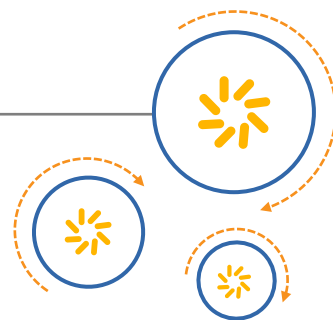




Qualcomm Technologies, Inc.



Qualcomm[®] Adreno[™] Vulkan

Developer Guide

80-NB295-7 A

August 8, 2017

Qualcomm Adreno is a product of Qualcomm Technologies, Inc. Other Qualcomm products referenced herein are products of Qualcomm Technologies, Inc. or its subsidiaries.

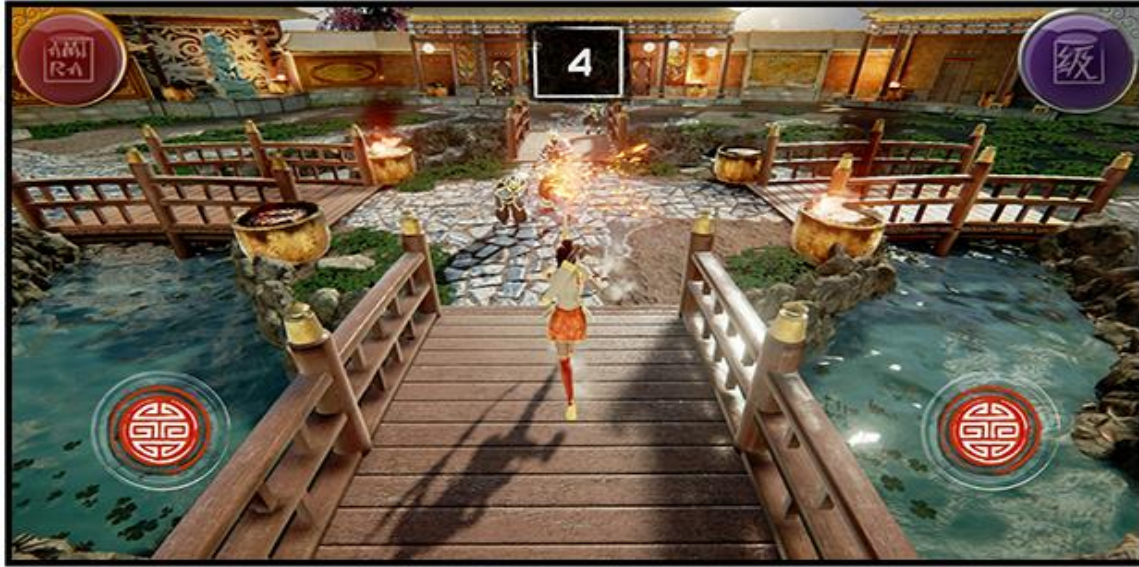
Qualcomm Adreno, Snapdragon, FlexRender, and Reign of Amira are trademarks of Qualcomm Incorporated, registered in the United States and other countries.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	August 2017	Initial release



This image is taken from the Reign of Amira™ demo developed by the Advanced Content Group at Qualcomm Technologies, Inc. The demo is running on a Qualcomm Snapdragon 835 device using the Vulkan 1.1 API and showcases an interactive twin-shooter arena game with Physically Based Rendering, ASTC Texture Compression, High Dynamic Range rendering, linear space lighting, dynamic shadows, light refraction, and other high end rendering techniques possible with Vulkan.

Contents

1 Introduction.....	8
1.1 Purpose.....	8
1.2 Conventions	8
1.3 Technical assistance.....	8
2 Overview.....	9
2.1 Vulkan.....	9
2.2 Learning Vulkan	10
2.3 Adreno GPU	10
2.3.1 Texture features	11
2.3.2 Visibility processing	14
2.3.3 Shader support	17
2.3.4 Low power performance.....	18
2.3.5 Universal bandwidth compression (UBWC)	18
2.3.6 Draw order independent depth rejection.....	18
2.3.7 Compute.....	18
2.3.8 Other supported features.....	19
2.3.9 Adreno APIs	20
2.4 Vulkan.....	20
2.4.1 Vulkan versions	20
2.5 About Android	21
2.5.1 Vulkan support on Android	21
2.5.2 Android and Vulkan on Adreno	21
3 Vulkan 1.0 with Adreno	22
3.1 Development environment.....	22
3.1.1 Development system.....	22
3.1.2 Target system.....	22
3.2 Setup instructions.....	23
3.2.1 Android development on Windows	23
3.3 Walkthrough of sample applications	24
4 Using Vulkan with Adreno	25
4.1 Application information.....	25
4.2 Physical device	25
4.3 Validation layers	26
4.4 Extensions	27
4.5 Queues	28
4.6 Logical device.....	29

4.7 Memory	29
4.8 Window system integration (WSI)	30
4.8.1 Surface abstraction.....	30
4.8.2 Swap chain.....	31
4.8.3 Hardware scaling	33
4.9 Command buffers	33
4.10 Pipelines.....	33
4.10.1 Pipeline caches.....	34
4.10.2 Building pipelines in parallel.....	34
4.10.3 Derivative pipelines	34
4.11 Synchronization	35
4.12 Renderpass	36
4.13 Framebuffers.....	37
4.14 Images	37
4.15 Descriptors/push constants.....	38
4.15.1 Descriptor set frequency	38
4.15.2 Descriptor pools.....	38
4.16 Shaders.....	38
4.17 Multi-buffer rendering	41
5 Understanding the developer tools	42
5.1 Snapdragon profiler	42
5.2 Adreno SDK for Vulkan	45
6 Optimizing applications	47
6.1 Shader tips.....	47
6.1.1 Build pipelines during initialization	47
6.1.2 Use built-ins.....	47
6.1.3 Use the appropriate data type	47
6.1.4 Reduce type casting	48
6.1.5 Pack scalar constants	48
6.1.6 Keep shader length reasonable	48
6.1.7 Sample textures in an efficient way	49
6.1.8 Threads in flight/dynamic branching	49
6.1.9 Pack shader interpolators.....	50
6.1.10 Minimize usage of shader GPRs.....	50
6.1.11 Minimize shader instruction count	50
6.1.12 Avoid uber-shaders.....	50
6.1.13 Avoid math on shader constants	51
6.1.14 Avoid discarding pixels in the fragment shader	51
6.1.15 Avoid modifying depth in fragment shaders	51
6.1.16 Avoid texture fetches in vertex shaders.....	51
6.1.17 Break up draw calls	51
6.1.18 Use medium precision where possible.....	52
6.1.19 Favor vertex shader calculations over fragment shader calculations.....	52
6.1.20 Measure, test, and verify results	52
6.1.21 Prefer uniform buffers over shader storage buffers	52
6.2 Optimize vertex processing.....	53
6.2.1 Use interleaved, compressed vertices	53

6.2.2 Consider geometry instancing	53
6.3 Texture compression strategies.....	54
6.3.1 Diffuse texture tests	55
6.3.2 Normal texture tests	59
6.4 Bandwidth optimization.....	63
6.5 Depth range optimization.....	64
6.6 Other optimizations.....	65
A Device limits (Adreno 530)	66
B Sparse properties (Adreno 530).....	69
C Features (Adreno 530)	70
D References.....	72
E Glossary.....	74

