The color volume of BT.2020 color gamut and BT.709 color gamut. Image credit: Sony.

Display technique

Modern display technology features increased bit widths, color gamut, and screen brightness:

- Maximum bit width support includes 6, 8, 10, 12 bits.
- Maximum luminance (brightness) support includes 100 nit, 500 nit, and 1000 nit.
- Color gamut support includes BT.709, DCI-P3, and BT.2020.

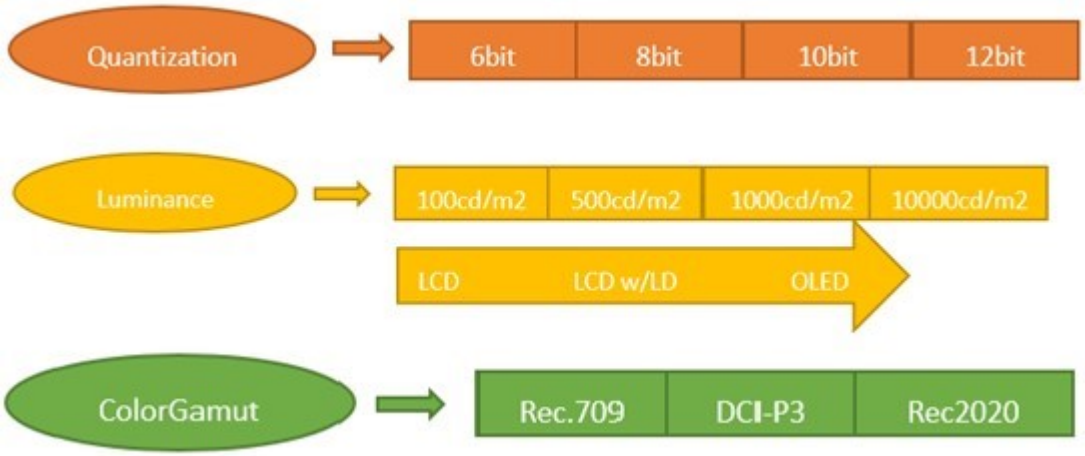


Figure7 The evolution of bit, brightness, and color gamut supported by display devices.

Light-to-display overview

Figure8 Environment photons to display photons.

When a photo is taken using a camera and viewed, the following happens:

- The camera converts light to video signal using an Optical to Electrical Transfer Function (OETF)
- The display converts video signal to light using the reverse function, Electrical to Optical Transfer Function (EOTF)

For SDR, gamma is used. For HDR, Perceptual Quantization (ST-2084) or Hybrid Log-Gamma (BBC / NHK) are used. ST-2084 defines following:

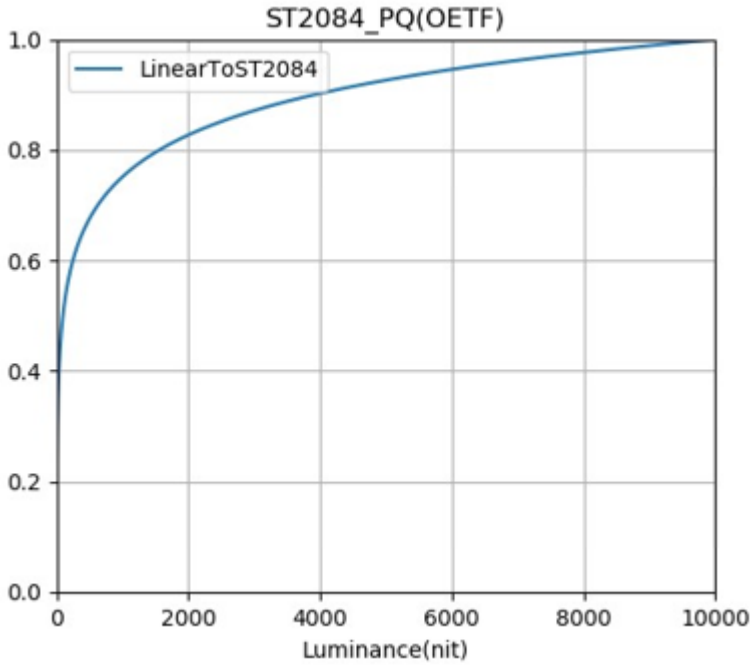


Figure9 SMPTE ST 2084 (PQ).

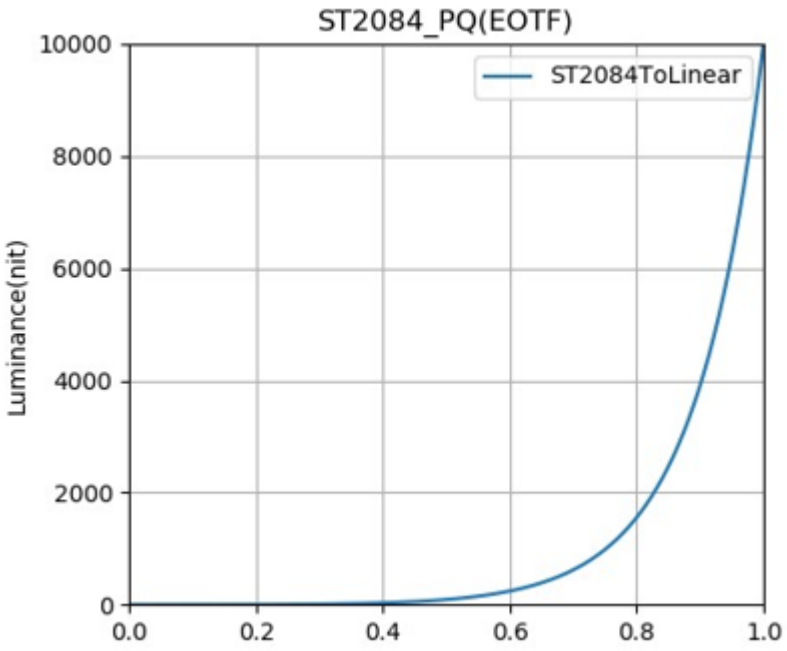


Figure10 SMPTE ST 2084 PQ EOTF(PQ).

HDR10

HDR10 is an open standard (announced on August 27, 2015). This standard is composed of the following specifications:

- **Bit depth:** 10-bit
- **Color representation:** ITU-R BT.2020
  - Chromaticity
- **Electro-Optical Transfer Function (EOTF):** SMPTE ST 2084
  - Maps non-linear signal value into display light
- **Static metadata:** SMPTE ST 2086
  - SMPTE ST2086 defines the “Master Display Color Volumes” metadata, which is used to describe the capability of the display. It includes:
    - CIE(x,y) chromaticity coordinates for Color primaries
    - CIE(x,y) chromaticity coordinates for White point
    - Min and max luminance
    - Others such as MaxFALL (Maximum Frame Average Light Level) and MaxCLL

(Maximum Content Light Level) static values

Color volume mapping

Color volume mapping is the process of mapping content that was created on a mastering display down to a playback display. It includes tone mapping for Luminance and gamut mapping for Chromaticity.

- **Tone mapping:** The peak luminance of a playback HDR display will often be lower than the peak luminance of the reference display that is used to master HDR content. So tone mapping is necessary to convert the high luminance HDR content to the dynamic range supported by the playback display.
- **Gamut mapping:** The primary colors of a playback HDR display are often less saturated than the reference HDR display. The wider content gamut must be mapped to the corresponding gamut of the playback display.

Note: When applying color volume mapping, the luminance and chromaticity attributes of both the reference display and the content must be passed to the display. These attributes are represented by the static metadata fields that are defined in the SMPTE ST 2086 standard.

Traditional Game HDR

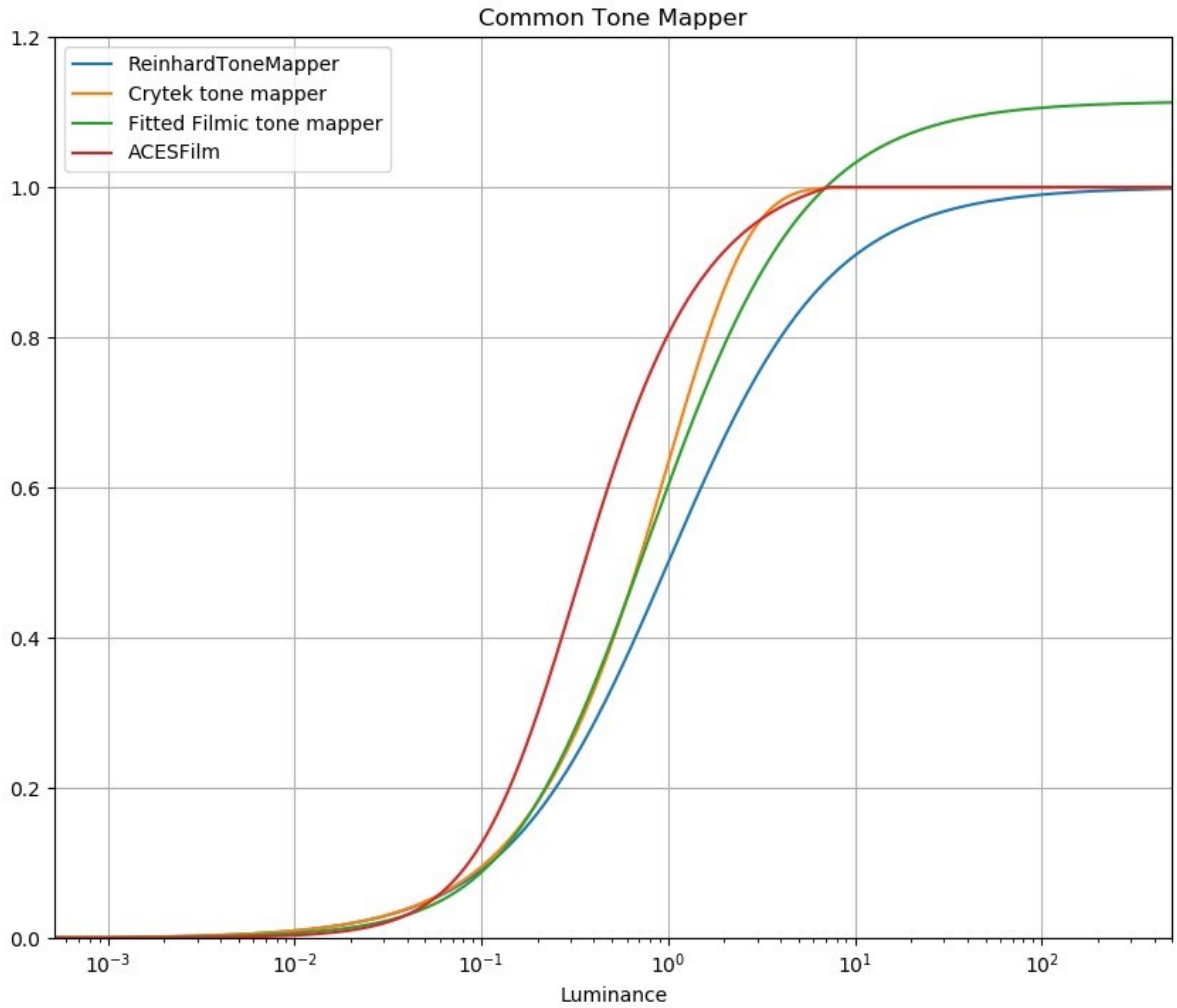
HDR images are more convincing than SDR images:

Figure11 sdr image(left) and traditional HDR image(right), images taken from Unreal.

Figure12 Traditional HDR pipeline.

A traditional game HDR pipeline takes the following steps:

1. **RenderedScene:** A scene is rendered on Render-Targets with HDR format, such as RGBA16, R11G11B10, R10G10B10A2.
2. **Postprocessing:** Bloom, vignette, grain jitter, motion blur, etc.
3. **ColorGrading:** Contrast, saturation and gamma, gain, etc.
4. **ToneMapping:** Maps the dynamic range data to standard range data (RGBA8888/sRGB). There are many traditional mapping curves:



1. **Reinhard tone mapping:** Simple and direct: the light dark, dark light, the overall image bias towards gray.
2. **Crytek tone mapping:** CryEngine2 implementation: the power function is used to form an s-curve, which is steeper than the slope of Reinhard tone mapping and sensitive to dark colors. The overall image has a higher contrast and brighter color.
3. **Filmic tone mapping:** Uncharted 2 implementation: the results image after tone mapped are close to the effects of manual adjustment, and the overall growth process is longer.
4. **ACES Film:** ACES simplified version: sample ODT(RRT(x)) transform and fit a simple curve.
5. **Gamma:** Gamma correction is an old, simple approach to brightening or darkening an image

True HDR

Compared to a traditional HDR image, a Qualcomm True HDR image exhibits a heightened contrast, with more details preserved in highlights and shadowed areas. Qualcomm True HDR gaming is compatible with the HDR10 standard; in fact, the DPU expects HDR10 data input.

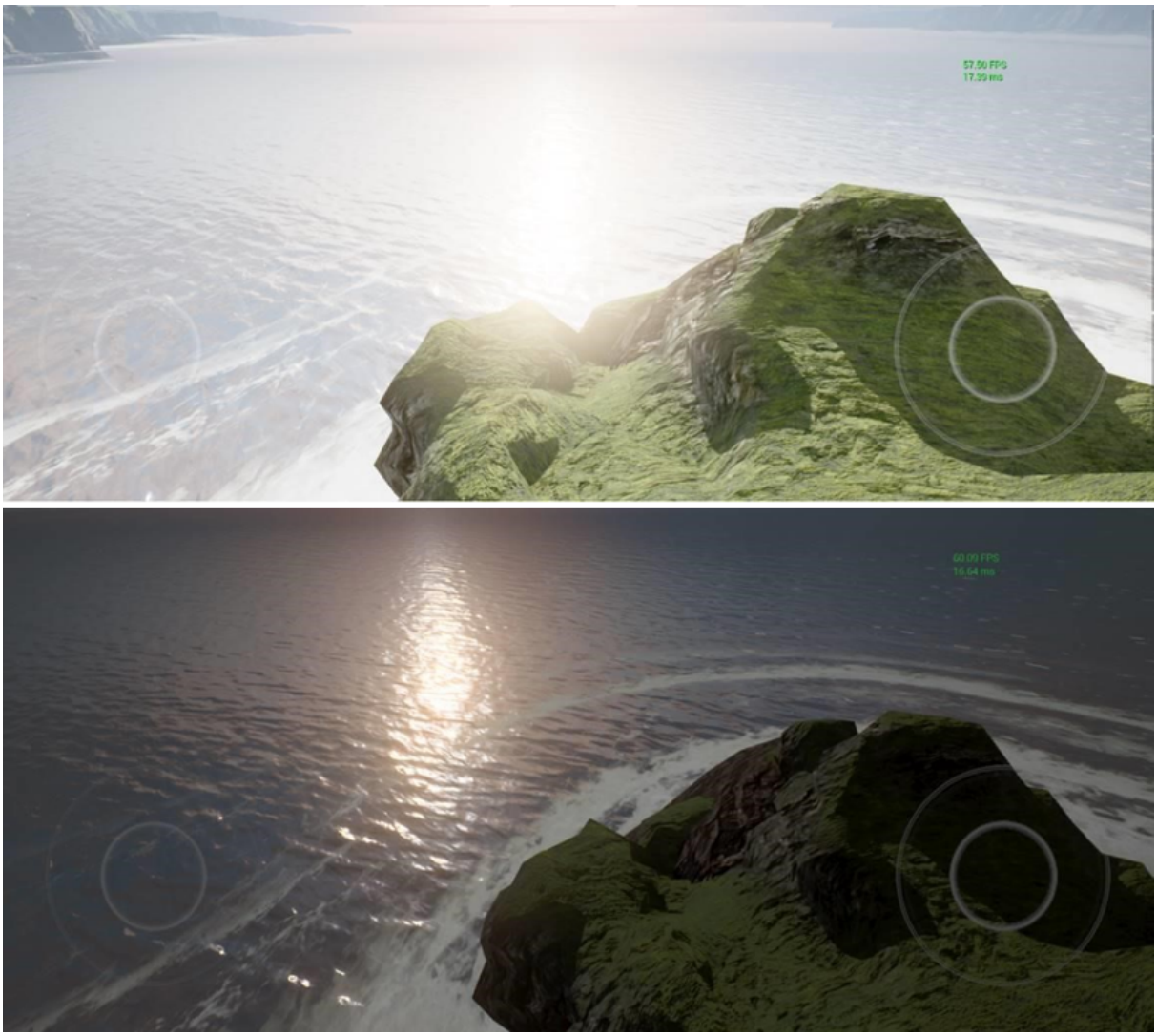


Figure13 Traditional HDR image(up), True HDR image(down).

True HDR pipeline

HDR Game content is produced on a reference display with an exceptionally high dynamic range and a wider gamut. The rendered scene is mapped to the dynamic range and color space supported by True HDR in the postprocessing phase – this final step depends on the DPU consuming HDR metadata.

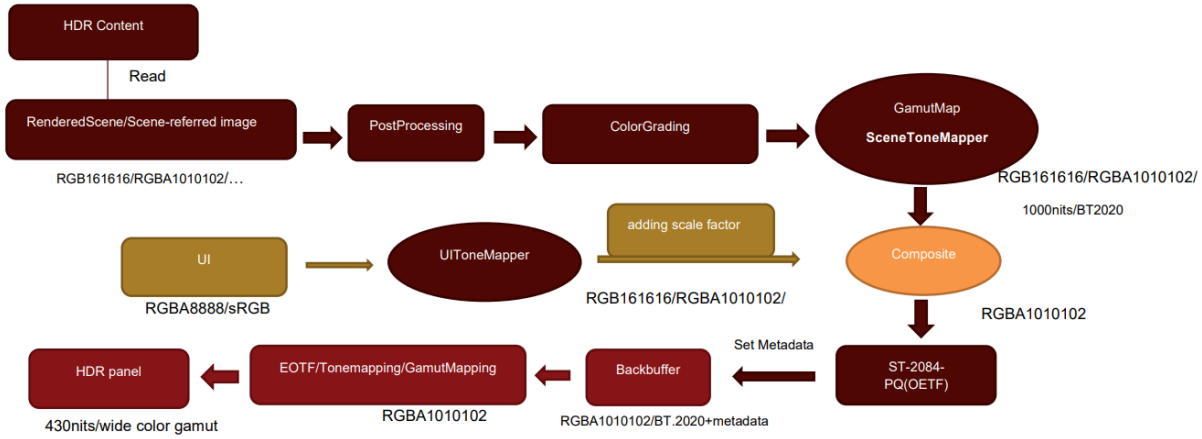


Figure14 Workflow of a True HDR pipeline for Game

The following provides a detailed overview of the True HDR pipeline and how it functions:

- **HDR Content:** Game content produced on a reference display with large color volume (exceptionally high dynamic range and wider color gamut).
- **Scene-referred image:** Takes HDR content as input and uses a realistic rendering approach (such as physically based rendering (PBR)) to generate a good HDR scene reference image with proper exposure, rich highlights, and shadows.
- **Postprocessing, ColorGrading:** Same as traditional game HDR.
- **GamutMap and Tonemapper:** Maps the dynamic range and color space of the RenderedScene to the dynamic range and color space supported by True HDR.
- **UIToneMapper:** Maps UI render data to HDR.
- **Adding a scale factor:** Adds a scale factor to the UI to match with the main rendered scene.
- **Composite:** Blends the scene HDR and UI HDR inputs. If the game relies on a lot of transparent UI, it would be easier to render the entire UI to a render target and then blend that render target with the back-buffer, rather than directly render UI to the back-buffer.
- **ST-2084-PQ(OETF):** The data required for a backbuffer using True HDR is called ST-2084-PQEncoded. The backbuffer should be encoded by ST-2084-PQ, and the encoded value mapped to the normalized range – [0.0,1.0].
- **EOTF/Tonemapping/GamutMapping**: Handled by Qualcomm DPU.

- **ST-2084-PQ(EOTF):** Inverse of ST-2084-PQ(OETF).
- **Tonemapping:** Maps output of EOTF to the actual color space used by the screen and the actual dynamic range.
- **GamutMapping:** Maps one color space to another. The usual method is linear stretch or compression. The color space required by HDR10 standard is BT2020, but the actual screen cannot meet this requirement. The Qualcomm DPU will perform the color gamut conversion.

Academy Color Encoding System (ACES) Tonemapping

Even with HDR screens, the dynamic range of support is limited. Scene-referred images can generate very high dynamic range images. A SceneToneMapper might map an input HDR scene of up to 10000 nits to a 1000 nit True HDR scene, with a UIToneMapper performing a similar mapping for the UI data.

The Academy Color Encoding System (ACES) is often used in games.

ACES pipeline

There are two color gamuts in ACES: AP0 and AP1.

AP0 and AP1 contain BT.709 and BT.2020, which means that ACES can handle several types of tone mapping and gamut mapping:

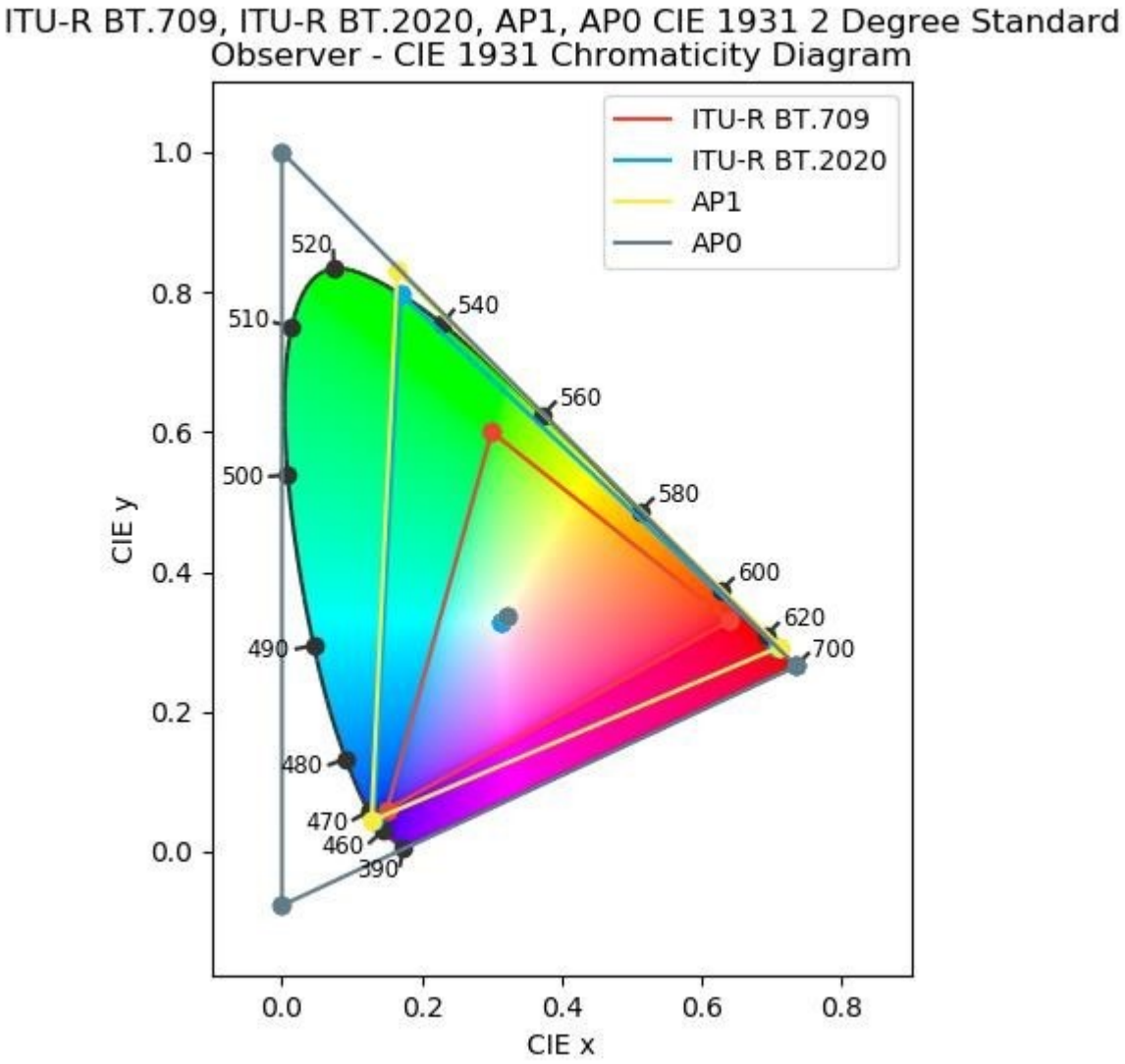


Figure15 Color gamut comparison among BT.2020, BT.709 and AP0/AP1 used by ACES.

Figure16 ACES pipeline.

For games, images after scene rendering are called a Scene-Referred image. The ACES pipeline involves these components:

- **Color grading:** According to the ACES Color Space.
- **Reference Rendering Transform (RRT):** Converts Scene-Referred colorimetry to Display-Referred: The color space is converted to the ACES defined AP0 color space and dynamic range is converted to 0-10000 nits, so the image can be renderered to any output device.

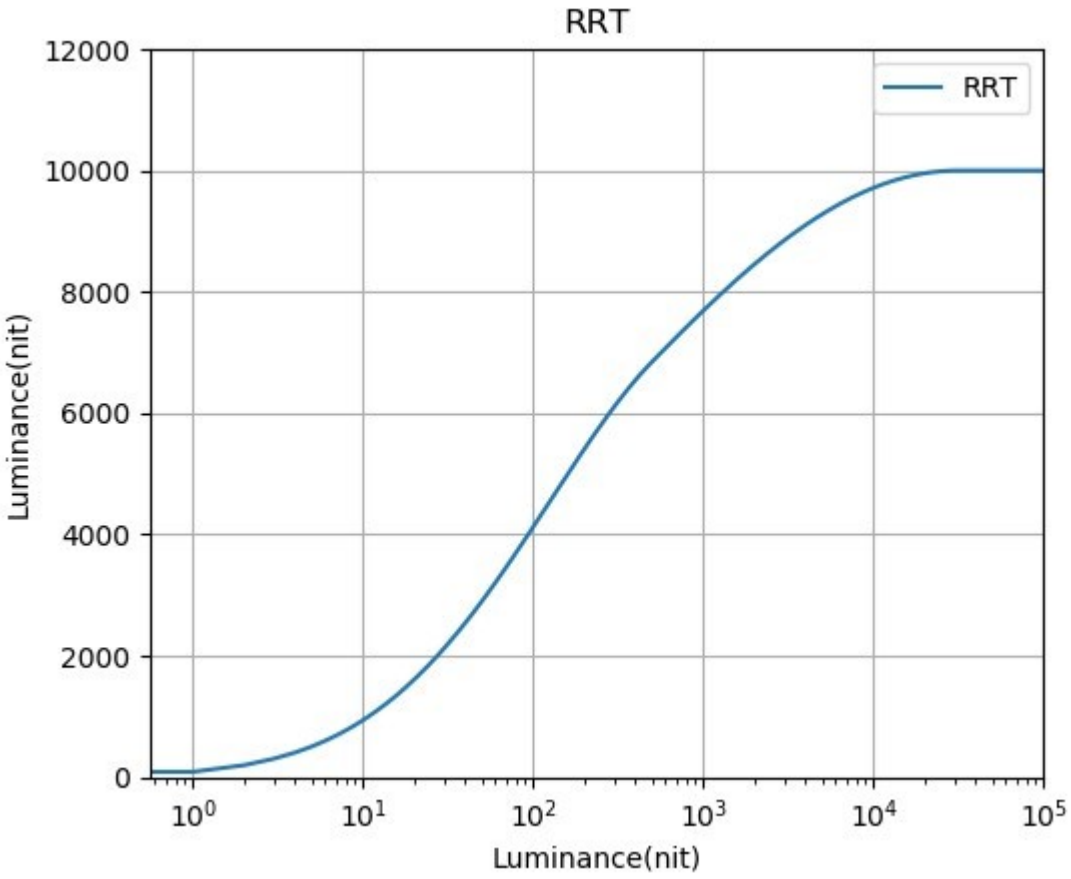


Figure17 Figure 18 RRT curve.

- **Output Device Transform (ODT):** Converts the color space into the ACES defined AP1 color space: both the dynamic range and the color space is mapped to the dynamic range and color space supported by the device. There are many ODTs: ODT_48nits, ODT_ 1000nits, ODT_2000nits, and others.

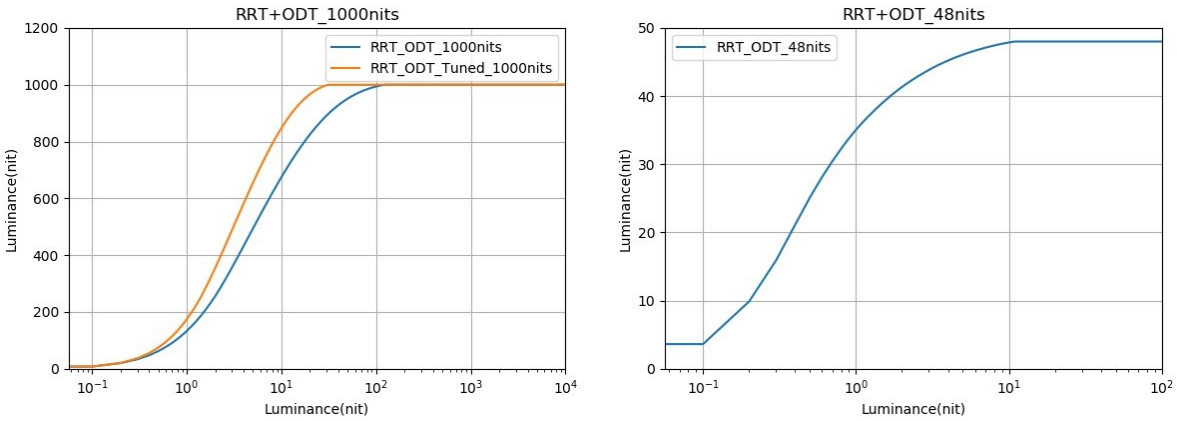


Figure18 RRT+ODT_1000 nits (left) and RRT+ODT_48 nits (right)

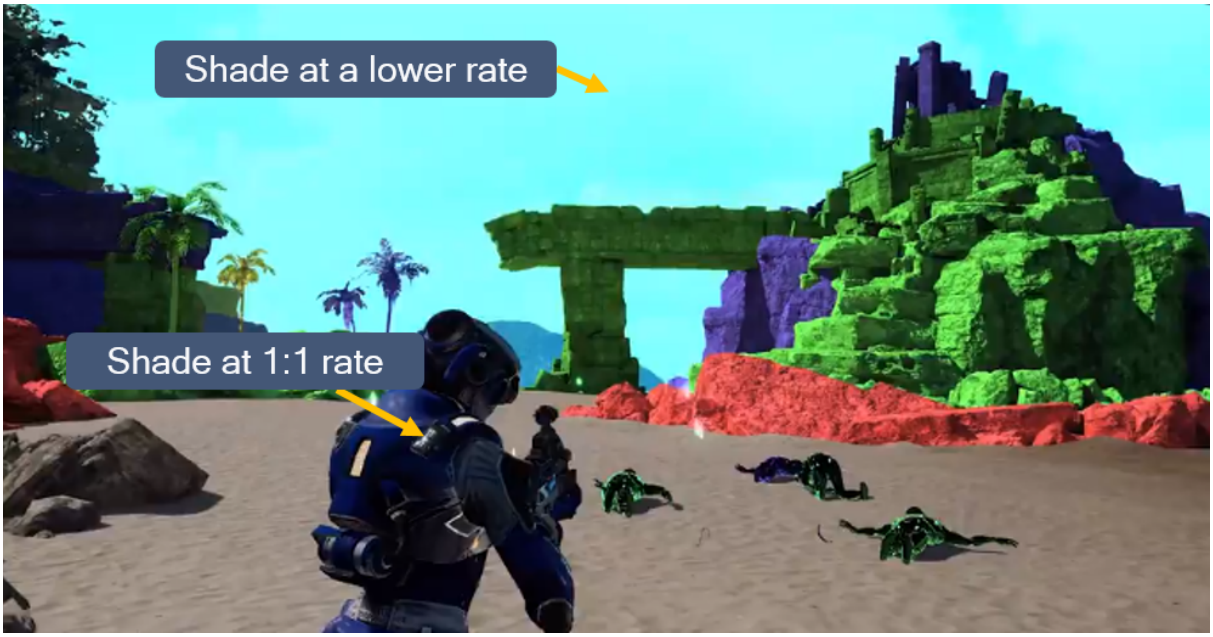
OpenGL ES: Here is a guide to setting up TrueHDR.

Variable Rate Shading (VRS)

Variable Rate Shading (VRS) allows a fragment shader to color one or more pixels at a time, such that a fragment can represent one pixel or a group of pixels. Think of VRS as performing the inverse of what anti-aliasing techniques do: anti-aliasing techniques sample each pixel more frequently to avoid aliasing and jagged edges by smoothing content with high variation – but if images do not have a high color variation or will be blurred on a subsequent pass (eg, with motion blur), we may be able to sample fewer pixels without harming visual quality.

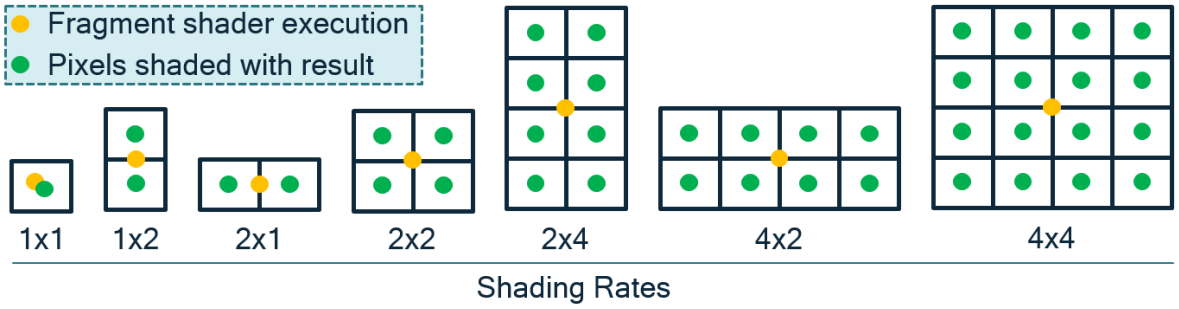
VRS allows developers to specify shading rates where only one shader computation is performed for a fragment and the result is applied to the specified pixel group configuration. When used properly, this results in no visual quality degradation while significantly reducing the GPU's work to render a frame, thus conserving power and improving performance. With several high display rate mobile devices commercially available, like those powered by our Qualcomm® Snapdragon™ mobile platforms and their embedded Qualcomm® Adreno™ GPU's, the need for shading every pixel for all surfaces being rendered is diminished.

The following screenshot from our Variable Rate Shading demo shows how higher rate (per-pixel) shading should be used on highly detailed areas, while lower rates (shading fragments comprising groups of pixels) can be done on lower detail areas.



How does Variable Rate Shading work?

When the GPU renders and rasterizes objects into a surface, it does so at a rate of one sample per pixel (assuming multisampling is not used, although this concept can be applied to multisampling as well). Through graphics API extensions, a developer can modify the shading rate of a given surface to be coarser than a pixel, as shown:



Effective ways to modify Shading Rate

These extensions can improve performance on heavy fragment-bound draw calls such as:

- Surfaces where the color variance is low
- Surface areas that do not require per-pixel shading accuracy, such as color targets that will be downscaled through motion blur and areas that have significant changes in their velocity fields. If depth of field is used, there will be areas outside of the focus point that will be blurred
- Effects like motion volumetric rendering, where a portion of the scene is processed at full shading rate and a portion can be processed at a reduced shading rate

---

Note: Reducing shading rate can impact the visual quality of a rendered object if used inappropriately. On mobile devices where performance and power are tightly coupled, the use of coarser shading rates can also improve power consumption and reduce the thermal profile of your game, ultimately increasing the user's play time.

Raytracing

Introduction

Ray tracing is a rendering technique that closely approximates the behavior of light in the real-world, and also has applications outside of rendering (path-tracing audio signals and AI line-of-sight, for example). Performance can scale from cinematic quality productions to mobile applications.

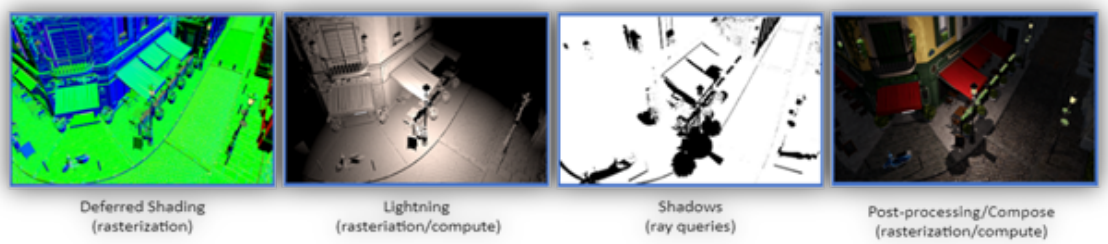
Qualcomm is engaged in continuous research and development, in partnership with many other research teams and game studios, to efficiently simulate light rays reflecting and refracting on and through surfaces to deliver beautiful images in real-time on Adreno.

How does raytracing differ from rasterization?

While ray tracing can replace standard rasterization, typically it's implemented as additional functionality on top of existing rasterization-based pipelines, providing more features (like accurate refraction) and simplifying some effects (like shadows, reflections and subsurface scattering) that a traditional rasterization-only pipeline would realize with less robust tricks and hacks.

Instead of (only) shading the visible pixels of triangles, rays are generated for pixels on screen. These rays can collide with different surfaces and reflect, be absorbed, refract, or some combination thereof. Many rays can be generated from a single intersection, and ray propagation can continue as long as desired, trading off visual quality against performance.

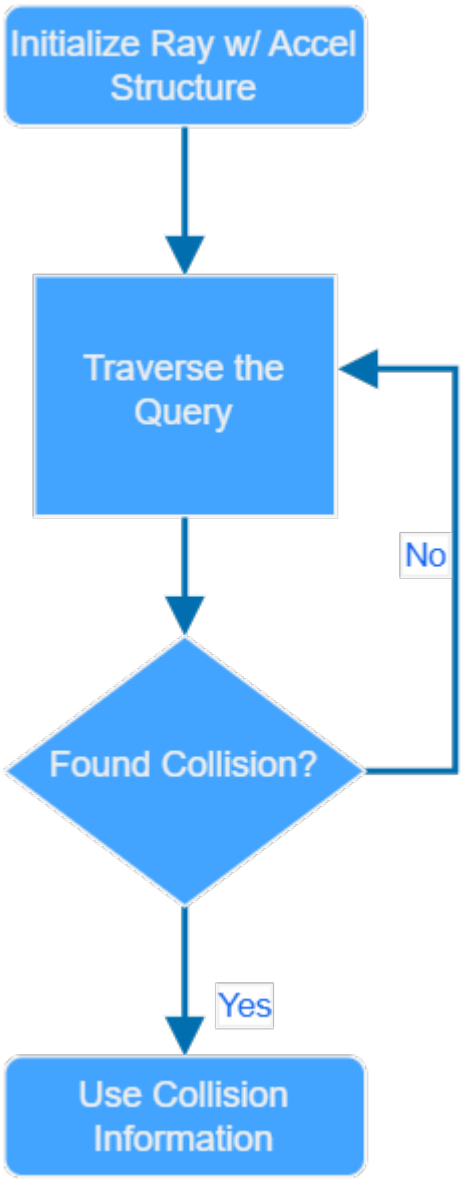
This required changes to all modern graphical APIs. Vulkan is the typical API for raytracing on mobile devices, and provides access to custom acceleration structures, ray-tracing pipelines, and ray queries – among many other supporting extensions outside of the scope of this article.



Hit detection

How can we generate rays efficiently on an HD screen? With a modest geometry budget of 100,000 visible, opaque triangles – even leaving the added complexity of transparency aside – the naive approach of casting a ray per pixel at every triangle results in about 207,360,000,000 intersection tests: unsuitable for offline rendering systems, let alone real-time renderers.

Acceleration structures are driver-optimized spatial partitioning datastructures that drastically reduce the number of intersection tests, with no noticeable degradation in accuracy. Read on to learn how to efficiently implement raytracing on Adreno.