Figures

Figure 2-1 Video texture example	11
Figure 2-2 Cube mapping	13
Figure 2-3 3D texture.....	13
Figure 2-4 Early Z rejection.....	14
Figure 2-5 Tiled/binning rendering architecture with Adreno 3xx, 4x, and 5x.....	16
Figure 2-6 Unified shader architecture	17
Figure 2-7 Flexibility in shader resources – Unified shader architecture	17
Figure 2-8 MSAA	19
Figure 5-1 Snapdragon profiler – realtime performance.....	42
Figure 5-2 Snapdragon profiler – trace	43
Figure 5-3 Snapdragon profiler – snapshot.....	44
Figure 5-4 Snapdragon profiler – Vulkan trace	44
Figure 5-5 Snapdragon profiler – Adreno SDK for Vulkan	45
Figure 6-1 Geometry instancing for drawing barrels.....	53
Figure 6-2 Diffuse texture used for the test	55
Figure 6-3 ATC compression result for GL_ATC_RGB_AMD	56
Figure 6-4 Noncompressed vs. ATC-compressed versions for GL_ATC_RGB_AMD.....	56
Figure 6-5 ETC1 compression result for GL_ETC1_RGB8_OES	57
Figure 6-6 Noncompressed vs. ETC1-compressed versions for GL_ETC1_RGB8_OES	57
Figure 6-7 ETC2 compression result for GL_COMPRESSED_RGB8_ETC2	58
Figure 6-8 Noncompressed vs. ETC2-compressed versions for GL_COMPRESSED_RGB8_ETC2.....	58
Figure 6-9 Normal texture used for the test	59
Figure 6-10 ATC compression result for GL_ATC_RGB_AMD	60
Figure 6-11 Noncompressed vs. ATC-compressed versions for GL_ATC_RGB_AMD.....	60
Figure 6-12 ETC1 compression result for GL_ETC1_RGB8_OES	61
Figure 6-13 Noncompressed vs. ETC1-compressed versions for GL_ETC1_RGB8_OES	61
Figure 6-14 ETC2 compression result for GL_COMPRESSED_RGB8_ETC2	62
Figure 6-15 Noncompressed vs. ETC2-compressed versions for GL_COMPRESSED_RGB8_ETC2....	62

Tables

Table 2-1 Adreno, Vulkan, and Android versions	21
Table 4-1 Validation layers library	26
Table 4-2 Queue capabilities.....	29
Table 4-3 Memory types supported on Adreno 5xx	30
Table 4-4 Memory heaps supported on Adreno 5xx.....	30
Table 4-5 Supported surface formats on Adreno 5xx	31
Table 6-1 Vertex data format support in Adreno architecture	64

1 Introduction

1.1 Purpose

This document is a guide for developing and optimizing Vulkan applications for Android on platforms containing Qualcomm® Adreno™ GPUs. Vulkan is a new, modern graphics API designed for current GPU hardware.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, `copy a:*. * b:.`

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at <https://createpoint.qti.qualcomm.com/>.

If you do not have access to the CDMA Tech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Overview

Vulkan is an explicit, streamlined, portable, and extensible API designed for use on both embedded systems which are constrained by processor capability, memory limitation, and power consumption limitations, as well as more complex systems with fewer constraints.

Khronos is a graphics standards organization that Qualcomm has a promoter member status with. Qualcomm has and continues to participate in the design and specification of the Vulkan API.

The document introduces the basics of Vulkan development with Adreno, detailing how to set up a development environment, and provides walkthroughs of sample applications. It also describes how to use debugging and profiling tools.

Detailed descriptions of the Adreno-specific implementations of Vulkan APIs are provided, and the developer tools provided by the Adreno SDK.

Good design practice with Vulkan is discussed, followed by advice on how to optimize applications specifically for the Adreno hardware. Rather than focus on how to program in Vulkan, this document focuses on optimizing Vulkan code for Qualcomm's Adreno family of GPUs. It is not intended as a primer for 3D graphics, nor as an introductory guide to Vulkan programming.

This document is intended for application developers, with a good working knowledge of a modern 3D graphics API such as OpenGL ES, OpenGL, Vulkan, or Microsoft Direct3D.

2.1 Vulkan

Vulkan is a new graphics API that allows applications to perform well on Adreno GPUs. Although there are several reasons for this, one key reason is that Vulkan's new multipass rendering support greatly benefits Adreno's tile-base architecture by eliminating bandwidth transfers to main memory. This support allows for the optimization of multiple passes over render targets with the same dimension.

Vulkan gives the developer the opportunity to get high performance from applications. Vulkan provides explicit low-level GPU control, and allows for a simpler driver design with less CPU overhead and more predictable processing.

Vulkan is designed from the ground up to be multithreading friendly. This greatly benefits applications designed for threading running on Snapdragon SOCs which have multiple cores. Threads can create command buffers and pipeline state objects in parallel.

Vulkan is cross-platform and supports both graphics and compute operations in the API. It's a single API for PC, console, mobile, and embedded hardware.

Vulkan supports a loadable layered architecture which allows for detailed error checking during development which is easily disabled for release builds. No validation or error-handling overhead needs to occur in production code or commercial releases.

Lastly, with Vulkan's cross vendor immediate language, SPIR_V, precompiled shader support there is no need for compilation to be done in the driver. Also developers can now hide proprietary shader code logic in commercial releases. Facilitates for runtime translation from GLSL to SPIR_V allow for existing shaders to work in Vulkan with minimal changes.

2.2 Learning Vulkan

Vulkan is a new graphics API with its first release in February 2016. Unlike its predecessor, OpenGL which was released back in 1992, there are limited albeit growing number of resources available to help learn the Vulkan API.

As a creator of the Vulkan standard (and many other standards), Khronos provides a number of resources and links on its [website](#). You'll find the up-to-date specification, reference pages, overview presentations, FAQs, and more. The site provides links to many excellent deep-dive Vulkan presentations recorded from recent tradeshow and events which give insights and tips into developing good Vulkan applications.

LunarG SDK

There is currently one published book on Vulkan, "Vulkan Programming Guide" by Graham Sellers which provides detailed coding examples and best practices on Vulkan.

There are a number of sample libraries available (beyond our Adreno SDK for Vulkan) that demonstrate Vulkan implements of common gaming and graphics rendering algorithms. Two excellent demo collections are [Sascha Willems's](#) and [Norbert Nopper's](#), both freely available on github.

2.3 Adreno GPU

The Adreno GPU is built in as part of the all-in-one design of the Qualcomm® Snapdragon™ processors. Accelerating the rendering of complex geometries allows the processors to meet the level of performance required by the games, user interfaces, and web technologies present in mobile devices today.

The Adreno GPU is built purposely for mobile APIs and mobile device constraints, with an emphasis on performance and efficient power use.

The original Adreno 130 variant provides support only for OpenGL ES 1.1. The Adreno 2xx series and onward supports OpenGL ES 2.0. The Adreno 3xx series adds support for OpenGL ES 3.0 and OpenCL. Adreno 4x adds support for OpenGL ES 3.1 and the AEP (Android Extension Pack). Adreno 5x continues with support for both OpenGL ES 1.1, the AEP, and Vulkan.

This section outlines the various technologies and subsystems provided by the Adreno GPU to support the graphics developer. Best practice for using these is discussed in later chapters.

2.3.1 Texture features

Multiple textures

Multiple texturing or *multitexturing* is the use of more than one texture at a time on a polygon. Adreno 4x supports up to 32 total textures in a single render pass, meaning up to 16 textures in the fragment shader and up to 16 textures at a time for the vertex shader. Effective use of multiple textures reduces overdraw significantly, saves Algorithmic Logic Unit (ALU) cost for fragment shaders, and avoids unnecessary vertex transforms.

To use multiple textures in applications, refer to the multitexture sample in the Adreno SDK OpenGL ES tutorials.

Video textures

More games and graphics applications today require video textures, which consist of moving images that are streamed in real time from a video file. Adreno GPUs support video textures.

Video textures are a standard API feature in Android today (Honeycomb or later versions). See the Android documentation for further details on surface textures at <http://developer.android.com/reference/android/graphics/SurfaceTexture.html>.

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



Figure 2-1 Video texture example

Texture compression

Texture compression can significantly improve the performance and load time of graphics applications since it reduces texture memory and bus bandwidth use. Compressed textures can be created using the Adreno Texture Compression and Visualization Tool and subsequently used by an OpenGL ES application.

Important compression texture formats supported by Adreno 3xx are:

- ATC – Proprietary Adreno texture compression format (for RGB and RGBA)
- ETC – Standard OpenGL ES 2.0 texture compression format (for RGB)
- ETC2 – Texture compression format that is supported in the OpenGL ES 3.0 API (for R, RG, RGB, and RGBA) and Vulkan API
- ASTC - Texture compression format that is supported in both OpenGL ES 3.0 API and Vulkan – allows the compression to be a variable block size.

Adreno 4x added support for ASTC LDR compression, which is made available through the Android Extension Pack.

To learn more about the use of texture compression, see the *Compressed Texture* tutorial in the Adreno SDK for OpenGL ES.

Floating point textures

Adreno 2xx, 3xx, and onward support the same texturing features, including:

- 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

Through the OpenGL ES 3.0 API, Adreno 3xx and onward also includes rendering support for FP16 (full support) and FP32 (no blending).

Cube mapping with seamless edges

Cube mapping is a fast and inexpensive way of creating advanced graphic effects, like 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 3xx and onward supports seamless-edge support for cube map texture sampling.

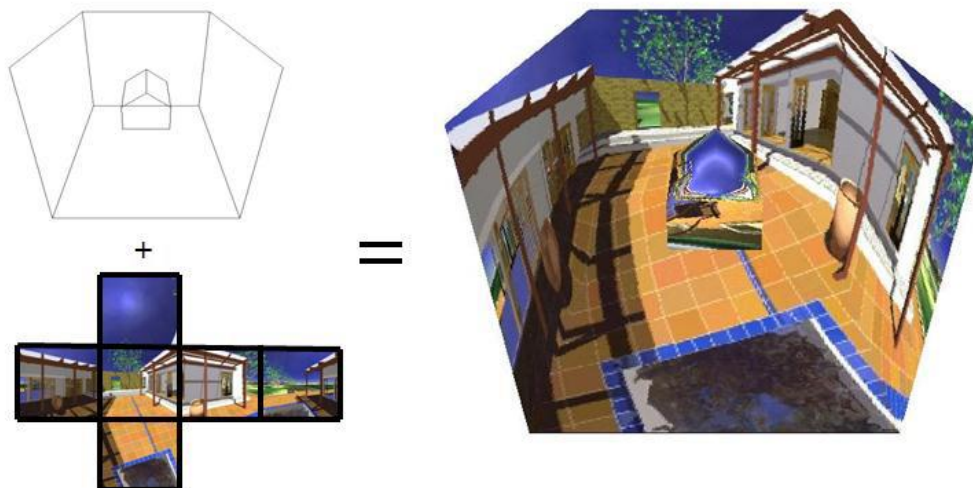


Figure 2-2 Cube mapping

3D textures

In addition to 2D textures and cube maps, there is a ratified OpenGL ES 2.0 extension for 3D textures called `GL_OES_texture_3D`. This extension allows 3D texture initialization and use for volumetric rendering purposes. This is a core functionality starting with OpenGL ES 3.0.

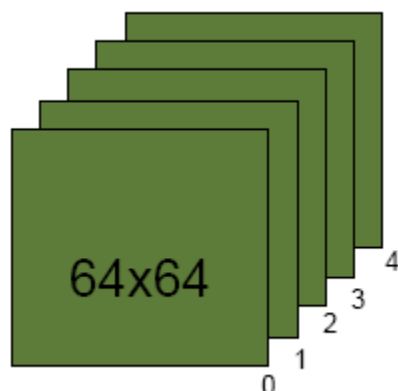


Figure 2-3 3D texture

Large texture size

Adreno 330 supports texture sizes up to 8192x8192x8192. Depending on memory availability, Adreno 420 can address textures of resolution up to 16384x16384x16384.

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. To get the best viewing experience with correct colors, it is important that the color space for render targets and textures matches the color space for the display, which is sRGB. Unfortunately, OpenGL ES assumes linear or RGB color space by default. As Adreno 3xx, 4x, and 5x support sRGB color space for render targets as well as

textures, it is possible to ensure a correct color viewing experience. Note that Vulkan fully handles sRGB in both textures and swapchain presentable images.

PCF for depth textures

Adreno 3xx, 4x, and 5x have hardware support for the OpenGL ES 3.0 and Vulkan feature of Percentage Closer Filtering (PCF). A hardware bilinear sample is fetched into the shadow map texture, which alleviates the aliasing problems that can be seen with shadow mapping in real time applications.

2.3.2 Visibility processing

Early Z rejection

Early Z rejection provides a fast occlusion method with the rejection of unwanted render passes for objects that are not visible (hidden) from the view position. Adreno 3xx and 4x can reject occluded pixels at up to 4x the drawn pixel fill rate.

Figure 2-4 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 trying to render a new primitive onto the same pixels of the existing color buffer that has the Z-buffer value of 2 (as shown in the second grid with green blocks), the conflicting pixels in this new primitive will be rejected as shown in the third grid representing the final color buffer. Adreno 3xx and 4x can reject occluded pixels at up to four times the drawn pixel fill rate.

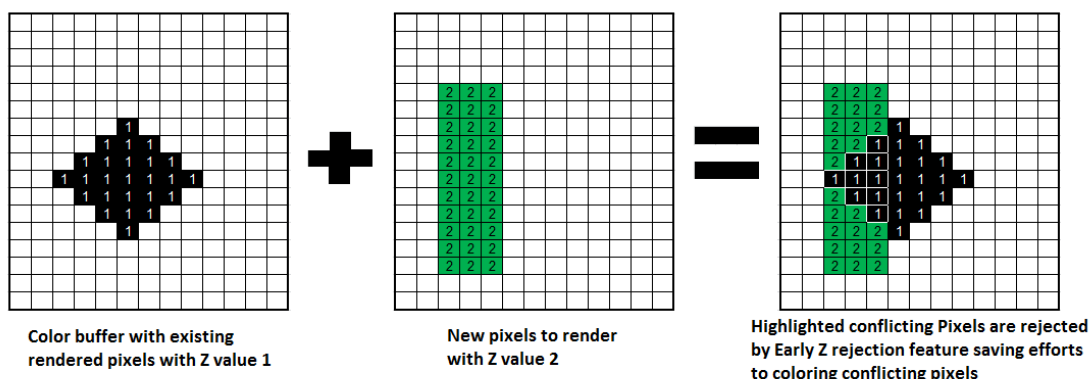


Figure 2-4 Early Z rejection

To get maximum benefit from this feature, QTI recommends 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 is useful for applications that have high-depth complexity.

FlexRender™ (hybrid deferred and direct rendering mode)

QTI introduced its new FlexRender solution as part of Adreno 3x, 4x, and 5x. FlexRender refers to the ability of the GPU to switch between indirect rendering (binning or deferred rendering) and direct rendering to the frame buffer.

There are advantages to both the direct and deferred rendering modes. The Adreno 3x, 4x, and 5x GPUs were designed to maximize performance by switching between the two modes in a

dynamic fashion. This works by the GPU analyzing the rendering for a given render target and selecting the mode automatically.

Tile-based rendering

To optimize rendering for a low-power and memory-bandwidth-limited device, Adreno GPUs use a tiled-based rendering architecture. This rendering mechanism breaks the scene frame buffer into small rectangular regions for rendering. The region sizes are automatically determined 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 deferred mode rendering mechanism of the Adreno GPU uses tile-based rendering and implements a binning approach is used to create bins of primitives are processed in each tile.

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

In [Figure 2-5](#), you see the two passes that are performed over the graphic primitives (Binning and Rendering). In this example there are 3 triangles that will be rendered in the frame buffer. The Binning Pass marks which bins a triangle is visible in (visibility stream). This stream is stored to system memory.

In the Rendering Pass, for each tile to be rendered, only the visible primitives are processed by reading the visibility stream. By using the 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.

It should be noted that Vulkan's Renderpass feature is highly advantageous to tiling architectures like Adreno, because many rendering passes can be done in GMEM and the costly resolve operations can be minimized.