Spec Sheets

Introduction

Information that applies to specific Adreno models is collected here:

Renderer Architecture

On A5X GPUs, the Triangle setup rate is 1 prim/clock (but user clip planes turn this rate into a 50 cycles entire-pipeline stall if a primitive is clipped).

Vertex and Index Buffers

On A7X, the vertex cache size is 32 4D vertices.

Texture Features

On A5X GPUs, ASTC is compressed in L2 and decompressed in L1. ETC formats stay compressed in L1.

A5X GPUs support up to 32 total textures in a single render pass – up to 16 textures in the fragment shader and up to 16 textures in the vertex shader.

Texture formats

The following table describes functionality available for each supported texture format.

A5X

***** only when **optimal** layout is used.

A5X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBWC*
A1_UNORM	1 bit	1x		√		
A8_UNORM	8 bit	8x		√	√	
R8_UNORM	8 bit	8x		√	√	
R8_SNORM	8 bit	8x		√	√	
R8_UINT	8 bit	8x		√	√	
R8_SINT	8 bit	8x		√	√	

A5X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBWC*	F
R4G4B4A4_UNORM	16 bit	8x		√	√	√	√
R4G4B4X4_UNORM	16 bit	8x		√	√	√	√
R5G5B5A1_UNORM	16 bit	8x		√	√	√	√
R5G5B5X1_UNORM	16 bit	8x		√	√	√	√
A1R5G5B5_UNORM	16 bit	8x		√			
X1R5G5B5_UNORM	16 bit	8x		√			
R5G6B5_UNORM	16 bit	8x		√	√	√	√
R8G8_UNORM	16 bit	8x		√	√	√	√
R8G8_SNORM	16 bit	8x		√	√	√	√
R8G8_UINT	16 bit	8x		√	√	√	√
R8G8_SINT	16 bit	8x		√	√	√	√
L8A8_UNORM	16 bit	8x		√	√		
A8L8_UNORM	16 bit	8x		√			
R16_UNORM	16 bit	8x		√	√	√	√
R16_SNORM	16 bit	8x		√	√	√	√
R16_FP16	16 bit	8x		√	√	√	√
R16_UINT	16 bit	8x		√	√	√	√
R16_SINT	16 bit	8x		√	√	√	√
R8G8B8_UNORM	24 bit	1x	√	√			
R8G8B8_SNORM	24 bit	1x		√			
R8G8B8_UINT	24 bit	1x		√			
R8G8B8_SINT	24 bit	1x		√			
R8G8B8A8_UNORM	32 bit	8x	√	√	√	√	√
R8G8B8X8_UNORM	32 bit	8x	√	√	√	√	√
R8G8B8A8_SNORM	32 bit	8x		√	√	√	√
R8G8B8A8_UINT	32 bit	8x		√	√	√	√
R8G8B8A8_SINT	32 bit	8x		√	√	√	√
R9G9B9E5_FLOAT	32 bit	8x		√	√		
R10G10B10A2_UNORM	32 bit	8x		√	√	√	√
R10G10B10A2_UNORM_FAST	32 bit	8x		√	√	√	√
R10G10B10X2_UNORM_FAST	32 bit	8x		√	√	√	√
R10G10B10A2_SNORM	32 bit			√			
R10G10B10A2_UINT	32 bit	8x		√	√	√	√
R10G10B10A2_SINT	32 bit			√			
A2R10G10B10_UNORM	32 bit	8x		√			
A2R10G10B10_UNORM_FAST	32 bit	8x		√			
X2R10G10B10_UNORM_FAST	32 bit	8x		√			

A5X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBWC*
A2R10G10B10_SNORM	32 bit			√		
A2R10G10B10_UINT	32 bit	8x		√		
A2R10G10B10_SINT	32 bit			√		
R11G11B10_FLOAT	32 bit	8x		√	√	√
R16G16_UNORM	32 bit	8x		√	√	√
R16G16_SNORM	32 bit	8x		√	√	√
R16G16_FP16	32 bit	8x		√	√	√
R16G16_UINT	32 bit	8x		√	√	√
R16G16_SINT	32 bit	8x		√	√	√
R32_UNORM	32 bit			√		
R32_SNORM	32 bit			√		
R32_FP32	32 bit	8x		√	√	√
R32_UINT	32 bit	8x		√	√	√
R32_SINT	32 bit	8x		√	√	√
R32_S15_16_FIXED	32 bit			√		
R16G16B16_UNORM	48 bit	1x		√		
R16G16B16_SNORM	48 bit	1x		√		
R16G16B16_FP16	48 bit	1x		√		
R16G16B16_UINT	48 bit	1x		√		
R16G16B16_SINT	48 bit	1x		√		
R16G16B16A16_UNORM	64 bit	8x		√	√	√
R16G16B16A16_SNORM	64 bit	8x		√	√	√
R16G16B16A16_FP16	64 bit	8x		√	√	√
R16G16B16A16_UINT	64 bit	8x		√	√	√
R16G16B16A16_SINT	64 bit	8x		√	√	√
R32G32_UNORM	64 bit			√		
R32G32_SNORM	64 bit			√		
R32G32_FP32	64 bit	8x		√	√	√
R32G32_UINT	64 bit	8x		√	√	√
R32G32_SINT	64 bit	8x		√	√	√
R32G32_S15_16_FIXED	64 bit			√		
R32G32B32_UNORM	96 bit			√		
R32G32B32_SNORM	96 bit			√		
R32G32B32_UINT	96 bit	1x		√		
R32G32B32_SINT	96 bit	1x		√		
R32G32B32_FP32	96 bit	1x		√		
R32G32B32_S15_16_FIXED	96 bit			√		

A5X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBWC*	F
R32G32B32A32_UNORM	128 bit			√			
R32G32B32A32_SNORM	128 bit			√			
R32G32B32A32_FP32	128 bit	8x		√	√	√	√
R32G32B32A32_UINT	128 bit	8x		√	√	√	√
R32G32B32A32_SINT	128 bit	8x		√	√	√	√
R32G32B32A32_S15_16_FIXED	128 bit			√			
UYVY_UNORM	YUV packed (16 bit)	1x		√			
YUY2_UNORM (YUYV)	YUV packed (16 bit)	1x		√	√		
NV12_UNORM	YUV planar (8/16 bit)	1x		√	√		
NV21_UNORM	YUV planar (8/16 bit)	1x		√			
IYUV_UNORM	YUV planar (8 bit)	1x		√			
Y8U8V8A8_UNORM	YUV packed (32 bit)	1x		√	√	√	
YVYU_UNORM	YUV packed (16 bit)	1x		√			
VYUY_UNORM	YUV packed (16 bit)	1x		√			
R24_UNORM_X8_TYPELESS	Depth/stencil (32 bit)	8x			√	√	√
ATI_TC_RGB	Compressed (64 bit)	1x	√	√	√		
ATI_TC_RGBA	Compressed (128 bit)	1x	√	√	√		
ATI_TC_RGBA_INTERP	Compressed (128 bit)	1x	√	√	√		
EACX2_RG11_UNSIGNED	Compressed (128 bit)	1x		√	√		
EACX2_RG11_SIGNED	Compressed (128 bit)	1x		√	√		
EAC_R11_UNSIGNED	Compressed (64 bit)	1x		√	√		
EAC_R11_SIGNED	Compressed (64 bit)	1x		√	√		
ETC1_RGB	Compressed (64 bit)	1x	√	√	√		
ETC2_RGB8	Compressed (64 bit)	1x	√	√	√		
ETC2A_RGBA8	Compressed (128 bit)	1x	√	√	√		
ETC2_PTA_RGBA8	Compressed (64 bit)	1x	√	√	√		
BC1_UNORM	Compressed (64 bit)	1x	√	√	√		
BC2_UNORM	Compressed (128 bit)	1x	√	√	√		
BC3_UNORM	Compressed (128 bit)	1x	√	√	√		
BC4_UNORM	Compressed (64 bit)	1x		√	√		
BC4_UNORM_FAST	Compressed (64 bit)	1x		√	√		
BC4_SNORM	Compressed (64 bit)	1x		√	√		
BC4_SNORM_FAST	Compressed (64 bit)	1x		√	√		
BC5_UNORM	Compressed (128 bit)	1x		√	√		
BC5_UNORM_FAST	Compressed (128 bit)	1x		√	√		
BC5_SNORM	Compressed (128 bit)	1x		√	√		
BC5_SNORM_FAST	Compressed (128 bit)	1x		√	√		

A5X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBWC*
BC6H_UFP16	Compressed (128 bit)	1x		√	√	
BC6H_SFP16	Compressed (128 bit)	1x		√	√	
BC7_UNORM	Compressed (128 bit)	1x	√	√	√	
ASTC_4X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X6	Compressed (128 bit)	1x	√	√	√	
ASTC_8X5	Compressed (128 bit)	1x	√	√	√	
ASTC_8X6	Compressed (128 bit)	1x	√	√	√	
ASTC_8X8	Compressed (128 bit)	1x	√	√	√	
ASTC_10X5	Compressed (128 bit)	1x	√	√	√	
ASTC_10X6	Compressed (128 bit)	1x	√	√	√	
ASTC_10X8	Compressed (128 bit)	1x	√	√	√	
ASTC_10X10	Compressed (128 bit)	1x	√	√	√	
ASTC_12X10	Compressed (128 bit)	1x	√	√	√	
ASTC_12X12	Compressed (128 bit)	1x	√	√	√	
ASTC_3X3X3	Compressed (128 bit)	1x	√	√	√	
ASTC_4X3X3	Compressed (128 bit)	1x	√	√	√	
ASTC_4X4X3	Compressed (128 bit)	1x	√	√	√	
ASTC_4X4X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X4X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X5X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X5X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X5X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X6X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X6X6	Compressed (128 bit)	1x	√	√	√	

A6X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBW

A6X

***** only when **optimal** layout is used.

A6X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBW
A1_UNORM	1 bit	1x		√		
A8_UNORM	8 bit	8x		√	√	
R8_UNORM	8 bit	8x	√	√	√	
R8_SNORM	8 bit	8x		√	√	
R8_UINT	8 bit	8x		√	√	
R8_SINT	8 bit	8x		√	√	
R4G4B4A4_UNORM	16 bit	8x		√	√	√
R4G4B4X4_UNORM	16 bit	8x		√	√	√
R5G5B5A1_UNORM	16 bit	8x		√	√	√
R5G5B5X1_UNORM	16 bit	8x		√	√	√
A1R5G5B5_UNORM	16 bit	8x		√		
X1R5G5B5_UNORM	16 bit	8x		√		
R5G6B5_UNORM	16 bit	8x		√	√	√
R8G8_UNORM	16 bit	8x	√	√	√	√
R8G8_SNORM	16 bit	8x		√	√	√
R8G8_UINT	16 bit	8x		√	√	√
R8G8_SINT	16 bit	8x		√	√	√
L8A8_UNORM	16 bit	8x		√	√	
A8L8_UNORM	16 bit	8x		√		
R16_UNORM	16 bit	8x		√	√	√
R16_SNORM	16 bit	8x		√	√	√
R16_FP16	16 bit	8x		√	√	√
R16_UINT	16 bit	8x		√	√	√
R16_SINT	16 bit	8x		√	√	√
R8G8B8_UNORM	24 bit	1x	√	√		
R8G8B8_SNORM	24 bit	1x		√		
R8G8B8_UINT	24 bit	1x		√		
R8G8B8_SINT	24 bit	1x		√		
R8G8B8A8_UNORM	32 bit	8x	√	√	√	√
R8G8B8X8_UNORM	32 bit	8x	√	√	√	√

A6X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	U
R8G8B8A8_SNORM	32 bit	8x		√	√	√
R8G8B8A8_UINT	32 bit	8x		√	√	√
R8G8B8A8_SINT	32 bit	8x		√	√	√
R9G9B9E5_FLOAT	32 bit	8x		√	√	
R10G10B10A2_UNORM	32 bit	8x		√	√	√
R10G10B10A2_UNORM_FAST	32 bit	8x		√	√	√
R10G10B10X2_UNORM_FAST	32 bit	8x		√	√	√
R10G10B10A2_SNORM	32 bit			√		
R10G10B10A2_UINT	32 bit	8x		√	√	√
R10G10B10A2_SINT	32 bit			√		
A2R10G10B10_UNORM	32 bit	8x		√		
A2R10G10B10_UNORM_FAST	32 bit	8x		√		
X2R10G10B10_UNORM_FAST	32 bit	8x		√		
A2R10G10B10_SNORM	32 bit			√		
A2R10G10B10_UINT	32 bit	8x		√		
A2R10G10B10_SINT	32 bit			√		
R11G11B10_FLOAT	32 bit	8x		√	√	√
R16G16_UNORM	32 bit	8x		√	√	√
R16G16_SNORM	32 bit	8x		√	√	√
R16G16_FP16	32 bit	8x		√	√	√
R16G16_UINT	32 bit	8x		√	√	√
R16G16_SINT	32 bit	8x		√	√	√
R32_UNORM	32 bit			√		
R32_SNORM	32 bit			√		
R32_FP32	32 bit	8x		√	√	√
R32_UINT	32 bit	8x		√	√	√
R32_SINT	32 bit	8x		√	√	√
R32_S15_16_FIXED	32 bit			√		
R16G16B16_UNORM	48 bit	1x		√		
R16G16B16_SNORM	48 bit	1x		√		
R16G16B16_FP16	48 bit	1x		√		
R16G16B16_UINT	48 bit	1x		√		
R16G16B16_SINT	48 bit	1x		√		
R16G16B16A16_UNORM	64 bit	8x		√	√	√
R16G16B16A16_SNORM	64 bit	8x		√	√	√
R16G16B16A16_FP16	64 bit	8x		√	√	√
R16G16B16A16_UINT	64 bit	8x		√	√	√

A6X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBW
R16G16B16A16_SINT	64 bit	8x		√	√	√
R32G32_UNORM	64 bit			√		
R32G32_SNORM	64 bit			√		
R32G32_FP32	64 bit	8x		√	√	√
R32G32_UINT	64 bit	8x		√	√	√
R32G32_SINT	64 bit	8x		√	√	√
R32G32_S15_16_FIXED	64 bit			√		
R32G32B32_UNORM	96 bit			√		
R32G32B32_SNORM	96 bit			√		
R32G32B32_UINT	96 bit	1x		√		
R32G32B32_SINT	96 bit	1x		√		
R32G32B32_FP32	96 bit	1x		√		
R32G32B32_S15_16_FIXED	96 bit			√		
R32G32B32A32_UNORM	128 bit			√		
R32G32B32A32_SNORM	128 bit			√		
R32G32B32A32_FP32	128 bit	4x		√	√	√
R32G32B32A32_UINT	128 bit	4x		√	√	√
R32G32B32A32_SINT	128 bit	4x		√	√	√
R32G32B32A32_S15_16_FIXED	128 bit			√		
UYVY_UNORM	YUV packed (16 bit)	1x		√		
YUY2_UNORM (YUYV)	YUV packed (16 bit)	1x		√	√	
NV12_UNORM	YUV planar (8/16 bit)	1x			√	√
NV21_UNORM	YUV planar (8/16 bit)	1x		√		
IYUV_UNORM	YUV planar (8 bit)	1x		√		
Y8U8V8A8_UNORM	YUV packed (32 bit)	8x		√	√	√
YVYU_UNORM	YUV packed (16 bit)	1x		√		
VYUY_UNORM	YUV packed (16 bit)	1x		√		
Y8_UNORM	Y/U/V planar (8 bit)3	1x			√	√
NV12_UV_UNORM	UV planar (16 bit)	1x			√	√
NV21_VU_UNORM	VU planar (16 bit)	1x		√		
NV12_4R_UNORM	YUV planar (8/16 bit)	1x		√	√	√
NV12_4R_Y_UNORM	Y planar (8 bit)	1x		√	√	√
NV12_4R_UV_UNORM	UV planar (16 bit)	1x		√	√	√
P010_UNORM	YUV planar (16/32 bit)	1x		√	√	√
P010_Y_UNORM	Y planar (16 bit)	1x		√	√	√
P010_UV_UNORM	UV planar (32 bit)	1x		√	√	√
TP10_UNORM	YUV planar ((32/3)/(32/3) bit)	1x		√	√	

A6X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	U
TP10_Y_UNORM	Y planar (32/3 bit)	1x		√	√	
TP10_UV_UNORM	UV planar (32/3 bit)	1x		√	√	
R24_UNORM_X8_TYPELESS	Depth/stencil (32 bit)	8x			√	√
ATI_TC_RGB	Compressed (64 bit)	1x	√	√	√	
ATI_TC_RGBA	Compressed (128 bit)	1x	√	√	√	
ATI_TC_RGBA_INTERP	Compressed (128 bit)	1x	√	√	√	
EACX2_RG11_UNSIGNED	Compressed (128 bit)	1x		√	√	
EACX2_RG11_SIGNED	Compressed (128 bit)	1x		√	√	
EAC_R11_UNSIGNED	Compressed (64 bit)	1x		√	√	
EAC_R11_SIGNED	Compressed (64 bit)	1x		√	√	
ETC1_RGB	Compressed (64 bit)	1x	√	√	√	
ETC2_RGB8	Compressed (64 bit)	1x	√	√	√	
ETC2A_RGBA8	Compressed (128 bit)	1x	√	√	√	
ETC2_PTA_RGBA8	Compressed (64 bit)	1x	√	√	√	
BC1_UNORM	Compressed (64 bit)	1x	√	√	√	
BC2_UNORM	Compressed (128 bit)	1x	√	√	√	
BC3_UNORM	Compressed (128 bit)	1x	√	√	√	
BC4_UNORM	Compressed (64 bit)	1x		√	√	
BC4_UNORM_FAST	Compressed (64 bit)	1x		√	√	
BC4_SNORM	Compressed (64 bit)	1x		√	√	
BC4_SNORM_FAST	Compressed (64 bit)	1x		√	√	
BC5_UNORM	Compressed (128 bit)	1x		√	√	
BC5_UNORM_FAST	Compressed (128 bit)	1x		√	√	
BC5_SNORM	Compressed (128 bit)	1x		√	√	
BC5_SNORM_FAST	Compressed (128 bit)	1x		√	√	
BC6H_UFP16	Compressed (128 bit)	1x		√	√	
BC6H_SFP16	Compressed (128 bit)	1x		√	√	
BC7_UNORM	Compressed (128 bit)	1x	√	√	√	
ASTC_4X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X6	Compressed (128 bit)	1x	√	√	√	
ASTC_8X5	Compressed (128 bit)	1x	√	√	√	
ASTC_8X6	Compressed (128 bit)	1x	√	√	√	
ASTC_8X8	Compressed (128 bit)	1x	√	√	√	
ASTC_10X5	Compressed (128 bit)	1x	√	√	√	

A6X Surface Format	Bit depth	Max MSAA	sRGB	Linear	Optimal	UBW
ASTC_10X6	Compressed (128 bit)	1x	√	√	√	
ASTC_10X8	Compressed (128 bit)	1x	√	√	√	
ASTC_10X10	Compressed (128 bit)	1x	√	√	√	
ASTC_12X10	Compressed (128 bit)	1x	√	√	√	
ASTC_12X12	Compressed (128 bit)	1x	√	√	√	
ASTC_3X3X3	Compressed (128 bit)	1x	√	√	√	
ASTC_4X3X3	Compressed (128 bit)	1x	√	√	√	
ASTC_4X4X3	Compressed (128 bit)	1x	√	√	√	
ASTC_4X4X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X4X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X5X4	Compressed (128 bit)	1x	√	√	√	
ASTC_5X5X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X5X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X6X5	Compressed (128 bit)	1x	√	√	√	
ASTC_6X6X6	Compressed (128 bit)	1x	√	√	√	
MIPI10_R_UNORM	MIPI packed 10 bit	1x		√		
MIPI10_R_UINT	MIPI packed 10 bit	1x		√		
MIPI12_R_UNORM	MIPI packed 12 bit	1x		√		
MIPI12_R_UINT	MIPI packed 12 bit	1x		√		
MIPI10_BAYER_UNORM	MIPI packed 10 bit	1x		√		
MIPI10_BAYER_UINT	MIPI packed 10 bit	1x		√		
MIPI12_BAYER_UNORM	MIPI packed 12 bit	1x		√		
MIPI12_BAYER_UINT	MIPI packed 12 bit	1x		√		
BAYER10_UNORM	16 bit	1x		√		
BAYER12_UNORM	16 bit	1x		√		
BAYER16_UINT	16 bit	1x		√		
BVH	128 bit	1x		√		
Y8_UINT	Y/U/V planar (8 bit)3	1x			√	√
NV12_UV_UINT	UV Planar (16bit)	1x			√	√
NV21_VU_UINT	VU Planar (16bit)	1x				
P010_Y_UINT	Y Planar (16bit)	1x			√	√
P010_UV_UINT	UV Planar (32bit)	1x			√	√
X24_TYPELESS_G8_UINT	Depth/stencil (32 bit)	8x			√	√

A5X GPUs support texture sizes up to 16384x16384x16384 (depending on memory availability).

Shaders

On A7X GPUs, every time a shader on the graphics queue uses more than a multiple of 2000 shader instructions, there is a potential performance impact. Compute shaders on LPAC have a different limit.

On A7X GPUs, every time a shader uses more than a multiple of 32 vertex buffers, there is a potential performance impact.

On A7X GPUs, every time a shader uses more than a multiple of 16 unique uniform buffers, there is a potential performance impact.

On A7X GPUs, each time a shader uses more than a multiple of 16 unique textures and SSBOs (summed), there is a potential performance impact.

On A5X GPUs, each time a shader uses more than a multiple of 4 unique samplers, there is a potential performance impact.

On A5X GPUs, one (1) texture per fragment at full rate is equivalent to 16 full precision ALUs.

Compute Shaders

On A7X GPUs, every time a compute shader on LPAC uses more than a multiple of 2256 shader instructions, there is a potential performance impact. Compute shaders on the Graphics queue have a different limit.

On A7X GPUs:

- every time the number of concurrent local groups exceeds a multiple of 64, there is a potential performance impact. If the shader spends a significant amount of time stalled, then the performance impact may occur for each multiple of 128 concurrent local groups
- there is another potential performance impact when the number of concurrent local groups exceeds a multiple of 2048

On A7X GPUs, every time the number of workgroups exceeds a multiple of 16 there is a potential performance impact. This multiple is 8 if the shaders read each others' memory

Queries

Occlusion queries

On A5X GPUs, no more than 512 occlusion queries should be active, with usually a 3 frame delay for results.

Adreno GPU on Mobile: Best Practices

Feature Summary

Adreno has a rich set of hardware and software features. Here is a quick-start overview of what's available, linking to more information:

- Renderer Architecture**: Make the most of Adreno's performance and battery usage by considering:**
  - renderpass and subpass management (including image layout, memory buffer usage, z-buffer setup and shader submission)
  - reduced fragment shader invocations
  - swapchain setup
  - less performant features to avoid
- Tile-based Rendering: FlexRender™, mid-frame, constantly chooses between binning/GMEM and direct/system-memory mode: optimize for both
- Render Surface Target: The image at the end of the frame involves sRGB and pixel formats, upscaling optimizations, and ensuring optimal layout
- Z-buffer: Depth buffers should use the optimal pixel format
- LRZ, Early-Z and Fast-Z: Hardware acceleration for depth buffering can dramatically improve performance
- Vertex and Index Buffers: Vertex and index buffer layouts impact performance, as does submission order of buffer changes
- Texture Features: Sampler setup and texture formats impact performance, as can leveraging driver-optimized texture features
- Shaders: Adreno's unified/scalar shader architecture enables many shader-level optimizations
- Compute Shaders: have some special considerations beyond other shaders
- GPU-Driven Rendering: involves using compute shaders to offload more work from the CPU onto the GPU
- Low Priority Asychronous Compute (LPAC): can efficiently perform some compute shaders concurrently with other processing on the Graphics Pipe
- OpenCL (download pdf): is supported
- 2D Operation Hardware Acceleration: is accessible through graphic APIs
- Queries: should be issued correctly to maximize performance and accuracy
- Qualcomm True HDR: mobile devices featuring OLED screens support a higher dynamic

range and wider color gamut: make use of this color depth

- Variable Rate Shading (VRS): allows a fragment shader to color one or more pixels at a time, so a fragment can represent one pixel or a group of pixels
- Raytracing: efficient simulation of light rays for beautiful imagery on low-power devices like mobile
- Querying the driver version: helps you implement workarounds for older drivers and devices

Best Practices Summary

Here is a quick-start overview of some of the most relevant development best practices, linking to more information:

Renderer Architecture

- Prefer Vulkan to OpenGL ES
- Follow Vulkan best practices
- Minimize renderpasses correctly
- Setup renderpasses efficiently
- Depth-only render efficiently
- Android phones: use swapchain efficiently
- Pass VK_PIPELINE_CREATE_LINK_TIME_OPTIMIZATION_BIT_EXT in shipping builds to maximize performance
- Use Vulkan Adreno Layer in nonshipping builds
- Use buffer best practices
- Separate graphics submits and compute dispatches
- Maximize indirect draw calls
- Avoid less performant features
- Use framerate extrapolation
- Ideal screenspace triangle size is at least 4 pixels, and not much larger than a bin

Tile-based Rendering

- Minimize the number of bins
- Prefer MSAA to other anti-aliasing techniques (for small-pixel form factors like mobile, consider no anti-aliasing)

Concurrent Binning

- Issue each draw call such that it produces enough fragment shader work to parallelize with the next draw call's binning
- Avoid VK_SHADER_STAGE_ALL when VK_SHADER_STAGE_VERTEX suffices
- Minimize dependencies – renderpass, barrier, Z-buffer-clear – that prevent concurrent binning
- Use the same Z-buffer (without clearing or invalidating it) between passes – or use separate Z-buffers per pass

Render Surface Target

- Use HDR (high dynamic range)
- Use the fastest possible sRGB pixel format
- Use the lowest render target resolution that looks good and upscale: prefer SGSR when possible, or frame buffer blits otherwise. On Android, consider relying on SurfaceFlinger's efficient bilinear rescale
- Maximize use of Variable Rate Shading

Z-Buffer

- Use the fastest possible depth format
- Don't disable LRZ (Low Resolution Z-Buffer) or Early Z Rejection

Vertex and Index Buffers

- Lay out vertex buffers optimally
- Minimize the size of vertex buffers
- Minimize the size of index buffers
- Batch vertex buffer object updates

Texture Features

Use ASTC texture compression in the sRGB format

Shaders

- Make instruction counts fit the instruction cache for binning, concurrent-binning, vertex, fragment, and compute shaders (LPAC compute shaders have a slightly larger instruction cache). Consider splitting up long shaders
- Minimize GPR (general purpose register) usage. Consider splitting up GPR-heavy shaders
- Don't reference too many unique vertex buffers, textures-and-SSBOs, samplers or uniform buffers
- Minimize texture samples in vertex shaders
- Separate graphics submits and compute dispatches
- Maximize half precision
- Minimize type casting
- Prefer built-in instructions
- Use hardware accelerated blits, Android-phone pre-rotation on Vulkan, convolution kernels, etc

Compute Shaders

- Most shader advice applies to compute shaders
- Prefer fragment shaders to compute shaders
- Instruction counts and GPR limits are similar – but not identical – to other shaders
- Minimize compute thread synchronization – when it's necessary, prefer atomics
- Tune workgroup sizes and number of workgroups, taking into account shader stalls and whether the shaders read each others' memory

Queries

- Use occlusion queries correctly for accuracy and efficiency
- Use GPU timestamp queries correctly for accuracy

Raytracing

- Minimize calls to rayQuery Proceed()
- Avoid proceed() calls in loops for non-opaque traversal that “accept first fit”
- Use only one ray query object; reuse as needed
- Access ray query data through intrinsics
- Prefer building acceleration structures on the CPU

Renderer Architecture

Here are some overarching issues to keep in mind while architecting a renderer that will make the most of Adreno’s performance and battery usage.

Graphics API

Prefer Vulkan to OpenGL ES. While Vulkan is not a perfect superset of OpenGL ES, in most respects it is more performant, more debuggable, and more fully featured.

Render Pass

Minimize the number of render passes – for example, any time several consecutive passes use the same formatted color buffer, combine them (disabling depth and/or stencil if one or both are unused). Snapdragon Profiler shows how renderpasses and subpasses are (or are not) merged on its Rendering Stages metric.

When combining the results of multiple passes (as in deferred rendering or blending transparent objects with opaque objects), prefer using alpha blending to the alternatives (such as the discard instruction, shader branching, stencil testing and compute shaders).

Many applications are fragment-shader bound, and Adreno has many tools to optimize such applications:

- Render target resolutions can be efficiently upscaled in several ways
- Variable Rate Shading shades fewer pixels
- Framerate Extrapolation can also nearly double a fragment-bound application’s framerate

Depending on content, these features can involve little to no visual quality degradation.

When performing depth-only rendering (like a depth-prepass), use an empty fragment shader and disable frame buffer writes to leverage Fast-Z.

Invalidate framebuffer contents as early as possible, so the driver doesn't wastefully resolve GMEM render target memory to system memory:

Vulkan

To invalidate framebuffer contents as early as possible, correctly use VK_ATTACHMENT_LOAD_OP_CLEAR and VK_ATTACHMENT_LOAD_OP_DONT_CARE on your renderpass (and VK_QCOM_render_pass_shader_resolve if a nonstandard shader resolve is desired).

Correct subpass handling is also essential.

Vulkan introduced render 'subpasses', which allow developers to set up render pipelines that explicitly state their usage, render target interactions, dependencies, transitions, etc. This allows GPUs to make informed decisions about how to handle these frame buffer transitions efficiently. To efficiently use GMEM, proper subpass use is crucial in tile-rendering architectures such as the Adreno GPU.

A properly structured renderpass allows Vulkan to instruct the GPU to execute all subpasses on a per-tile basis. That is, the full subpass chain can be executed for each tile, thus avoiding the need to resolve subpasses to system memory after each pass. Proper setup of these subpasses is required for the Vulkan driver to "merge" the subpasses into one. This can result in gains of over 10% frametime depending on subpass chain complexity and configuration.

Generally, a good Vulkan renderpass involves the following:

- Subpass count > 1
- Renderpass has input targets
- Each resolve attachment is used in an exactly one subpass
- *srcAccessMask* is not *VK_ACCESS_SHADER_WRITE_BIT*, and *dstAccessMask* is not *VK_ACCESS_SHADER_READ_BIT*
- Starting from the second subpass where the *input_attachments* field is used, the *dstAccessMask* must be set to *VK_ACCESS_INPUT_ATTACHMENT_READ_BIT*

Note: Subpass merging only applies when the given surface is being rendered in binning mode. Snapdragon Profiler 'Rendering Stages' metric in Trace capture identifies the mode these surfaces are being rendered with, and if proper merging has been done. Additionally, using the Vulkan Adreno Layer can also help identify if subpasses were not be merged properly by logging the *VKDBGUTILWARN003* flag.

Sample Code:

- SubPass
- Tonemapping Efficiently

OpenGL ES

To invalidate framebuffer contents as early as possible, there are several options:

- glInvalidateFramebuffer() states that this framebuffer is not being used until further notice, and ensures no GMEMLoad's or GMEMStore's will be performed with the frame buffer – until the developer makes further API calls on it
  - glInvalidateSubFramebuffer() makes a similar guarantee for a subrectangle of the framebuffer
- glClear() minimizes GMEMLoad's and GMEMStore's performed with the frame buffer – while memset'ing pixel memory as required. Sample Code: Avoid GMEMLoads
- EXT_discard_framebuffer can also be used to avoid GMEMLoad's
- Minimizing GMEMStore's performed with the frame buffer is also important: Sample Code: Minimize GMEMStores

Vulkan

Pass VK_PIPELINE_CREATE_LINK_TIME_OPTIMIZATION_BIT_EXT in shipping builds to maximize performance.

Prefer uniform buffers instead of push constants for performance.

OpenGL ES

- Avoid glClientWaitSync() and glFenceSync() (unless performing GPU write-back, which is discouraged for performance reasons) – the CPU needn't be explicitly told to wait on the GPU, since glSwapBuffers() already handle this
- Use the latest version of OpenGL ES – there are numerous reasons (bugfixes and optimizations) to be on the latest, and few reasons not to be
- Use the KHR_No_Error gles extension for shipping builds to maximize performance

Shader mode switching

Minimize pipeline and compute kernel switching to avoid unnecessary internal synchronization – and particularly minimize alternating between graphics submits and compute dispatches.

In other words, separate graphics submits and compute dispatches as much as possible – ideally issue a series of only graphics submits followed by a series of only compute dispatches.

And even within the phase (or, possibly less optimally, phases) of the frame where graphics submits take place, minimize interleaving pipelines – for example, prefer (SubmitGraphicsPipelineA 2 times, SubmitGraphicsPipelineB) to (SubmitGraphicsPipelineA, SubmitGraphicsPipelineB, SubmitGraphicsPipelineA). The same rationale applies to dispatching compute kernels.

Images

Vulkan image layout datastructures should be as specific as possible. This is significantly more important for performance on Adreno hardware than on many other GPU's.

VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT should be avoided as much as possible on all devices prior to Adreno750.

Buffer Best Practices

Flag all buffers read-only as much as possible.

Prefer Vertex Buffer Objects (VBOs) when possible.

Otherwise prefer uniform buffers (UBOs) provided the sum of their sizes for a given shader remain under 90% of 8K – 0.9*8192 = 7372 bytes. Note that the maximum size reported by graphics APIs (in Vulkan, vkPhysicalDeviceLimits::maxUniformBufferRange) is only a correctness limit – not a performance limit. Note that the sum of all uniform buffers used by a shader – not just each uniform buffer individually – must stay under this limit to avoid this possible performance reduction.

For larger data sizes prefer textures over Shader Storage Buffer Objects (SSBOs).

If a UBO exceeds its optimal size, the compiler will attempt to determine which portion of the UBO may be accessed by the shader, and map only those portions of the UBO to constant RAM. Dynamic or indirect indexing might prevent this optimization, so if your UBO might exceed its optimal size (and you choose not to use textures or SSBOs), prefer static indexing.

Vulkan: use VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT for buffers that are not read from outside of the renderpass (especially MSAA attachments, which are larger than non-MSAA attachments). For example, a Z-buffer that exists only to be cleared and used for typical z-buffering within a single renderpass should use this flag.

Vulkan

Avoid less performant features of the API

If you encounter VK_DEVICE_LOST, calling VK_EXT_device_fault is your only option – calling any other part of the API is undefined.

Vulkan image layout datastructures should be as specific as possible.

Swapchain

- **On Android phones:**
  - use VK_PRESENT_MODE_FIFO_KHR and minImageCount=3 to most efficiently utilize the GPU (VK_PRESENT_MODE_MAILBOX_KHR can sometimes help with frame pacing/latency, but often can cost significant battery and generate significant heat for no benefit, as it may renders frames that are not presented to the player)
  - Vulkan: use prerotation

Triangle Screen Size

Ideally every triangle rasterized shades at least 4 pixels; tune level-of-detail (LOD) systems and content accordingly.

If a triangle spans multiple tiles in binning mode, the full triangle will be rasterized per tile – there are no added vertices at tile boundaries. Therefore, many triangles much larger than the bin size in screenspace can be inefficient.

Features to avoid for performance reasons

- **Vulkan:**
  - VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT: since the driver doesn't know which view formats will be paired with the image, this often reduces performance
  - VK_EXT_conditional_rendering is unlikely to improve performance unless it's skipping a large amount work, as its overhead is substantial
  - VK_EXT_vertex_input_dynamic_state should be avoided when possible. Static pipelines are strictly more performant
  - VK_EXT_vertex_input_dynamic_state should be used minimally – static pipelines are strictly more performant
- Tessellation hardware stages (hull shader, tessellator, domain shader)
- Client-side vertex arrays

- User clip planes

Tile-based Rendering

Bin Minimization

Some performance bottlenecks for a render target can be alleviated by reducing the number of bins the driver generates (which, in turn, often increases the number of pixels each bin contains). The developer has several options:

- reduce frame buffer resolution
- use Variable Rate Shading (including foveated rendering) to render fewer fragments
- use fewer MSAA samples
- render to fewer render targets at once
- simplify vertex shaders
- use vertex buffer best practices

FlexRender™ technology (Hybrid Deferred and Direct Rendering mode)

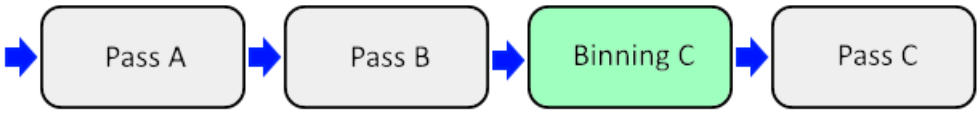
The driver heuristics that determine when to run binned or direct mode are not exposed to the developer, but generally these scenarios trigger direct mode:

- High ratio of texture samples in vertex shaders to vertices
- Small number of vertices and/or draws
- Use of tessellation or geometry shaders

Concurrent Binning

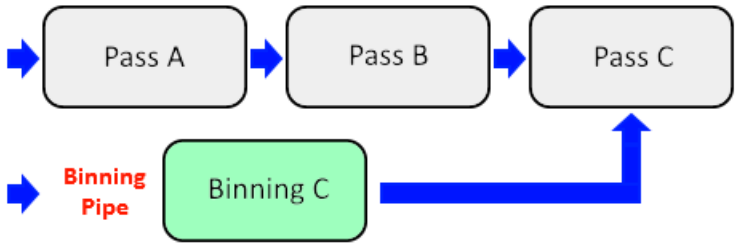
No additional steps are required to activate concurrent binning – but care should be taken to maximize its benefits.

Sequential render passes with dependencies might require synchronous binning – concurrent binning is only possible when there are “bubbles” in the render graph. Consider the renderpass chain:



Pass C uses binning but – unfortunately – its binning process depends on *Pass B*, which depends on *Pass A*.

If we remove the binning dependencies from *Pass B* – so *Pass C*'s vertex shader doesn't require any inputs from *Pass B* (perhaps by revisiting renderpass setup and barriers) – the driver might asynchronously perform binning during *PassB* and/or *PassC* on the concurrent binning pipe:



Another way concurrent binning can be prevented is reusing the same Z-buffer attachment with clears within a frame (you might issue these clears if, for example, the Z-buffer is being used for multiple purposes within a frame). These clears define dependencies, and thus prevent concurrent binning for every render pass or compute operation that uses this Z-buffer attachment.

Try to use the same Z-buffer – without clears or invalidations – over multiple render passes to allow concurrent binning. (If this is not possible, giving each renderpass its own Z-buffer also allows concurrent binning, though this risks using more memory bandwidth and costs memory)

Another common case that fails to leverage concurrent binning is when the application is VSYNC limited, and the first surface processes all the geometry. In such a scenario, try to schedule some independent work before submitting that geometry-heavy render pass so both the independent work and the first surface's geometry binning can happen in parallel.

Vulkan: Minimize the use of VK_SHADER_STAGE_ALL – exclusively prefer VK_SHADER_STAGE_VERTEX on descriptor bindings that are accessed only in the vertex stage.

Render Surfaces

sRGB textures and render targets

Use SRGB textures:

Vulkan

Vulkan fully handles sRGB in both textures and swapchain presentable images.

OpenGL ES

Unfortunately, OpenGL ES assumes linear or RGB color space by default. As Adreno GPUs support sRGB color space for render targets and textures, it is possible to ensure a correct color viewing experience with some extra work.

The best-performing color format is:

- R10G10B10A2, as the hardware is optimized for this format
- R11G11B10A0 if even greater color depth is required, and alpha transparency is not
- RGBA16 if graduated transparency is required, or the (considerable) color depth of the above formats is insufficient

Vulkan Sample: Qualcomm TrueHDR

Upscaling

To reduce the load on the rendering hardware, an application can reduce the size of the render target used, e.g., if the native screen resolution is 1080p (1920x1080), it could be rendered to a 720p (1280x720) render target instead. Since the aspect ratio of the two resolutions is identical, the proportions of the image will not be affected.

The reduced-size render target will not completely fill the screen, but there are numerous ways of upscaling the reduced-size render target to match the full native display size.

This technique has been used successfully in console games, many of which make heavy demands on the GPU and, if rendering were done at full HD resolution, could miss framerate targets due to too much time spent during fragment shading.

Upscale to improve fragment-bound frames with:

Snapdragon Game Super Resolution

Snapdragon GSR is a single pass spatially-aware super resolution technique developed by Qualcomm Snapdragon Studios to achieve optimal super scaling quality at the best performance and power savings – it uses optimized image processing to, for most content, achieve a sharper, higher-quality image than the typical bilinear filtering approaches.