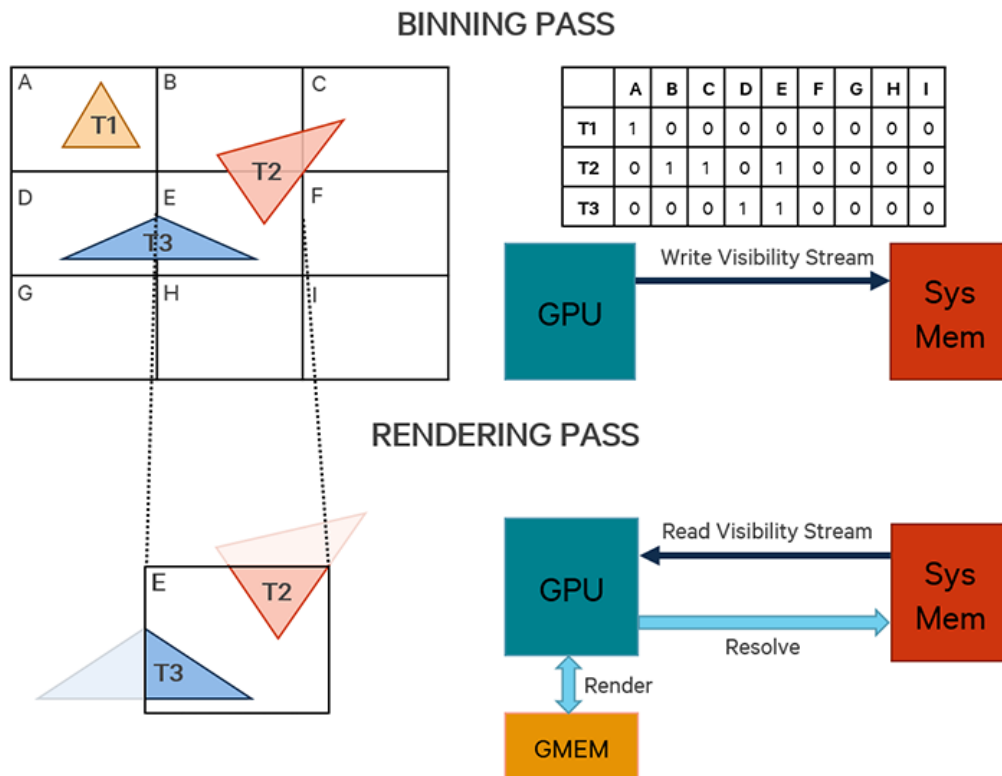


Figure 2-5 Tiled/binned rendering architecture with Adreno 3xx, 4x, and 5x

2.3.3 Shader support

Unified shader architecture

All Adreno GPUs support the Unified Shader Model, which allows for use of a consistent instruction set across all shader types (vertex and fragment shaders). In hardware terms, Adreno GPUs have computational units, e.g., ALUs, that support both fragment and vertex shaders.

Adreno 4x uses a shared resource architecture that allows the same ALU and fetch resources to be shared by the vertex shaders, pixel or fragment shaders, and general purpose processing. The shader processing is done within the unified shader architecture, as shown in [Figure 2-6](#).

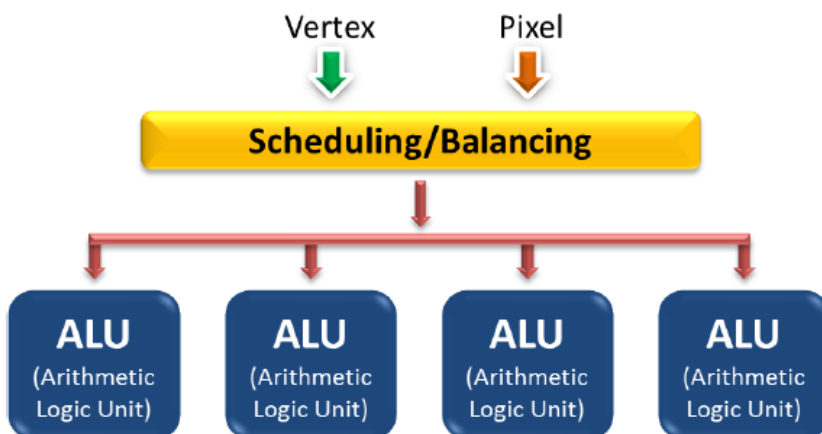


Figure 2-6 Unified shader architecture

[Figure 2-6](#) 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 [Figure 2-7](#). This allows for greater flexibility of pixel and vertex load balances.

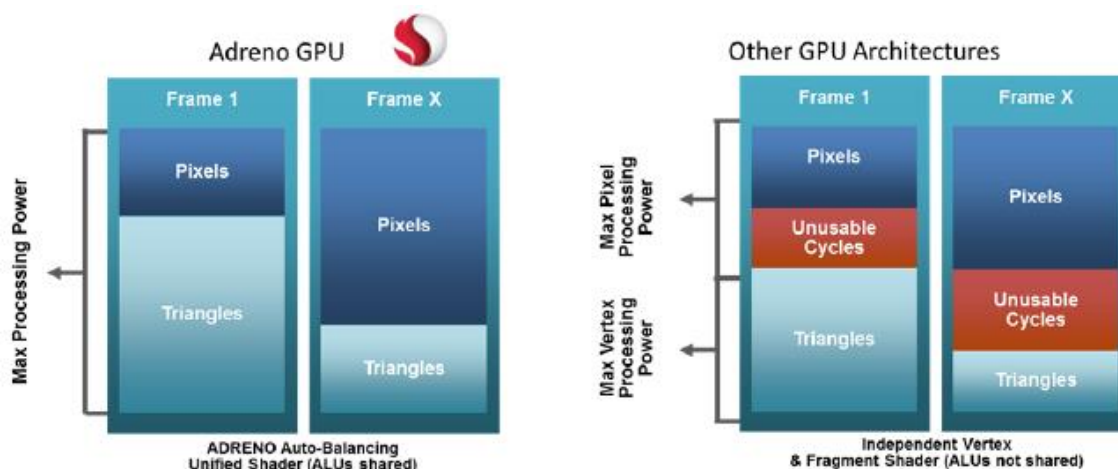


Figure 2-7 Flexibility in shader resources – Unified shader architecture

The Adreno 4x and 5x shader architecture is also multithreaded, e.g., if a fragment shader execution stalls due to a texture fetch, the execution is given to another shader. Multiple shaders are accumulated as long as there is room in the hardware.

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

Scalar architecture

Adreno 4x and 5x has a scalar component architecture. The smallest component Adreno 4x can support natively is a scalar component. This results in more efficient hardware resource use for processing scalar components, and it does not waste a full vector component to process the scalar.

The scalar architecture of Adreno 4x and 5x can be twice as power-efficient and deliver twice the performance for processing a fragment shader that uses medium-precision floating point (mediump), compared to other mobile GPUs today, which use vector architecture. For Adreno 4x, and 5x, mediump is a 16-bit floating point and highp is a 32-bit floating point.

2.3.4 Low power performance

Adreno GPUs continue improving their performance each generation because of constant architectural improvements to the physical chip design. Measurements of both graphics performance and power usage while using industry benchmarks, improve significantly when compared to the previous hardware. This “performance per milliwatt” enhancement of the Adreno GPU is an important trend that is expected to continue in the future.

2.3.5 Universal bandwidth compression (UBWC)

Universal bandwidth compression (UBWC) is supported by all Adreno 5x GPUs. UBWC is a unique predictive bandwidth compression scheme used by the Adreno GPU to improve throughput to system memory. By minimizing the bandwidth of data, significant DDR power savings can be achieved. Combined with the binning architecture, UBWC provides a bandwidth advantage over an equivalent direct rendered without UBWC.

UBWC works across many components in the Snapdragon chip including the GPU, Display, Video, and Camera. The compression supports YUV and RGB formats, and reduces memory bottlenecks.

2.3.6 Draw order independent depth rejection

Adreno 5x GPUs support a draw order independent depth rejection feature which has the goal of preventing the processing and rendering of occluded primitives. This depth rejection occurs independent of the draw order.

This feature has the advantages of reducing memory access, reducing rendered primitives, not requiring an application to draw front to back, and allowing for an improved frame rate.

2.3.7 Compute

Adreno 5x significantly improved the GPGPU/Compute and Quality of Service.

2.3.8 Other supported features

Index types

A geometry mesh can be represented by two separate arrays, one array holding the vertices, and another holding sets of three indices into that array, which together define a triangle.

Adreno 4x natively supports 8-bit, 16-bit, and 32-bit index types. Most mobile applications use 16-bit indices.

Multisample anti-aliasing (MSAA)

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, multisampling is efficiently supported by Adreno 4x. As shown in [Figure 2-8](#), 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. And 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. Adreno 4x supports the use of two or four samples per pixel.

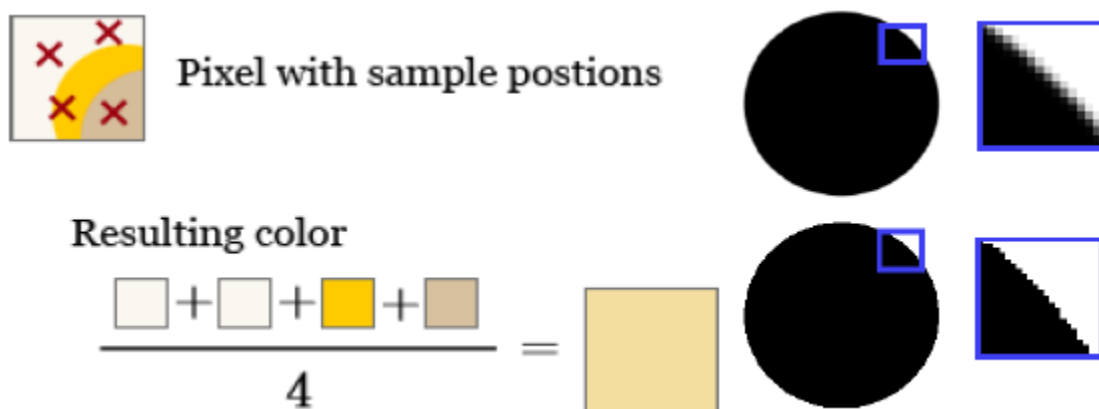


Figure 2-8 MSAA

Vertex texture access or vertex texture fetch

With the advantage of having shared resources to process vertex and fragment shaders, in the Adreno GPUs the vertex shader has direct access to the texture cache. It is simple to implement vertex texture algorithms for function definitions, displacement maps, or lighting level-of-detail (LoD) systems on Adreno GPUs. Vertex texture displacement is an advanced technique that is used to render realistic water in games on a desktop and for consoles. The same can now be implemented in applications running on Adreno GPUs.

The following is an example of how to do a texture fetch in the vertex shader:

```
/////vertex shader
attribute vec4 position;
attribute vec2 texCoord;
uniform sampler2D tex;
void main() {
```

```
float offset = texture2D(tex, texCoord).x;  
...  
gl_Position = vec4(...);  
}
```

2.3.9 Adreno APIs

Adreno 4x supports Khronos standard APIs including:

- OpenGL ES 1.x (fixed function pipeline)
- OpenGL ES 2.0 (programmable shader pipeline)
- OpenGL ES 3.0
- EGL
- OpenCL 1.1e
- DX11 FL 9.3

Adreno 4xx additionally supports:

- OpenGL ES 3.1 (most recent version of this API)
- AEP
- OpenCL 1.2full

Adreno 5.x additionally supports:

- Vulkan 1.0
- OpenCL 2.0 Full Profile
- DX12 FL 12

Along with the OpenGL ES APIs, the extensions to these APIs are also supported.

2.4 Vulkan

Vulkan is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems. It is a new API designed for modern GPUs such as Adreno. It consists of an equal desktop feature set, creating a flexible and powerful low-level interface between software and graphics acceleration.

2.4.1 Vulkan versions

Vulkan 1.0

Vulkan is a new API with a different approach to the rendering pipeline. It was formally released in February, 2016.

For a complete description of the API, see <https://www.khronos.org/api/vulkan>.

2.5 About Android

Android is a mobile operating system based on the Linux kernel and is currently developed by Google.

2.5.1 Vulkan support on Android

Android supports one version of the Vulkan API.

- Vulkan 1.0 – This API specification is supported by Android 7.0 (Nougat) (API level 24) and higher.

2.5.2 Android and Vulkan on Adreno

The Adreno GPU versions support the Vulkan 1.0 specification. there is a minimum version of the Android OS required. [Table 2-1](#) lists these requirements.

Table 2-1 Adreno, Vulkan, and Android versions

Adreno ver	Vulkan ver supported	Android ver required
Adreno 5xx	1.0	Android 7.0 (Nougat)

3 Vulkan 1.0 with Adreno

3.1 Development environment

Before developing Vulkan applications, it is necessary to set up a suitable development environment. A development system is needed, which can be based on Windows, Linux, or OSX. There must also be a target system for testing the application. For the purposes of this document, that means an Android device or emulator.

3.1.1 Development system

There are a number of software pieces that are required to create the development toolchain. The required software packages are as follows.

Adreno SDK for Vulkan

The Adreno SDK for Vulkan is based on using the Android Studio IDE and contains numerous examples which use a common framework which encapsulates much of the Vulkan code. The SDK requires the use of a device with Vulkan drivers, as there is no desktop emulation supported. The SDK was developed using a Windows version of Android Studio, thus there may be some differences if developing on macOS or Linux.

Adreno SDK for Vulkan offers an demos, tutorials, and documentation.

Android developer tools

When developing using Android Studio, be sure to use the latest versions of both the Android Studio IDE, Gradle plugin, Gradle, and NDK. You can download Android Studio at <https://developer.android.com/studio/index.html>

Android NDK

The Android NDK is a toolset that allows code implementation using native languages, such as C and C++. All of the Adreno SDK for Vulkan samples are developed in this manner, so it is necessary to install the NDK to take advantage of these samples. You can download the NDK at <https://developer.android.com/ndk/downloads/index.html>.

3.1.2 Target system

When deploying and testing an application, there is there only one possibility as discussed here. Currently there is no Vulkan emulator support for Adreno Studio.

Existing Adreno-based consumer devices

The Adreno GPU is used in a broad range of tablet and mobile phone devices from major manufacturers. Many Adreno GPUs 5xx and higher will have support for Vulkan.

3.2 Setup instructions

This section lists the steps needed to install the software toolchain for building Vulkan applications for Android. This material also provides information and instructions on how to set up a working toolchain, which can lead to the building, installation, and running of any sample application in the Adreno SDK.

3.2.1 Android development on Windows

These instructions cover the setup of a development environment on a Windows x64 development system with an Android target device. If a different platform is necessary for the development system, the details of the installation may differ.

Install the Adreno SDK for Vulkan together with a number of Android development packages. These instructions aim to get a development project up and running in the shortest time possible.

3.2.1.1 Set up the Adreno SDK for Vulkan

1. Download the Adreno SDK for Vulkan from the Qualcomm Developer Network website at <https://developer.qualcomm.com/download/>.
2. Extract all files to a folder, e.g., C:\AdrenoSDKVulkan_Windows.
3. Follow the instructions in the extracted DeveloperGuide.pdf file in the documents folder.
4. You can also view the SampleGuide.pdf to get an overview of the samples in the SDK.

3.2.1.2 Android NDK

1. Download the Windows 64-bit NDK from the Android Developers site at <http://developer.android.com/tools/sdk/ndk/index.html>.
2. Extract all files into the Android directory, e.g., C:\Android\android-ndk-r14.
3. Add the environment variable ANDROID_NDK_ROOT=C:\Android\android-ndk-r14 to the Windows system.

NOTE: This path may change depending on the latest version number of the NDK.