Android

For applications written in Java, configure the fixed-size property of the GLSurfaceView instance (available since API level 1). Set the property using the setFixedSize function, which takes two arguments defining the resolution of the final render target.

For applications written in native code, define the resolution of the final render target using the function NativeWindow_setBuffersGeometry, which is a part of the NativeActivity class, introduced in Android 2.3 (API level 9).

At every swap operation, the operating system takes the contents of the final render target and scales it up so that it matches the native resolution of the display.

You can also do this yourself with your preferred graphics API. The upscaling can be done either at the end of the rendering process or at some point in the rendering pipeline – for example, one approach is to render the geometry at 1:1 resolution, but apply postprocessing effects using render targets of a lower resolution.

Vulkan

vkCmdBlitImage() provides a hardware-accelerated path to upscaling an image prior to display.

OpenGL ES

OpenGL ES 3.0 introduced support for frame buffer blit operations, so the contents of a draw buffer can be blit from one frame buffer to another. As part of the blit operation, the API also supports upscaling, which can be used to copy the contents of a texture of a smaller resolution to another texture of a larger resolution.

Note: Upscaling using a frame buffer blit is faster than the alternative approach of rendering a full screen quad directly to the back buffer, taking the reduced-size render target as a texture input.

On Android, at the end of the frame, if the final render target is a different resolution than the device's native resolution then SurfaceFlinger will bilinearly rescale the final image to native resolution with efficiency.

Bandwidth Optimization

Applications becoming memory-bandwidth limited can be a bottleneck due to the physical limitation of the GPU's data access rate. The rate is not constant and varies according to many factors, including but not limited to:

1. Location of the data: is it stored in RAM, VRAM, or one of the GPU caches?
2. Type of access: is it a read or a write operation? Is it atomic? Does it have to be coherent?
3. Viability of caching the data: can the hardware cache the data for subsequent operations that the GPU will be carrying out, and would it make sense to do this?

Cache misses can cause applications to become bandwidth-limited, which significantly reduces performance. These cache misses are often caused when applications draw or generate many primitives, or when shaders need to access many locations within textures.

Here are two ways to minimize cache misses:

1. Improve the transfer speed rate: ensure that client-side vertex data buffers are used for as few draw calls as possible – ideally, an application should never use them
2. Reduce the amount of data the GPU needs to access to perform the dispatch or draw call that is hitting this constraint

Graphics APIs provide several methods that developers can use to reduce the bandwidth needed to transfer specific types of data.

The first method is compressed texture internal formats, which sacrifice texture quality for the benefit of reduced size. Many of the compressed texture formats supported divide the input image into 4x4 blocks and perform the compression process separately for each block, rather than operating on the image as a whole. While this seems inefficient from the point of view of data compression theory, doing so has the advantage of each block being aligned on a 4-pixel boundary. This allows the GPU to retrieve more data with a single fetch instruction, because each compressed texture block holds 16 pixels instead of a single pixel, as in the case of an uncompressed texture. Also, the number of texture fetches can be reduced, provided that the shader does not need to sample texels that are too far apart.

The second method is to always use packed vertex data formats, unless the vertex data sets would suffer greatly from a reduction in the precision of their components.

The third method is to always use indexed draw calls, and always use an index type that is as small as possible while still being able to address all the vertices for the mesh being drawn. This reduces the amount of index data that the GPU needs to access for each draw call, at the expense of slightly more complicated application logic.

Z-Buffer

The best-performing depth format is:

- D16, if stencil is unnecessary and this provides sufficient precision – which, in many cases where the developer is tuning a z-reverse implementation appropriately to content, it will
- D24_S8, if stencil is required
- D32, if stencil is not required and D16 does not provide sufficient precision

UBWC supports all these formats.

LRZ, Early-Z and Fast-Z

Low Resolution Z pass

Warning: Several conditions cause the driver to disable LRZ for a period of time between two operations.

LRZ (test and write operations) will be disabled until the next API-surface-clear by performing any of the below, and then executing a depth-write:

- changing the depth direction (eg the sign of the forward axis)
- setting the depth function to ALWAYS or NOT_EQUAL

LRZ write operations will be disabled until next API-surface-clear (note that test operations on the existing LRZ buffer will still be enabled) by performing:

- any of the below, and then later performing a depth-write in the same draw:
  - blending implemented with:
    - the fixed-function pipeline
    - logical operations that read from the framebuffer
  - a color-masked write (eg writing to a subset of a buffer's color channels using graphics API calls rather than shader code)
  - a partial Multiple Render Target (MRT) write (eg writing to a subset of all specified render targets)
- in any subpass, a depth-write and reading from any attachment modified in any prior subpass of this renderpass (the order of operations is irrelevant – it doesn't matter if the depth-write happens before or after the read from the previously-modified-attachment)
- any stencil operation (reads or writes) in almost any situation

- a framebuffer fetch
- “advanced” blending with extensions

LRZ (test and write operations) will be disabled for the current draw (note that by default the next draw will fully re-enable LRZ) if the fragment shader writes to:

- a UAV (any kind of buffer)
- depth or stencil

LRZ write operations will be disabled for the current draw (note that test operations on the existing LRZ buffer will still be enabled, and that by default the next draw will fully re-enable LRZ) if:

- the draw’s active pipeline has alpha-to-coverage enabled
- a fragment-shader:
  - uses discard
  - outputs sample coverage

Vulkan: LRZ (test and write operations) will be disabled if secondary command buffers are used (this limitation applies only for devices prior to Adreno650)

LRZ is fully active during direct rendering (presuming it isn’t disabled for some other reason), but typically produces much less performance benefit than with binned rendering

Early Z rejection

Warning: Early Z-rejection is disabled if:

- Z-Buffer is written to from a fragment shader
- “Discard” shader instruction is used
- A fragment shader writes depth or stencil values – if these writes are necessary, perform them as late in the frame as possible
- Alpha-to-Coverage is enabled (minimize the use of this feature)

Adreno GPUs can reject occluded pixels at up to 4x the drawn pixel fill rate.

Fast-Z

Hint the driver to engage Fast-Z by using an empty fragment shader and disabling frame buffer write masks for renderpasses that modify Z values only.

If fast-Z doesn't activate by default for indirect draws, it can be switched on with glsl.

Vertex and Index Buffers

Vertex buffer layout

Pass a single vertex buffer to the vertex shader.

The vertex buffer should begin with an interleaved array of all attributes that are used by the vertex shader to calculate final vertex positions – often this is nothing more than vertex positions arranged like: (xyz|xyz) – followed by one interleaved array of all other attributes.

Compress all attributes as much as possible.

For any asset with a coordinate range known in advance, try to map the data onto one of the supported, packed vertex data formats. Taking normal data as an example, it is possible to map XYZ components onto the GL_UNSIGNED_INT_2_10_10_10_REV (OpenGL ES) format by normalizing the 10-bit unsigned integer data range of <0, 1024> onto the floating-point range <-1, 1>.

Vulkan

Vulkan does not yet support half-precision in vertex shaders.

OpenGL ES

OpenGL ES supports half-precision in vertex shaders with GL_OES_vertex_half_float.

The Adreno GPU provides hardware support for the following vertex formats:

- GL_BYTE and GL_UNSIGNED_BYTE
- GL_SHORT and GL_UNSIGNED_SHORT
- GL_FIXED
- GL_FLOAT
- GL_HALF_FLOAT
- GL_INT_2_10_10_10_REV and GL_UNSIGNED_INT_2_10_10_10_REV

Always use the vertex format that provides acceptable precision and takes the least amount of

space.

The compiler will usually optimize the binning vertex shader to use only the vertex attributes the vertex shader – called a position-only vertex shader – that needs to compute final vertex positions – any attributes that are simply forwarded to the fragment shader (by Vulkan's "layout"/"out" or OpenGL's "varying/flat") will be "refactored" into code that runs immediately before the fragment shader, resulting in optimal performance.

In addition to the above, vertex information can be laid out in a cache-friendly way over a range of devices, such that triangles sharing the same vertex are more often clustered together – doing so can reduce vertex cache misses and increase performance.

Batching vertex buffer object updates

If you must modify Vertex Buffer Object contents on-the-fly when rendering a frame, batch as many of the VBO updates as possible before issuing any draw calls that use the modified VBO region. If using multiple VBOs, batch the updates for all the VBOs first, and then issue all the draw calls.

Not doing so might cause the driver to maintain multiple copies of an entire VBO, which reduces performance.

Index types

A geometry mesh can be represented by two separate arrays. One array holds the vertices, and the other holds sets of three indices into that array. Together, they define a set of triangles.

Adreno GPUs natively support 8-bit, 16-bit, and 32-bit index types.

Prefer 8-bit indices when possible, 16-bit indices when necessary, and try to avoid 32-bit indices.

Texture Features

Sampling in Vulkan

Use VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER to access the GPU's more performant Bindless mode. We've seen separate samplers exhibit 2-5% less fill rate compared to combined samplers.

Multiple textures

Multiple texturing or *multitexturing* is the use of more than one texture at a time on a polygon.

Effective use of multiple textures significantly reduces overdraw, saves ALU cost for fragment shaders, and avoids unnecessary vertex transforms.

Large texture size

Devices have different maximum dimensions for textures.

Texture compression

See Texture Compression Examples.

Floating point textures

Adreno GPUs support floating point texturing features including the following:

- Texturing and linear filtering of FP16 textures via the GL_OES_texture_half_float and GL_OES_texture_half_float_linear extension
- Texturing from FP32 textures via GL_OES_texture_float

For a complete listing of supported texture and surface formats, refer to the Texture formats Feature Table.

Video textures

Video textures are a standard API feature in Android (Honeycomb or later versions). Refer to Android documentation for additional details on surface textures at http://developer.android.com/reference/android/graphics/SurfaceTexture.html.

Apart from using the standard Android API as suggested, the standard OpenGL ES extension can also be used, e.g., if an application requires video textures. For more information, refer to http://www.khronos.org/registry/gles/ extensions/OES/OES_EGL_image.txt.

Shaders

Half-Precision

Adreno's scalar architecture can be twice as power-efficient and deliver twice the performance while processing a fragment shader – if that fragment shader uses medium-precision 16-bit floating point (mediump) processing instead of high-precision 32-bit (highp) floating point.

Use strict half-precision types as much as possible. When necessary, relaxed-precision will often produce 16-bit floating point code.

Instruction count

A shader's instruction count fitting the instruction cache is usually critical for avoiding shader stalls and thus achieving optimal performance.

If a shader must exceed a target device's instruction limit (particularly if that shader rarely stalls on texture fetches and other slow operations that might allow the driver to swap in a different shader), consider splitting the shader into multiple parts.

Compute shaders that run on LPAC have a slightly higher instruction limit than other shaders. Compute shaders than run on the Graphics Pipeline have the typical (slightly lower) limit.

A low % Wave Context Occupancy can indicate long shaders thrashing the instruction cache.

GPR minimization

Keeping every shader's register usage (called GPRs or General Purpose Registers) under the device limits will ensure that the maximum number of simultaneous waves execute, maximizing performance.

Modifying GLSL to save even a single instruction can save a GPR. Not unrolling loops can also save GPRs, but that is up to the shader compiler. Always profile shaders to make sure the final solution chosen is the most efficient one for the target platform. Unrolled loops tend to put texture fetches toward the shader top, resulting in a need for more GPRs to hold the multiple texture coordinates and fetched results simultaneously.

For example, if unrolling the loop presented below:

```
for (i = 0; i < 4; ++i) {
    diffuse += ComputeDiffuseContribution(normal, light[i]);
}
```

The code snippet would be replaced with:

```
diffuse += ComputeDiffuseContribution(normal, light[0]);
diffuse += ComputeDiffuseContribution(normal, light[1]);
```

```
diffuse += ComputeDiffuseContribution(normal, light[2]);
diffuse += ComputeDiffuseContribution(normal, light[3]);
```

Avoid “uber-shaders” – they combine multiple shaders into a single shader that uses static branching. Using them makes sense if trying to reduce state changes and batch draw calls. However, this often increases GPR count, which can reduce performance.

If GPR usage is too high, consider splitting the shader into multiple parts.

A low % Wave Context Occupancy can indicate thrashing GPRs with too much register usage.

Split up draw calls

If a shader has too many instructions to fit the Instruction Cache, or uses too many GPRs, splitting it into multiple shaders may improve performance.

Vulkan’s subpasses, correctly authored, keep intermediate results in GMEM, and is often the fastest approach.

Alternatively, values generated by ShaderA can be written to a texture or SSBO and later retrieved via by ShaderB – or the results of ShaderA can be alpha-blended into the results of ShaderB. Be aware that this approach risks a memory bandwidth bottleneck, so make sure there is plausible headroom in % Texture Pipes Busy before attempting this optimization, and profile before and after making the change.

Minimize ALU Cost

Even when a shader fits within the instruction cache, instruction choices involving type-casting (including converting floating point values from 32-bit to 16-bit precision), control flow (branches and loops), built-in shader instructions and more all impact ALU efficiency.

Minimize type casting

Minimize the number of type cast operations performed.

For example, assigning a float to a vec4 data type could prevent the compiler from performing optimizations.

For another example, the following code might be suboptimal:

```
uniform sampler2D ColorTexture;
in vec2 TexC;
vec3 light(in vec3 amb, in vec3 diff)
{
    vec3 Color = texture(ColorTexture, TexC);
```

```
    Color *= diff + amb;
    return Color;
}
```

Here, the call to the texture function returns a vec4. There is an implicit type cast to vec3, which requires an additional instruction. Changing the code as follows might not require the additional instruction:

```
uniform sampler2D ColorTexture;
in vec2 TexC;
vec4 light(in vec4 amb, in vec4 diff)
{
    vec4 Color = texture(Color, TexC);
    Color *= diff + amb;
    return Color;
}
```

For another example, the following code should take a single instruction:

```
int4 ResultOfA(int4 a) {
    return a + 1;
}
```

Now suppose a slight error is introduced into the code. For the example, the floating-point constant value 1.0 is used, which is not the appropriate data type:

```
int4 ResultOfA(int4 a) {
    return a + 1.0;
}
```

The code could now require eight instructions: the variable *a* is converted to vec4, then, the addition is done in floating point. Finally, the result is converted back to the return type int4.

This discussion generally applies to converting floating point values from 32-bit to 16-bit precision as well.

Control Flow

Minimizing non-uniform looping and branching often helps reduce GPR and ALU usage.

Every time the branch encounters divergence, or where some elements of the thread branch one way and some elements branch in another, both branches will be taken and the results of the irrelevant branch ignored.

Here are the three types of branches, listed in order (starting with best performance):

- Branching on a constant, known at compile time
- Branching on a uniform variable
- Branching on a variable modified inside the shader

Branching on a constant may yield acceptable performance.

Pack shader interpolators

Shader-interpolated values (GLES varyings or Vulkan out variables) require a GPR (general purpose register) to hold data being fed into a fragment shader. Therefore, minimize their use.

Use constants where a value is uniform. Pack values together as all shader interpolated values have four components, whether they are used or not. Putting two vec2 texture coordinates into a single vec4 value is a common practice, but other strategies employ more creative packing and on-the-fly data compression.

Note: OpenGL ES 3.0 and ES 3.1 introduce various intrinsic functions to carry out packing operations.

Pack scalar constants

Packing scalar constants into 4-vectors – or, failing that, 2-vectors – substantially improves hardware fetch effectiveness.

Consider the following code:

```
float scale, bias;
vec4 a = Pos * scale + bias;
```

By changing the code as follows, fewer total instructions may be generated, because the compiler might replace several instructions with the more efficient “mad” instruction:

```
vec2 scaleNbias;
vec4 a = Pos * scaleNbias.x + scaleNbias.y;
```

In this case, while the compiler might infer that the *scale* and *bias* variables should be stored in a single GPR to enable the “mad” instruction, it’s much more likely to store *scaleNbias* in a single GPR, which in turn increases the likelihood of optimal instruction generation.

Note: OpenGL ES 3.0 and ES 3.1 introduce various intrinsic functions to carry out packing operations.

Use built-in shader instructions

Built-in functions are an important part of the glsl specification and should always be preferred to writing custom implementation. These functions are often optimized for specific shader profiles and for the capabilities of the hardware for which the shader was compiled.

Note: gl_VertexID and gl_InstanceID are removed as per the GL_KHR_vulkan_glsl extension, but gl_InstanceIndex is available.

Note: gl_Position, gl_PointSize, gl_ClipDistance, gl_CullDistance are available in non-fragment stages

---

Refer to the GL_KHR_vulkan_glsl extension for details on changes to GLSL built-ins in Vulkan.

Avoid discarding pixels in the fragment shader

Some developers believe that manually discarding (also known as killing) pixels in the fragment shader boosts performance. However:

- If some pixels in a thread are killed and others are not, the shader still executes
- It’s hard to predict how the shader compiler will generate microcode involving discard, so even in the unlikely case where performance is increased on one device or driver, this gain may be reversed on another device or driver

In theory, if all pixels in a thread are killed, the GPU will stop processing that thread as soon as possible. In practice, “as soon as possible” isn’t soon enough to help performance, and discard operations – like modifying depth in fragment shaders often disable hardware optimizations.

If a shader cannot avoid discard operations, the cost can be mitigated by executing the shader after (ideally all) opaque draw calls – generally, the later in the frame a discard operation takes place, the better.

Avoid modifying depth in fragment shaders

Similar to discarding fragments, modifying depth in the fragment shader can disable hardware optimizations.

Stay under performance-optimal resource limits

Stay under target device performance limits with respect to the number of unique resources referenced by a shader:

- vertex buffers
- uniform buffers
- samplers
- textures and SSBOs (textures and SSBOs share hardware resources)

Minimize texture fetches in vertex shaders

Adreno is based on a unified shader architecture, which means vertex processing performance is similar to fragment processing performance. However, for optimal performance it is best to minimize texture sampling in vertex shaders – and when this cannot be avoided, it is important to ensure that texture fetches in vertex shaders are localized (eg cache coherent) and always operate on compressed texture data.

Latency Hiding and the Texture Pipe

To hide latency, and thus avoid shader stalls and increase performance, the Snapdragon™ Adreno™ GPU shader compiler converts some memory access requests into texture fetches.

To allow this, the data buffer must be read-only and a storage type – in Vulkan, this means a SSBO or Shader Storage Buffer Object (VK_DESCRIPTOR_TYPE_STORAGE_BUFFER).

Even with a read-only storage buffer, the Adreno driver will sometimes choose a global memory access over a texture fetch if it decides that the texture pipe is a bottleneck. Snapdragon Profiler can profile texture pipe usage.

Texture Fetch Bottleneck Optimization

If texture fetches are a performance bottleneck, here are some optimization strategies:

- When possible, use vertex buffer objects instead of storage buffers. Note that most GPU-driven rendering architectures are more likely to bottleneck the texture pipe, since they typically use storage buffers instead of vertex buffers
- Use uniform buffers objects instead of textures or storage buffers if the data fits in the optimal uniform buffer size.
- Minimize texture fetches
- Minimize texture cache misses by rewriting shaders to access one cluster of data before moving to the next – texture cache misses significantly contribute to any texture pipe bottleneck. Hardware operates on blocks of 2x2 fragments, so shaders are more efficient if they access neighboring texels within a single block
- Avoid 3D textures, since fetching data from volume textures is expensive due to the complex filtering that needs to be performed to compute the result value
- Don't exceed the performance limit the number of samplers, as well as the sum of textures and SSBOs referenced by the shader (textures and SSBOs share hardware resources)
- Compress all textures to allow better memory usage, translating to a lower number of texture stalls in the rendering pipeline
- Use mipmaps: they help to coalesce texture fetches and can help improve performance at the cost of increased memory usage

Texture filtering can influence the speed of texture sampling. Filter performance is architecture/chip dependent, and you might see a benefit by using bilinear or nearest filtering over trilinear or anisotropic. Mipmap clamping may reduce the cost of using trilinear filtering, so the average cost might be lower in real-world cases. Nonetheless, adding anisotropic filtering multiplies with the degree of anisotropy – in other words, a 16x anisotropic lookup can be 16 times slower than a regular isotropic lookup. However, because anisotropic filtering is adaptive, this hit is taken only on fragments that require anisotropic filtering, which could be only a few fragments altogether. A rule of thumb for real world cases is that anisotropic filtering is, on average, less than double the cost of isotropic.

Shader-specific gradients, based on the dFdx and dFdy functions, cost more than a regular texture sample. These shader-specific gradients cannot be stored across lookups, so if a texture lookup is done again with the same gradients in the same sampler, it will incur the cost again.

In summary: usage of trilinear, anisotropic filtering, wide texture formats, 3D textures, texture lookup with gradients of different Lod, or gradients across a pixel quad may also increase texture sampling time.

DXC and Glslang and the Texture Pipe

Many games and engines generate their shaders in one language and use conversion tools to translate these shaders into other languages – for example, HLSL can be converted to SPIRV using DXC or glslang.

These conversion tools might convert HLSL buffer types to GLSL storage buffers and vice versa – and, since storage buffers utilize the texture pipe, this can result in unexpected increases in texture usage.

The DXC documentation details what to expect when converting from HLSL to SPIRV: https://github.com/Microsoft/DirectXShaderCompiler/blob/main/docs/SPIR-V.rst#constanttexturestructuredbyte-buffers

DXC's texture format conversion is specifically documented as well: https://github.com/microsoft/DirectXShaderCompiler/blob/main/docs/SPIR-V.rst#textures.

Aside from texture and buffer conversions, while most code generated from these tools works well on our hardware (and whenever possible we engage the creators of these tools to maintain and improve such code generation), sometimes these tools use instructions and idioms that perform well on PC and console, but have suboptimal performance on mobile GPUs.

Compiling and linking during initialization

The compilation and linking of shaders is a time-consuming process that can produce framerate hitches if performed naively at runtime. It is recommended that shaders are loaded and compiled during initialization.

Vulkan

Performing all calls to vkCreateGraphicsPipelines() and vkCreateComputePipelines() during initialization should eliminate any framerate hitches due to shader compilation.

OpenGL ES

After shaders have been loaded and compiled, glUseProgram should then be invoked to switch between shaders as necessary during the rendering phase.

For OpenGL ES 2.0, ES 3.0, and ES 3.1 contexts, the use of blob binaries is recommended. After compiling and linking a program object, it is possible to retrieve the binary representation, or blob, using one of the following functions:

- glGetProgramBinary (if using an OpenGL ES 3.0 or 3.1 context, then this function is considered core functionality)
- glGetProgramBinaryOES (if using an OpenGL ES 2.0 context and the GL_OES_get_ program_binary extension is available)

The blob can then be saved to persistent storage. The next time the application is launched, it is not necessary to recompile and relink the shader. Instead, read the blob from persistent storage and load it directly into the program object using glProgramBinaryOES or glProgramBinary. This can significantly speed up application launch times.

Warning: Many OpenGL ES implementations incorporate build time GL state into program objects when they are linked. If that program is then used for a draw call issued in the context of a different GL state configuration, then the OpenGL ES implementation must rebuild the program on-the-fly. This behavior is legal in the OpenGL ES specification – however, it can cause serious problems for developers. It often takes a significant amount of time to rebuild the program object, which can lead to severe frame drops. The rebuild was not requested by the application, so the delay is unexpected and the reason for it may not be apparent.

On Adreno platforms, this is not an issue. The Adreno drivers never recompile shaders, so it's safe to assume that program objects will never be rebuilt unless specifically requested.

Compute Shaders

Prefer fragment shaders to compute shaders when possible, since compute shaders require kernel output to be written to memory before the next kernel can begin execute. By comparison fragment shaders use Adreno's concurrent resolve hardware, which can write the results of a fragment program while allowing another fragment program to begin executing simultaneously.

Keep instruction counts below device limits like any other shader (although issuing shaders to LPAC has a slightly higher instruction limit).

Tune workgroup number (depending on whether shaders read each other's memory) and workgroup sizes (depending on whether or not the shader significantly stalls in ways that cannot be optimized away) to avoid potential performance bottlenecks. Note the driver may not be able to execute the requested workgroup number or size optimally if the shader's GPR usage or instruction count is too large to accommodate the simultaneous execution of the requested number of

workgroups.

Warning: OpenGL ES: When calling glDispatchIndirect() with any workgroup size smaller than 64 – for example (32, 1, 1) = 32x1x1 = 32 – the Cpu may wait on the Gpu as a result of the driver mapping the indirect buffer to a larger workgroup size, since doing so involves a command buffer flush. To avoid this, use a workgroup size that is at least 64 – for example (64, 1, 1) = 64x1x1 = 64.

```
layout(local_size_x = 32, local_size_y = 1, local_size_z = 1) in;
//workgroup size = local_size_x*local_size_y*local_size_z = 32x1x
1 = 32 -- make sure this equation produces at least 64 to avoid a
Cpu-wait-on-Gpu
```

Avoid synchronizing compute threads if possible – if synchronization is required, prefer atomic operations between local groups to shader barriers.

Separate graphics submits and compute dispatches as much as possible.

GPU-Driven Rendering

Geometry instancing

Geometry instancing calls include:

Vulkan

vkCmdDraw, vkCmdDrawIndexed, vkDrawIndirectCommand, and vkDrawIndexedIndirectCommand support instanced rendering.

OpenGL ES

glDrawArraysInstanced, and glDrawElementsInstanced support instanced rendering.

Indirect draw calls

For example, if the renderer uses a scene graph, it is possible to cache the draw call arguments necessary to render a mesh's nodes in a buffer object created at loading time. The buffer can then be used at render-time as an input to the glDrawArraysIndirect or glDrawElementsIndirect functions.

To ensure fast-z mode for depth-only renderpasses that use indirect draws and SSBOs, activate SPIR-V ExecutionMode 4 (EarlyFragmentTest) by adding this qualifier in your GLSL shader:

```
layout(early_fragment_tests) in;
```

Vertex streaming versus attribute fetching

Vertex streaming is more performant than attribute fetching when architecting your renderer in the GPU-driven style.

Low Priority Asychronous Compute (LPAC)

LPAC has a slightly larger instruction cache than shaders than run on the Graphics Pipe.

2D operation hardware acceleration

Adreno hardware accelerates common 2D operations:

- blits – with Vulkan, use vkCmdBlitImage()
- surface clears
- Android-phone pre-rotation on Vulkan (recent OpenGL ES drivers do this automatically)
- rotation
- convolution kernels – for example, with Vulkan's VK_QCOM_Image_Processing

Graphics APIs like Vulkan and OpenGL ES often map to Adreno's specialized, high-performance hardware.

Queries

Occlusion queries performance

The number of queries and their frame-latency should be respected to maximize performance.

The performance of occlusion queries is further affected by the number of bins – the higher the bin count the more expensive the queries.

Run occlusion queries in direct mode whenever possible. One way to ensure this occurs is to issue all the queries for a frame in one batch after a flush; for example: Render Opaque -> Render Translucent -> Flush -> Render Queries -> Switch FBO.

The driver has a heuristic which understands that only queries have been issued to the surface – and switches to direct mode.

Note: The overhead of queries will show up as a higher "% CP Busy" metric in Snapdragon Profiler.

We've seen cases where issuing many queries to a binned surface caused a CP overhead increase of 20-40% – and drop to 4-6% in direct mode.

Timer Query Accuracy

Timer queries should always be issued within a renderpass to maximize accuracy.

Timer queries are calculated over the entire set of binned tiles when in binning mode. For example, let's assume that we have 50 draw calls and a render target that requires 8 tiles to render. Let's also assume we want to measure draw call 10 and instrument it with timer queries.

The entire command stream of 50 draws will be captured and run through the binning process to generate the visibility streams. During the rendering pass, the draw calls will be rendered according to the visibility stream of each tile. Even if the geometry for draw call 10 only contributes to one tile, it will incur a small overhead for each tile (while processing the visibility stream). This overhead and the actual rendering time will be accumulated and presented in the resulting timer query.

Note: The overhead mentioned above is small (2-5μs) but can add up if the draw call count is high and draws are present in many tiles. Starting with A5X, GPU optimizations to the visibility stream have been added to reduce this overhead by "trimming" the end of the stream of draw calls that do not contribute to the tile. This optimization can be nullified if something like a full screen pass is issued as the last draw call to a render target.

Qualcomm True HDR setup

To enable Qualcomm True HDR in OpenGL ES, the following extensions must be supported:

- EGL_EXT_gl_colorspace_display_p3
- EGL_EXT_gl_colorspace_bt2020_pq
- EGL_EXT_surface_SMPTE2086_metadata

Note: Vulkan Swapchain/WSI for Android only supports VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT. For additional information on how to enable this for Vulkan, check out the Enhancing graphics with wide color content guide from the Android Developers Documentation.

Set EGLSurface format

Set the EGLSurface format to R10G10B10A2:

```
EGLConfig EGLConfigList [1]; int ConfigAttributes [] = {
    EGL_RED_SIZE,10
    EGL_GREEN_SIZE,10
    EGL_BLUE_SIZE,10
    EGL_ALPHA_SIZE,2
    EGL_COLOR_COMPONENT_TYPE_EXT,
EGL_COLOR_COMPONENT_TYPE_FIXED_EXT, EGL_NONE};

eglChooseConfig (eglDisplay,ConfigAttributes,EGLConfigList,1, eglNumC
onfigs);
```

Set color space

Set the color space of eglWindowSurface to EGL_GL_COLORSPACE_BT2020_PQ_EXT.

```
EGLint attribs[] = {EGL_GL_COLORSPACE_KHR,EGL_GL_COLORSPACE_BT2020_PQ
_EXT,EGL_NONE };
EGLSurface eglSurface=eglCreateWidowSurface(eglDisplay,eglConfigParam
, InWindow, attribs);
```

Set metadata

Set the metadata attributes of eglSurface.

```
EGLint SurfaceAttribs [] = {
EGL_SMPTE2086_DISPLAY_PRIMARY_RX_EXT,       EGL_SMPTE2086_DISPLAY_PRI
MARY_RY_EXT,
    EGL_SMPTE2086_DISPLAY_PRIMARY_GX_EXT,
    EGL_SMPTE2086_DISPLAY_PRIMARY_GY_EXT,
    EGL_SMPTE2086_DISPLAY_PRIMARY_BX_EXT,
    EGL_SMPTE2086_DISPLAY_PRIMARY_BY_EXT,
    EGL_SMPTE2086_WHITE_POINT_X_EXT,
    EGL_SMPTE2086_WHITE_POINT_Y_EXT,
    EGL_SMPTE2086_MAX_LUMINANCE_EXT,
    EGL_SMPTE2086_MIN_LUMINANCE_EXT
};
static const DisplayChromacities DisplayChromacityList [] = { {{0.708
00f, 0.29200f, 0.17000f, 0.79700f, 0.13100f, 0.04600f, 0.31270f, 0.32
900f}}, // DG_Rec2020 };
for (uint32_t i = 0; i < 8; i++)
{
    eglSurfaceAttrib(PImplData->eglDisplay,eglSurface, SurfaceAttribs
[i],EGLint(DisplayChromacityList[0].ChromaVals[i]* EGL_METADATA_SCALI
NG_EXT));
}
```

Get the luminance of display on Android

See:
https://developer.android.com/reference/android/view/Display.HdrCapabilities.html#getDesired

. . . for methods like:

- MaxAverageLuminance
- MaxLuminance
- MinLuminance

Variable Rate Shading (VRS)

Vulkan

VRS is exposed through the VK_KHR_fragment_shading_rate extension. VK_KHR_fragment_shading_rate takes a VkExtent2D argument where developers specify the width and height of the desired fragment size.

OpenGL ES

VRS is exposed through the QCOM_shading_rate extension. This extension includes a number of enumerations (e.g., GL_SHADING_RATE_1X1_PIXELS_QCOM, GL_SHADING_RATE_1X2_PIXELS_QCOM, etc.) for controlling the different fragment sizes. You can see a demonstration of the extension's usage on our new Adreno GPU OpenGL ES Code Sample Framework in GitHub.

XR/VR/AR (Extended Reality/Virtual Reality/Augmented Reality)

The following API calls supports efficient stereographic rendering – the driver captures issued commands and replays them for other eye. This saves CPU time (but has no impact on the GPU).

Vulkan

VK_KHR_multiview

OpenGL ES

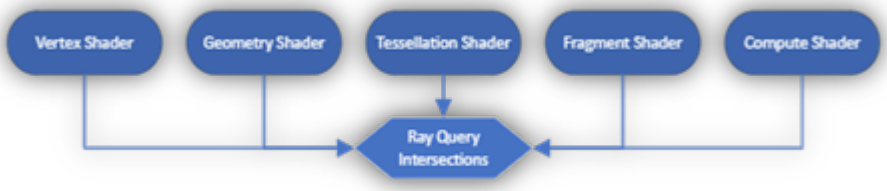
GL_OVR_multiview

Raytracing

Hit detection

Vulkan implements hit detection with the VK_KHR_acceleration_structure extension. This structure can be constructed on the CPU or GPU (we recommend constructing it on the CPU). Unless your scene is entirely static, the structure will need to be updated as geometry and/or lights move. While this updating cost can be substantial, it can also be amortized over several frames, often with few or zero artifacts.

Developers can query ray-hit information during any shader stage via the VK_KHR_ray_query extension – any algorithm that requires ray-occlusion information likely starts here:

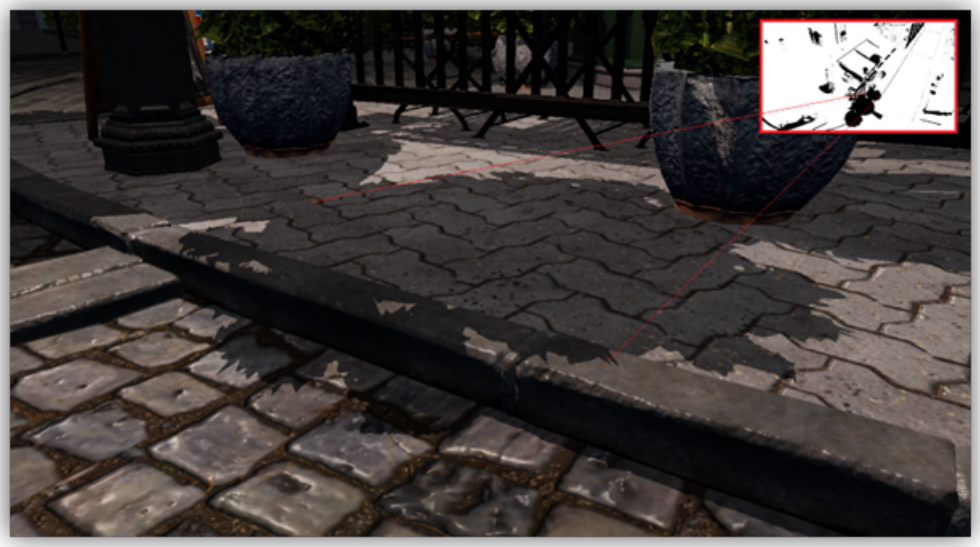


Application example: shadow generation

With a rasterization-only pipeline, shadow generation can be expensive and may not generate consistently good visual results.

With ray tracing, shadows can be added with a few lines of shader code and a reasonably optimal acceleration structure. Fine detail can be achieved without requiring intermediate surfaces or other algorithms – just querying per-pixel visibility provides precise occlusion information, the results of

which can be further manipulated by other post-processing techniques.



This sample generates a shadowmap in a subpass by raytracing from the main light source (a point-light). This query logic and the acceleration structure management are the only additions raytracing needs to determine pixel occlusion – the surrounding code remains the traditional rasterization approach.

```
// Initialize the query object
rayQueryEXT rayQuery;
rayQueryInitializeEXT(
    rayQuery,
    accelStructure,
    gl_RayFlagsTerminateOnFirstHitEXT,
    cullMask,
    WorldPos,
    minDistance,
    DirectionToLight,
    LightDistance);

    // Traverse the query -- do once for the first hit
    while(rayQueryProceedEXT(rayQuery))
    {
        // Hit something! Logic can be added here depending on the t
ype of intersection
    }

    // Get the last intersection information
    if(rayQueryGetIntersectionTypeEXT(rayQuery, true) != gl_RayQueryC
```

```
ommittedIntersectionNoneEXT)
    {
        // Got an intersection -- this pixel is occluded, so retrie
ve distance
        float intersectionDistance = rayQueryGetIntersectionTEXT(ray
Query, true);

        // ...
    }
```

Optimization

Minimize calls to rayQueryProceed()

Our Vulkan SPIR-V compiler currently does not support function calls – therefore all calls to rayQueryProceed() are inlined. For example, if the application wraps calls to rayQueryProceed() in a ClosestHitTrace() function, and then calls ClosestHitTrace() in 10 different places in the code, the compiler could generate 10 copies of the traversal loop. The traversal loop might be 300 instructions, so this can easily result in shaders that don't fit the instruction cache and consequently perform poorly.

We've found that typically a couple of calls to proceed() doesn't hurt performance. However, for larger numbers of traversal loops it becomes more challenging for our compiler to achieve reasonable register allocation, which results in GPR spilling and other performance degradations.

Reuse one ray query object

Reuse the single ray query object as needed. A ray query object costs enough GPRs that the concurrent wave count for the shader may be reduced. Thus make every effort to avoid more then one ray query object, reusing that one object across different calls to rayQueryProceed(). This reuse should be possible unless recursion is used – which itself is usually avoidable.

Access ray query data through intrinsics

Avoid copying data from a ray query into large, custom data structures – this will increase GPR usage. Instead, use intrinsics to access the query object's data.

Avoid proceed() calls in loops for non-opaque traversal that “accept first fit”

When using gl_RayFlagsTerminateOnFirstHitEXT or gl_RayFlagsCullOpaqueEXT, there is no need to call rayQueryProceedEXT in a while loop – this easy simplification helps the compiler generate code with fewer branches.

For example:

```
rayQueryEXT rayQuery;
rayQueryInitializeEXT(rayQuery, rayTraceAS, gl_RayFlagsTerminateOnFir
stHitEXT | gl_RayFlagsCullOpaqueEXT, cullMask, WorldPos, minDistance,
DirectionToLight, LightDistance);

// Traverse the query. No need for a while(), since we want first h
it or
// non-opaque intersections only
rayQueryProceedEXT(rayQuery));

// Determine if the shadow query collided
if(rayQueryGetIntersectionTypeEXT(rayQuery, true) != gl_RayQueryCommi
ttedIntersectionNoneEXT)
{
    // Got an intersection == Shadow, do something
}
```

Instruction cache

Good instruction cache practices are even more important than usual for ray tracing shaders – remove dead code, factor away non-uniform branches and looping and shrink each shader’s total instruction size so it fits target devices’ instruction cache.

16-bit Precision

Half-precision is, as ever, an easy performance win that often involves few to no additional artifacts. The Vulkan API supports this with VK_KHR_shader_float16_int8.

Culling

Simplifying acceleration structures speeds raytracing. Aggressively cull your geometry as much as your content allows – frustum culling, portal culling, level-of-detail, and other techniques may apply.

Acceleration structures

Acceleration structures are typically built fastest on the CPU with vkCopyAccelerationStructureToMemoryKHR and vkCopyMemoryToAccelerationStructureKHR as opposed to the GPU.

Try to cache as much of your acceleration structure as long as you can, amortizing any rebuilds over multiple frames to stay within your frame budget.

Variable-Rate Shading (VRS)

Fewer pixels shaded means fewer rays queried – often with few to no additional artifacts. Use VRS as aggressively as your content allows.

Querying the driver version to implement version-specific workarounds

Despite our best efforts, sometimes drivers ship with bugs.

When a future driver fixes some bugs, you may want to implement a workaround for certain versions of a driver and not others. Here's how:

Vulkan

"driverVersion" can be queried from VkPhysicalDeviceProperties – it returns the complete major/minor/patch number of the driver version.

For Adreno, the 32-bit number of the driverVersion is encoded as major-minor-patch. The first 10 bits is the major version, the second 10 bits is the minor version and the remaining 12 bits is for the patch version.

The major number is usually fixed, while the minor number changes with each release. Patches are the variations of a minor release. We primarily use the minor number to identify different driver versions, while the patch number is generally used to identify workarounds and fixes for a given driver version.

Sample code:

```
#define VK_VERSION_MAJOR(version) ((uint32_t)(version) >> 22)
```

```
#define VK_VERSION_MINOR(version) (((uint32_t)(version) >> 12) & 0x
3FFU)

#define VK_VERSION_PATCH(version) ((uint32_t)(version) & 0xFFFU)

// Workarounds for Adreno driver 676
if (VK_VERSION_MINOR(version) == 676)
{
   // Guard for known Adreno issue on patch < 17 and driver 676
   if (VK_VERSION_PATCH(version) < 17)
   {
      doWorkaround();
   }
   // After patch 17 we know the issue was addressed by the vendor
   else
   {
      doNormalFlow();
   }
}
```

adb

To check the driver version, run the command:

```
adb shell dumpsys SurfaceFlinger | grep GLES
```

Output is of the form:

```
GLES: Qualcomm, Adreno (TM) 740, OpenGL ES 3.2 V\@0676.0 (GIT\@6f08dd
b, I5e1ee3b043, 1669189803) (Date:11/23/22)
```

Given: V@0676 as above, 676 is the driver promotion/minor number.

Shader stats (like for Snapdragon Profiler) for vulkan needs driver support: 636 promotion/minor number and above qualifies.

Adreno APIs

Adreno GPUs support industry-standard APIs including:

- OpenGL ES 1.x (fixed function pipeline)
- OpenGL ES 2.0 (programmable shader pipeline)
- OpenGL ES 3.0
- OpenGL ES 3.1 + AEP
- OpenGL ES 3.2
- EGL
- Vulkan 1.0
- Vulkan 1.1
- OpenCL 1.1e
- OpenCL 2.0 Full Profile
- DirectX 11 FL 9.3
- DirectX 12 FL 12

Texture Compression Examples

As an example, we show one diffuse and one object-space normal RGB texture, compressed using the (legacy) Adreno Texture Tool. The figures show the original textures and the compressed versions using each of the compression formats. For each example, there is a difference image showing the absolute difference between the original and the compressed version.

Diffuse texture test

The diffuse texture used for this test is shown below:

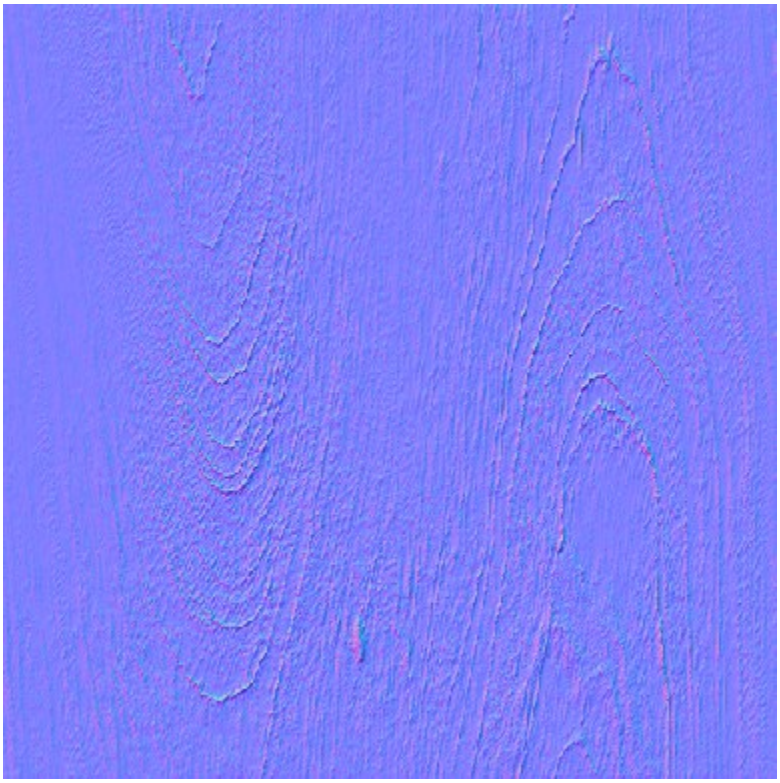
ATC Compression

ETC1 Compression

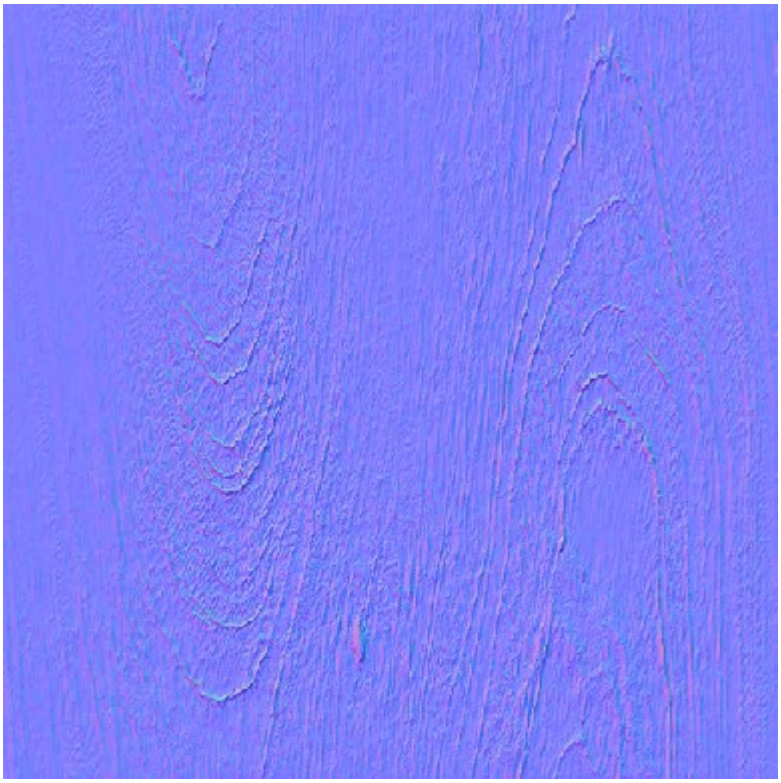
ETC2 Compression

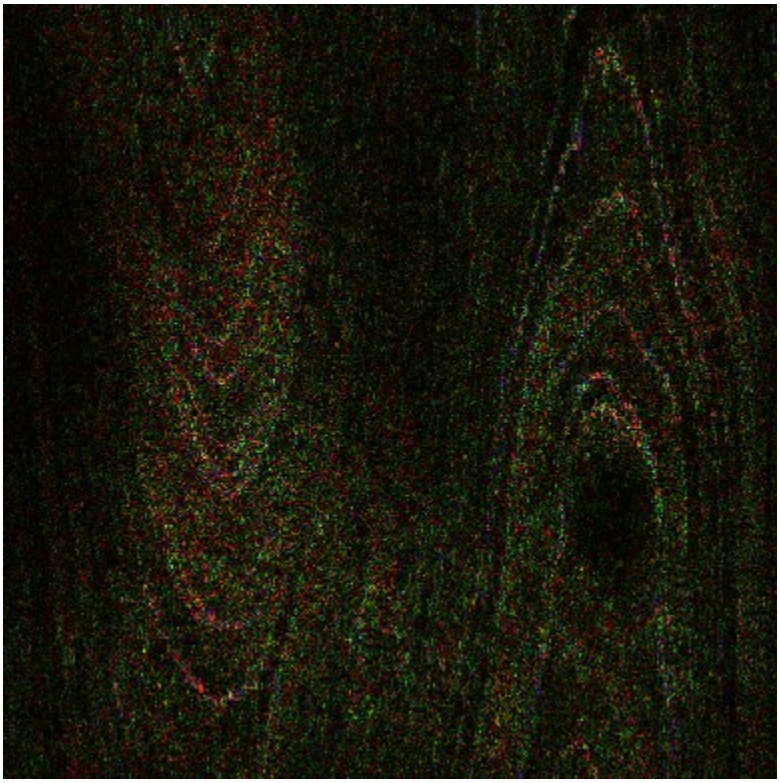
Normal texture test

Texture data can be compressed to any of these texture compression formats. The normal texture used for this test is shown below:

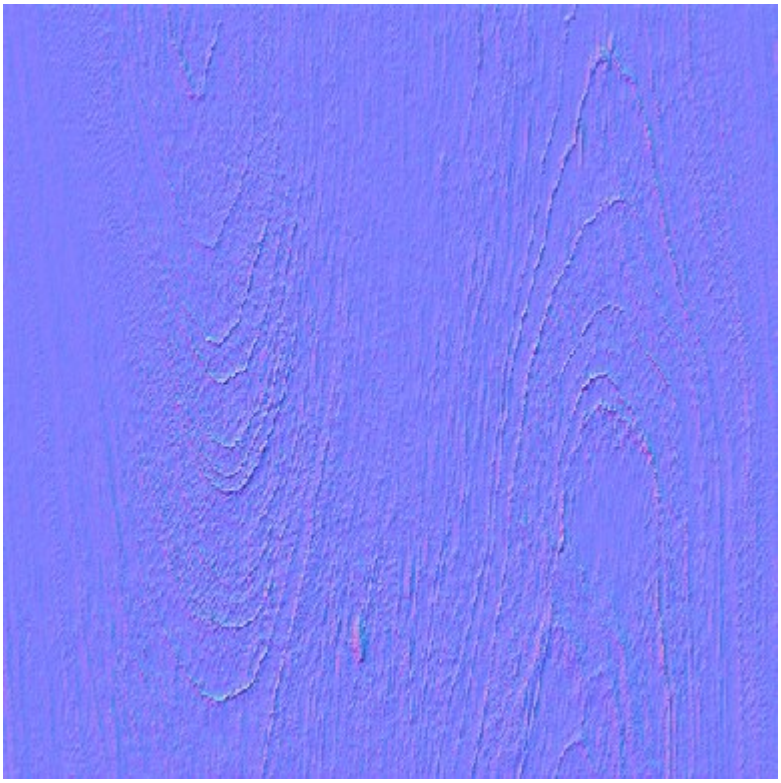


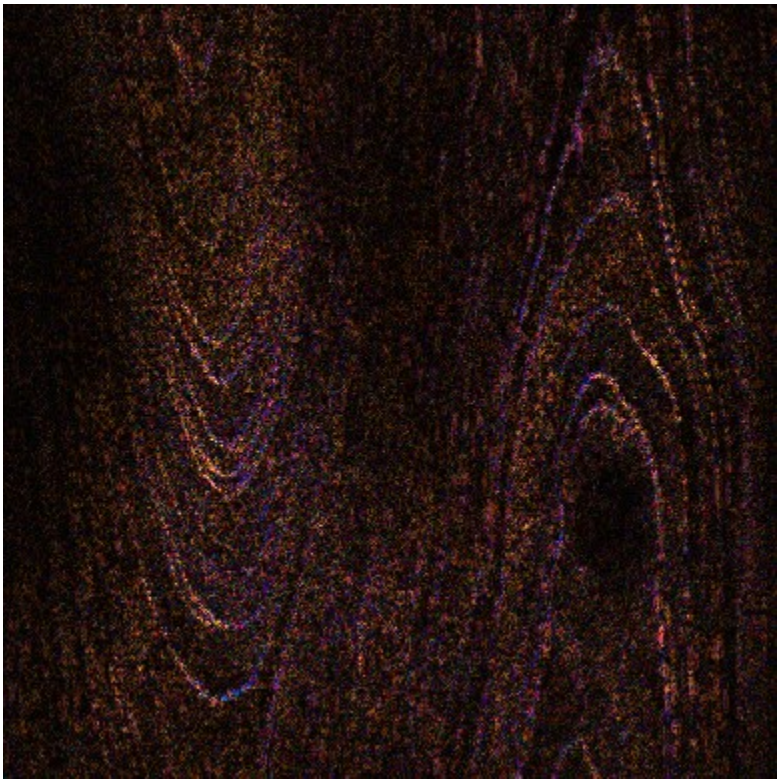
ATC Compression



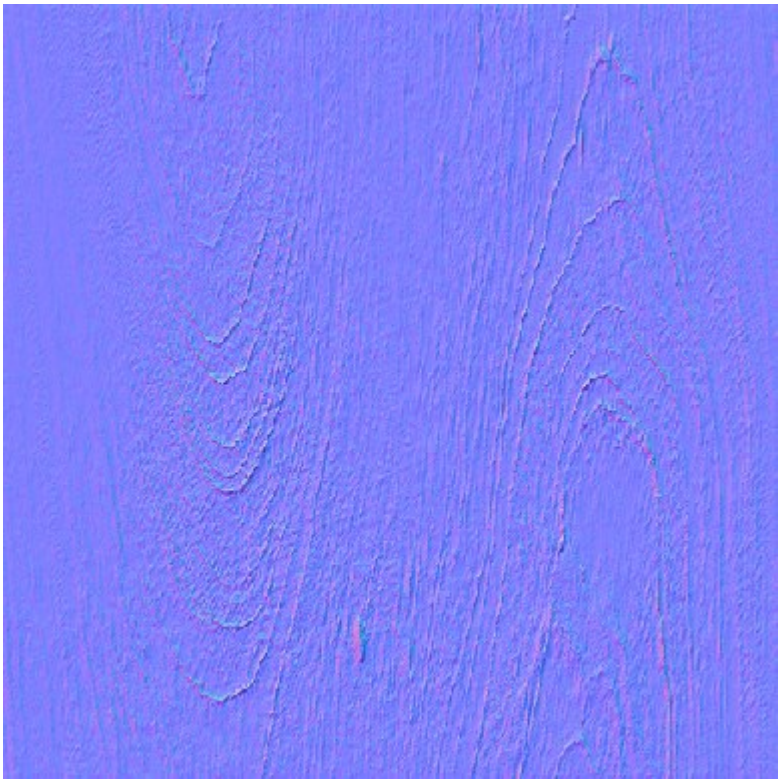


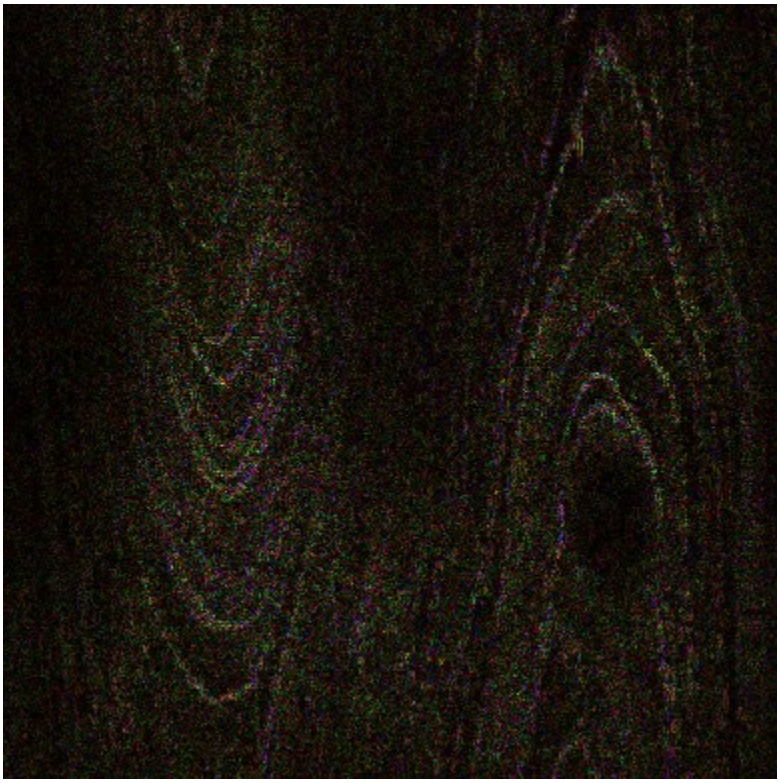
ETC1 Compression





ETC2 Compression





Qualcomm® Oryon™ CPU

This document provides game programmers with general optimization guidance for achieving the best performance on Qualcomm Snapdragon X CPUs. This document covers different optimization topics that can be applied to both general programming and game programming specifically.

Tools

There are a number of tools that are useful in debugging and developing on Windows with Snapdragon. This section briefly covers a number of tools that support Windows on Snapdragon and can provide valuable insight into what is happening during game execution.

Event Tracing for Windows

Event Tracing for Windows (ETW) is a logging mechanism shipped with Microsoft that can trace and log events that are raised by user-mode applications and kernel-mode drivers. It is built into the Windows operating system. There are third party tools that can leverage this infrastructure to log events. One of these tools is Superluminal, discussed in the next section.

Superluminal

Superluminal is a profiling tool specifically for Windows. It supports Windows running on an Arm machine and can profile both native Arm64, as well as Arm64EC code. It provides visualization, filtering, and analysis tools for data captured with both Superluminal tools and through ETW. The Unreal Engine also has support for Superluminal, allowing events in Unreal to be captured by Superluminal. Frame pointers must be turned on in order to obtain full traces. This can be accomplished by putting the following into the `Engine/Saved/UnrealBuildTool/BuildConfiguration.xml` file:

```
<BuildConfiguration>
    <bOmitFramePointers>false</bOmitFramePointers>
</BuildConfiguration>
```

However, recently, there were some changes to the Unreal build files and source code to fully support Superluminal on Windows on Arm.

Visual Studio

Visual Studio fully supports Windows on Arm, including developing, compiling, and debugging. It can be downloaded and run on a Snapdragon Windows machine for either full development or debugging of executables. Visual Studio also supports remote debugging for developing on a different machine than the target Snapdragon machine. It is also possible to use remote desktop on a development machine, log into the Snapdragon target running Visual Studio, and point to the source files on the development machine.

Architecture

Overview

The Qualcomm Snapdragon X CPU is a high performance, ARMv8-compliant 64-bit CPU. It is a custom processor based on the ARMv8 Instruction Set Architecture (ISA) specification. The Snapdragon X processor only supports AArch64 instructions; AAarch32 instructions are not supported. It implements the NEON instructions for vectorized operations. The Snapdragon X CPU family comes in multiple core configurations.

Processor	Configuration
Snapdragon X Elite	12 performance cores

General primer on the ARM architecture

The ARM architecture is a Reduced Instruction Set Computer (RISC) based architecture with the following features:

- Low-cycle execution time
- Heavy pipelining of instructions
- Low-power execution
- High-performance

The ARM system employs load/store instructions to fetch data into and write data from internal registers. Almost all operations are done on register values, using registers values as both source and destination. Because ARM is a RISC architecture, the instructions tend to be focused on performing a single operation, allowing simplified instruction decode and, subsequently, reducing the overall complexity of the chip.

There are three ARM architecture profiles:

- **A-profile (Applications)** – Designed for high performance systems and made to run complex operating systems such as Windows and Linux.
- **R-profile (Real-time)** – Designed for systems with real-time requirements and used in

networking and embedded control systems.

- **M-profile (Microcontroller)** – Designed for microcontrollers and found in many IoT devices.

Snapdragon X is a custom designed CPU conforming to the ARM A-profile.

Within a profile, there are various versions of that architectural specification, which change over time as features are improved or new features added. The Snapdragon X CPU implements version 8 of the ARM A-profile specification, referred to as ARMv8-A.

Within a version of a profile, there may be minor versions that are released to add new features. These are referred to as .x extensions. For example, the ARMv8.1-A extension adds atomic memory instructions. Each extension includes mandatory features and optional features. Any processor that is said to be compliant at a particular extension level means that it implements all the mandatory features of that extension level and the mandatory features of all previous extensions.

The Snapdragon X processor is ARMv8.7-A compliant, which means it implements the mandatory features of all extensions from ARMv8.0-A (the base specification) to ARMv8.7-A. Some of these notable extensions include atomic instructions, CRC instructions, and floating point to integer instructions. Some notable features NOT implemented include AArch32 support, Big-endian support, and SVE instructions.

Introduction to ARM Assembly

The ARMv8 class of processors are 64-bit processors, which have 64-bits of data and address, 31 64-bit general purpose internal registers, a set of 32 128-bit wide vector registers, and are designed to run complex operating systems like Windows or Linux. ARM processors are the dominant processor for cellphones and mobile devices.

Example instructions include loading a register from memory, writing a memory location from a register, or adding two register values together and placing the result in a register. Registers in ARM64 are numbered from R0 to R30, with R31 being a dedicated zero register that always reads/writes as 0. The full 64-bits can be accessed in assembly as *Xn* or the lower 32-bits can be accessed as *Wn*. Most instructions fall into one of the following categories: load/store instructions, data processing instructions, and branch instructions. There are other instructions that perform more specialized functions, such as setting system bits, but this primer focuses on basic instructions that fall into the three categories listed above.

Values are moved into and out of registers to perform any operations on them. There are load/store instructions to move data from/into memory into/from registers. Operations (e.g., add) are performed on data that resides in the registers. In ARM assembly, the resultant register is typically the first listed in the instruction.

For example, the following instruction adds two numbers together (the number stored in x3 and the number stored in x2) and stores the result in another register (x4):

```
add         x4, x3, x2
```

An immediate value can be used in many instructions with the prefix #. Modifying the case above to add the number 17 to what is in x3, the instruction would be:

```
add         x4, x3, #17
```

To load a register with data from memory, one of the load instructions could be used:

```
ldr         x1, [x7]
```

This instruction treats the data in x7 as an address to load from (a pointer). This will load the data from that memory address into register x1. A store instruction is similar:

```
str         x1, [x7]
```

Note that in the case of a store instruction, the source register is the first parameter, and the destination address is the second parameter. Putting this all together to load two 64-bit numbers, add them together, and store the result, the assembly would look like:

```
ldr         x8, [x8]
ldr         x9, [x9]
add         x8, x8, x9
str         x8, [x10]
```

And, likewise, doing this operation using only 32-bit numbers would look like:

```
ldr         w8, [x8]
ldr         w9, [x9]
add         w8, w8, w9
str         w8, [x10]
```

Unconditional branching can be achieved with the branch instruction:

```
b           label
```

Where label is the assembly label of the location to which to branch. This label will be replaced with an actual address during compilation.

An example of a conditional branch would be:

```
ldr         w8, [x9]
cmp         w8, #10
ble         label
```

This statement indicates that if the value of w8 is less than 10, then branch to the address at the label.

In addition to the 31 general purpose registers, Snapdragon X has 32 128-bit vector registers to

support special vector operations. These vector operations are supported by advanced Single Instruction Multiple Data (SIMD) instructions. The SIMD instructions support both floating point (single and double precision) and integer operations and can operate on multiple lanes of data simultaneously.

ARM64 and ARM64EC

The basic application binary interface (ABI) for Windows when compiled and run on ARMv8 processors in 64-bit mode, for the most part, follows the ARM standard AArch64 EABI. The ABI defines the calling convention, stack usage, and data alignment used for executables running under Windows. Windows uses ARM64 when referring to the use of the ARM64 ABI as the calling convention within an executable.

ARM64EC (*Emulation Compatible*) is a new ABI for building apps for Windows 11 on ARM. ARM64EC enables ARM64 binaries to run natively and interoperably with x64 code. It is a Windows 11 feature that requires the use of the Windows 11 SDK and is not available on Windows 10 on ARM. This makes ARM64EC code and x64 code interoperable. The operating system emulates the x64 portion of the binary.

Code built as ARM64EC is interoperable with x64 code running under emulation within the same process. The ARM64EC code in the process runs with native performance, while any x64 code runs using emulation that comes built-in with Windows 11. ARM64EC guarantees interoperability with x64 by using the same calling conventions, stack usage, and preprocessor definitions as x64. Code using the ARM64EC ABI is not interoperable with code written using the ARM64 ABI, as the stack layout and calling conventions are different.

It is important to note that ARM64EC code is native ARM64 code. It is not emulated code, though it can interoperate with emulated x64 code running on a Windows 11 ARM64 machine. ARM64EC was designed to deliver native-level functionality and performance, while providing transparent and direct interoperability with x64 code running under emulation.

Differences from X64

The ARM architecture is a RISC-based architecture as opposed to X64's Complex Instruction Set Computer (CISC) based architecture. A RISC system focuses on doing fewer operations per instruction while having a low clock cycle/instruction. In comparison, a CISC system focuses on completing more operations per instruction while having a generally higher clock cycle/instruction. For example, a single instruction in a CISC-based CPU may be able to read a value from memory, increment it, and store the result back to memory. A RISC-based CPU, however, would use three separate instructions, one for the load, one for the increment, and one for the store. However, this does not necessarily directly translate to 3X the clock cycles necessary to complete the instructions in comparison to a CISC machine. Both systems must fetch from memory, perform the operation, and store the value back out to memory. A CISC system simply breaks the single instruction down into micro operations to perform the required operations.

X64 systems typically have fewer general purpose registers than ARM64, with X64 systems having 16 64-bit general purpose registers and ARM64 having 31 64-bit general-purpose registers. x64 systems have either 16 128-bit floating point registers (AVX), 16 256-bit floating point registers (AVX2), or 32 512-bit floating point registers (AVX-512). ARM64 has 32 128-bit floating point registers. In both architectures, the registers can be used in floating point operations and Single Instruction Multiple Data (SIMD) operations.

The ARM64 processors do not employ simultaneous multithreading, which is referred to as Hyperthreading in x86-64 cores. Snapdragon X ARM64 processors do employ out of order execution, which can provide a significant performance improvement. These differences between RISC and CISC in instruction complexity, it also implies a more subtle difference between what is referred to as weakly ordered versus strongly ordered. The ARM architecture is referred to as a weakly ordered system, while the Intel architecture is referred to as a strongly ordered system.

Performance optimization

Clocks and counters

It is common to need timing functionality when optimizing software. It is best to leverage the existing OS-supplied timing functionality. These will typically use the underlying hardware counters of the system and provide the easiest way to obtain accurate information. In those situations where direct underlying access to counter registers are required, Snapdragon X supports both the ARM standard cntpct and cntvct system counters, as well as a cycle counter via the PMUv3 architecture.

`cntpct_el0` is a physical register that holds the current value of the system counter. The `cntvct_el0` register is a virtual counter that holds the value of the physical system counter minus an offset set by the OS. Both the `cntpct_el0` and `cntvct_el0` registers are available to be read from user space. This counter is incremented at the frequency set in the `cntfrq_el0` register. The `cntfrq_el0` register is not populated by hardware, but rather set by software at the highest level exception of the system. It can be read and used by the by the OS and is also readable from user space. Timing in software can be determined by using the physical counter registers and the value in the `cntfrq` register. The value in `cntfrq`, and the frequency if the system counters, is not the raw system clock, but rather a lower frequency, usually between 1 MHz and 50 MHz. The use of system counters is the most commonly used method for timing functions. To query the `cntvct_el0` register, the MRS assembly instruction can be used – `mrs x0, cntvct_el0`. It is preferable to precede the reading of the counter with an ISB instruction to ensure that it the reading is not speculatively executed.

The `pmccntr_el0` register is a 64-bit wide cycle counter register, implemented as part of the PMUv3 architecture. It is subject to any changes in the clock frequency, meaning that it will not be incremented when waiting in a WFI or WFE. This should be kept in mind when attempting to time any portions of code that may be subject to `sleep()` calls or interrupt waiting. To query the counter, the MRS assembly instruction can be used – `mrs X0, pmccntr_el0`.

Note that this register can only be read from user space if it is enabled in the `pmuserenr` register. This register can be read from user space through the `pmuserenr_el0` register – `mrs X0,`

`pmuserenr_el0`. If bit 2 is set, then the `pmccntr_el0` cycle counter can be read from user space.

Vectorization

Vectorization is important in game programming, as most 2D and 3D games heavily resort to using vector mathematics during gameplay. Optimizing vector calculations is an important aspect of optimizing the performance of the overall game. This section highlights various ways that vectorization can be used and optimized on Snapdragon X CPUs.

Vector support in Snapdragon X

The Snapdragon X CPU supports the use of vectorized code by employing NEON SIMD instructions. NEON was designed as an additional load/store architecture to provide good vectorizing compiler support for languages such as C/C++. It implements the ability to perform vector operations on data to increase performance by providing:

- 32 128-bit vector registers, each capable of containing multiple lanes of data.
- SIMD instructions to operate simultaneously on those multiple lanes of data.

These SIMD instructions can improve the speed of many computationally intensive algorithms, including audio and video processing and 3D space transformations.

NEON instructions allow up to:

- 16x8-bit, 8x16-bit, 4x32-bit, and 2x64-bit integer operations
- 8x16-bit, 4x32-bit, and 2x64-bit floating point operations

This allows, for example, the simultaneous addition of 4 32-bit floating point numbers to another 4 32-bit floating point numbers in a single operation. This can greatly improve performance of computation heavy routines.

Note that the Snapdragon X CPU does not support the SVE extensions from ARM.

There are a number of ways to exploit NEON support in software:

- Using specialized libraries that provide vectorized versions of common functions.
- Using auto-vectorization features of compilers that automatically optimize code to take advantage of NEON.
- Leveraging NEON intrinsics, which are function calls that the compiler replaces with appropriate NEON instructions.
- Using specialized compilers, libraries, and coding techniques, such as the Intel ISPC, which allows a C programmer to specify parts of the code that should be vectorized. ISPC leverages NEON to create performant code on ARM systems.

Specialized libraries

Specialized libraries that leverage vectorized NEON optimizations exist for some common math, audio, and video functions. Some of these are open-source libraries and some are royalty-free licenses. Some of these libraries can be found on the ARM website.

NEON intrinsics

NEON intrinsics provide a way to write NEON code that is easier to maintain than assembler code, while still enabling control of the generated NEON instructions. NEON intrinsics are function calls that the compiler replaces with an appropriate NEON instruction or sequence of NEON instructions. Intrinsic functions and data types, or intrinsics in the shortened form, provide access to low-level NEON functionality from C or C++ source code. However, the code must be optimized to take full advantage of the speed increases offered by the NEON unit. For example, instead of a single uint32_t variable, which holds single 32-bit integer, with intrinsics there is a single uint32x4_t variable, which can hold 4 32-bit integers in a single 128-bit vector register.

Software can pass NEON vectors as function arguments or return values and declare them as normal variables. In the following example, two vectors are added together, with the function taking two pointers to memory that each point to four consecutive 32-bit integers and returning essentially four 32-bit numbers. The intrinsic call to `vld1q_u32()` takes a pointer and loads four 32-bit integers into a single NEON vector register. Likewise, the `vaddq_u32()` function adds two NEON vector registers together, each register holding four 32-bit integers, and puts the resulting four 32-bit numbers into another NEON register.

Listing 1 Source code

```
uint32x4_t sumVector(uint32_t* A, uint32_t* B)
{
    uint32x4_t temp = vld1q_u32(A);
    uint32x4_t temp1 = vld1q_u32(B);
    uint32x4_t vec128 = vaddq_u32(temp, temp1);

    return vec128;
}
```

GCC	Clang	MSVC

`ldr q0, [x0]`
`ldr q31, [x1]`
`add v0.4s, v0.4s, v31.4s`
`ret`
	`ldr q0, [x0]`
	`ldr q1, [x1]`
	`add v0.4s, v1.4s, v0.4s`
	`ret`
		`ldr q17,[x1]`
		`ldr q16,[x0]`
		`add v0.4s,v16.4s,v17.4s`
		`ret`

In this case, all compilers give the same assembly due to the use of intrinsics. This allows the programmer more control over the final assembly, at the cost of requiring a deeper understanding of the instruction architecture.

Intrinsics provide almost as much control as writing assembly language, but leave the allocation of registers to the compiler, so that programmers can focus on the algorithms. Also, the compiler can optimize the intrinsics such as normal C or C++ code, replacing them with more efficient sequences if possible. It can also perform instruction scheduling to remove pipeline stalls for the specified target processor. This leads to more maintainable source code than using assembly language. The downside to intrinsics is that they are specific to the ARM64 architecture and are not cross platform.

For game programming, the use of intrinsics may be good in those cases where already-written code could take advantage of specific intrinsic routines or there are specific areas in existing ARM64 code that could be targeted for performance improvements with intrinsics.

Auto-vectorization

Auto-vectorization is the generation of NEON code by a compiler by analyzing the source code of a program. This is an architecture independent approach that allows the game programmer to focus at the algorithmic level and not be concerned with specific architectural details of the target. Most of the major compilers (i.e., GCC, Clang, MSVC) will do auto-vectorization when the appropriate flags are set. Different compilers will vectorize to different degrees and the same code may get vectorized by one compiler and not get vectorized by another. Sometimes the compiler can generate NEON code without source modification, but using certain coding styles can promote more optimal output. A rule of thumb is to try not to do the compiler's job. It is best to use straightforward code and let the compiler's optimizer create the best code. There are times, however, that restructuring loops can help the compiler better optimize.

The generation of NEON instructions is enabled in the GCC compiler with the use of a `-march` flag of ARMv8-a and above. However, the Snapdragon X CPU is 8.7 compliant, so the `-march=armv8.7-a` flag can be used in the latest versions of GCC. This will not only enable the generation of NEON instructions but provide support for other features in the Snapdragon X CPU. GCC feature flags that should not be enabled include `+profile`, `+memtag`, and `+sve`.

As an example of auto-vectorization, consider the sum of two 3D vectors. Because this operation is so fundamental within gameplay, it is important to optimize these types of vector operations as much as possible. The table below shows a simple vector addition function and the resultant assembly using GCC, Clang, and MSVC.

Listing 2 Source code

```
struct FloatVector
{
    float V[3];
};
```

```
void sumVector(FloatVector& __restrict A, FloatVector& __restrict B,
FloatVector& R)
{
    for (int i = 0; i < 3; i++)
    {
        R.V[i] = A.V[i] + B.V[i];
    }
}
```

GCC 13.2 -O2 -march=armv8.7-a	Clang 17.0 -O2 -march=armv8.7-a	MSVC 19.0 /O2 /arch:armv8.7

ldr d28, [x0]
ldr d30, [x1]
ldr s31, [x0, 8]
ldr s29, [x1, 8]
fadd v30.2s, v30.2s, v28.2s
fadd s31, s31, s29
str d30, [x2]
str s31, [x2, 8]
ret
	ldr d0, [x0]
	ldr d1, [x1]
	ldr s2, [x0, #8]
	ldr s3, [x1, #8]
	fadd v0.2s, v0.2s, v1.2s
	fadd s1, s2, s3
	str d0, [x2]
	str s1, [x2, #8]
	ret
		ldp s19,s18,[x1]
		ldp s17,s16,[x0]
		fadd s16,s16,s18
		fadd s17,s17,s19
		stp s17,s16,[x2]
		ldr s17,[x0,#8]
		ldr s16,[x1,#8]
		fadd s16,s17,s16
		str s16,[x2,#8]
		ret

In this example, GCC and Clang produce identical code. MSVC declined to use any SIMD instructions, as the compiler indicated that there were not enough instructions to warrant vectorization. However, note that it required an extra fadd instruction over GCC and Clang.

If the number of floats in the vector is increased from three to four, then all compilers will vectorize the code the same way.

GCC	Clang	MSVC

`ldr q31, [x0]`
`ldr q30, [x1]`
`fadd v31.4s, v31.4s, v30.4s`
`str q31, [x2]`
`ret`
	`ldr q0, [x0]`
	`ldr q1, [x1]`
	`fadd v0.4s, v0.4s, v1.4s`
	`str q0, [x2]`
	`ret`
		`ldr q17,[x0]`
		`ldr q16,[x1]`
		`fadd v16.4s,v17.4s,v16.4s`
		`str q16,[x2]`
		`ret`

This simple example helps highlight the importance of not only using vectorized code, but packing as many vector operations as possible to gain as much performance as possible.

This example still only adds two vectors at a time. By using forethought in the design of the software, the performance for doing multiple vector additions can be increased. In the next example, a structure of arrays approach is used to pack four vectors into a single struct and the computations are done on all four vectors at a time.

Listing 3 Source code

```
struct FloatVectorSoA
{
    float X[4];
    float Y[4];
    float Z[4];
    float W[4];
};
void sumSoAFloat(FloatVectorSoA& __restrict A, FloatVectorSoA& __rest
rict B, FloatVectorSoA& R)
{
    for (int i=0;i<4;i++)
    {
        R.X[i] = A.X[i] + B.X[i];
        R.Y[i] = A.Y[i] + B.Y[i];
        R.Z[i] = A.Z[i] + B.Z[i];
        R.W[i] = A.W[i] + B.W[i];
    }

}
```

GCC

```
ldp     q24, q29, [x0
]
ldp     q28, q25, [x1
]
ldp     q30, q31, [x0
, 32]
ldp     q26, q27, [x1
, 32]
fadd    v28.4s, v28.4
s, v24.4s
fadd    v29.4s, v29.4
s, v25.4s
fadd    v30.4s, v30.4
s, v26.4s
fadd    v31.4s, v31.4
s, v27.4s
stp     q28, q29, [x2
]
stp     q30, q31, [x2
, 32]
ret
```

Clang

```
ldp     q0, q3, [x1]
ldp     q1, q2, [x0]
fadd    v0.4s, v1.4s,
v0.4s
ldp     q1, q5, [x0,
#32]
fadd    v2.4s, v2.4s,
v3.4s
ldp     q4, q3, [x1,
#32]
fadd    v1.4s, v1.4s,
v4.4s
fadd    v3.4s, v5.4s,
v3.4s
stp     q0, q2, [x2]
stp     q1, q3, [x2,
#32]
ret
```

MSVC

```
 add          x9,x2,#0
x20
 add          x8,x1,#0
x10
 sub          x13,x0,x
1
 sub          x12,x2,x
1
 sub          x11,x0,x
2
 mov          x10,#4
|$LL22@S|
 ldur         s17,[x8,
#-0x10]
 sub          x10,x10,
#1
 ldr          s16,[x0]
,#4
 fadd         s16,s17,
s16
 ldr          s17,[x13
,x8]
 stur         s16,[x9,
#-0x20]
 ldr          s16,[x8]
 fadd         s16,s17,
s16
 ldr          s17,[x11
,x9]
 str          s16,[x12
,x8]
 ldr          s16,[x8,
#0x10]
 fadd         s16,s17,
s16
 ldr          s17,[x8,
#0x20]
 add          x8,x8,#4
 str          s16,[x9]
,#4
 ldr          s16,[x0,
#0x2C]
 fadd         s16,s17,
s16
 str          s16,[x9,
#0xC]
 cbnz         x10,|$LL
```

Again, GCC and Clang generated essentially the same vectorized code. However, note that MSVC could not vectorize the code due to it the vectorizer determining that the *loop contains loop-carried data dependencies that prevent vectorization*. There are two important takeaways in this example. The first is that the exact same code may produce very different results, depending on the compiler and the flags used. It is important to check the produced code to see how well the compiler vectorized the code. The second is that careful architectural planning in the software can dramatically improve performance. In this case, structuring the software and underlying data in such a fashion that it allows the compiler to leverage the underlying vector capabilities of the hardware increases performance.

While auto-vectorization can produce some good results in certain cases, it is compiler dependent and may not always result in the expected vector improvements. It does require careful software planning to obtain the best utilization of the vector capabilities of the system.

ISPC

The Intel Implicit SPMD Program Compiler (ISPC) is a vectorizing compiler using code syntax very similar to, and intended to integrate with, C and C++. This compiler is NOT an auto-vectorizing compiler, but rather a compiler for code intentionally written to be vectorized. The advantage of this approach is that it forces the software developer to think about how vectorization fits into the entire game design. With this forethought in the design, the ISPC creates very optimized code in an architecture independent manner. ISPC v1.19 and later generates NEON code for ARM64 CPUs on Windows, as well as vectorized code for Intel platforms, making it suitable for cross-platform development. It is an open-source compiler with a BSD license and leverages the LLVM compiler for its backend.

With ISPC, code is written with a C syntax with added features to support the ability to write Single Program Multiple Data (SPMD) programs in a straightforward manner. The ISPC compiler generates regular object files using standard C/C++ calling conventions, allowing straightforward integration with existing C/C++ files.

ISPC leverages the width of the vector unit of the CPU to maximize the benefit of vector code. For Snapdragon X, the vector unit is 128-bits wide. This allows it to perform calculations on multiple lanes of data at the same time. For example, an add instruction could operate on four lanes of 32-bit numbers at the same time, or two lanes of 64-bit numbers at the same time. Pipelining inside the Snapdragon X increases throughput by allowing back-to-back instructions to be scheduled, increasing the performance impact of well-created vectorized code.

The data should be laid out correctly to gain the most benefit from ISPC. Similar to the SoA example shown above, ISPC prefers data laid in SoA or AoSoA to produce the best use from the vector instructions. There are gather/scatter instructions, but rearranging data can be costly if done often. It is best to keep data rearrangement to a minimum and leverage successive vector instructions as much as possible to obtain the best performance.

Consider the following example, which is equivalent to the SoA vectorization approach above.