4. Add this to the Windows PATH environment variable %ANDROID_NDK_ROOT%
5. Update the project structure in Android Studio for the NDK path (File/Project Structure/SDK Location) that will be used for new and existing projects. You should also check that any sdk.dir files in your projects are updated with the appropriate NDK path.

3.2.1.3 JDK

JDK is a requirement for Android Studio and will be needed before Android Studio can run.

1. Download the Windows x64 Oracle JDK from the Oracle Technology Network site at <http://www.oracle.com/technetwork/java/javase/downloads/>.
2. Install the JDK to C:\Program Files\Java.

3. Add this environment variable to the Windows system `JAVA_HOME=C:\Program Files\Java\jdk1.8.0_05`.

NOTE: This path may change depending on the latest version number of the JDK.

4. Add this entry to the PATH variable `%JAVA_HOME%\bin`.

3.2.1.4 Build and run sample application

1. Verify you have an Android device with Vulkan drivers connected to your computer. You should be able to see the device when running the command “adb devices”
2. Open Android Studio
3. Open the Triangle sample by selecting (File/Open...) and selecting the Triangle folder in the Adreno SDK for Vulkan sample's folder.
4. Select Build/Clean Project
5. Select Run/Run ‘app’
6. The app should get installed to the device and be executed. A colored triangle will be rendered in the app using the Vulkan api.

3.2.1.5 Vulkan video tutorials

There are several Vulkan Video tutorials on our developer site that will be helpful when starting out with Vulkan development using our Adreno SDK for Vulkan.

These videos give an overview of Vulkan on Qualcomm Snapdragon Processors, using Vulkan geared for OpenGL ES developers, capturing Vulkan Applications with the Snapdragon Profiler, and An Introduction to Qualcomm Adreno SDK for Vulkan.

You can find the videos at <https://developer.qualcomm.com/software/adreno-gpu-sdk/tutorial-videos>

3.3 Walkthrough of sample applications

You will find the various sample walkthrough videos useful to understanding how to develop a Vulkan application. In particular the videos “Triangle Sample Walkthrough”, and “Cornell Lights Sample Walkthrough” will cover the basic steps of implementing Vulkan Rendering.

You can also find videos which accompany some of the tutorials at <https://developer.qualcomm.com/software/adreno-gpu-sdk/tutorial-videos>

4 Using Vulkan with Adreno

4.1 Application information

Applications begin using Vulkan by creating a Vulkan Instance. The instance logically provides your application with Vulkan state that is different than other Vulkan applications that might be using Vulkan. By itself, Vulkan doesn't contain any global state about running applications.

With Adreno drivers, you can create as many Vulkan instances as you like, although there is not currently a known use case for having more than one.

Adreno Vulkan drivers ignore the free form Application information passed in a `VkApplicationInfo` structure when calling `vkCreateInstance`. The information may be used by third party extension layers during their debugging or reporting, so it is good to still fill in these values.

4.2 Physical device

When running on Adreno-based devices, after a `VkInstance` is created, apps will detect a single `VkPhysicalDevice` when enumerating the devices with `vkEnumeratePhysicalDevices`.

Device properties are returned with `vkGetPhysicalDeviceProperties`. Here are typical values that are returned using a recent Vulkan build on a device with an Adreno 530 GPU.

```
apiVersion: 1.0.31 (use VK_VERSION_MAJOR, VK_VERSION_MINOR,  
VK_VERSION_PATCH macros)  
driverVersion: 0.338.119431 (use VK_VERSION_MAJOR, VK_VERSION_MINOR,  
VK_VERSION_PATCH macros)  
vendorID: 0x5143  
deviceType: VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU  
deviceName: Adreno (TM) 530  
pipelineCacheUUID: [170][36][21][0][67][81][0][0][0][0][2][0][3][5][0][0]  
limits: (see Appendix A)  
  
sparse properties: (see Appendix B)
```

Device features are returned with `vkGetPhysicalDeviceFeatures`. These are option features that may be supported by a physical device such as supporting an image cube array or ETC2 texture compression. You can find the list of advertised features in Appendix C.

Checking the API version is an important consideration to determine if the drivers are sufficiently robust enough to run with your commercial game. Your testing should reveal the minimum acceptable Vulkan api to run your game, and opt to not use Vulkan otherwise (i.e. use a GL rendering path instead).

4.3 Validation layers

Performing error checking in a graphics API is problematic because of performance concerns. Vulkan provides an elegant solution for error-checking that is both comprehensive in the coding areas that are checked, but also easy to disable when creating production builds.

Validation Layers are how error checking is implemented in Vulkan. These layers are optional binaries that are loaded at run time and intercept Vulkan calls. If the layer does not support the current API entry points, the system passes the entry point onto the next layer. Eventually the entry point reaches the driver where it is executed.

The layers are specialized in various areas of the API that they check. They are defined in the Android NDK and are loaded by the Vulkan loader. The libraries shown in [Table 4-1](#) implement the validation:

Table 4-1 Validation layers library

Filename	Name
libVkLayer_threading.so	VK_LAYER_GOOGLE_threading
libVkLayer_parameter_validation.so	VK_LAYER_GOOGLE_parameter_validation
libVkLayer_object_tracker.so	VK_LAYER_LUNARG_object_tracker
libVkLayer_core_validation.so	VK_LAYER_LUNARG_core_validation
libVkLayer_image.so	VK_LAYER_LUNARG_image
libVkLayer_swapchain.so	VK_LAYER_LUNARG_swapchain
libVkLayer_unique_objects.so	VK_LAYER_LUNARG_uniqueobjects

On Android, you must first build your project with the validation layers so they are included in your apk.

Once the layers are included with your project, you can enumerate them using `vkEnumerateInstanceLayerProperties`. You need to both check that the libraries are available and then request them by passing their name to `vkCreateInstance` when the instance is created.

It's important to request the layers in the order shown in the table above, so that basic error checking can be done before more specialized error checking is done.

You need to use the Debug Report extension, `VK_EXT_debug_report`, so that your application can get notified when validation errors occur. You can check if the extension is available using `vkEnumerateInstanceExtensionProperties` and request the extension by passing its name (`VK_EXT_DEBUG_REPORT_EXTENSION_NAME`) to `vkCreateInstance` when the instance is created.

You then create the callbacks using `vkCreateDebugReportCallbackExt`, specifying which debug reports you want to be notified on

```
VK_DEBUG_REPORT_INFORMATION_BIT_EXT = 0x00000001,
VK_DEBUG_REPORT_WARNING_BIT_EXT = 0x00000002,
VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT = 0x00000004,
VK_DEBUG_REPORT_ERROR_BIT_EXT = 0x00000008,
VK_DEBUG_REPORT_DEBUG_BIT_EXT = 0x00000010,
```

When your callback is called, you can then filter based on the debug report flag, and optionally print the report using `__android_log_print`.

Validation errors aren't perfect as you'll get a few false positives. As the error checking improves, this will likely decrease. Still getting the reports is a terrific way to both find unobvious errors in your code, and strengthen your code for running on Vulkan implementation on other platforms. We strongly recommend running with them turned on when developing your application.

You can also inspect the source code for the validation errors in the NDK at `..\sources\third_party\vulkan\src\layers` or in the source code from [LunarG](#). Although using the NDK libraries is convenient, downloading and building the Validation Layers from the most current Lunar G SDK, will provide most up to date error checking

This is useful for better understanding the logic the validation layers use and get a deeper understanding of the problem being reported. If you confident the error is a false positive, you can easily add code in your callback function to ignore the debug report. You might also choose to report the issue to LunarG. In the LunarG SDK there is a document, `validation_layer_details.html`, which details what the validation layers check for.

This process of using Validation Layers is defined on the [Android NDK developer site](#) and implemented in the framework code in our Adreno SDK for Vulkan.

4.4 Extensions

Extensions may define new Vulkan commands, structures, and enumerants which are defined in the Khronos-supplied `Vulkan.h` file together with the core API. Commands defined by extensions may not be available for static linking- in which case function pointers to these commands should be queried at runtime.