Listing 4 Source code

```
struct FloatVector
{
    float V[4];
};

FloatVector operator+(const FloatVector& A, const FloatVector& B)
{
    FloatVector R;

    for (uniform int i=0; i< 4; i++)
    {
        R.V[i] = A.V[i] + B.V[i];
    }

    return R;
}
```

ISPC
--target=neon --arch=aarch64

```
ldp     q0, q1, [x0]
ldp     q2, q3, [x1]
ldp     q4, q5, [x0, #32]
ldp     q6, q7, [x1, #32]
fadd    v0.4s, v0.4s, v2.4s
fadd    v1.4s, v1.4s, v3.4s
fadd    v2.4s, v4.4s, v6.4s
fadd    v3.4s, v5.4s, v7.4s
ret
```

Note that in this case, the resultant assembly code is essentially the same as GCC and Clang assembly code for the SoA case. ISPC will automatically try to generate code such that all lanes of the SIMD unit will be filled. This makes intentionally writing vectorized code easier, as the programmer can focus on single vector operations and the compiler will attempt to vectorize the code to use all available lanes. While the code appears to be written as only computing one (x,y,z,w) vector, the code actually can do four of them at the same time. This does assume, however, that the data arrangement allows this approach.

Unreal Engine

The Unreal Engine from Epic Games uses ISPC within the game engine to provide vectorized implementations of computationally intensive routines. Currently, it is used within the Chaos physics system and in the animation system. Epic supports using it in custom code as well. All that is needed is to add the ISPC module to the `build.cs` file, add ISPC files to the project, include the auto-generated C++ headers, and the Unreal build system will take care of the rest. Epic recommends using it for dense compute-bound workloads such as physics and cloth simulations or vertex transformations. It is best used with contiguous memory loads, manipulations, and stores, for example, the Unreal TArray structures. It is best used when there are no data dependencies between operations, and can be especially useful when combined with ParallelFor constructs and batching.

Topology and threading/affinity

Properly understanding, detecting, and using the system topology is critical to obtaining the best gaming performance. In the current desktop and mobile market, processors come in a variety of different core configurations. Some configurations may be homogeneous, where every core is the same, or heterogeneous, where there are different core types within the same chip. Additionally, there may be a difference in the clocking speeds of the cores, regardless of whether the cores are the same across all processors.

The Snapdragon X processor comes in symmetrical or asymmetrical configurations. In a symmetrical configuration, all cores are the same and run at the same speed. In an asymmetrical configuration, all cores are of the same type, but there may be a mix of lower speed efficiency cores and higher speed performance cores. An example configuration might be a 12-core version which has 4 cores clocked at 2.5 GHz and 8 cores clocked at 3.4 GHz. To optimize gaming, it may be best to run rendering threads or time-critical code on the fastest processors.

The number of physical cores on an Snapdragon X system is always the same as the number of logical cores. This is important to keep in mind as gaming code may consider both the number of physical cores and the number of virtual cores when determining how many and what types of threads are created. For example, the Unreal Engine determines the number of foreground and background threads based on the number of virtual cores, which for Snapdragon X is the same as the number of physical cores.

The process to determine how many cores there are and the speed at which each is running may differ across operating systems. Generally, the best guidance is to use the least complex solution that obtains the desired information and consistently use it across the codebase. For Windows, there are several methods to determine the target CPU's topology. The Windows API provides several methods, such as `GetLogicalProcessorInformation` and `GetSystemCPUSetInformation`, that allow users to fully enumerate logical processors.

Example

The following example shows how to use `GetLogicalProcessorInformationEx` to query the number of processors and the speeds at which they run.

```
#include <Windows.h>
#include <iostream>

extern "C" {
#include <Powrprof.h>
}

#include <vector>

#pragma comment(lib, "Powrprof.lib")

typedef struct _PROCESSOR_POWER_INFORMATION {
    ULONG  Number;
    ULONG  MaxMhz;
    ULONG  CurrentMhz;
    ULONG  MhzLimit;
    ULONG  MaxIdleState;
    ULONG  CurrentIdleState;
} PROCESSOR_POWER_INFORMATION, * PPROCESSOR_POWER_INFORMATION;

typedef BOOL(WINAPI* LPFN_GLPI)(PSYSTEM_LOGICAL_PROCESSOR_INFORMATION
, PDWORD);
typedef BOOL(WINAPI* LPFN_GLPIEX)(LOGICAL_PROCESSOR_RELATIONSHIP, PSY
STEM_LOGICAL_PROCESSOR_INFORMATION_EX, PDWORD);
typedef DWORD(WINAPI* LPFN_GAPC)(WORD);

#define ALL_PROCESSOR_GROUPS 0xffff

// Helper function to count set bits in the processor mask.
DWORD CountSetBits(ULONG_PTR bitMask)
{
    DWORD LSHIFT = sizeof(ULONG_PTR) * 8 - 1;
    DWORD bitSetCount = 0;
    ULONG_PTR bitTest = (ULONG_PTR)1 << LSHIFT;
    DWORD i;

    for (i = 0; i <= LSHIFT; ++i)
    {
        bitSetCount += ((bitMask & bitTest) ? 1 : 0);
        bitTest /= 2;
```

```
    }

    return bitSetCount;
}

DWORD GetInstalledProcessorCount()
{
    // on Windows 7 and later, use GetActiveProcessorCount() ...

    LPFN_GAPC gapc = (LPFN_GAPC)GetProcAddress(GetModuleHandle(TEXT("
kernel32")), "GetActiveProcessorCount");
    if (gapc)
        return gapc(ALL_PROCESSOR_GROUPS);

    // on Vista and later, try GetLogicalProcessorInformationEx() n
ext ...

    LPFN_GLPIEX glpiex = (LPFN_GLPIEX)GetProcAddress(GetModuleHandle(
TEXT("kernel32")), "GetLogicalProcessorInformationEx");
    if (glpiex)
    {
        std::vector<BYTE> buffer;
        PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX info = NULL;
        DWORD bufsize = 0;

        // not using RelationGroup because it does not return accura
te info under WOW64...
        while (!glpiex(RelationProcessorCore, info, &bufsize))
        {
            if (GetLastError() != ERROR_INSUFFICIENT_BUFFER)
                return 0;

            buffer.resize(bufsize);
            info = (PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX)&buffer[
0];
        }

        DWORD logicalProcessorCount = 0;

        while (bufsize >= sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION
_EX))
        {
            // for RelationProcessorCore, info->Processor.GroupCount
is always 1...
```

```
            logicalProcessorCount += CountSetBits(info->Processor.Gro
upMask[0].Mask);
            bufsize -= info->Size;
            info = (PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX)(((LPBYT
E)info) + info->Size);
        }

        return logicalProcessorCount;
    }

    // on XP and later, try GetLogicalProcessorInformation() next...

    LPFN_GLPI glpi = (LPFN_GLPI)GetProcAddress(GetModuleHandle(TEXT("
kernel32")), "GetLogicalProcessorInformation");
    if (glpi)
    {
        std::vector<BYTE> buffer;
        PSYSTEM_LOGICAL_PROCESSOR_INFORMATION info = NULL;
        DWORD bufsize = 0;

        while (!glpi(info, &bufsize))
        {
            if (GetLastError() != ERROR_INSUFFICIENT_BUFFER)
                return 0;

            buffer.resize(bufsize);
            info = (PSYSTEM_LOGICAL_PROCESSOR_INFORMATION)&buffer[0];
        }

        DWORD logicalProcessorCount = 0;

        while (bufsize >= sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION
))
        {
            if (info->Relationship == RelationProcessorCore)
            {
                // A hyperthreaded core supplies more than one logic
al processor.
                logicalProcessorCount += CountSetBits(info->Processor
Mask);
            }

            bufsize -= sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION);
            ++info;
```

```
        }

        return logicalProcessorCount;
    }

    // fallback to GetSystemInfo() last ...

    SYSTEM_INFO si = { 0 };
    GetSystemInfo(&si);

    return si.dwNumberOfProcessors;
}

int main(int argc, char* argv[])
{
    DWORD dwNumProcessors = GetInstalledProcessorCount();

    std::vector<PROCESSOR_POWER_INFORMATION> a(dwNumProcessors);
    DWORD dwSize = sizeof(PROCESSOR_POWER_INFORMATION) * dwNumProcess
ors;
    std::cout << "Number of processors: " << dwNumProcessors << std::
endl;
    CallNtPowerInformation(ProcessorInformation, NULL, 0, &a[0], dwSi
ze);
    std::cout << "Processor\tMHz\tMaxMHz\tMHz Limit\tIdle State\tMax
Idle State" << std::endl;
    for (int i = 0; i < dwNumProcessors; i++)
    {
        std::cout << a[i].Number << "\t\t";
        std::cout << a[i].CurrentMhz << '\t';
        std::cout << a[i].MaxMhz << '\t';
        std::cout << a[i].MhzLimit << "\t\t\t";
        std::cout << a[i].CurrentIdleState << "\t\t";
        std::cout << a[i].MaxIdleState << "\t" << std::endl;
    }



    a.clear();

    system("pause");
    return 0;
}
```

Thread priority

Once the topology has been determined, developers can decide if certain threads should be run on higher performance cores. Thread priority has historically been used by game developers to accomplish two things: high priority, interrupt-style work and work directly related to the frame rate. Typically, these kinds of tasks are put on high priority threads to ensure that things like audio processing (bursty, but time-dependent work) or rendering (directly impacting framerate) work gets as many resources as possible. For processors with homogeneous, symmetrical cores, thread priority by itself may work fine. However, in Snapdragon X processors, which have asymmetrical cores (cores running at different frequencies), using thread priority alone may not result in optimum performance. Though a thread may be given high priority to run, the OS may wind up choosing to place that thread on a lower frequency core.

Thread affinity

Thread affinity attempts to restrict a particular thread to a set of cores, while still allowing the OS to move threads around the indicated subset. It is recommended that the developer allows the OS to determine the affinity of threads and programs run on Snapdragon X. Today's operating systems do a good job of balancing performance and power consumption and allowing them to control where processes are run is typically the best course of action.

In those special cases where more control is needed, some major game development platforms may already have software infrastructure to set the affinity for certain threads or tasks. Unreal Engine 5 has implemented functions to set affinity for all the platforms it supports. The Unreal GameThread, for example, can have its affinity mask set via the `FPlatformAffinity::GetMainGameMask()` function, which gets called during the initialization of the engine. There are also specific calls for setting the affinity mask for the rendering thread, RHI thread, audio thread, and foreground and background threads. This can be overwritten in a platform specific file to return the appropriate mask for the specific Snapdragon X chip in use.

Best Practices

Compilation

Use updated compilers and set the flags correctly.

GCC	`-O2 -march=armv8.7-a`
Clang	`-O2 -march=armv8.7-a`
MSVC	`/O2 /arch:armv8.7 /Qpar`

Vectorization

Where possible, design with vectorization in mind. Leverage ISPC for cross-architectural, portable code. This not only allows the creation of cross-architectural code, but encourages the intentional implementation of vectorized code. Understanding how your game can be vectorized and building it in across all aspects of the code allows you to create the vectorized code that is high-level and maintainable.

Use an autovectorizing compiler for existing code and use the `-march=armv8-a` flag or above. Check to ensure generated code utilizes SIMD instructions.

Threading and affinity

Given the advanced nature of today's operating systems, it is recommended to allow the OS to decide the scheduling and affinity of threads running on Snapdragon X. For the rare cases where more control is required, there are OS-provided mechanisms to query the CPUs to determine which CPUs are performance cores and which are efficiency cores. Using this information, affinity can be set for particular threads.

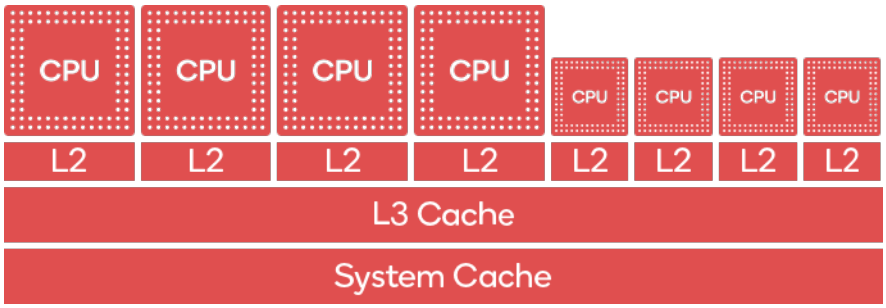
Qualcomm® Kryo™ CPU

Overview

Kryo CPUs are ARM-based big.LITTLE architecture. To optimize CPU utilization, we recommend looking at the Qualcomm Products and/or the Device Finder to identify the families of Kryo CPU chipsets you would like to target and which ARM Cortex CPU is it based on. For example, Qualcomm Snapdragon 865 has the Kryo 585 CPU which is based on the Cortex-A77.

ARM has family specific software optimization guides that provide details on how to optimize applications for those CPUs.

big.LITTLE Architecture



big.LITTLE architecture is composed of *big* cores which are optimized for performance and *little* cores optimized for power efficiency. Understanding this distinction and how to work with this

architecture is crucial to optimizing performance and power efficiency, which translates to longer play sessions and a cooler thermal profile for the game.

For additional information on big.LITTLE architecture visit https://www.arm.com/why-arm/technologies/big-little. To understand how to take advantage of thread placement on your game, refer to Threading and Core Affinity.

Understanding CPU utilization

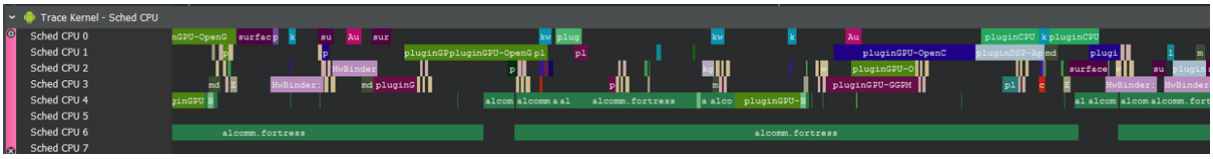
Measuring CPU Utilization of a game can be more nuanced than measuring GPU Utilization. Being ‘CPU bound’ does not necessarily mean the CPU is fully utilized. An important aspect of game development is understanding the threaded nature of the game and game engine.

Several games and engines are single or dual threaded by maintaining a main thread and a render thread at minimum. It is worth noting that when a game relies heavily on these threads to do all the work, the game can be bottlenecked by a single thread on a single core and become CPU bound.

On Snapdragon Profiler and an 8 core CPU platform, you might encounter a CPU Utilization metric reporting ~12.5% (100% / 8 Cores). This might not seem like the source of the bottleneck for this game – but look closer!

The following example game demo illustrates this scenario:

Using Snapdragon Profiler’s CPU Utilization Realtime metric does not seem to indicate a CPU bottleneck. Upon closer inspection, using Trace mode reveals one thread is fully utilized and almost all other cores are idle most of the time.



Threading and Core Affinity

Task scheduling on Kryo CPUs is performed at the hardware level. This scheduling can be guided by the OS through platform APIs that allow game developers to provide “hints” to the system to balance power and performance. On Snapdragon platforms, these hints are provided via the Heterogeneous Compute SDK which, among other useful APIs, provides developers the ability to control task execution and map tasks to a desired core or core type.

sched_setaffinity() from sched.h is another method of mapping threads to cores.

Often intelligently mapping threads to the appropriate cores will outperform letting Android schedule any thread on any core.

Little cores should be leveraged as much as possible. Given a frame budget of 16ms (60FPS) a developer can identify tasks using tools such as Snapdragon Profiler that would be good candidates to be moved to little cores.

For example, a game with a cloth simulation solver that takes 3ms to execute on a big core may take 10ms on a little core. As long as this execution time is acceptable (which in this case it is because our budget is 16ms), this task can and should be moved to the little cores. This both conserves power (with less drain on the battery) and leaves faster cores for more processing-intensive threads.

Another example: audio should usually be scheduled on a little core (the slowest), since such cores are usually more than fast enough for typical audio processing, particularly if that’s the only functionality the game requests of that core.

Prefer the biggest core for processing tasks that are sufficiently latency-sensitive to be too slow on the smaller cores – for example, the render thread should be scheduled on the fastest core for latency-minimization and maximizing frame pacing consistency; it’s the core least likely to be interrupted, and will process most quickly.

Power considerations

Another area to watch for is excessive or unnecessary wakeups of idle cores, which can result in more power consumption. One solution is to map tasks to the same core without overloading the core. Fortunately, runtimes for big.LITTLE platforms, such as those in Snapdragon mobile platforms, are engineered to employ three common task scheduling methods to help take care of such issues for you:

- **Core clustering**: Cores of the same size are treated as one cluster. The most appropriate cluster is chosen based on system demands. This can be efficient if the whole cluster migrates to the big core. However, this is less optimal than the following two methods because the CPU frequency driver tells the OS kernel the required frequency and voltage
- **In-kernel switching or CPU migration**: A big and a little core are paired into a virtual core in which only one of the two physical cores in that virtual core is used (which core depends on demand). This provides increased efficiency over core clustering, but requires that core

capabilities are the same

- **Global task scheduling**: All physical cores are available all of the time. The global task scheduler allocates tasks on a per-core basis depending on demands. This is the optimal method because the OS scheduler can allocate work on any core, all cores, or any combination, while unused cores are automatically turned off. This method does not require matching core configurations. It can react more quickly to load changes, and work can be allocated at a finer level of granularity than by the CPU frequency driver used in core clustering

Developers also have the ability to control power modes on Snapdragon platforms via the Power Optimization SDK. This SDK allows developers to adjust the power mode based on the game or task use case and performance/power demands with the following modes:

- **Efficient mode**: Achieves close-to-best performance with power savings
- **Performance burst mode**: Supports all cores at the maximum frequency for a short duration of time. It is useful for bursts of intensive computation to gain performance
- **Saver mode**: Provides around half of the peak performance of the system and helps when performance requirements of the application are small
- **Window mode**: Allows for fine tuning of performance/power balance using arguments to set the minimum and maximum frequency percentages relative to the maximum frequency that cores can use
- **Normal mode**: Returns system to its default state

NEON Intrinsics and SIMD

Arm Neon is an advanced SIMD architecture for ARM processors. The NEON instruction set makes a bank of specialized registers available for SIMD processing.

The instruction set includes typical SIMD operations for moving data between Neon and general purpose registers, as well as data processing and type conversion. Effective use of Neon through hand-coded assembly, intrinsic functions, or automatic vectorization by the compiler can lead to tremendous performance gains for multimedia applications. Qualcomm's SIMD/FPU co-processor is compliant with the Arm Neon instruction set.

For more information on Neon intrinsics, refer to https://developer.arm.com/architectures/instruction-sets/simd-isas/neon.

Interfacing with DSP and GPU

Some less-latency-sensitive processing can be offloaded to the DSP. This is generally more performant than any kind of GPU readback.

Best practices

Summary

- Schedule latency-sensitive, heavy threads (like rendering) on the fastest available core
- Schedule other threads (like audio) on the slowest cores that can handle them
- Use NEON Intrinsics
- Offload tasks to the DSP
- Don’t ship a CPU-side Vulkan memory allocator

Avoid single-thread 100% CPU Utilization bottleneck

Pay close attention to the threaded nature of your game and task system to avoid bottlenecking a single core with a thread while most other cores are idle or under-utilized.

Tools such as Snapdragon Profiler can help identify problematic areas using the Android Trace User Markers for instrumentation, or Snapdragon Profiler Sampling mode, which can generate a flamegraph of the most utilized functions across all cores.

Avoid disk I/O or blocking operations on main/render threads

Overusing main/render threads on Android can be problematic and can lead to ‘Application Not Responding’ (ANR) errors – especially when using disk I/O or blocking operations.

For more information on ANRs in Android refer to, ANRs.

Use Neon for SIMD operations

When possible, use Neon intrinsics. If you are using a third party game engine or middleware, check for Neon optimizations and enable them when possible.

Use big cores for compute intensive tasks

When a task requires a large amount of compute work, prefer the big cores. This can help alleviate bottlenecks on a single core and maintain a more consistent FPS.

Use little cores for power efficient tasks

Running most of the game's logic on little cores can lead to greater power benefits and better battery life, lower temperature, and prolonged gaming sessions by users.

Multi-thread game initialization

Game loading can be memory and compute intensive, which can lead to long wait times for end-users. Using multiple threads to distribute game loading can result in decreased load time and improve user experience.

Vulkan

Minimize dynamic memory allocation – ideally all memory requests from Android happen once at startup, and then are managed by the application from that point forward. This avoids fragmentation.

Don't ship a CPU-side Vulkan memory allocator – it will be less performant than the default allocator. (A CPU-side Vulkan allocator can still be useful during development/debugging, however).

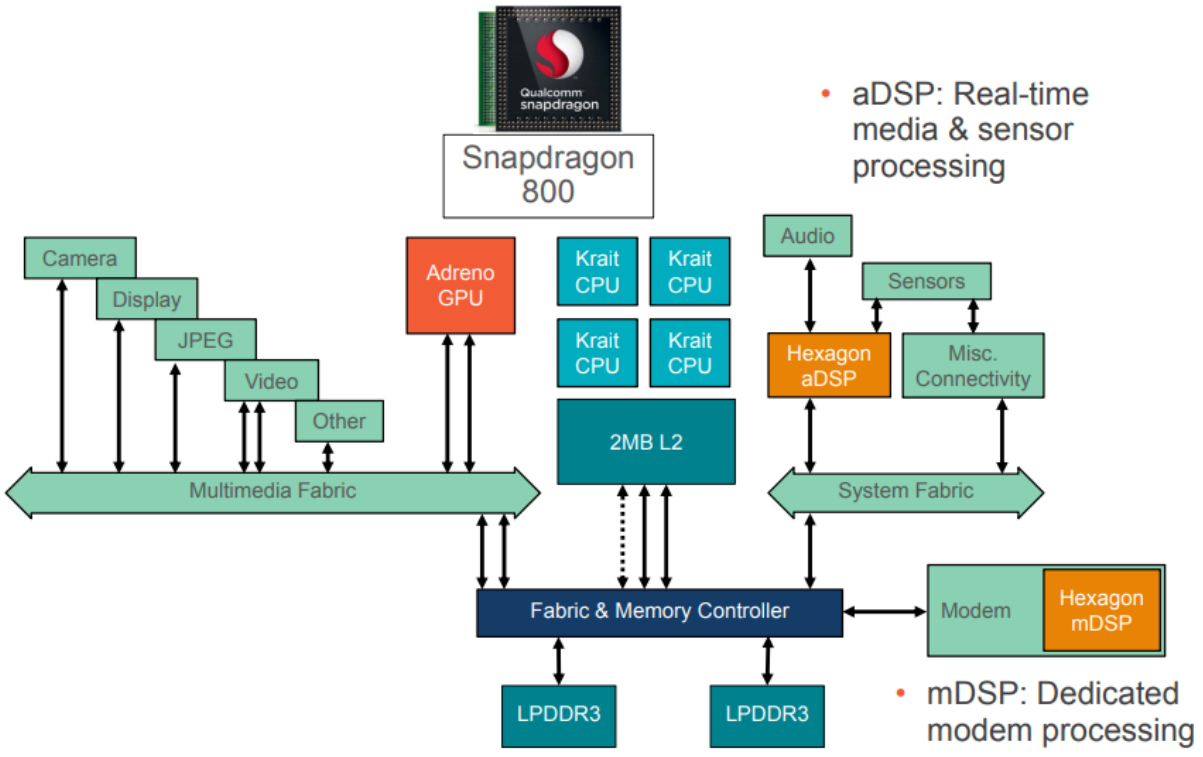
Qualcomm® Hexagon™ DSP

Overview

Game developers have another tool to create best-in-class mobile experiences that are uniquely available for Qualcomm platforms. Snapdragon SoCs have Hexagon, which is a programmable digital signal processor (DSP) alongside the Kryo CPU and Adreno GPU. Hexagon was designed for modem and multimedia functions, and is optimized for performance and power efficiency.

Instead of pushing performance with large frequency values, Hexagon allows for high throughput of work per cycle and a lower clock speed. While Hexagon is not designed for game development, its architecture and programmable nature allows game developers to offload tasks to it, reduce work on other cores, and conserve power.

This guide introduces basic concepts and techniques relevant for game development. For more in-depth information about programming for the Hexagon, please visit Hexagon DSP SDK.



FastRPC (Remote Procedural Call) and Android Native Hardware Buffers

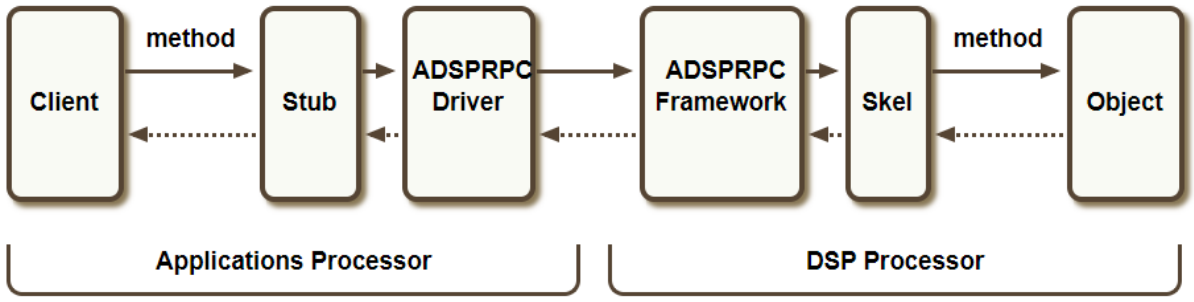
An important aspect of Hexagon, is that communication with DSP is done through inter-process communication (IPC). To accomplish this, using the FastRPC (Remote Procedure Call) library is required. This process has a higher latency than a roundtrip to the GPU, so a careful investigation of tasks that could be offloaded to the GPU is necessary to identify the best usage of this path and to stay within your game frame budget. The DSP does not share caches with other cores and data transfers must be done through DRAM. Handing buffers to and from the CPU, GPU, and DSP is streamlined in Android via Android Native Hardware Buffers which allow for no-copy buffer management.

Hexagon Programming Primer

Hexagon is programmed by the Hexagon SDK. This SDK comes with a comprehensive set of documentation describing Hexagon DSP, how to program for it, and includes sample applications.

As shown in the picture below, a DSP program is split into two main parts. The portion that interfaces with the client method in the application's space uses a *Stub*, which is the compiler generated functionality that interfaces with the DSP RPC driver.

It is worth noting that DSP offloading is serial at this point, and the calls done to FastRPC are blocking. The portion that interfaces with the DSP level is handled by an auto generated *Skel* code that takes care of unmarshaling parameters, and interfaces between executing in the application's DSP and the DSP RPC Framework.



Hexagon Vector eXtensions: HVX SIMD

One of the most powerful aspects that game developers can leverage to make optimal use of Hexagon is proper use of its co-processor. The co-processor allows for single instruction multiple data (SIMD) vector operations via the Hexagon Vector eXtensions (HVX) instruction set.

For example, in Hexagon DSP v66, SIMD operations execute on large vector registers of up to 1024 bits each. On top of this, multiple SIMD instructions can be executed in parallel.

For in-depth documentation on leveraging this HVX set, visit the HVX Reference Manual.

0.3 Components

Software to help make better software more quickly:

Snapdragon Profiler

To quick-start your performance analysis, go here.

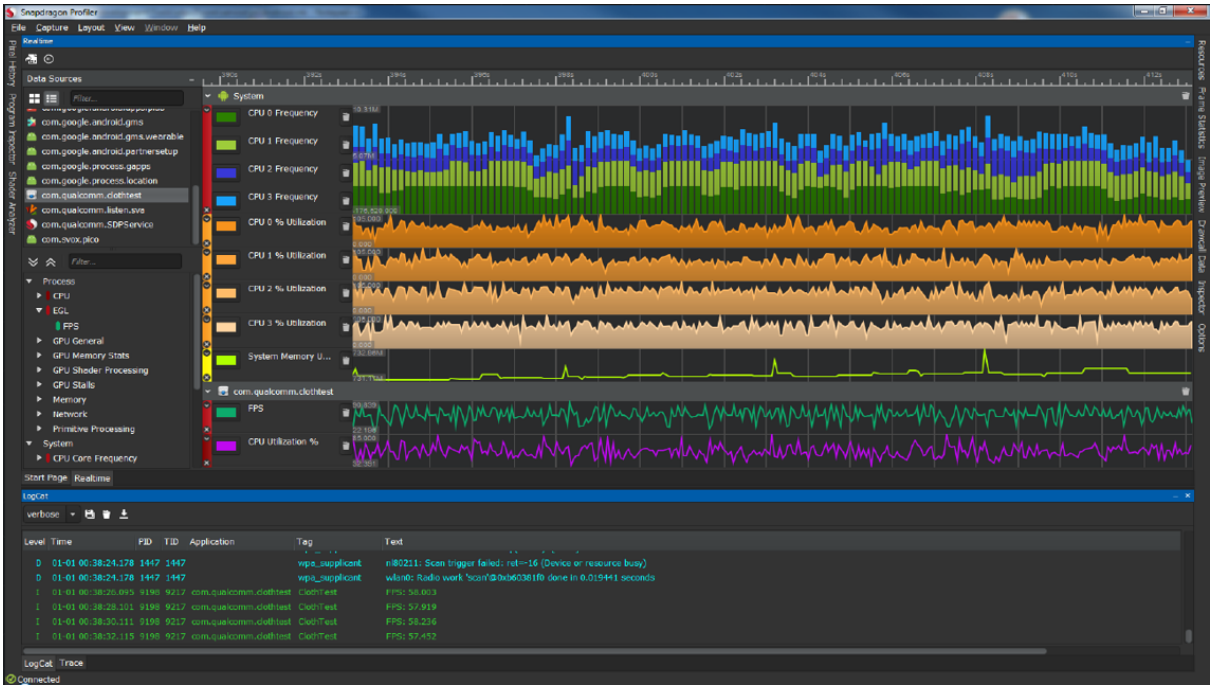
Improving performance of a 3D application is a challenging process. Without the proper tools, developers often find themselves resorting to trial and error to identify a bottleneck or the source of a visual glitch. Each attempt forces an application rebuild. This is a time-consuming and cumbersome process.

Developers rarely have access to a raw list of the graphics API commands that an application issues. This makes the optimization process tricky. The developer is likely to know and understand what their application does in general, but might not have complete visibility into the rendering process.

The reason for poor rendering performance is often unclear at first. Having real-time insight into detailed GPU utilization, texture cache misses, or pipeline stall statistics would make it easier to identify the reason for a slowdown. Sometimes the rendering process consists of many draw calls, making it difficult to identify specifically which of those draw calls is responsible for rendering a broken mesh. The ability to highlight the geometry that was drawn because of any specific draw call is helpful.

The following figures show how the tool works in three different modes, each suiting a different purpose.

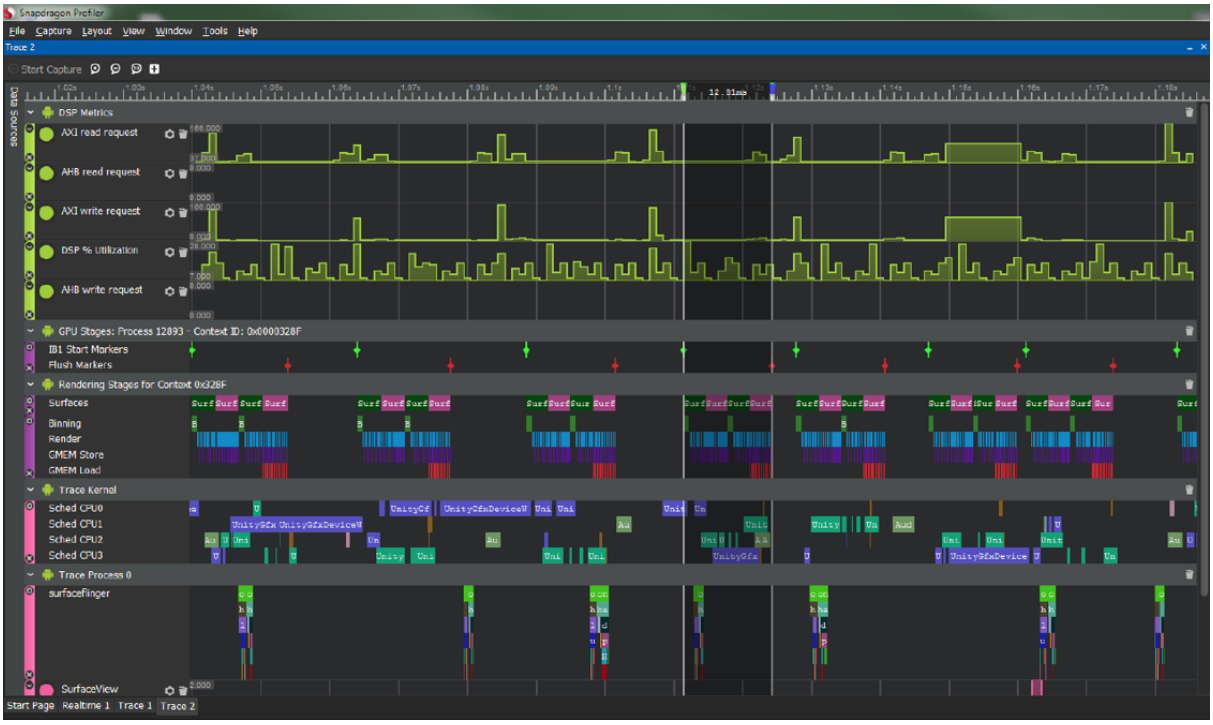
Realtime



Real-time performance visualization allows for a wide range of both system and per-process metrics and counters to be graphed in real time. Categories of metrics include:

- CPU
- EGL
- GPU
- Memory
- Network
- Power
- Primitive processing
- System memory
- Thermal

Trace



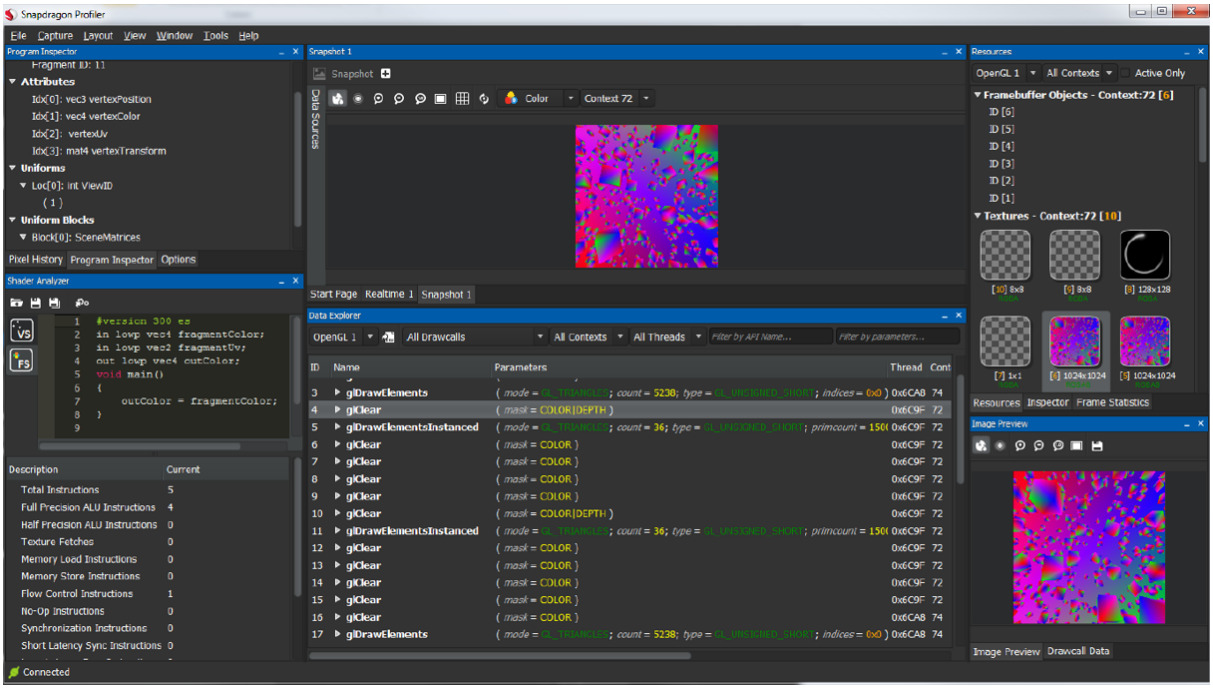
Trace allows for a specific time performance capture to provide a detailed visualization of the system and driver work being performed.

After capture, the user can zoom in, inspect, and measure detailed attributes. Traces can be captured from the following Android components:

- OpenGL ES
- DSP
- Activity Manager
- Audio
- Camera
- CPU
- Dalvik VM
- Disk I/O
- Graphics
- Hardware modules
- Input

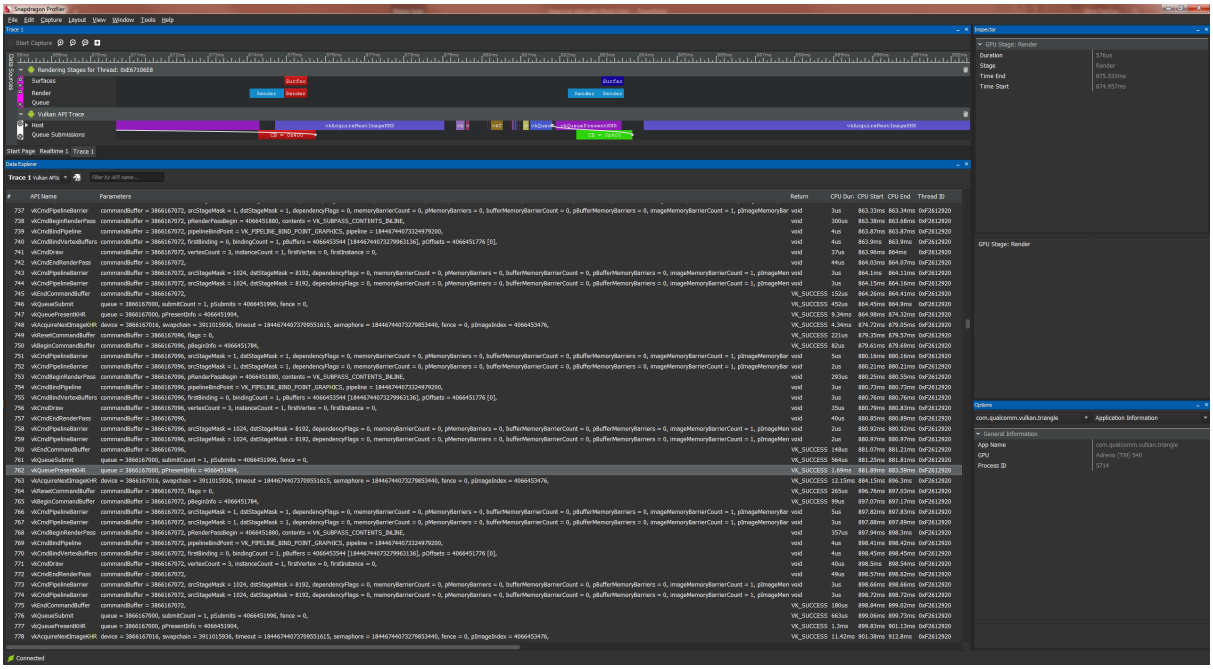
- Kernel Workqueues
- RenderScript
- Resource Loading
- Synchronization Manager
- Video
- View
- WebView
- Window manager

Snapshot



Snapshot mode provides the user with a complete view of a rendered frame. There is a detailed draw call list, resources (framebuffers, textures, shaders), shader complexity analysis, pixel history, overdraw analysis, texture preview, frame statistics, and more.

Vulkan in Snapdragon Profiler



A Vulkan application can be profiled with Snapdragon Profiler to get insights on Vulkan-specific rendering information. The user creates a trace and selects both Vulkan Rendering stages and Vulkan API trace metrics.

After a capture is made, a detailed graph is presented with the surface and rendering stages information, as well as a detailed table containing Vulkan API calls, parameters, and timing information.

You can find out more about Snapdragon Profiler and download a copy from the Qualcomm Developer Network at https://developer.qualcomm.com/software/snapdragon-profiler.

Quick Start with Performance Analysis

For your initial analysis, we recommend Tracing with the following metrics. (We also provide some metric values that are reasonable for many applications on many devices).

This will probably provide a good overview of any performance problems.

Metrics marked (Slow To Trace) introduce significant overhead – you might want to omit them from initial traces until they become more obviously relevant.

- **GPU Primitive Processing:**
  - Average Polygon Area – ideally at least 4, but not much larger than the bin dimensions
  - % Prims Clipped – Lower is better; ideally less than 2%

- % Prims Trivially Rejected – Lower is better; ideally less than 2%
- Reused Vertices – Higher is better; usually indicates indexed draws

•GPU Shader Processing:

- (Slow To Trace): % Linear Filtered – Lower is better; Linear Filtering tends to be expensive
- (Slow To Trace): % Nearest Filtered – Higher is better; Nearest Filtering tends to be performant
- % Shader ALU Capacity Utilized – Higher is better; ideally 50-100%
- % Shaders Busy – Higher is better; ideally 50-100%
- % Shaders Stalled – Lower is better; ideally less than 10%
- % Texture Pipes Busy – Minimize within aesthetic constraints for battery consumption – even low values like 20%-30% can sometimes bottleneck the system, but values nearing 100% can sometimes not be a bottleneck
- % Time ALUs Working – Higher is better; ideally 50%-100%
- % Time Compute – Higher is better; when compute is active, ideally this metric might report near 100%. The percentage of the frame compute is active will vary dramatically based on the app
- % Time EFUs Working – Higher is better; ideally at least 20%
- % Time Shading Fragments – Higher is better; if you're using a traditional vertex-and-fragment shading pipeline (instead of the compute-heavy GPU-driven approach), ideally this metric might report 100% for at least 60% of the frame
- % Time Shading Vertices – Higher is better; if you're using a traditional vertex-and-fragment shading pipeline (instead of the compute-heavy GPU-driven approach), ideally this metric might report 100% for less than 10%-20% of the frame
- % Wave Context Occupancy – Higher is better; ideally at least 50% on average (likely with spikes)
- ALU/Fragment – Higher is better (likely with spikes)
- ALU/Vertex – Higher is better (likely with spikes)
- Fragment ALU Instructions (Full) – Lower is better (ideally much lower than "Fragment ALU Instructions (Half)")
- Fragment ALU Instructions (Half) – Higher is better (ideally much higher than "Fragment ALU Instructions Full")
- Interpolation Instructions / Fragment – Lower is better (can be a source of stalls)
- Textures/Fragment – Lower is better (particularly on a vertex shader)

- **GPU Stalls:**
  - % Instruction Cache Miss – Lower is better; generally best to minimize, but in some cases even spikes as high as 80% may not bottleneck performance
  - % Stalled on System Memory – Lower is better; ideally usually less than 2%, perhaps with short spikes up to 30%
  - % Texture Fetch Stall – Lower is better; ideally usually less than 2%, perhaps with short spikes up to 20% (a sustained ~16% or higher is usually too high)
  - % Texture L1 Miss – Lower is better; ideally vacillating between 0% and under 50%, with occasional higher spikes
  - % Texture L2 Miss – Lower is better; ideally vacilliting between 0% and under 40%, with occasional higher spikes
  - % Vertex Fetch Stall – Lower is better; ideally usually 0%, with occasional spikes not exceeding 70%
- **Vulkan (for high-level visibility – binning passes are best kept to 10%-20% of renderpass; 30% is usually too much):**
  - Concurrent binning
  - Per drawcall stages
  - Rendering stages
  - Rendering Workloads
- **GPU General**
  - % CP Overhead – close to 0% at all times; should never exceed 20%
  - **GPU % Bus Busy – Varies:**
    - up to 25% average for a downclocked-GPU/battery-conscious app
    - up to 90% for a max-performance app
  - **GPU % Utilization – Varies:**
    - up to 40% average for a downclocked-GPU/battery-saver app
    - up to 90+% for a max-performance app
- **(Slow To Trace) GPU Memory Stats:**
  - Avg Memory Latency Cycles – Lower is better; large spikes indicate slow shaders
  - Texture Memory Read BW – Lower is better; large spikes indicate slow shaders
  - Vertex Memory Read – If binning is slow, this can indicate why (high reads = low bandwidth; low reads = stalls)

– Write Total – Lower is better; writes to main memory tend to be expensive

Testing Suggestions

Play the game for at least 10 minutes, tracking CPU and GPU clock-rates. If rendering workloads are relatively consistent, but clock-rates on the CPU and GPU get significantly reduced sometime during play (typically either both or neither processors will be affected), your device is experiencing thermal throttling. Chances are you'll want to optimize your game enough to avoid such thermal throttling.

If there is no thermal throttling, then after the tenth minute or so try to Trace a few of the most expensive frames for analysis.

Note that Snapdragon Profiler has a roughly 5% CPU overhead while just tracing framerate and nothing else – so keep in mind your frame timings will almost certainly be at least a little bit slower than playing the game without the profiler attached.

We recommending allowing at least 20 minutes of cooldown at room temperature (about 70 degrees F or 21 deg C) between testing sessions, particularly if your last session exhibited thermal throttling.

If your game has loading screens, be aware that loading screens usually have very different performance characteristics than in-game.

Vulkan Adreno Layer

Overview

The Vulkan Adreno layer detects optimizations that can be made on Adreno GPUs and offers suggestions on how to improve the usage of Vulkan APIs via logcat messages. Users can configure this layer to disable/enable rules using vkal_config.txt. See readme.txt for more details.

You can download the Vulkan Adreno Layer from the Qualcomm Developer Network at https://developer.qualcomm.com/software/adreno-gpu-sdk/tools.

The following fault codes are related to messages generated by the Vulkan Adreno Layer.

Fault Codes

Fault Codes	Description
VKADRENOFAULT001	Unknown struct provided in VkCommandBufferAllocateInfo
VKADRENOFAULT002	Unknown struct provided in VkSubmitInfo
VKADRENOFAULT003	Unknown struct provided in VkCommandBufferBeginInfo
VKADRENOFAULT004	Unknown struct provided in VkCommandBufferInheritanceInfo
VKADRENOFAULT005	Unknown struct provided in VkImageViewCreateInfo
VKADRENOFAULT006	Unknown struct provided in VkPipelineLayoutCreateInfo
VKADRENOFAULT007	Unknown struct provided in VkDescriptorSetLayoutCreateInfo
VKADRENOFAULT008	Unknown struct provided in VkPipelineShaderStageCreateInfo
VKADRENOFAULT009	Unknown struct provided in VkGraphicsPipelineCreateInfo
VKADRENOFAULT010	Unknown struct provided in VkComputePipelineCreateInfo
VKADRENOFAULT011	Unknown struct provided in VkDescriptorSetLayoutCreateInfo
VKADRENOFAULT012	Unsupported VkDescriptorType
VKADRENOFAULT013	Unknown object type in VkAdrenoLayerAppStats::AddToMap
VKADRENOFAULT014	Unknown object type in VkAdrenoLayerAppStats::AddToMapInternal
VKADRENOFAULT015	Unknown object type in VkAdrenoLayerAppStats::PrintUsage
VKADRENOFAULT016	Layer cannot enable support for VK_EXT_device_memory_report
VKADRENOFAULT017	Driver does not support VK_EXT_device_memory_report
VKADRENOFAULT018	Invalid VkDeviceMemoryReportEventTypeEXT value
VKADRENOFAULT019	Invalid VkObjectType value
VKADRENOFAULT020	VKAL_LOG with LogGroupsInternal is disabled. Use VKAL_LOG_INTERNAL instead
VKADRENOFAULT021	Disabled log. VKAL_LOG_INTERNAL must be provided with an external log group

Vulkan Debug Utils

The following warning and error codes are related to messages generated by VK_EXT_debug_utils.

Warning Codes

Warning Code	Description
VKDBGUTILWARN001	Framebuffer is not qualified for multipass
VKDBGUTILWARN002	Multipass disabled in Renderpass due to memory barriers
VKDBGUTILWARN003	Renderpass is not qualified for multipass due to a given subpass

Error Codes

Error Code	Description
VKDBGUTILERRO001	Shader compilation failed
VKDBGUTILERRO002	Shader linking failed

0.4 Tutorials

Platform-specific programming guides:

Android

Android OS on Snapdragon

Introduction

This guide is for Android game developers of native-C++ performance-intensive games that are intended to dominate the phone (unless the player is interrupted by a phone call, text, etc), and not interact with any other Android services, processes or Activity's.

These are Qualcomm's best recommendations for management of memory and your app's Android lifecycle.

Memory Management on Android

Snapdragon hardware uses unified memory, which means CPU and GPU allocations coexist in the same memory space. This means it's possible for GPU allocations to cause the CPU to run out of memory and vice versa.

Keeping this in mind, let's examine some of the tools available to profile CPU and GPU memory usage on Android.

Common engines and platforms provide their own memory profiling tools:

- **Unreal5** : https://docs.unrealengine.com/5.0/en-US/memory-insights-in-unreal-engine/
- **Unreal4** : https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/PerformanceAndProfiling/Profiler/
- **Unity :**
  - https://docs.unity3d.com/Manual/ProfilerMemory.html
  - https://blog.unity.com/engine-platform/analyzing-physical-memory-footprint-using-memory-profiler

- **Oculus Quest :**
  - https://developer.oculus.com/blog/getting-a-handle-on-meta-quest-memory-usage/
  - https://developer.oculus.com/documentation/unity/ts-gpumeminfo/
  - https://developer.oculus.com/documentation/native/android/ts-ovrmetricstool/

You may have some of the following questions (especially if you're not using one of the above engines):

In Kotlin/Java, what is the best approximation of "how much memory can my app allocate?"

```
ActivityManager::MemoryInfo.availMem - ActivityManager::MemoryInfo.threshold
```

. . . in bytes

https://developer.android.com/reference/android/app/ActivityManager.MemoryInfo

To be clear: "availMem" is not the amount of memory Android believes is available for your app – it's "availMem - threshold".

Note that even if the user is only playing your game, it's possible for background processes to allocate more memory at any time, reducing the amount of memory your app can consume. See the onTrimMemory() callbacks to deal with this scenario.

What is the easiest way to see what memory your app has allocated over time?

Android Studio's Profiler tracks a running app's memory footprint over time, classifying allocated memory into several broad categories (for example, Java, C++, and several subcategories of graphics): https://developer.android.com/studio/profile

For an in-depth CPU memory allocation analysis, the Perfetto tool has a learning curve, but is powerful: https://perfetto.dev/docs/quickstart/android-tracing

If neither the Android Studio nor Perfetto tools suit you, read on for Kotlin/Java and adb methods of profiling memory.

In Kotlin/Java what is the best approximation of memory your app has natively allocated?

```
Debug::MemoryInfo.getTotalPrivateDirty()
```

. . . in kilobytes. See: https://developer.android.com/reference/android/os/Debug.MemoryInfo

In Kotlin/Java, what is the best approximation of graphics memory used by your app?

```
Debug::MemoryInfo.debugMemoryInfo.getMemoryStat("summary.graphics")
```

. . . in kilobytes. See: https://developer.android.com/reference/android/os/Debug.MemoryInfo#getMemoryStat(java.lang.String)

In adb, what is the best approximation of total memory used by your app?

```
adb shell dumpsys meminfo <your_app_name>
```

. . . outputs something like:

```
                  Pss  Private  Private  SwapPss     Heap     Heap
 Heap
                Total    Dirty    Clean    Dirty     Size    Alloc
 Free
               ------   ------   ------   ------   ------   ------
------
 Native Heap  2136406  2136216        0        0  4214784  2114445  2
100338
 Dalvik Heap     1245     1228        0        0     2431      895
 1536
Dalvik Other      972      972        0        0
       Stack       64       64        0        0
      Ashmem        2        0        0        0
     Gfx dev     1304      912      392        0
   Other dev       12        0       12        0
    .so mmap     4217      376      620        0
   .apk mmap      398        0       36        0
   .ttf mmap       47        0        0        0
   .dex mmap     5082        4     4084        0
   .oat mmap      109        0        4        0
   .art mmap     4172     3788      172        0
  Other mmap       22        4        0        0
  EGL mtrack    44400    44400        0        0
   GL mtrack    19168    19168        0        0
     Unknown     1095     1092        0        0
       TOTAL  2218715  2208224     5320        0  4217215  2115340  2
101874
```

The relevant column, in kilobytes, is “Private Dirty”:

```
TOTAL:              2208224
```

Note that this doesn't include any memory that's shared between your app's process and other processes (for example, *.so libraries shared between multiple processes), and that while "Pss Total" adds a fraction of shared memory usage to "Private Dirty", it doesn't account for all of it. https://developer.android.com/topic/performance/memory-management

In adb, what is the best approximation of memory available across the entire device?

```
adb shell cat proc/meminfo
```

. . . outputs a number of fields; the relevant one is:

```
MemAvailable:       2093732 kB
```

Note that MemAvailable is the best approximation from adb, but is usually less accurate than ActivityManager::MemoryInfo in the app's Kotlin/Java code, as shown above.

In adb, what is the easiest way of approximating an app's memory footprint over time?

```
adb shell dumpsys procstats --hours 1
```

. . . outputs something like:

```
 * <your_app_name> / u0a280 / v1:
   TOTAL: 6.3% (1.1GB-1.6GB-2.1GB/1.1GB-1.6GB-2.1GB over 2)
     Top: 6.3% (1.1GB-1.6GB-2.1GB/1.1GB-1.6GB-2.1GB over 2)
(Cached): 0.65%
```

In adb, what's the best way to get the total graphics memory footprint across the entire device?

(Requires root access)

```
adb shell "cat /sys/class/kgsl/kgsl/page_alloc"
```

. . . outputs, in bytes:

```
611381248
```

Note that two invocations of this command "before running your app" and "after running your app" typically isn't equivalent to the "Graphics" field below, since Android's memory management includes purposeful overallocation, sharing common memory objects, and other features.

In adb, what's the best way of examining an app's graphics memory footprint?

```
adb shell dumpsys meminfo <your_app_name>
```

. . . outputs, in kilobytes:

```
Gfx dev:256448
EGL mtrack:44400
GL mtrack:263124
```

. . . and the sum of these three values is output like:

```
Graphics: 563972
```

In adb, what's the best way of getting a finer-grained picture of an app's graphics memory usage?

On some devices, with root access, it's possible to more precisely classify graphics memory allocations. However, this doesn't work on all Android installations due to kernels' varying security settings.

Attempt more precise classification with:

```
adb shell cat /sys/kernel/debug/kgsl/proc/<proc-id>/mem
```

For example, if process 4190 is an OpenGL app, then:

```
adb shell cat /sys/kernel/debug/kgsl/proc/4190/mem > C:\kgsl_mem.txt
```

. . . might produce allocations classified as one of:

- command
- texture
- gl
- any
- arraybuffer
- renderbuffer
- egl_surface
- renderbuffer

If process 4190 is a Vulkan app, it might produce allocations classified as one of:

- vk_any

- vk_cmdbuffer
- vk_descriptorpool
- vk_device
- vk_devicememory
- vk_image
- vk_querypool
- vk_queue

In adb, what is the best way of approximately reporting an app's memory leaks?

```
adb shell dumpsys meminfo --unreachable <your_app_name> | (find "unre
achable allocations")
```

. . . outputs something like:

```
1886402728 bytes in 3605 unreachable allocations
```

(This app purposefully leaked a lot of memory – more typically only kilobytes of "leaked" memory is reported. This command typically reports a small amount of "leaked" memory, even for apps that manage memory perfectly).

What is the best way of investigating an app's memory corruption?

Android Game Development Environment for Visual Studio investigates a wide array of memory errors: https://developer.android.com/games/agde/address-sanitizer

How can I instrument C++ malloc calls?

https://android.googlesource.com/platform/bionic/+/main/libc/malloc_hooks/README.md

Android Application Lifecycle Guidelines

NativeActivity Dependency Minimization

Ideally, your app keeps all its resources within its process – for example, within its NativeActivity. This guarantees that Android will reclaim all resources (memory and otherwise) upon closing the app.

Avoid launching other services as much as possible.

Lifecycle Callbacks

onDestroy() is not guaranteed to be called; don't use it.

onPause() means your app is backgrounded (though sometimes still fully visible, as in multi-window mode), and hence more vulnerable to being killed. onPause() is often given a very brief execution window, so don't use it – or, if you must, perform only fast operations (like logging the event).

onResume() means the user has restored the app after an onPause(). The app is now less vulnerable to being killed.

onStop() means the app is no longer visible, and is more likely to be killed than after onPause(). onStop() is where any resources outside of the Activity should be released to avoid leaking them (Android will reclaim all resources within the process, so such resources require no work from you).

If the user restores the app, the app receives onRestart().

onStop() won't be called if the system is under extreme memory pressure, in which case the system can kill the application process at any time. It is best to keep all resources in the NativeActivity (which keeps them in the app's process).

onStop() is a good place to perform any "save-game" processing so the player doesn't lose progress if the app is killed – even though this event may not occur when "under extreme memory pressure":

For more, see:

- https://developer.android.com/reference/android/app/Activity
- https://developer.android.com/guide/components/activities/activity-lifecycle

Lowmemorykiller Management

Android almost always gives an app one or more warnings before terminating it to redistribute the app's memory to processes Android considers more important.

Handling onTrimMemory() callbacks is the most reliable way of discouraging Android from killing your app, since as of this writing the newer MemoryAdvice method isn't supported across all devices.

TRIM_MEMORY_RUNNING_MODERATE, TRIM_MEMORY_RUNNING_LOW and TRIM_MEMORY_RUNNING_CRITICAL are warnings your app would ideally respond to by:

- reducing memory usage as much as possible
- performing any "save-game" processing if this has not been done recently, so that if the app is killed after all, the user loses little to no progress

TRIM_MEMORY_RUNNING_CRITICAL means your app might be killed – in some cases, without even calling onStop().

Upon receiving onStop() or any TRIM_MEMORY_RUNNING_* onTrimMemory() callback, we recommend performing any "save-game" processing such that the player doesn't lose progress if the app is killed.

Since TRIM_MEMORY_RUNNING_MODERATE, TRIM_MEMORY_RUNNING_LOW or TRIM_MEMORY_RUNNING_CRITICAL onTrimMemory() callbacks can happen repeatedly and quickly, consider performing "save-game" processing immediately upon first receiving such a callback, and then again every minute so long as these callbacks continue.

TRIM_MEMORY_UI_HIDDEN means that your app's UI is no longer visible, so this is a good place to release large resources that are used only by your UI. This is a lifecycle event notification and an opportunity to release UI memory – not a low-memory warning.

When your app is backgrounded, you might get some onTrimMemory() warnings (onTrimMemory() tends to return TRIM_MEMORY_COMPLETE repeatedly while the app is in the background, but unlike the TRIM_MEMORY_RUNNING warnings this does not necessarily indicate a low-memory scenario).

We recommend that after you perform any "save-game" processing, decide if you want to try reducing your app's footprint in response to these warnings, or just risk the app being killed and forcing the player to reload from the last saved state.

Prior to Oct 18, 2011 (Android 4.0/Ice Cream Sandwich), the onTrimMemory() callbacks don't exist, and ActivityManager::MemoryInfo.lowMemory flag is all you have to indicate a low-memory scenario that is more likely to kill your app – possibly without even calling onStop(). Hopefully you are comfortable ignoring devices with Android installations this old.

For more, see:

https://developer.android.com/topic/performance/memory

Multi-Window Disabling

We recommend you explicitly disable multi-window (for simplicity) with the following:

```
<application
    android:name=".MyActivity"
    android:resizeableActivity="false"
    android:supportsPictureInPicture="false"
```

If resizeableActivity is set to true, the activity can be launched in split-screen and free-form modes.

If the attribute is set to false (as we recommend), the activity does not support multi-window mode – and if the user attempts to launch the activity in multi-window mode anyway, the activity takes over the full screen instead.

For more, see:

https://developer.android.com/guide/topics/manifest/activity-element

Identifiers That Can't Change After Publishing

Know which identifiers your app cannot change after publishing:
https://android-developers.googleblog.com/2011/06/things-that-cannot-change.html

Understanding and resolving Graphics Memory Loads

Certain programming techniques that work well on platforms such as PCs and gaming consoles may not port well to mobile because of different hardware in mobile GPUs.

Graphics Memory (GMEM) Loads are among the most common problems affecting GPU performance in mobile applications. In this section, we show you how to use Snapdragon Profiler to find GMEM Loads in your application code.

In a nutshell

Clear or invalidate all framebuffer attachments. This will indicate to the GPU not to load tile data from system memory into GMEM.

What are Graphics Memory Loads (Unresolves)?

The tiling architecture pipeline of the Qualcomm® Adreno™ GPU includes a render pass. Each tile is rendered into GMEM during the render pass. Typically, the driver loads previous frame buffer data from main memory into GMEM for each tile. This operation is called a GMEM Load (or unresolve).

Why are Graphics Memory Loads expensive?

The problem is that every GMEM Load slows processing. If content of the frame buffer is cleared or invalidated, the driver can clear that tile in GMEM. While that involves an additional graphics call and its associated overhead, it is less expensive than loading the frame buffer back into GMEM for every bin being rendered.

The two main causes of GMEM Loads are:

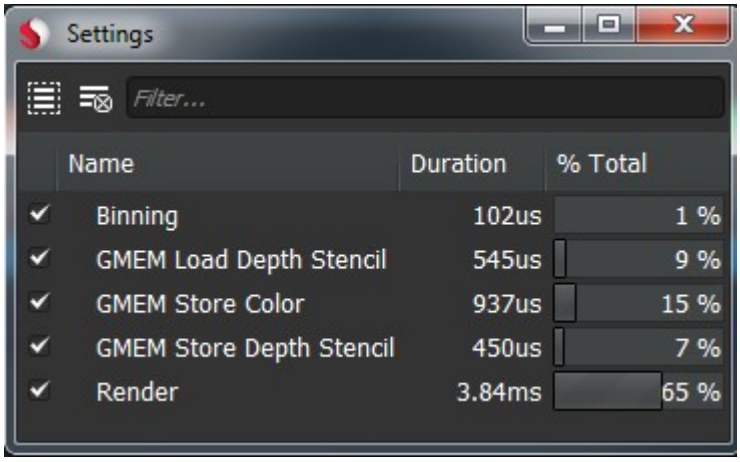
- **Improper hints to the driver** — The application code makes the driver think that previous content of the frame buffer is needed (usually by neglecting to clear the buffer). That boils down to a relatively easy fix that pays off well in reduced render time. It applies mostly to OpenGL ES programming since Vulkan handles the condition explicitly.
- **Algorithm** — Certain APIs such as glReadPixels and glFlush force pipeline flushes to get results. Doing this mid-frame will cause GMEM Loads when you resume drawing frame content. You can usually avoid GMEM Loads by modifying the algorithm.

Detecting Graphics Memory Loads in Snapdragon Profiler

Using Snapdragon Profiler in Trace Capture mode, you can allow the Rendering Stages metric to highlight GMEM Loads in their own track.

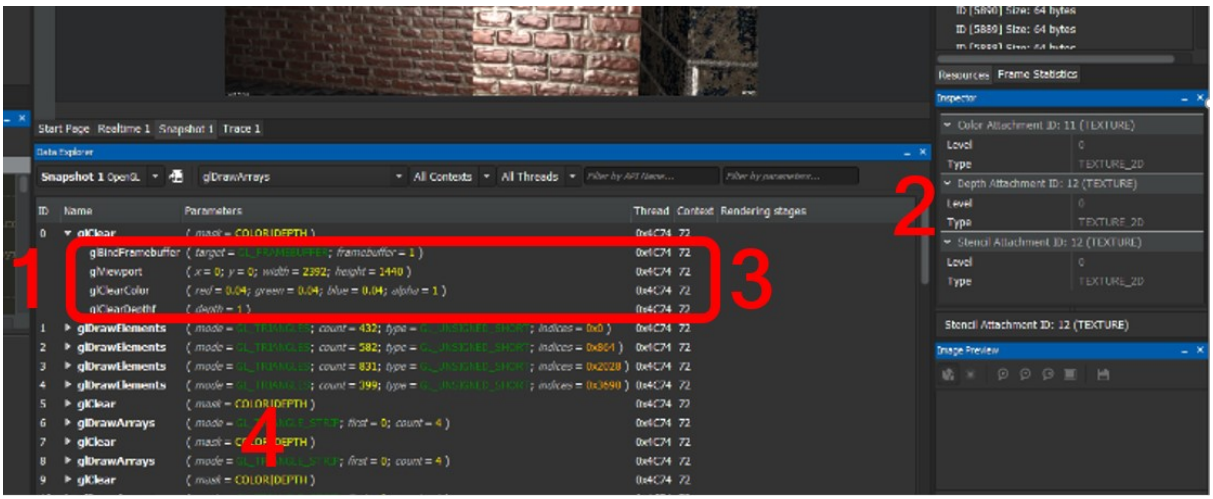
In the screenshot below (based on the Depth of Field demo application from the the Adreno SDK), red blocks show that GMEM Loads (Depth Stencil) are taking place as four different surfaces (0, 16, 32 and 48) are rendered:

The Settings dialog for Rendering Stages shows that those GMEM Loads take up about 9% of total rendering time.



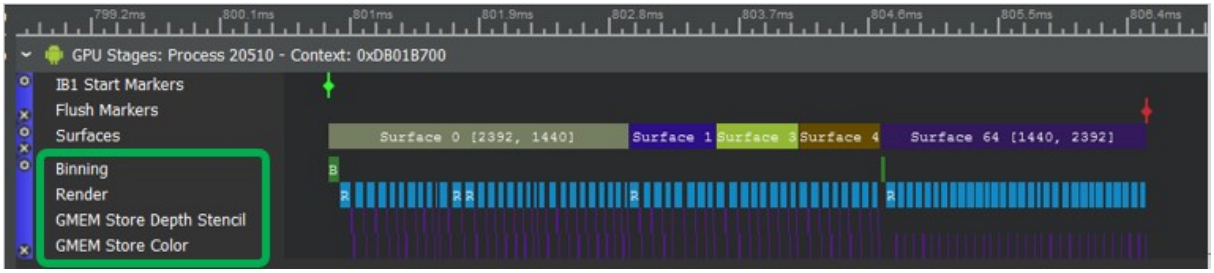
If GMEM Loads are unnecessary, you can reclaim about 9% of frame time.

You can then use Snapshot Capture mode in Snapdragon Profiler to determine the cause of the GMEM Loads.



The first surface with GMEM Loads is the first surface bound — glBindFrameBuffer. The framebuffer param is set to 1: # Select frame buffer object 1 (FBO 1) to examine Resources. The Inspector View shows that the surface has attachments for Color, Depth, and Stencil. # The glClearColor and glClearDepth calls leave the GPU thinking that the content of the Stencil attachment is relevant for the next frame. This causes GMEM Loads. Similarly, the other three surfaces (IDs 5, 7 and 9 here) have a Stencil attachment and do not clear it.

After modifying the code to explicitly clear Stencil content from the frame buffer, you can validate the results in Trace Capture mode, which no longer shows the GMEM Load Depth Stencil track:



In this case, rendering time is shortened by about 9%.

For more information on GMEM Loads (especially in extended reality apps), see the QDN blog post Profiling VR Apps for Better Performance.

Identify application bottlenecks

How many frames per second?

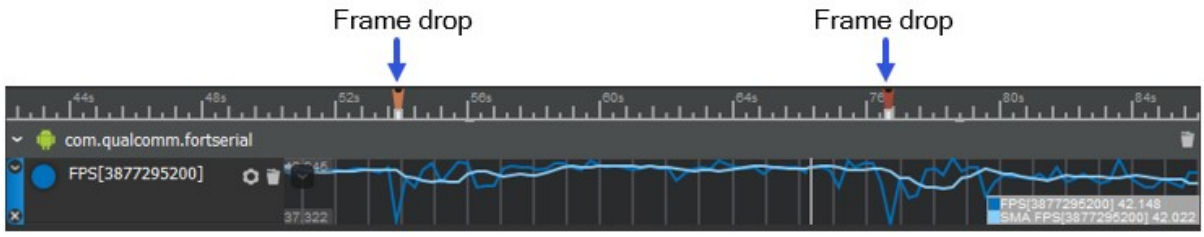
You may already know that you have a performance problem before you start using Snapdragon Profiler. Even if you do not, it is recommended to examine the app's current, overall performance to identify bottlenecks.

Frame rate is an ideal place to start. Games usually run best at 30 or 60 frames per second (fps), and sometimes higher for virtual and extended reality (VR/XR) applications.

One aspect of this is the average frame rate, which measures how fast the app runs on average. Another aspect is consistency of the frame rate. Even though your average stays close to your target frame rate, occasional long frames may miss that target. User experience then suffers from stuttering and glitches, and motion is not smooth, so your app would benefit from optimization.

If your app is not optimized, the average frame rate may be lower than your target and the app will not reach its desired performance level. If you have optimized your app, you may be closer to your average but still have periodic spikes in the frame rate. The spikes hamper animation, so you will want to smooth those out by identifying the problems and modifying your code.

In both cases, Snapdragon Profiler can take you straight to the performance level of your app in fps. The screenshot below shows an average (light blue line) of 42.022 fps.



While 42 fps may suffice as an average, the range (blue line) periodically falls as low as 37.322 fps. That suggests that the app is dropping frames, which harms performance.

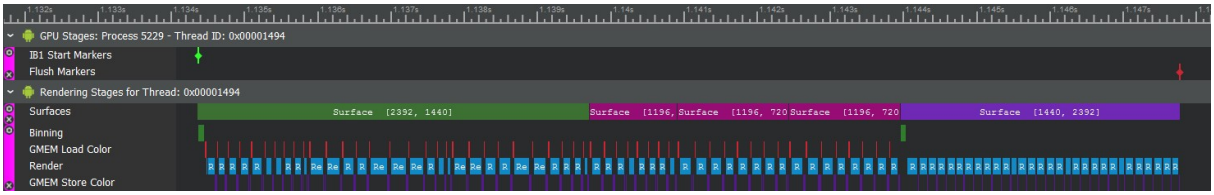
Note also that applications should choose to hit frame rates that are divisors or multiples of their platform's Vsync rate. Since a typical platform has a 60Hz Vsync, 30Hz or 60Hz are the only acceptable targets.

Exploring potential bottlenecks

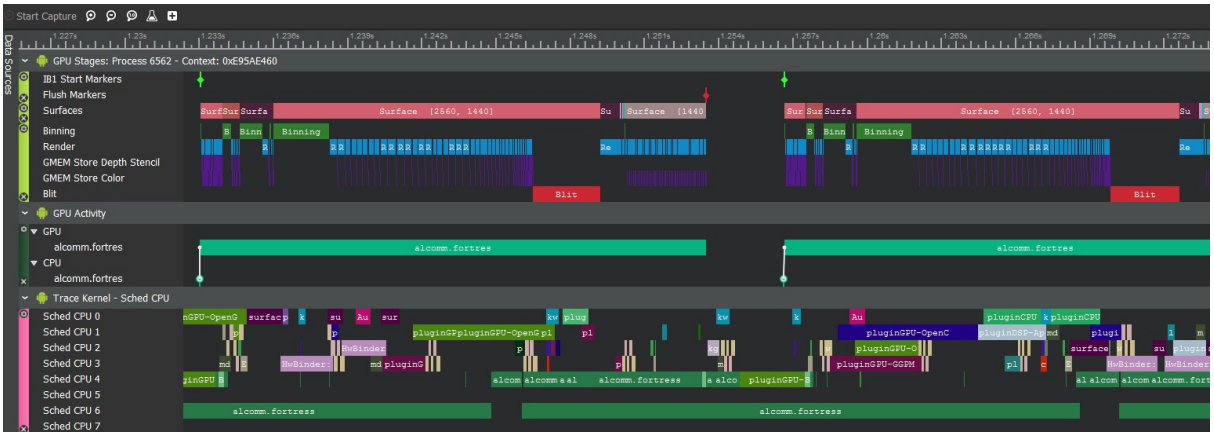
While no single metric can show you where your performance problems lie and how to address them, Snapdragon Profiler lets you examine dozens of metrics to understand how your app is interacting with the hardware.

The sections in the Trace Capture mode screenshot correspond to three important metrics to start with:

- **Rendering Stage** – Each Rendering Stage in Snapdragon Profiler is a metric that represents the app's execution on the GPU. Each data track is a subset of the metric, and the number of tracks varies depending on the application. In the following screenshot, the green, magenta, and purple bars show individual surfaces and the tracks under the Surface bars represent related rendering stages.



- **GPU Activity** – A system metric that shows the interaction between the CPU and GPU.
- **CPU Scheduling ("Trace Kernel - Sched CPU")** – A system metric that provides an overview of the app's execution on each CPU core. You can see which parts of your app are running where, and whether you have a scheduling or thread contention problem.



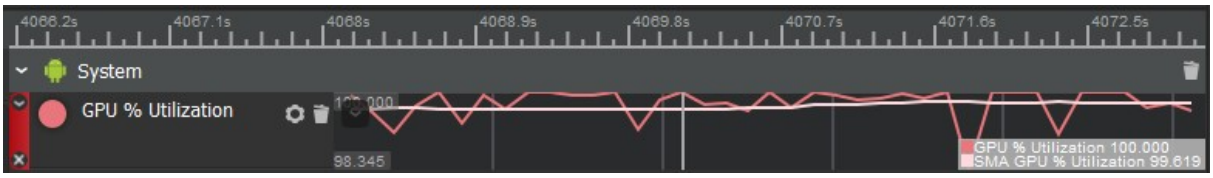
These metrics can help you explore three of the main areas where performance may be limited: the GPU, the CPU and Vsync, or the vertical sync refresh on the display.

GPU-bound application

In a graphics-intensive application, it is recommended to start the process of elimination with the GPU.

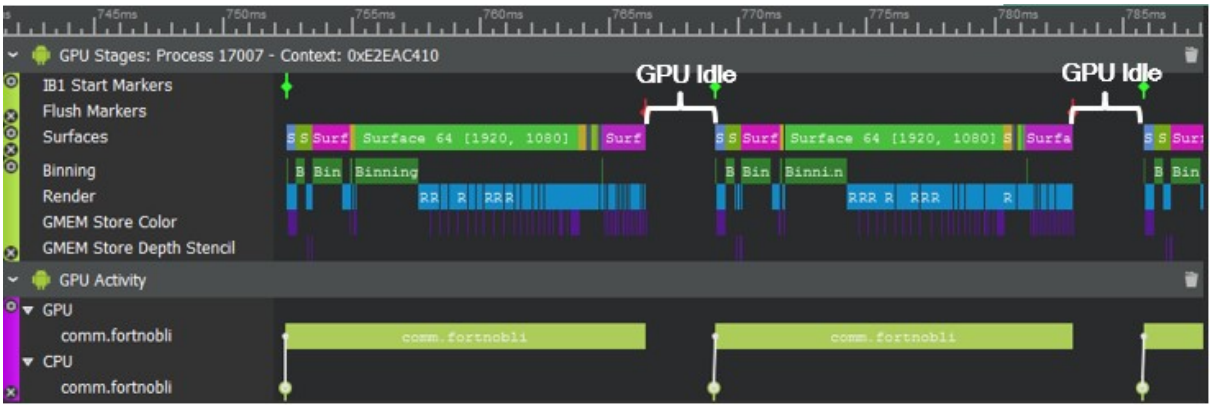
In the Real-Time view of Snapdragon Profiler, GPU % Utilization is a top-level indicator. The screenshot below shows utilization in the range of 26-38%.

Compare that to the following screenshot, showing utilization in the range of 98-100%:

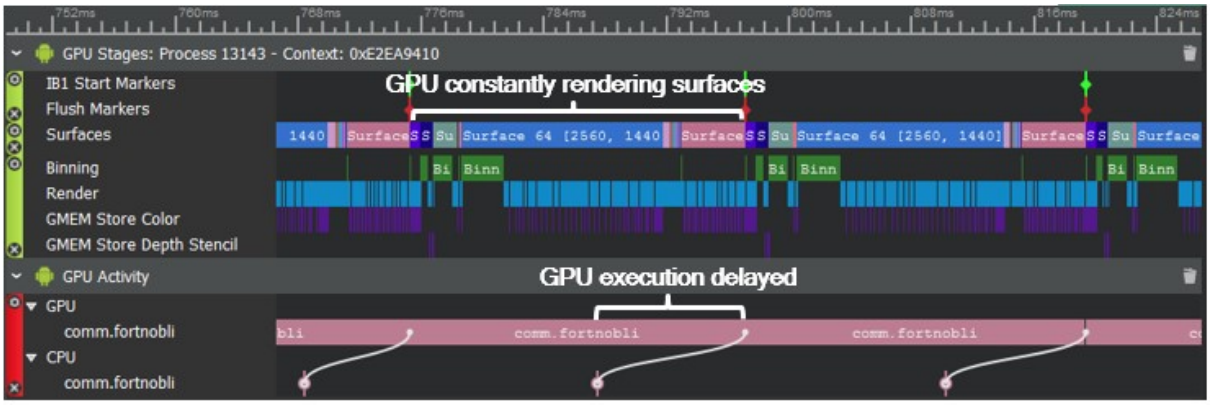


The latter is a strong indication that the app is GPU-bound.

Besides the Real-Time view, Trace Capture mode in Snapdragon Profiler offers another point of reference. If the app is not GPU-bound, gaps in GPU activity are likely to show up.



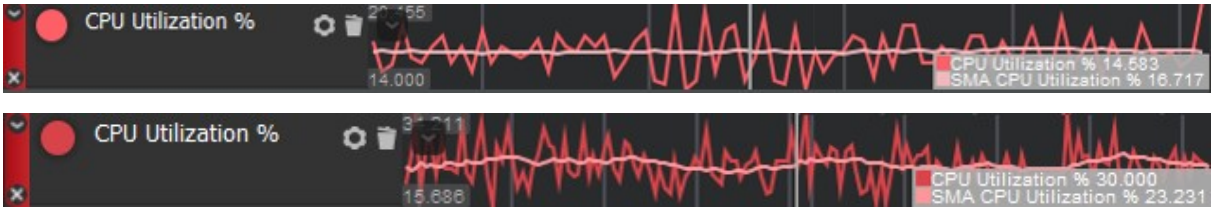
If the app is GPU-bound, then GPU execution might be delayed and the GPU might be constantly rendering surfaces.



CPU-bound app

If the app is not GPU-bound, determine whether the app is CPU-bound or not.

Unlike a GPU-bound app, the CPU % Utilization in the Real-Time view is not a reliable indicator of a CPU-bound app.



Average CPU utilization is 16%in the first app and 23% in the second. The first is CPU-bound, but the difference between them is not as striking as the GPU-bound app above, so the app may appear not to be CPU-bound.