To enable all Android specific extensions, be sure to define `VK_USE_PLATFORM_ANDROID_KHR` before including the Vulkan header file:

```
#define VK_USE_PLATFORM_ANDROID_KHR
#include <vulkan/vulkan.h>
```

Extensions can be available to both the created Vulkan instance as well as the Vulkan device.

The following instance extensions are currently supported in the latest Adreno drivers. You can get a list of extensions supported in the driver installed on your device by calling the `vkEnumerateInstanceExtensionProperties` api. Generally, you make 2 calls to the api; once to get the number of extensions so that you can allocate a structure large enough to hold the names, and a second time to get the extension names.

`VK_KHR_surface` – required to activate the general WSI surface extension

`VK_KHR_android_surface` – required to activate the Android specific WSI surface extension

`VK_EXT_debug_report` – required to receive callbacks when using validation layers

`VK_KHR_get_physical_device_properties2` – Provides new entry points to query device features, device properties, and format properties in a way that can be easily extended by other extensions, without introducing any further entry points. Also uses `sType/pNext` members for core queries.

`VK_ANDROID_native_buffer` – `vkCreateImage` extension structure for creating an image that is back by a `gralloc` (graphics memory allocator) buffer.

VK_KHR_maintenance1 – Adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release including vkCmdFillBuffer on transfer-only queue, copying images between 2D and 3D images, supporting left handed NDC space (no need to invert y coordinate in vertex shader), and others.

VK_KHR_get_memory_requirements2 – (TBD)

The following device extensions are currently supported in the latest Adreno drivers. You can get a list of extensions supported in the driver installed on your device by calling the vkEnumerateDeviceExtensionProperties api. Generally, you make 2 calls to the api; once to get the number of extensions so that you can allocate a structure large enough to hold the names, and a second time to get the extension names.

VK_KHR_swapchain – Device-level companion to the VK_KHR_surface extension. It introduces VkSwapchainKHR objects, which provide the ability to present rendering results to a surface.

VK_KHR_maintenance1- Adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release

More details of these extensions can be found on the [Khronos](#) site.

Aside from using/enabling the Validation Layers in production code, application may opt to query the VkResult value for some or all of the Vulkan API calls. Errors such as VK_ERROR_DEVICE_LOST, VK_OUT_OF_DEVICE_MEMORY, and VK_ERROR_EXTENSION_NOT_PRESENT may be caught and cause the application to log more information, die gracefully, or attempt to recover.

4.5 Queues

Vulkan Queues are where much of the GPU work is done. Devices have one or more queues which belong to queue families. A Queue family is a set of queues that have similar properties and can be run in parallel.

There are several capabilities of a queue, depending on the work it can perform:

VK_QUEUE_GRAPHICS_BIT = 0x00000001

Supports graphics operations such as drawing geometry

VK_QUEUE_COMPUTE_BIT = 0x00000002

Supports compute operations such as dispatching compute shaders

VK_QUEUE_TRANSFER_BIT = 0x00000004

Supports transfer operations such as copying images and buffers

VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008

Supports sparse memory bindings operations

Queues also have a value for the Timestamp valid bits which is the number of bits that are valid when timestamps are taken from the queue.

Queues also have a value for the minimum image transfer granularity which is the minimum image size for transfer (if the queue supports transfers).

On Adreno only queues with both Graphics and Compute capabilities can be created. As shown in [Table 4-2](#), vkGetPhysicalDeviceQueueFamilyProperties returns the following on Adreno:

Table 4-2 Queue capabilities

Queue	Flags	Count	Timestamp valid bits	Minimum image transfer granularity
0	3: Graphics Compute	3	48	[1] [1] [1]

You may want to have a dedicated submit queue thread to improve performance.

This Queue family also supports presenting, which can be confirmed by call `vkGetPhysicalDeviceSurfaceSupportKHR`.

4.6 Logical device

A Vulkan logical device is created after choosing a physical device, the type and number of queues, and determining which layers and extensions are needed. The Vulkan api `vkCreateDevice` is used to create the logical device.

A queue priority (0.0 to 1.0) can be assigned for each queue. On Adreno, there are 3 values that are internally used (0.0,0.5,1.0). It's important to note that these priorities are relative to all processes running on the device. In general, you should use a 1.0 priority, except with VR applications where you should choose a 0.5 value so that VR services running on the device needed for predicable display purposes can have higher priority.

4.7 Memory

Device memory in Vulkan refers to memory that is accessible to the device and is used to store textures, buffers, and other data. There are several types of data, enumerated by `VkMemoryPropertyFlags`:

```
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT = 0x00000001
```

Memory is local to the device

```
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT = 0x00000002
```

Memory allocations with this type can be mapped and read or written by the host

```
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004
```

Memory is coherent (up-to-date) when accessed by both host and device

```
VK_MEMORY_PROPERTY_HOST_CACHED_BIT = 0x00000008
```

Memory is cached by the host so reads are faster

```
VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT = 0x00000010
```

Memory allocated isn't necessarily consumed immediately and may be deferred by the driver

As shown in [Table 4-3](#) and [Table 4-4](#), using `vkGetPhysicalDeviceMemoryProperties` on Adreno returns the following information:

Table 4-3 Memory types supported on Adreno 5xx

Memory type	Property flags	Heap index
0	1: Device Local	0
1	11: Device Local Host Visible Host Cached	0
2	15: Device Local Host Visible Host Cached Host Coherent	0

Table 4-4 Memory heaps supported on Adreno 5xx

Memory type	Flags	Size
0	1: Local to the device	2895253504 bytes

Memory suballocation recommendation – allocating a single memory allocation and using that memory block for many objects

Even better single memory allocation, single buffer, but bind the same buffer with different offsets. One buffer can have many types of data.

On Adreno, the page alignment size can be quite large, so hence the recommendation to minimize actual allocations done with calls to `vkAllocateMemory`.

We discourage the attempt of using a custom memory allocator, except for debugging purposes such as logging memory allocations. It is doubtful that developing a custom memory would provide any performance improvement.

4.8 Window system integration (WSI)

WSI refers to a set of optional Vulkan extensions that support the display of results onto the platform's windowing system. Vulkan can be used without displaying results, hence WSI support is needed for any type of visual presentation.

The WSI Platform refers to the Operating System and Window System that Vulkan is being run on. For Adreno implementations of Vulkan, only the Android Operating System is covered in this document.

4.8.1 Surface abstraction

On Android, a surface abstraction is first needed when using WSI. There is an extension, `VK_KHR_surface_extension`, that allows a `VkSurfaceKHR` object to be created. This object abstracts the native platform window which to render to. This extension uses a `VkAndroidSurfaceCreateInfoKHR` structure to describe which Android Native Window the surface is matched to. You can refer to the framework code in the Adreno SDK for Vulkan for how this Window handle is accessed and how the `VkSurfaceKHR` object is created.

A check should be done using `vkGetPhysicalSurfaceSupportKHR` to confirm that the physical device supports presenting of the `VkSurfaceKHR`.

You can call `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` to get more information about the surface. The below values are returned on Adreno 5xx

```
minImageCount 2
maxImageCount 3
currentExtent [1600][2392] or [1440][2392]
minImageExtent [1][1]
maxImageExtent [4096][4096]
maxImageArrayLayers 1
supportedTransforms 271 =  IDENTITY_BIT |
                          ROTATE_90_BIT |
                          ROTATE_180_BIT |
                          ROTATE_270_BIT |
                          INHERIT_BIT
currentTransform = IDENTITY_BIT
supportedCompositeAlpha 8 = INHERIT_BIT
supportedUsageFlags 159 =  TRANSFER_SRC_BIT |
                          TRANSFER_DST_BIT |
                          SAMPLED_BIT |
                          STORAGE_BIT |
                          COLOR_ATTACHMENT_BIT |
                          INPUT_ATTACHMENT_BIT
```

As shown in [Table 4-5](#), you can also get information about the formats that the surface supports by calling `vkGetPhysicalDeviceSurfaceFormatsKHR`. On Adreno 5xx, the following formats are supported:

Table 4-5 Supported surface formats on Adreno 5xx

Formats	Color space
VK_FORMAT_R8G8B8A8_UNORM	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
VK_FORMAT_R8G8B8A8_SRGB	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
VK_FORMAT_R5G6B5_UNORM_PACK16	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR

We recommend using the first format.