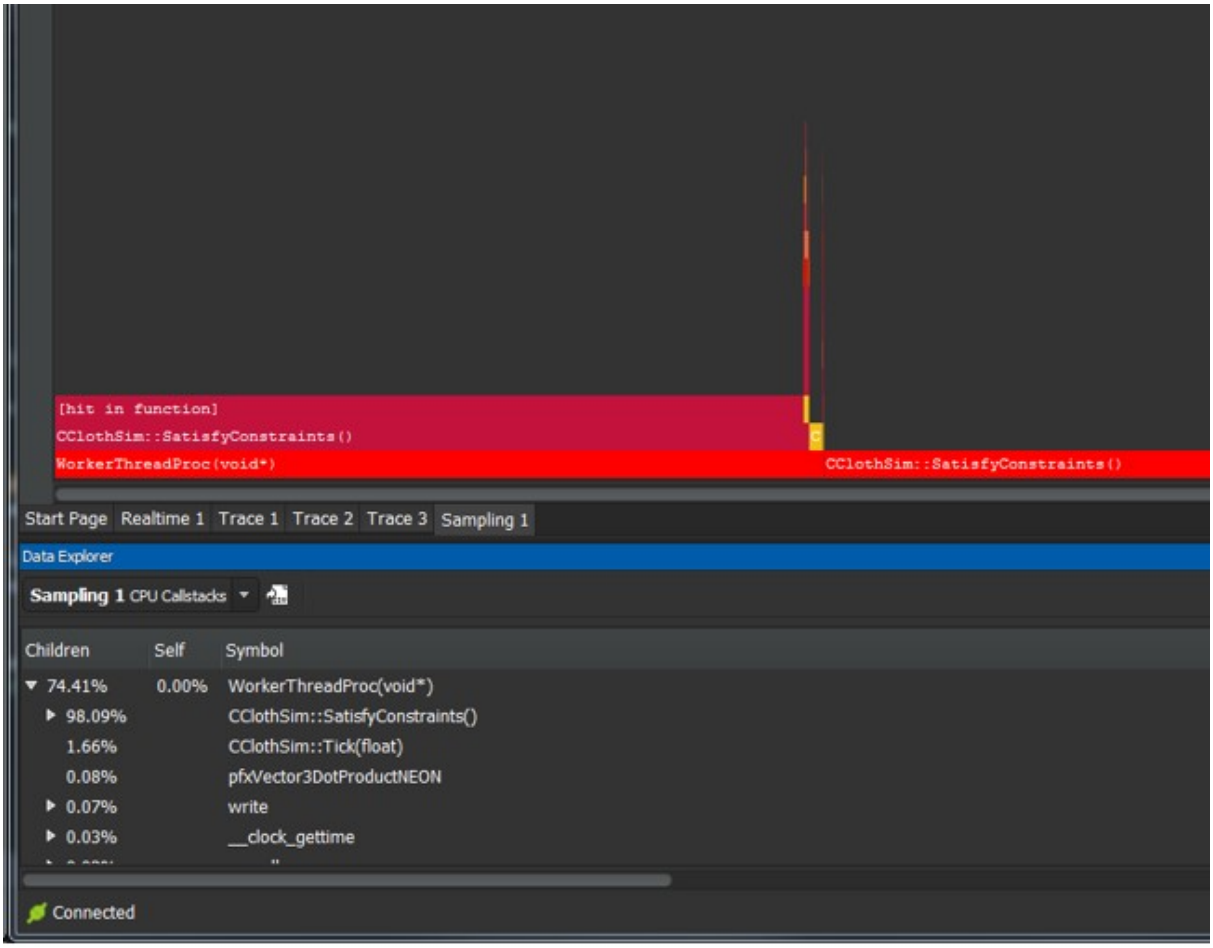
Two strong criteria for a CPU-bound app are that it is not GPU-bound and has an average frame time exceeding 16ms. To determine whether an app is CPU-bound, consider the multi-threaded nature of the app and the CPU. It is also helpful to go beyond CPU % Utilization and examine metrics like frequencies and thread scheduling.

In the Trace Capture mode screenshot below, the thread in Sched CPU 6 appears to bottleneck the CPU:

The next step is to look deeper into that thread to identify hotspots and candidates for multi-threading. The Sampling capture mode periodically samples the CPU program counter at a fixed interval and identifies CPU hot-paths. It offers a statistical representation of activity, including the time spent in each function and library. You can see which functions in your code take the most

execution time.

The following capture shows both the functions and the sequence of functions running on the CPU to render cloth textures. The red and orange blocks indicate hotspots in the app code.



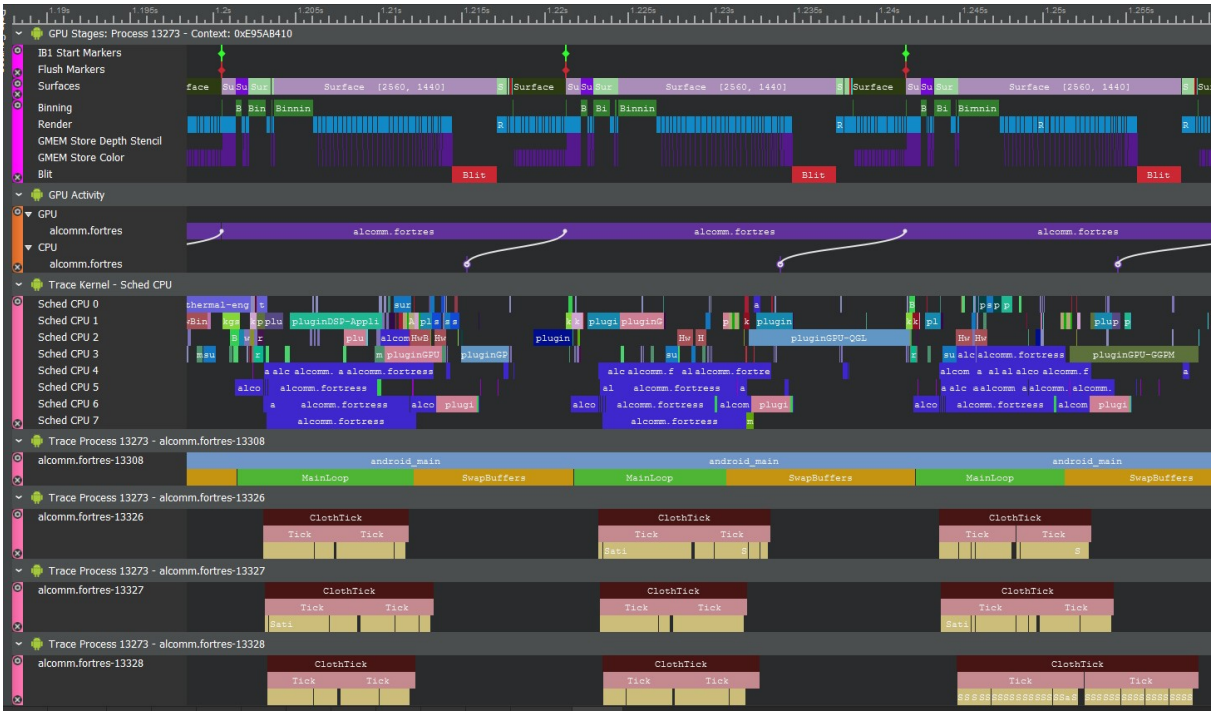
In this case, CPU sampling shows SatisfyConstraints() as the biggest hotspot, with 98% of activity.

To delve even deeper, Snapdragon Profiler supports user markers and the Native Tracing API built into the Android NDK. Android developers can use that API to insert trace markers into the app code and see this data in Snapdragon Profiler.

In the folllowing example, you can use android/trace.h to instrument the SatisfyConstraints() call and trace it back to the main thread, which is a worker function.

A texture in the rendered frame requires one function per cloth, which ticks as needed for the elapsed time. Once the app code is modified to thread the functions out, Snapdragon Profiler can

display the following inclusive, correlated view of all activity at once.

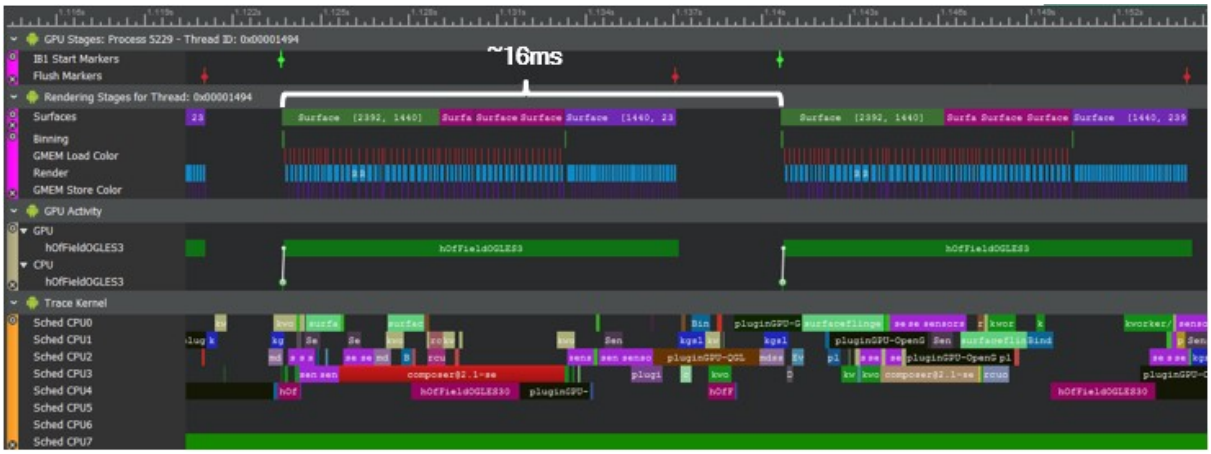


Now, cloth computation is taking place in multiple threads. The output in CPU Scheduling shows more threading. The app is no longer CPU-bound, so the CPU waits for draws to be dispatched.

That process is typical for identifying, diagnosing, and solving a performance problem with Snapdragon Profiler.

Vsync-bound app

If you determine that the app is neither CPU nor GPU-bound, then it may be Vsync-bound, i.e., it may be running as fast as the display hardware can accommodate. Using Trace Capture mode in Snapdragon Profiler, you might see something similar to this:



Note that the frame time is right around 16ms (1 second divided by 60 fps) and there is a gap near the end of the frame during which both GPU and CPU are waiting for an available surface to render.

While the goal of many applications is to run as fast as the display will allow, there are still opportunities for optimization even when an application is Vsync-bound and with benefits beyond a higher frame rate.

Most problems in mobile applications come down to power consumption battery life. Even if your application can meet the target frame rate by being Vsync-bound (instead of CPU or GPU bound), you may still want to optimize your code. If it can be modified to use less power and run at a lower temperature, the application would meet the target frame rate while doing less work. It might also offer better performance on lower-end devices.

Windows on Snapdragon

Windows on Snapdragon Detection

C++

The native architecture of the system can be queried using IsWow64Process2.

```
USHORT process, nativeMachineArchitecture;
BOOL success = IsWow64Process2(GetCurrentProcess(), &process, &native
MachineArchitecture);
if (success == FALSE) {
    std::cout << std::format("Error: IsWow64Process2() failed, GetLas
tError() {:x}\n", GetLastError());
```

```
    return 0;
}

const bool bWindowsOnSnapdragon = nativeMachineArchitecture == IMAGE_
FILE_MACHINE_ARM64;
```

Build architecture can be determined at compile time by checking for certain pre-processor definitions.

```
// Target processor architecture for build
enum class BuildArchitecture { X86, X64, ARM64, ARM64EC, Unknown };

// Get build architecture for this build
constexpr BuildArchitecture gBuildArchitecture =
// Microsoft-specific predefined macros
#if defined(_X86_)
BuildArchitecture::X86
#elif defined(_M_ARM64)
BuildArchitecture::ARM64
#elif defined(_M_X64) && defined(_M_AMD64) && defined(_M_ARM64EC)
BuildArchitecture::ARM64EC
#elif defined(_M_X64) && defined(_M_AMD64) && !defined(_M_ARM64EC)
BuildArchitecture::X64
#else
BuildArchitecture::Unknown
#endif
;
```

Once native machine architecture and build architecture are known, the two can be compared to determine whether running in emulation.

```
// ...

// Determine if x86/x64 application is running in emulation
const bool bEmulation = bWindowsOnSnapdragon && (
    gBuildArchitecture == BuildArchitecture::X64 ||
    gBuildArchitecture == BuildArchitecture::X86 ||
    gBuildArchitecture == BuildArchitecture::ARM64EC );

std::cout << std::format("Windows on Snapdragon: {}\nEmulation: {}\n"
, bWindowsOnSnapdragon, bEmulation);
```

Adreno GPU Detection

This DXGI example showcases how to detect the Qualcomm Adreno GPU, by enumerating adapters and checking for the Qualcomm vendor ID.

The Qualcomm vendor ID is the ASCII values for Q, C, O, and M, which evaluates to `0x4d4f4351`.

```
// Qualcomm Vendor Id
constexpr UINT QCOMVendorId = 'Q' | ('C' << 8) | ('O' << 16) | ('M' <
< 24);
```

Create a DXGI Factory which is needed to enumerate adapters.

```
// Create DXGI factory to enumerate adapters
winrt::com_ptr<IDXGIFactory7> factory;
constexpr UINT dxgiFactoryFlags = 0;
HRESULT result = CreateDXGIFactory2(dxgiFactoryFlags, IID_PPV_ARGS(&f
actory));
assert(SUCCEEDED(result));
```

Now check each adapter for the Qualcomm vendor ID.

```
// For each adapter, check if Qualcomm vendor id
winrt::com_ptr<IDXGIAdapter1> adapter;
BOOL bQCOMAdapterFound = FALSE;
for (UINT adapterIndex = 0;
    DXGI_ERROR_NOT_FOUND != factory->EnumAdapters1(adapterIndex, adap
ter.put());
    ++adapterIndex)
{
    // Get adapter description to check Vendor ID
    DXGI_ADAPTER_DESC1 adapterDesc{};
    adapter->GetDesc1(&adapterDesc);

    // Disregard Basic Render Driver adapter
    if (adapterDesc.Flags & DXGI_ADAPTER_FLAG_SOFTWARE)
    {
        continue;
    }

    // Use first Qualcomm adapter
    if (adapterDesc.VendorId == QCOMVendorId)
    {
        bQCOMAdapterFound = TRUE;
```

```
        std::wcout << std::format(
            L"Qualcomm Adreno GPU found:\n"
            L"\tDescription         = {}\n"
            "\tVendor ID           = {:X}\n"
            "\tDedicated Video Mem = {} bytes\n"
            "\tDedicated Sys Mem   = {} bytes\n"
            "\tShared Sys Mem      = {} bytes\n",
            adapterDesc.Description,
            adapterDesc.VendorId,
            adapterDesc.DedicatedVideoMemory,
            adapterDesc.DedicatedSystemMemory,
            adapterDesc.SharedSystemMemory);

        break;
    }
}
```

Once the Adreno GPU adapter is found, the D3D11 or D3D12 device can be created using the adapter.

```
// If Qualcomm GPU adapter available create the device
if (bQCOMAdapterFound)
{
    // Create the device using the found Qualcomm Adreno GPU adapter
    // using D3D12CreateDevice or D3D11CreateDevice
}
else
{
    std::wcout << "No Qualcomm GPU found\n";
}
```

Get Started with Windows on Snapdragon Development

Introduction

In the last several years, Arm chips have made the leap from powering consumer cell phones to laptops. Powerful, lightweight, and always connected, Arm presents a new future for personal computing devices that users can take with them anywhere. There are numerous Windows 10 Arm devices on the market from manufacturers such as Microsoft, Lenovo, and Samsung. Users can run 32-bit `x86` and 64-bit `x86-64` versions of their favorite programs in emulation, but that comes with performance trade-offs. As the install base grows, there is increasing demand for applications and games that can run natively on this hardware and take advantage of all it has to offer. Developers want software to be available to as many potential users as possible, and for that software to run without compromise. However, additional platforms often carry a significant amount of additional work.

This general development guide describes the work and collaboration between Microsoft and Qualcomm® to make porting software to Windows 10 on Arm as streamlined as possible. The following sections walk users through these processes and features from the comfort of their current x86-64 development environment:

- Necessary setup for Visual Studio
- Adding `Arm64` as a build target to a project
- How to enable remote debugging on an Arm device to assess performance and issues on `Arm64`
- Key platform details that set `Arm64` apart from `x86-64` and `x86` systems that will be important to consider during development

---

Note: This guide assumes existing knowledge and experience with software development, basic familiarity with Visual Studio and associated tools, and an understanding of `x86`/`x86-64` systems. It also assumes that development work will be performed on an `x86-64` Windows 10 device. However, it is also possible to run Visual Studio natively on `Arm64` as of Visual Studio 2022 version 17.4 - see https://learn.microsoft.com/en-us/visualstudio/install/visual-studio-on-arm-devices

---

Getting Started in Visual Studio

This section provides the necessary steps to enable Arm64 Development in Visual Studio. Visual Studio 2022, Visual Studio 2017, and Visual Studio 2019 are configured similarly. Most examples provided in this guide use Visual Studio 2017.

Prerequisites

- A 64-bit host development system running Windows 10. This is where you will do your development and the bulk of debugging.
- A target system running Windows 10 Arm64. This is where you will ultimately run and test software.

Installing and configuring Visual Studio on the Development Host

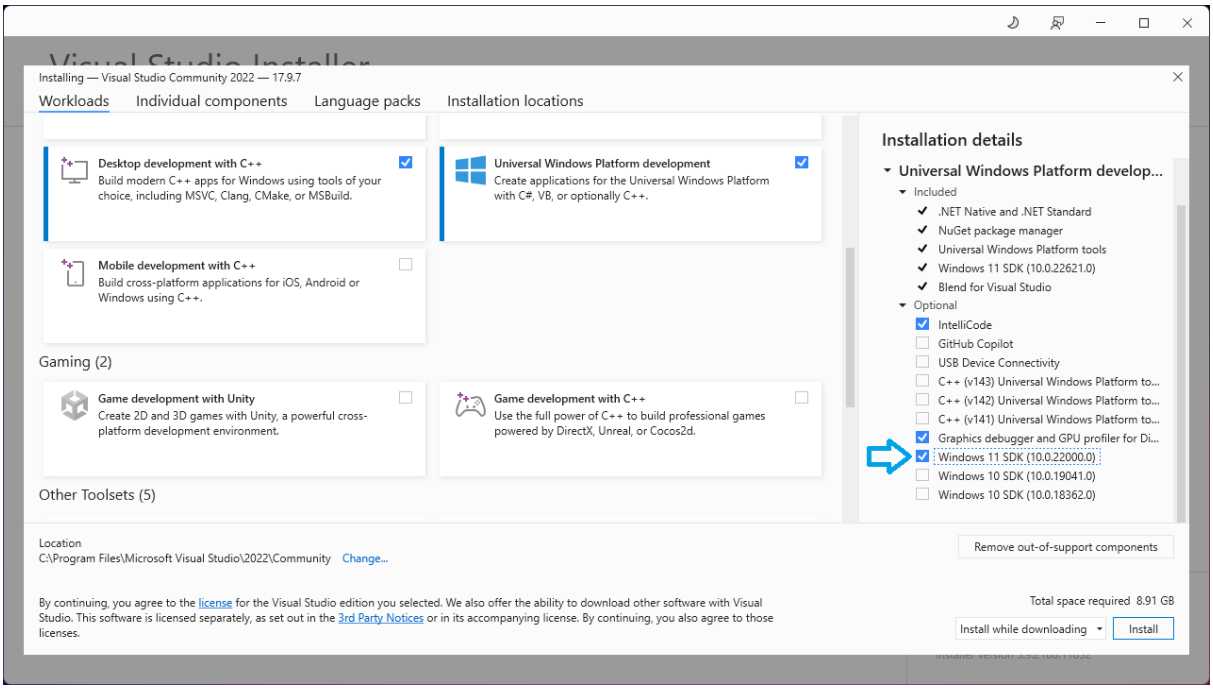


Figure19 Visual Studio Installer

1. Run the latest version of Visual Studio Installer.
2. Install Visual Studio with the following workloads:
    - Desktop Development with C++
    - Universal Windows Platform Development
    - Game development with C++ (for game development only)
3. If using Visual Studio 2019 or 2017, install the following individual components:
    - Visual C++ compilers and libraries for Arm
    - Visual C++ compilers and libraries for Arm64

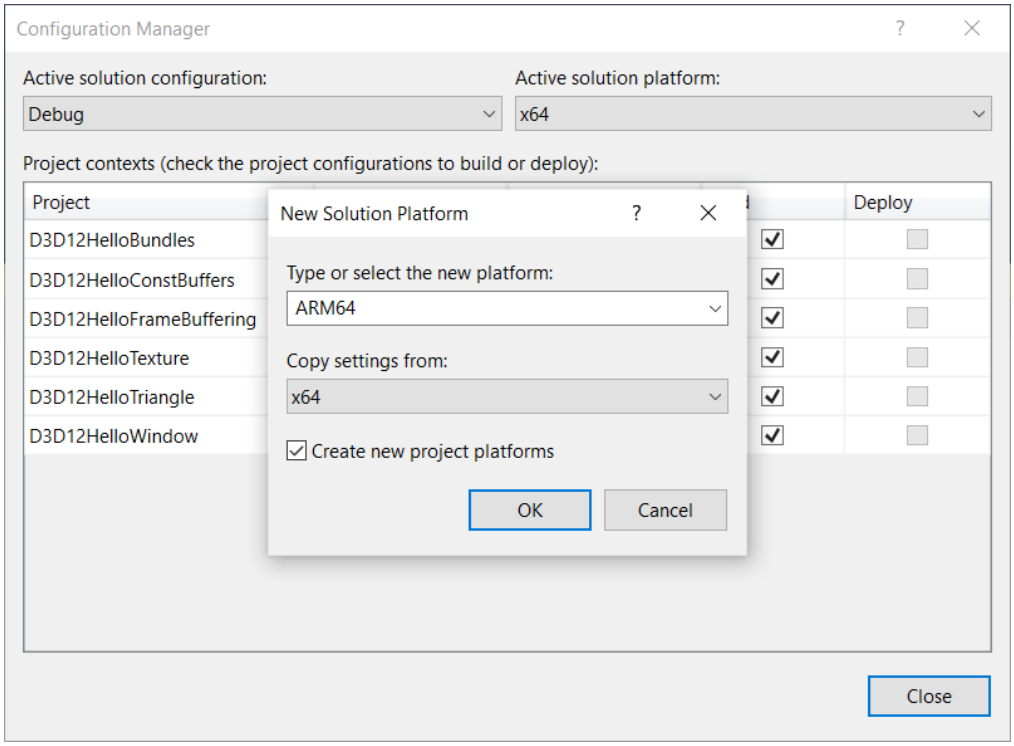
- Visual C++ runtime for UWP
- C++ Universal Windows Platform tools for Arm64
- Latest Windows 11, or Windows 10 SDK (v10.0.18362 or newer)

Installing and configuring the Arm64 target device

Note: Ensure that the latest Windows updates are installed before configuring the Arm64 target device.

---

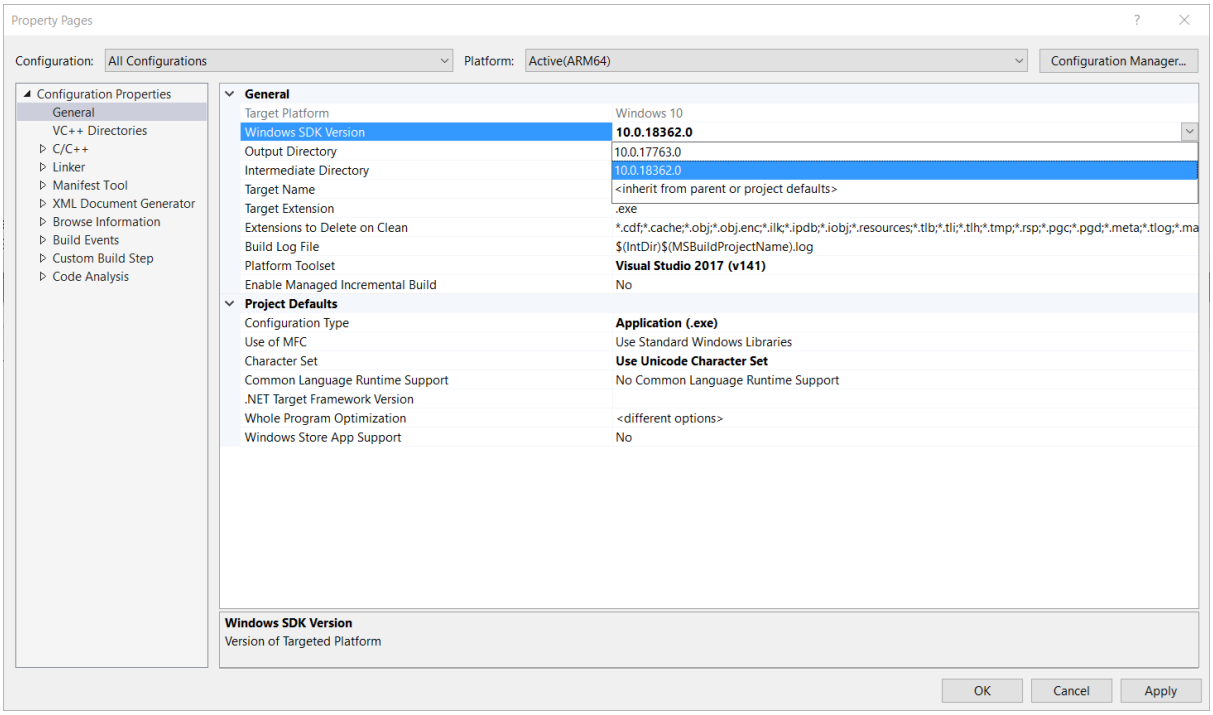
1. Download the Windows 11 SDK (or Windows 10 SDK v10.0.18362 or newer) from: https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/
2. During installation, select **Debugging Tools for Windows**.
3. Add the Arm64 platform to Visual Studio projects:
    a. In Visual Studio, click **Build > Open Configuration Manager**.
    b. In the **Active Solution Platform** list, select **<New. . . >**.
    c. In the **New Solution Platform** dialog box, select the following values:
        - **Type or select the new platform** list: **Arm64**
        - **Copy settings from** list: **x64**



d. Click **OK**.

4. Open the Property Pages of the project by either right-clicking the project in the Solution Explorer window, or from the View menu.

5. Update the Windows SDK version to 10.0.18362.0 or newer.



Debugging

On-device and remote debugging are both valid options for debugging an application. This guide focuses on remote debugging so that users can continue to work from the comfort of their development environment.

For more information about on-device debugging on Windows 10 Arm64, refer to: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-arm64

If you want to give on-device debugging a try, Microsoft has written a general starting guide to debugging on Windows 10 Arm 64.

Remote debugging with Visual Studio

Remote debugging allows users to debug their application directly from Visual Studio on their development host machine. This allows users easy access to their favorite tools even if they do not have native Arm64 equivalents.

Prerequisites

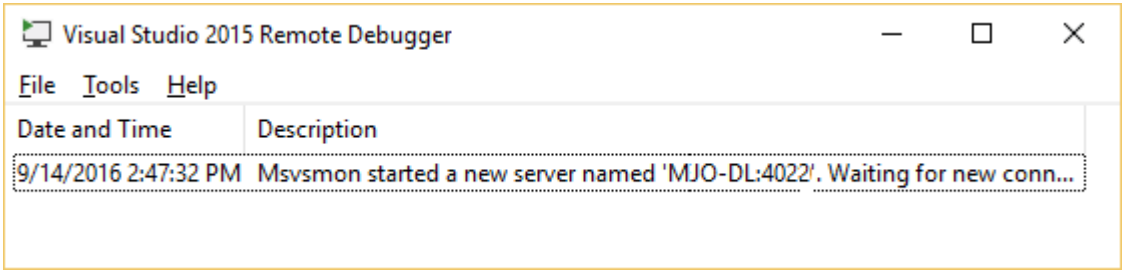
Ensure that the Development Host and Arm64 target device are on the same network (Domain, Private, or Public).

Setup

1. **Download and install the Arm64 or x64 version of Remote Tools:**
    - https://learn.microsoft.com/en-us/visualstudio/debugger/remote-debugging
2. Open **Remote Debugger**. If you do not have administrator privileges on the target device, right-click **Remote Debugger** and select **Run as Administrator**.
    a. On the first run, perform the following in the **Remote Debugging Configuration** dialog box:
        - Select at least one network type to use with Remote Tools
        - Install the Windows Web Services API (if not already installed)
3. Click **Configure remote debugging** to set the appropriate firewall rules and start the remote debugger.

Configuration is complete when the Remote Debugger window appears. The target device will be ready and listening for a connection.

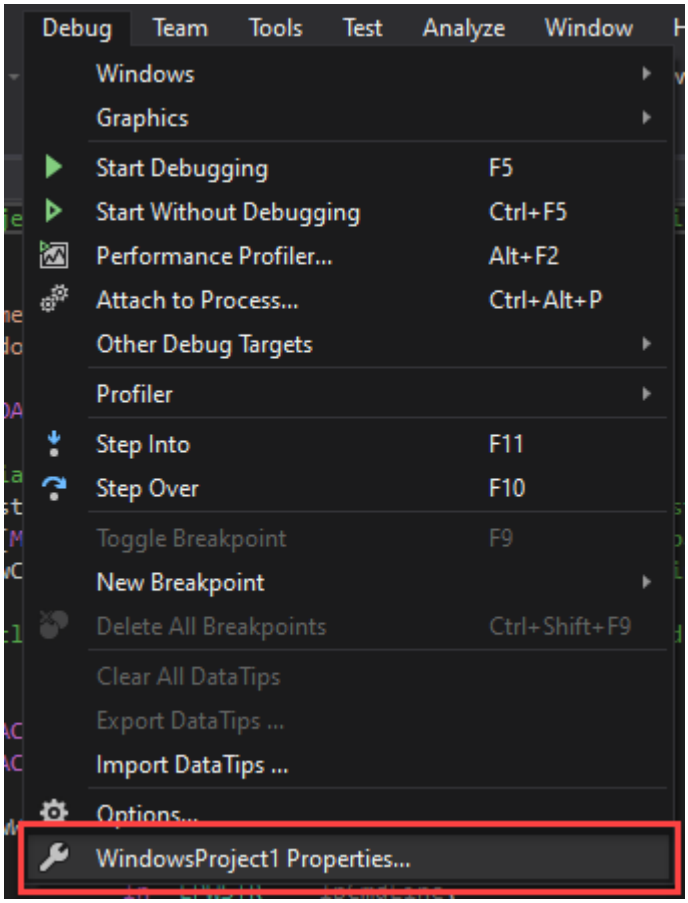
Note: The server name and port number (MJO-DL:4022) are needed to connect from Visual Studio on the development machine.



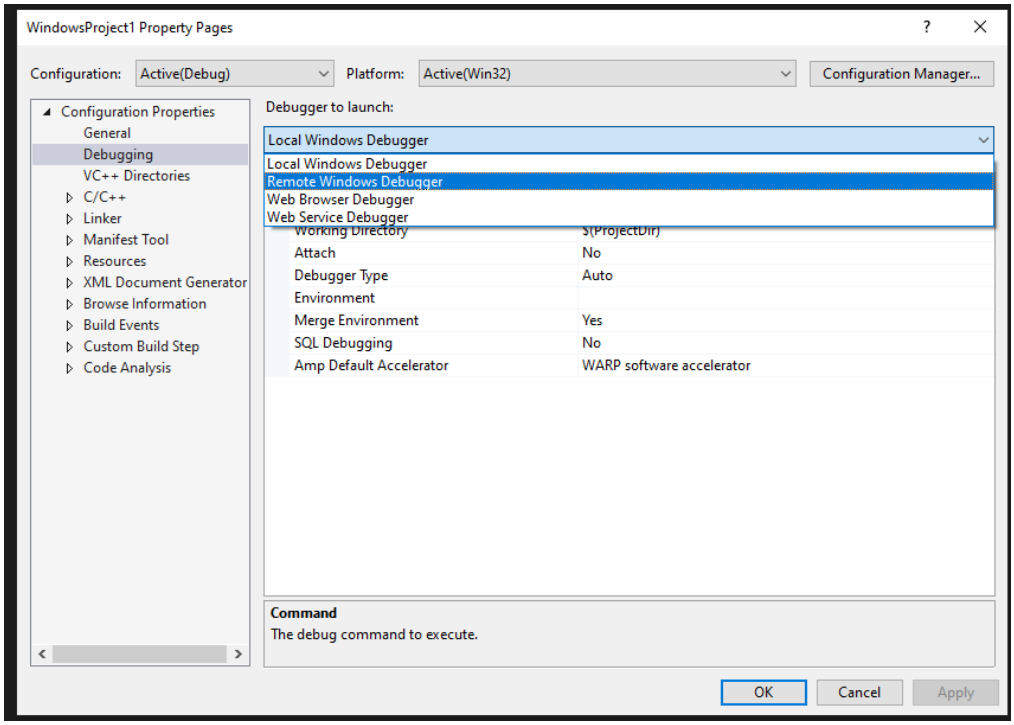
Connecting to Remote Debugger from Visual Studio

Once the Remote Debugger is running on the Arm64 device, users can connect to it directly from Visual Studio on the x86-64 Development machine.

1. Launch the Visual Studio project.
2. Click **Debug** and select **<Project Name> Properties**.



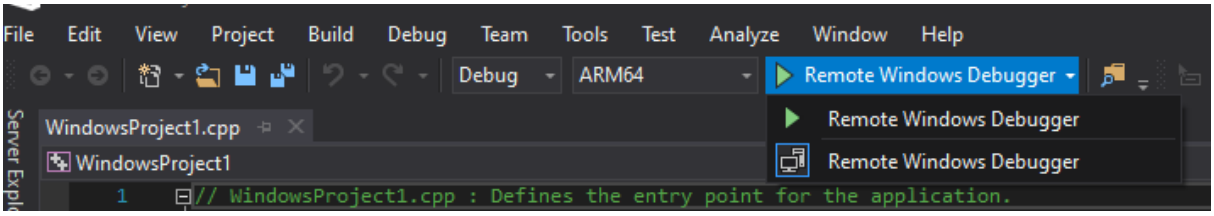
   a. From the **Configuration Properties** list, select **Debugging**.
   b. From the **Debugger to launch** list, select **Remote Windows Debugger**.



c. In the **Remote Server Name** field, enter the server name from the Remote Debugger on the Arm64 device.

d. Click **OK**.

3. Launch the application in debug directly to the remote device by changing the target at the top of the Visual Studio window.



Distributing the application

Applications compiled by Visual Studio depend on the libraries of the Visual Studio runtime. These libraries were installed on the Arm64 development target when the Windows 10 SDK and the remote debugging tools were installed.

When distributing an application, users must ensure that the installer, or the distribution platform being used, is installing the Visual Studio redistributables.

The installers are available to download from Microsoft:
https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads

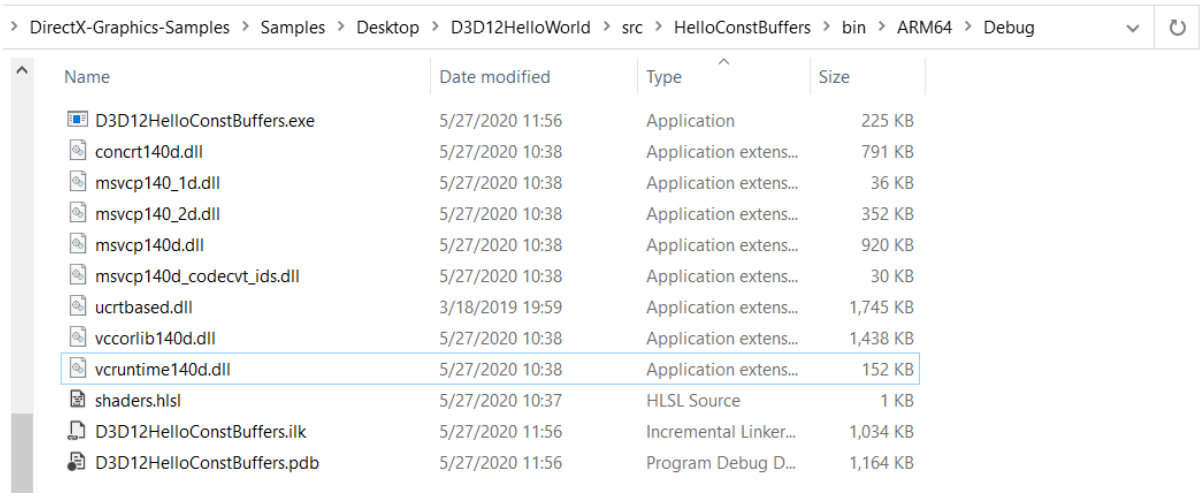
Non-redistributable debug runtime

To run, or remotely debug Arm64 executables compiled in the Debug configuration, the debug version of the runtime libraries must be installed on the Arm64 system. These libraries are not provided by the Visual Studio runtime redistributable (see Distributing the application).

The libraries are located in the **Debug_NonRedist** folder of the Visual Studio installation (`C:\Program Files (x86)\Microsoft Visual Studio`).

This process also requires the debug version of universal C runtime from the Windows SDK, **ucrtbased.dll** (`C:\Program Files (x86)\Windows Kits\10\bin\<version>\arm64\ucrt`).

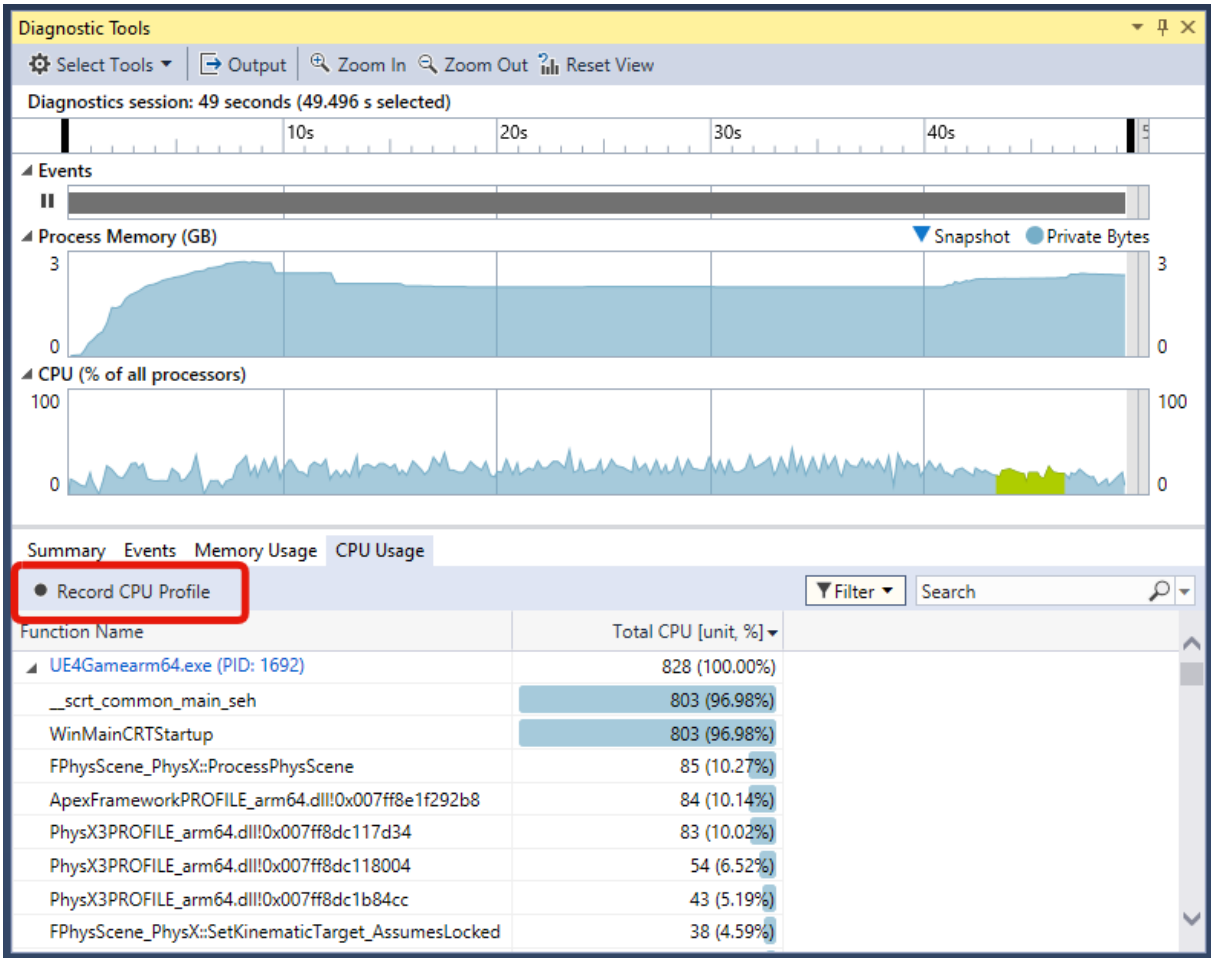
The image below shows a DirectX12 sample built for Arm64 in Debug configuration and its dependent debug libraries.



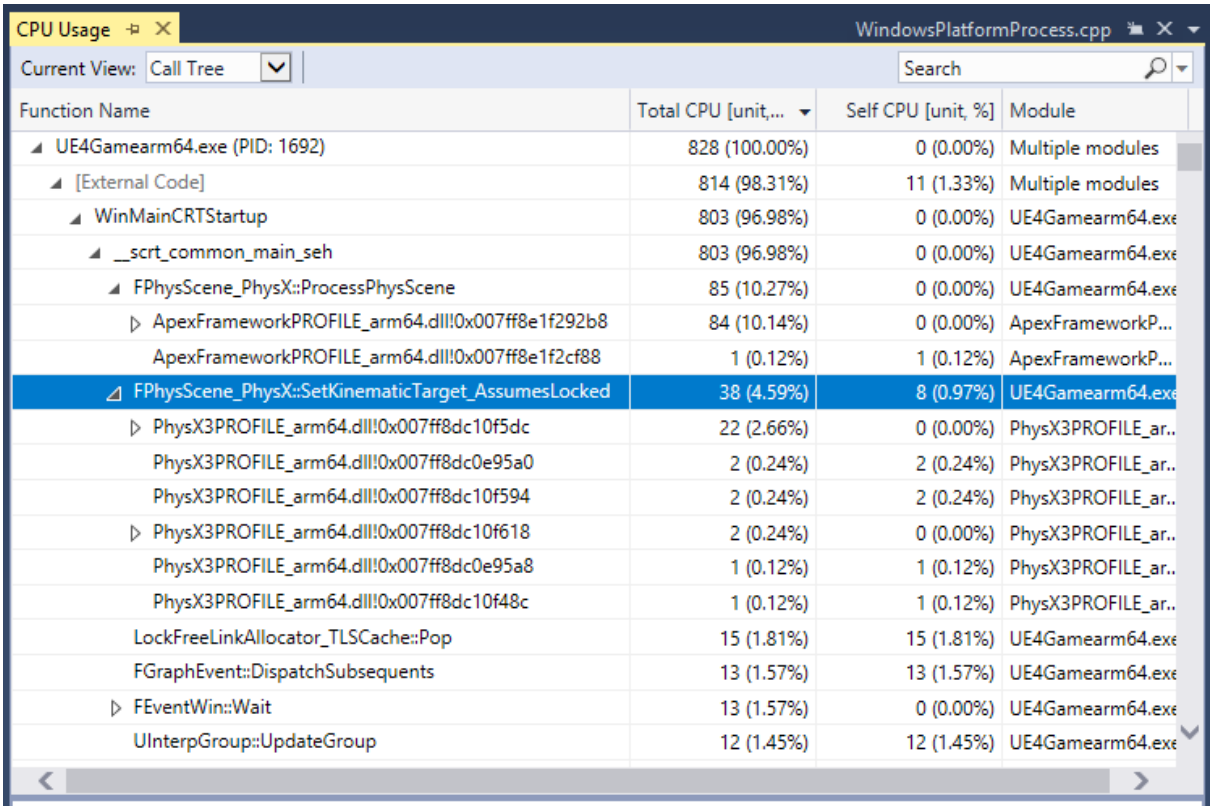
Visual Studio Performance Profiler

To use the Visual Studio Performance Profiler attached to a remote process on the Arm64 target:

1. Click Debug > Performance Profiler.
2. Click the CPU Usage tab and select Record CPU Profile.



3. In the Current View list, select Caller/Callee or Call Tree with timing data.



CPU and GPU profiling with PIX

A CPU/GPU profiler is crucial for game development because it helps identify any potential bottlenecks a game may encounter on-system.

PIX is Microsoft's CPU and GPU debugger/profiler for DirectX 12 games on Windows and supports remote profiling on Arm64 devices.

For tutorials and articles covering advanced PIX usage, refer to Microsoft's PIX on Windows.

DirectX 12

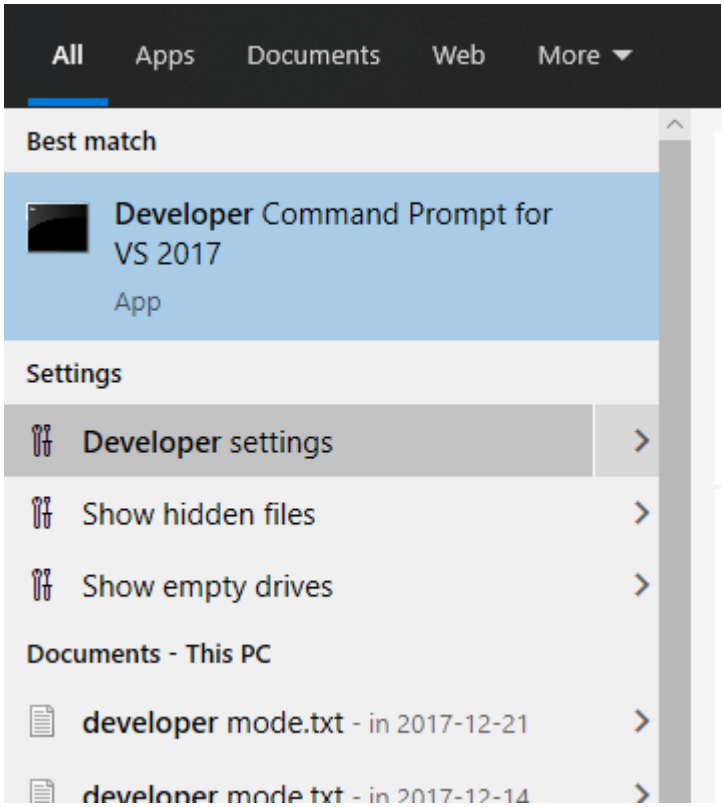
The Arm64 hardware platform and drivers are optimized for DirectX 12. To ensure the best performance and reliability in a game, use a DirectX 12 render path over DirectX 11 wherever possible.

Setup on Development machine

Install the latest version of PIX (https://devblogs.microsoft.com/pix/download/).

Setup on Arm64 target device

1. Enable Developer Mode on the device:
    a. Open the **Start** menu. Search for and select **Developer Settings**.



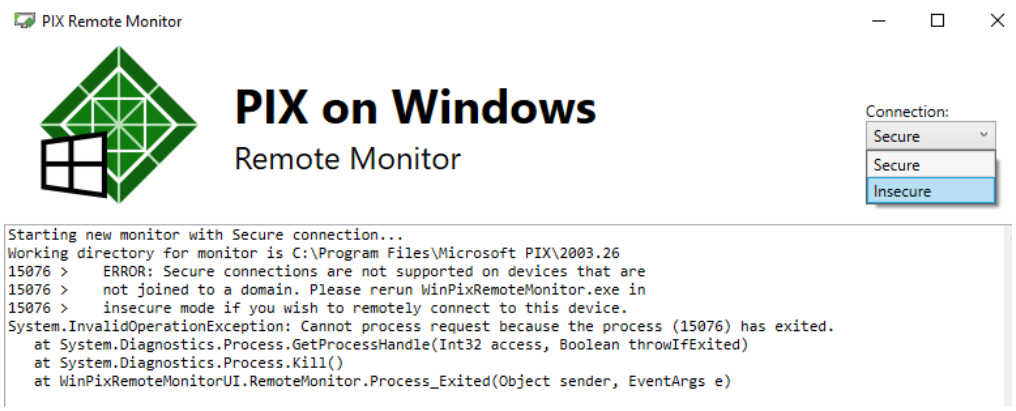
    b. Under the **Use developer features** group, click **Developer mode**.

Installation is complete when “Developer Mode package installed” appears.

2. Copy the entire contents of the latest installed version of PIX from the Development machine (i.e., `C:\Program Files\Microsoft PIX\<pix version>`) to the Arm64 Target Device.

    a. Right-click **WinPixRemoteMonitorUI.exe** and select **Run as Administrator**.

    b. [If you are not connected to a domain ONLY] From the **Connection** list, select **Insecure**.

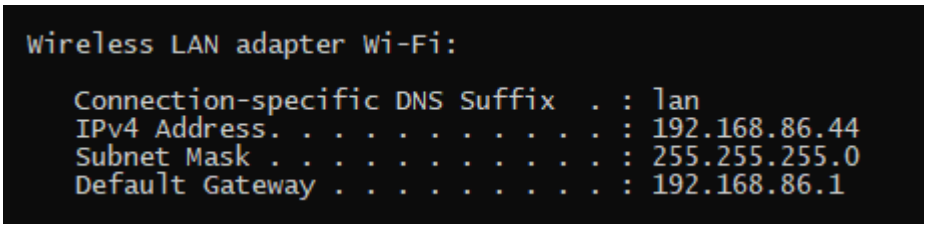


3. Find the IP Address of the Arm64 Target Device.

    a. Open a command prompt.

    b. Enter “ipconfig” and press **Enter**.

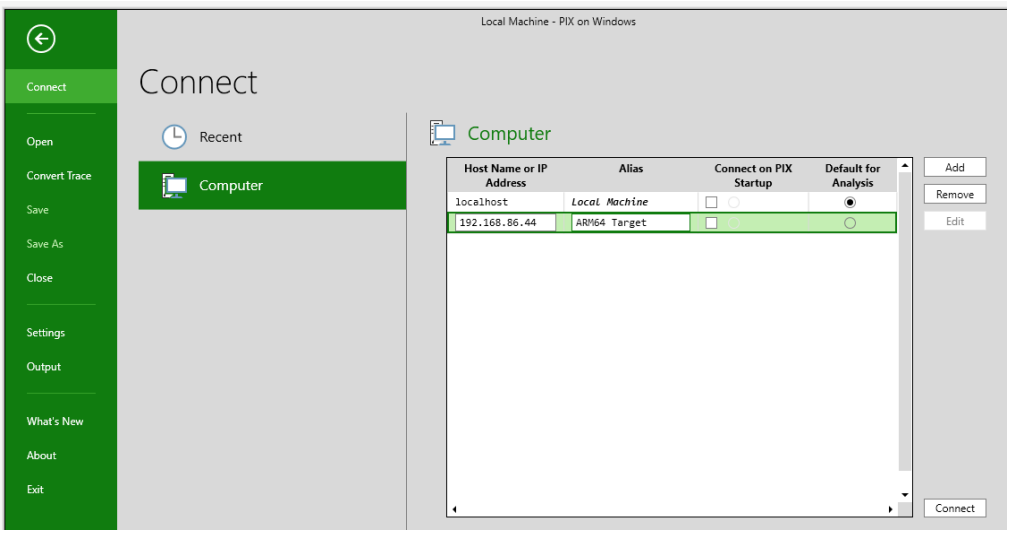
    c. Find the entry for the current active network connection (wired or wireless) and note the IPv4 address.



Connecting to a remote PIX debugger

After setup is complete on both machines:

1. Open PIX on the Development machine.
2. Click the Home tab, then **Connect > Computer > Add**.
    a. Enter the IP address noted when setting up PIX on the Arm64 device (see step 3c of section Setup on Arm64 target device) in the **Host Name or IP Address** field.
    b. Enter a name for the connection in the **Alias** field.
    c. Click **Connect**.



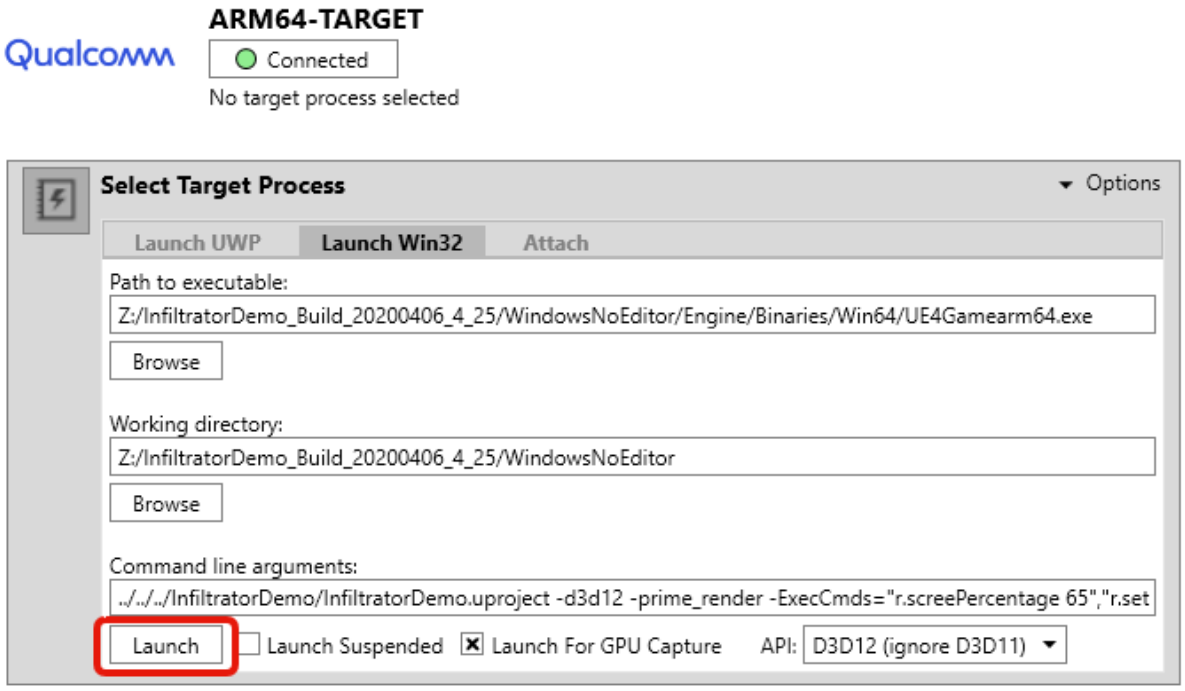
From PIX on the Development machine, users can launch an executable directly on the Arm64 Target Device, or attach to a current, running application to begin profiling and debugging a game.

A common path would be to launch a game for remote debugging from Visual Studio to the Arm64 Target Device. PIX then connects to the game and selects the debug game's current, running process.

For more information about PIX, refer to Microsoft's tool documentation: https://devblogs.microsoft.com/pix/documentation/

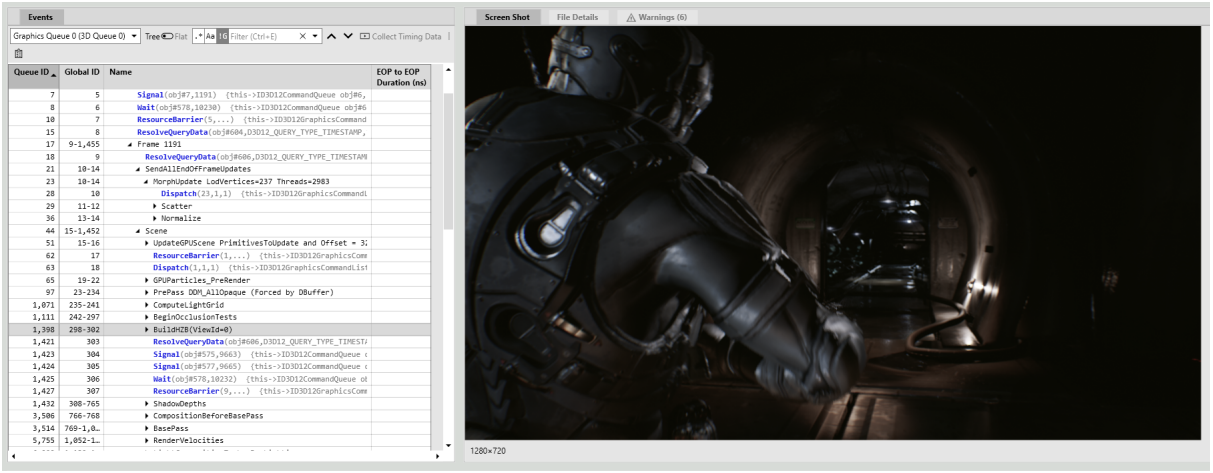
Examples

Launching for GPU capture on the Arm64 target

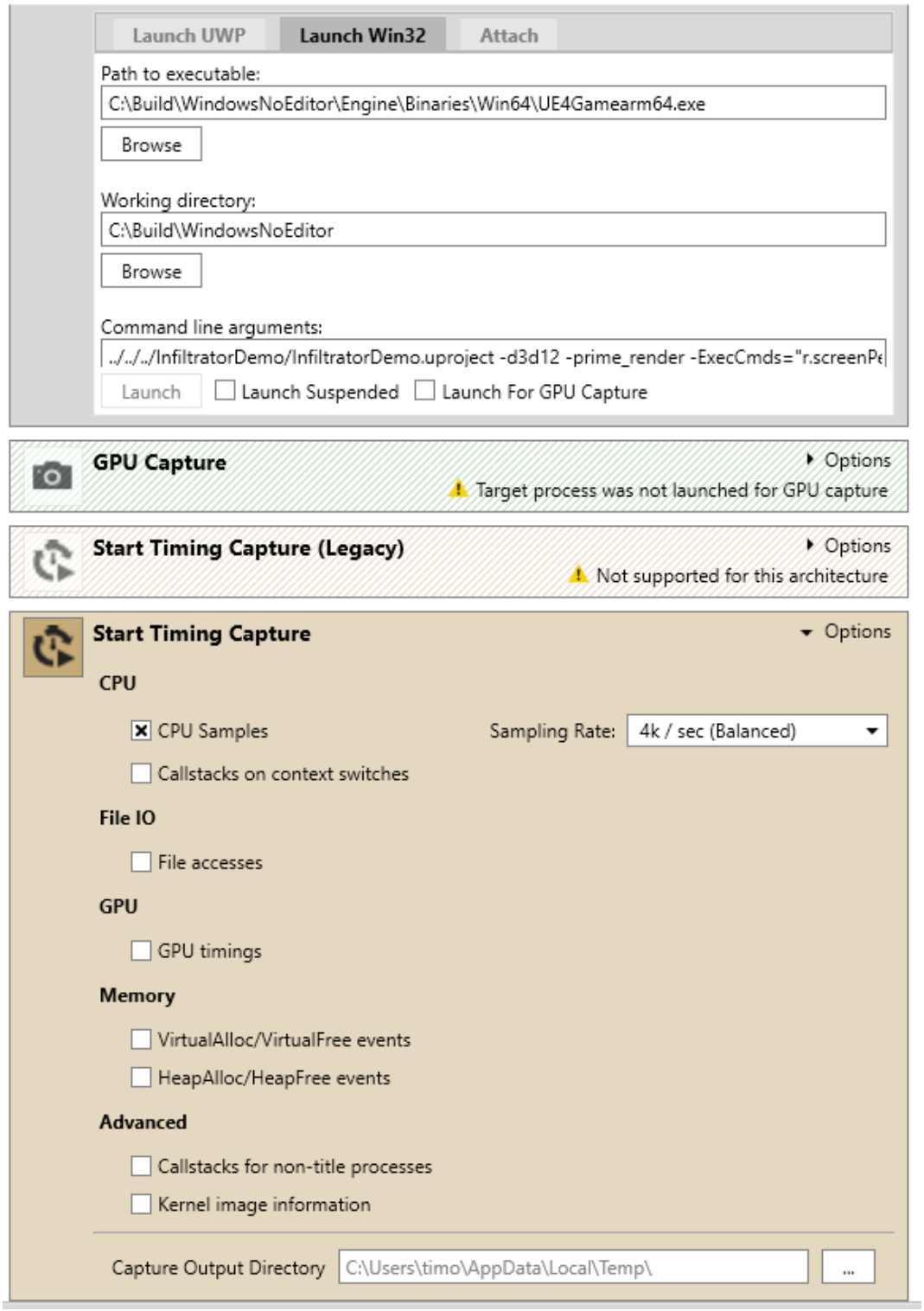


GPU capture (Unreal Engine 4.25 for Arm64)

In this example, the engine inserts PIX events in the GPU command stream with the help of the WinPixEventRuntime library. This helps trace frame render operations.



Launching for CPU capture



CPU capture

This example shows a CPU capture for a single render frame of UE4 4.25. PIX events are reported for both CPU and GPU captures. Cores utilization 4 through 7 are the faster Gold cores.

Troubleshooting PIX

- If the computer is not on a Domain, WinPixRemoteMonitorUI.exe provides a warning during startup that it cannot run on a secure connection. To proceed, switch the Connection mode to Insecure in the UI.
- If GPU analysis is not working, ensure that you are running WinPixRemoteMonitorUI.exe as an Administrator.

Platform details

Snapdragon 8cx

The Snapdragon 8cx SoC has the following components:

- Four Silver Kryo 495 Cortex-A76 Armv8.2-A cores (1.80 GHz)
- Four Gold Kryo 495 Cortex-A76 Armv8.2-A cores (2.84 GHz)
- Adreno 680 GPU (1842.5 GFLOPs)
- 8-channel low power DDR SDRAM at 2133 MHz (68.26 GB/s)

The Gold cores offer higher performance, while the Silver cores have better thermal and power characteristics. The variation in performance between these cores are because of frequency and architectural differences.

When porting high performance applications to 8cx, it is essential to configure thread affinity between the Gold and Silver cores to balance performance and power consumption.

For more details on this heterogeneous processing architecture, see Arm big.LITTLE.

Memory model

The Armv8 architecture employs a weakly-ordered memory model. To optimize performance, reads and writes to central memory may happen in a different order than stated by program instructions.

Other architectures, such as amd64 also use weakly-ordered memory. If you are using high-level threading APIs, these problems are handled for you. However, users who are porting their own implementation of low-level multithreaded code must use the appropriate memory barriers and other configuration for the hardware platform.

For more information, refer to the Memory Ordering chapter of the Programmer's Guide for Armv8-A.

Neon SIMD

If your software uses SIMD for parallel data processing, the SSE code must be ported over to the Neon instruction set. For more information, refer to the Neon Documentation.

Visual Studio supports Neon intrinsics. New Neon code can be written in parallel with existing SSE code and enabled at compile time for porting. It is also worth noting that the Arm Compute Library is available, and provides Neon-optimized implementation of commonly used algorithms.

Visual Studio also supports the auto-vectorization of source code. This is controlled by the /**arch** option.

Key differences between Adreno and Desktop GPUs

Adreno 680 in the Qualcomm® Snapdragon™ 8cx Compute Platform is a tile-based GPU optimized to deliver high performance with low power consumption.

Tiled-based rendering is a render pipeline architecture that splits the screen into several tiles and renders each tile in succession. Compared to the direct rendering model favored by older desktop GPU technology, this greatly reduces memory bandwidth, which reduces power consumption and improves performance.

While this technology was pioneered on mobile devices, modern and discrete desktop GPUs from major vendors all use tiled rendering as well.

If you are porting from an older rendering architecture, ensure that the rendering pipeline is optimized for a tiled renderer. Using a deferred renderer and reducing shader counts will yield the best performance improvements when moving from an immediate mode GPU to a tiled GPU.

The Adreno 680 also benefits from the unified memory architecture of the Snapdragon 8cx SoC. Compared to discrete desktop GPUs, there is no distinction between operating system memory and GPU memory, which allows the following:

- Better use of resources depending on CPU or GPU intensive tasks

- No transfers between central and GPU memory
- Less overall power consumption

For more details, refer to the various sections of the developer guides.

References

Qualcomm Technologies, Inc.

- Windows on Snapdragon https://www.qualcomm.com/developer/windows-on-snapdragon
- Snapdragon Developer Tools
  https://www.qualcomm.com/developer/windows-on-snapdragon#tools

Windows

- Windows 10 on Arm documentation https://docs.microsoft.com/en-us/windows/arm/

Platform details

Programmer's Guide for Arm A-Profile
https://www.arm.com/en/architecture/learn-the-architecture/a-profile

- Memory Systems, Ordering, and Barriers
  https://developer.arm.com/documentation/102336/latest
- Neon SIMD https://developer.arm.com/documentation/102474/latest
- Windows on Arm – An assembly language primer
  https://www.codemachine.com/article_armasm.html

Unreal Engine 5 for Windows on Snapdragon

First, complete Epic's on-boarding steps to be able to access Unreal Engine source on GitHub.

Then, view Snapdragon Studio's forked copy of Unreal Engine here:
https://github.com/SnapdragonStudios-UnrealEngine/UnrealEngine Specifically, view the **windowsARM64_5.4** branch found here:
https://github.com/SnapdragonStudios-UnrealEngine/UnrealEngine/tree/windowsARM64_5.4

The `README.md` on the **windowsARM64_5.4** details additional pre-requesites and setup that must be done to package applications for `Arm64` and `Arm64EC`, which includes installing some additional Visual Studio Installer components and running a setup batch file.

Unreal Engine 4 for Windows on Snapdragon

Attention: These steps are for those using the previous version of Unreal Engine, UE4. If using Unreal Engine 5, follow the steps in Unreal Engine 5 for Windows on Snapdragon instead.

Before you begin

Note: All development work and work in the Unreal Editor must take place on a Windows x64 system.

The following instructions help users produce a project build that can run on a Windows 10 Arm device:

1. Follow the steps provided by Epic Games to gain access to the Unreal Engine source code on GitHub.
2. Follow instructions from the pull request for Unreal Engine 4.25 to create a custom build of the engine to add Arm for Windows 10 support.
   https://github.com/EpicGames/UnrealEngine/pull/6975

Technical Prerequisites

Development machine

Install:

- Git
- Visual Studio 2019
  - Workloads
    - Desktop Development with C++
    - Universal Windows Platform Development Workload
  - Individual components
    - .NET Framework 4.6.x
    - Visual C++ compilers and libraries for Arm and Arm64
    - Visual C++ Runtime for UWP
    - Windows 10 SDK 10.0.18362.0

Target Arm64 Windows 10 machine

Install the latest Visual Studio C++ Redistributable for ARM64.

Building Unreal Engine from source to enable Windows Arm cross-compile

To produce a `UE4Editor-Debug.exe` in `Engine\Binaries\Win64\` you must:

1. Clone the latest version of Unreal Engine 4.25 source from the 4.25 branch.

   https://github.com/EpicGames/UnrealEngine/tree/4.25

2. Incorporate changes from the Arm Support Pull Request.

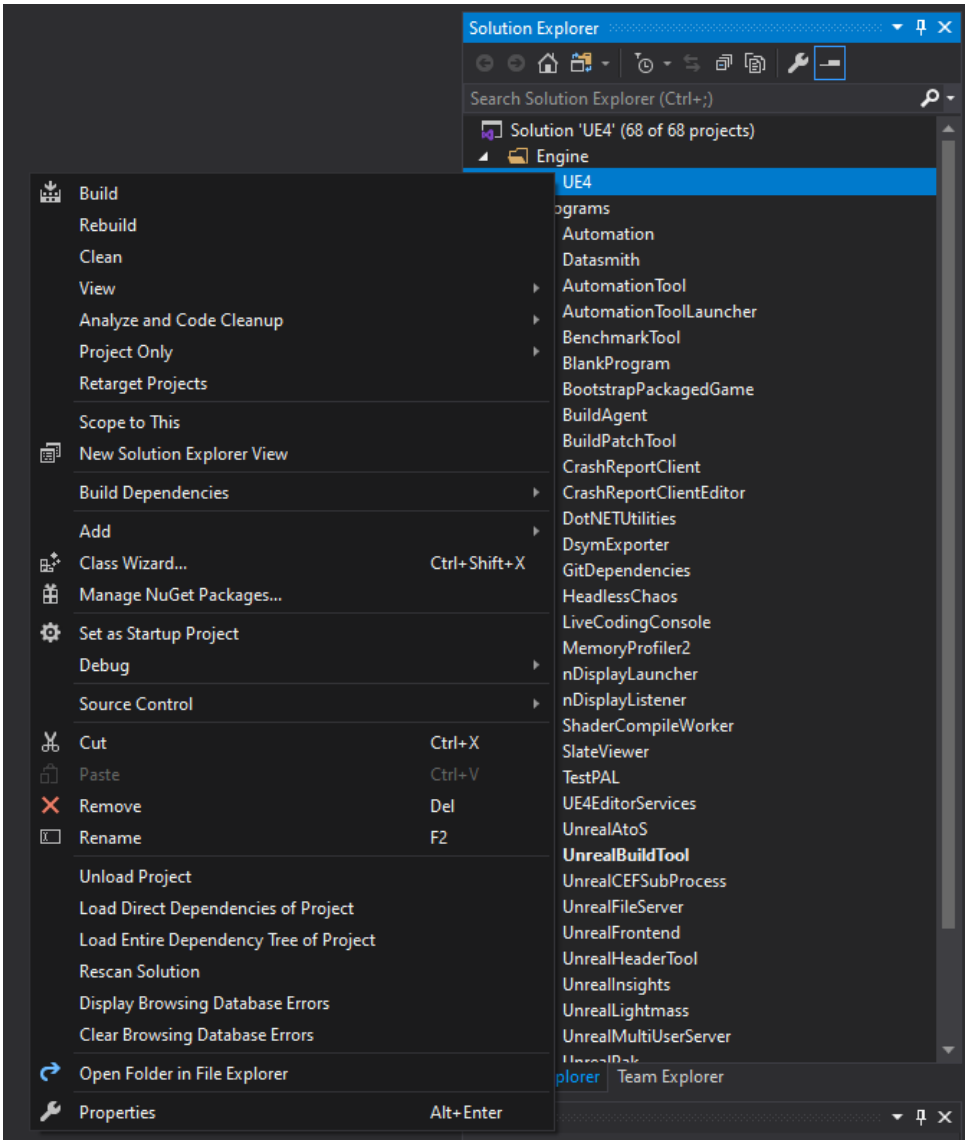
   https://github.com/EpicGames/UnrealEngine/pull/6975

   a. Run **Setup.bat**.

   b. Run **GenerateProjectFiles.bat**.

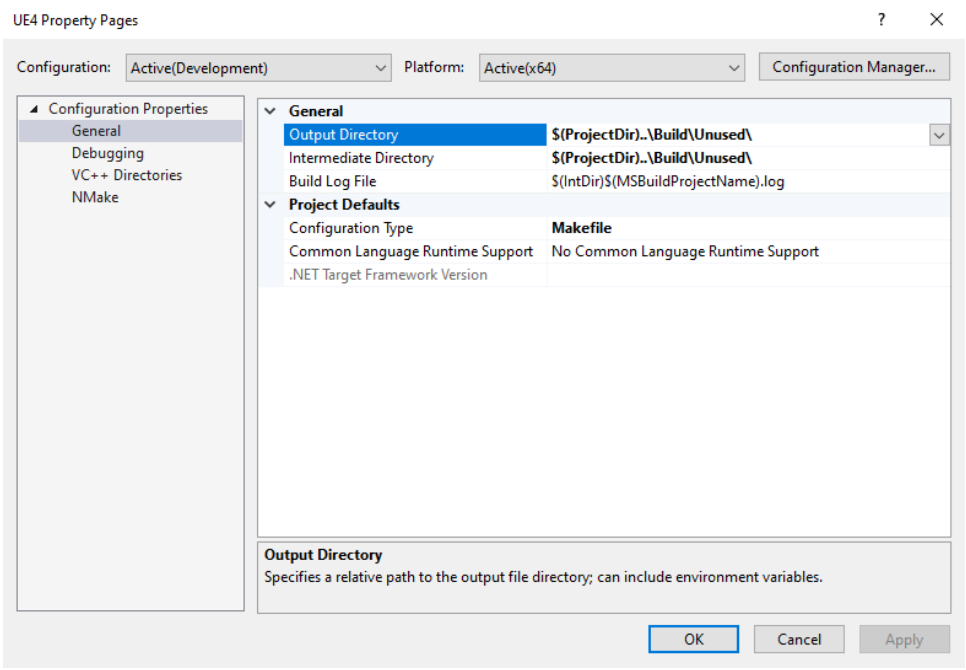
3. Open **UE4.sln** in Visual Studio 2019.

   a. In Solution Explorer, right-click **UE4** and select **Set as Startup Project**.



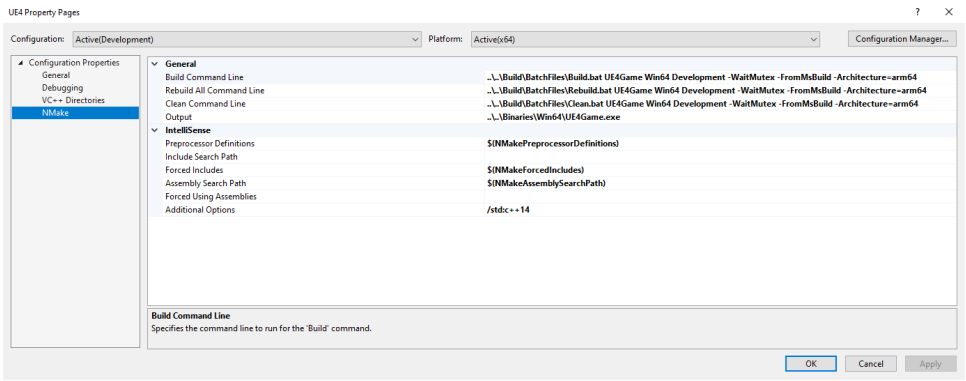
b. Right-click **UE4** > **Properties**.

c. From the **Configuration** list, select **Active (Development)**.



d. Under Configuration Properties, click **NMake** and append `` -Architecture=arm64`` to the following entries:

- Build Command Line
- Rebuild All Command Line
- Clean Command Line



e. Click **Apply**.

4. Right-click **UE4** in the Solution Explorer and select **Build**.

Producing builds

1. Open UE4 Game Project in the editor that was built in the previous section.

   If the project was created in a different editor version, we recommend that you open a copy of the project.

2. To create a packaged build, go to **File > Package Project > Windows (Snapdragon)**

Additional notes

Remote debug

It is possible to remote debug with Visual Studio, as outlined in Microsoft Remote Debugging. The remote debugging tools for Visual Studio 2019 and the Arm64 platform must be installed on the engineering sample and started up.

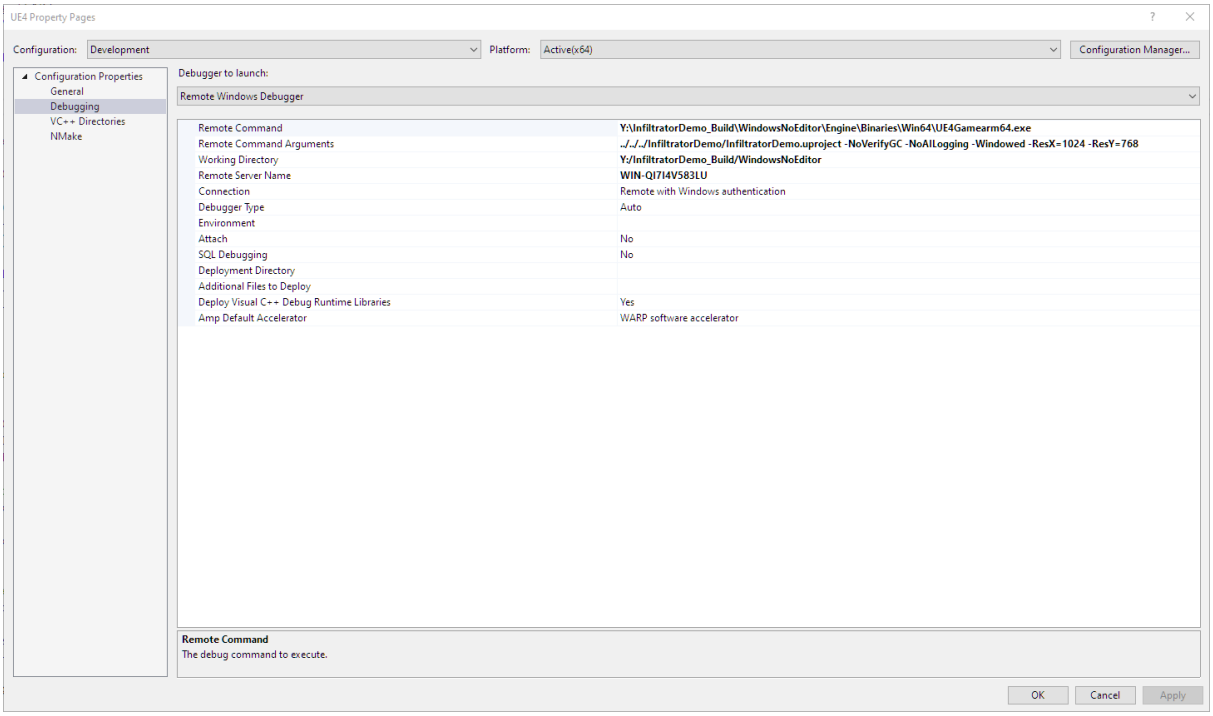


Figure20 An example remote debug configuration

Build optimization

This build is optimized for the Microsoft SQ1 Arm processor, and available in the Surface Pro X tablet in particular. The thread affinity setup is optimized for the DX12 RHI on the SQ1 and takes advantage of the faster “big cores”. For more details, review the additions to Engine/Source/Runtime/Core/Public/Windows/WindowsPlatformAffinity.h.

Third-party libraries

Several third party libraries have been recompiled for Windows 10 Arm64 to support this work. This was often done outside of the Unreal Engine 4 source tree, using each library’s build system. Refer to the README-arm64.txt files for notes on how the libraries were configured and compiled and details on the source version used:

- `Engine/Source/ThirdParty/PhysX3/README-arm64.txt`
- `Engine/Source/ThirdParty/Vorbis/README-arm64.txt`

References

- Unreal Engine source code: https://www.unrealengine.com/en-US/ue-on-github
- Visual Studio Redistributable for Arm64: https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist?view=msvc-170
- Microsoft Remote Debugging: https://learn.microsoft.com/en-us/visualstudio/debugger/remote-debugging?view=vs-2019

LEGAL INFORMATION

Your access to and use of this material, along with any documents, software, specifications, reference board files, drawings, diagnostics and other information contained herein (collectively this "Material"), is subject to your (including the corporation or other legal entity you represent, collectively "You" or "Your") acceptance of the terms and conditions ("Terms of Use") set forth below. If You do not agree to these Terms of Use, you may not use this Material and shall immediately destroy any copy thereof.

1) Legal Notice.

This Material is being made available to You solely for Your internal use with those products and service offerings of Qualcomm Technologies, Inc. ("**Qualcomm Technologies**"), its affiliates and/or licensors described in this Material, and shall not be used for any other purposes. If this Material is marked as "**Qualcomm Internal Use Only**", no license is granted to You herein, and You must immediately (a) destroy or return this Material to Qualcomm Technologies, and (b) report Your receipt of this Material to qualcomm.support@qti.qualcomm.com. This Material may not be altered, edited, or modified in any way without Qualcomm Technologies' prior written approval, nor may it be used for any machine learning or artificial intelligence development purpose which results, whether directly or indirectly, in the creation or development of an automated device, program, tool, algorithm, process, methodology, product and/or other output. Unauthorized use or disclosure of this Material or the information contained herein is strictly prohibited, and You agree to indemnify Qualcomm Technologies, its affiliates and licensors for any damages or losses suffered by Qualcomm Technologies, its affiliates and/or licensors for any such unauthorized uses or disclosures of this Material, in whole or part.

Qualcomm Technologies, its affiliates and/or licensors retain all rights and ownership in and to this Material. No license to any trademark, patent, copyright, mask work protection right or any other intellectual property right is either granted or implied by this Material or any information disclosed herein, including, but not limited to, any license to make, use, import or sell any product, service or technology offering embodying any of the information in this Material.

THIS MATERIAL IS BEING PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY LAW, QUALCOMM TECHNOLOGIES, ITS AFFILIATES AND/OR LICENSORS SPECIFICALLY DISCLAIM ALL WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, COMPLETENESS OR ACCURACY, AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MOREOVER, NEITHER QUALCOMM TECHNOLOGIES, NOR ANY OF ITS AFFILIATES AND/OR LICENSORS, SHALL BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY EXPENSES, LOSSES, USE, OR ACTIONS HOWSOEVER INCURRED OR UNDERTAKEN BY YOU IN RELIANCE ON THIS MATERIAL.

Certain product kits, tools and other items referenced in this Material may require You to accept additional terms and conditions before accessing or using those items.

Technical data specified in this Material may be subject to U.S. and other applicable export control laws. Transmission contrary to U.S. and any other applicable law is strictly prohibited.

Nothing in this Material is an offer to sell any of the components or devices referenced herein.

This Material is subject to change without further notification.

In the event of a conflict between these Terms of Use and the *Website Terms of Use* on www.qualcomm.com, the *Qualcomm Privacy Policy* referenced on www.qualcomm.com, or other legal statements or notices found on prior pages of the Material, these Terms of Use will control. In the event of a conflict between these Terms of Use and any other agreement (written or click-through, including, without limitation any non-disclosure agreement) executed by You and Qualcomm Technologies or a Qualcomm Technologies affiliate and/or licensor with respect to Your access to and use of this Material, the other agreement will control.

These Terms of Use shall be governed by and construed and enforced in accordance with the laws of the State of California, excluding the U.N. Convention on International Sale of Goods, without regard to conflict of laws principles. Any dispute, claim or controversy arising out of or relating to these Terms of Use, or the breach or validity hereof, shall be adjudicated only by a court of competent jurisdiction in the county of San Diego, State of California, and You hereby consent to the personal jurisdiction of such courts for that purpose.

2) Trademark and Product Attribution Statements.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. The Bluetooth® word mark is a registered trademark owned by Bluetooth SIG, Inc. Other product and brand names referenced in this Material may be trademarks or registered trademarks of their respective owners.

Snapdragon and Qualcomm branded products referenced in this Material are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.