4.8.2 Swap chain

The swap chain is a separate Vulkan object used to work with the native window system to create images that are drawn into and then presented onto the Vulkan surface. Logically, the swap chain is a set of images in a queue that the application accesses to get an image to render to, and then hand back to the swap chain for presentation. This system efficiently allows one image to be presented, while another image is being drawn to.

There is an extension `VK_KHR_swapchain` that is needed to create the swap chain. This extension is always supported in Adreno drivers and should be enabled when the Vulkan device is created.

To create a swapchain `vkCreateSwapchainKHR` is called using a `VkSwapchainCreateInfoKHR` structure filled out with information about the surface, image format, color space, dimensions, present mode and other data.

We recommend the following values:

```
VkSwapchainCreateInfoKHR swapchainCreateInfo = {};
swapchainCreateInfo.sType =
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
swapchainCreateInfo.surface = mSurface;
swapchainCreateInfo.minImageCount =
surfaceCapabilities.maxImageCount;
swapchainCreateInfo.imageFormat = VK_FORMAT_R8G8B8A8_UNORM;
swapchainCreateInfo.imageColorSpace =
VK_COLOR_SPACE_SRGB_NONLINEAR_KHR;
swapchainCreateInfo.imageExtent.width = mWidth (see Section 4.8.2 below);
swapchainCreateInfo.imageExtent.height = mHeight (see Section 4.8.2 below);
swapchainCreateInfo.imageUsage =
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
swapchainCreateInfo.preTransform =
VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
swapchainCreateInfo.queueFamilyIndexCount = 0;
swapchainCreateInfo.pQueueFamilyIndices = nullptr;
swapchainCreateInfo.imageArrayLayers = 1;
swapchainCreateInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
swapchainCreateInfo.compositeAlpha =
VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR;
swapchainCreateInfo.presentMode = VK_PRESENT_MODE_FIFO_KHR;
swapchainCreateInfo.clipped = VK_TRUE;
```

The `minImageCount` parameter is used to define the number of images in the swapchain. On Adreno `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` will return the `minImageCount` as 2 and the `maxImageCount` as 3. Setting to 2 enables double buffering and setting to 3 enables triple buffering. We recommend using 3 images in the swapchain. Using 2 will reduce memory use, but has the potential for some jittering.

Image usage should just be set to minimum number of usages needed; preferably just `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` is specified.

Sharing mode defines how the images are shared across queues and recommend using `VK_SHARING_MODE_EXCLUSIVE` which will ignore the queue family values in the structure. This will work on Adreno hardware because the single queue family supports both graphics and presenting.

`PreTransform` should be set to identity for most Android applications. Landscape and Portrait display support is best handled with your application's screen orientation manifest setting (`AndroidManifest.xml`). If you do use the transform bit, you need to also make sure you provide the correctly rotated dimensions for that surface (instead of having the driver figure it out). Otherwise you may have performance problems as well as some corruption issues.

`CompositeAlpha` defines how transparent images are composited by the native window system. On Android for most applications, use `VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR`

There are several present modes including FIFO, FIFO relaxed, mailbox, and immediate. Immediate will likely show visual tearing, but does have the lowest latency. FIFO is an excellent choice for most applications that are running slower than the refresh rate of the device. FIFO relaxed and mailbox are good choices for applications running faster than the refresh rate, however mailbox may result in having the GPU render more frames that will be required.

4.8.3 Hardware scaling

An effective way to improve performance in your application is to reduce the rendering frame size, if you can do so without compromising on image quality. Depending on your implementation of scaling, a side benefit will be that rendering performance of your application will be similar regardless of the screen size.

In Vulkan on Android, when the swapchain image size does not match the surface size (return by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`), it will be scaled by the hardware to match the surface.

Some background info on this Android feature can be found at <https://android-developers.googleblog.com/2013/09/using-hardware-scaler-for-performance.html>

4.9 Command buffers

We strongly recommend the use of secondary command buffers that can be reused frame after frame. Secondary command buffers do not share state with the primary command buffer (aside from using the same attachments), nor do they handle any renderpass setup commands. Secondary buffers are ideal for rendering static geometry and other scene elements that don't need to be recorded every frame.

In a primary command buffer, you will get better performance by minimizing pipeline changes (i.e. sort drawing by pipelines).

In a secondary command buffer, you will get better performance by sorting draws front to back relative to the scene camera (e.g. draw skybox last).

We recommend use threads to record multiple command buffers, as the process can be slow. When threading use one command pool per thread.

Avoid creating command buffers for simple geometry (single quad). Because of their overhead, command buffers should have a reasonable amount of work recorded in them. Also, be reasonable in the number of draws that are recorded in a command buffer. Similar strategies used in OpenGL applications like batching draw calls based on similar state and shaders (i.e. pipelines) still apply in Vulkan.

We discourage the use of `VkCmdClear*` functions as they have low performance (the preferred way is to use the load operations in the renderpass). Command Buffers should generally be reallocated when they need to be recorded.

4.10 Pipelines

The Vulkan pipeline is a complex object representing state, shaders, and other data needed to render geometry. There are 2 types of pipelines: graphics and compute. Both function similarly, with the differences involving the type of state data and shaders needed for the different processing.

Because pipelines contain much information that needs to be organized, particularly shader code which needs to be compiled and linked, their creation with the `vkCreateComputePipelines` and `vkCreateGraphicsPipelines` APIs is very time-consuming. One common technique is to frontload the creation of all needed pipelines to the startup phase or non-critical path of the program, so no delays are experienced during the program execution.

4.10.1 Pipeline caches

Vulkan provides a feature to cache pipelines, so that their creation can be accelerated on subsequent runs. We strongly recommend this technique to improve the perform of your program.

Creation of a pipeline cache is done with `vkCreatePipelineCache`. The cache should be passed in as a parameter to `vkCreateComputePipelines` and `vkCreateGraphicsPipelines`. Once all pipelines are created, you should call `vkGetPipelineCacheData` to obtain the cache, and then save it to disk so that it can be reused on subsequent runs.

When reading the cache, you can inspect its header to confirm the cache is valid for the device by using the `vendorID`, `deviceID`, and `uuid` fields in the `VkPipelineCacheHeader` structure. These fields are available from the data returned `vkGetPhysicalDeviceProperties` and documented in Appendix A.

Because pipeline caches can be large, you should destroy them (after preserving them to disk for a subsequent run) with `vkDestroyPipelineCache`.

You can find a sample of using Pipeline Caches in the Adreno SDK for Vulkan.

4.10.2 Building pipelines in parallel

As mentioned in the previous section, building pipelines in parallel is a useful technique to utilize the multiple cores on the device and speed up the CPU intensive activity.

Each thread can build up a separate cache which Vulkan treats as thread safe. After all pipelines are created, you can then merge the caches using `vkMergePipelineCaches` to create a single pipeline cache which can then be saved and used on subsequent runs.

4.10.3 Derivative pipelines

Using derivative pipelines is an excellent idea, particularly when you encounter similar pipelines sharing shaders. The key advantage is that the cost to create a child pipeline that is similar to a parent pipeline is less than the cost to create the child pipeline without knowledge of a parent pipeline. This is primarily due to not needed to recompile the shaders. Also during run time, switching from/to derivative pipelines is quicker. Vulkan doesn't specify how much in common the pipelines need to have, but using the same shaders is a good prerequisite for derivation.

First make sure the parent pipeline is created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT`. When calling `vkCreateGraphicsPipelines` to create the child pipeline, use either the `basePipelineHandle` parameter or the `basePipelineIndex` provided you are also passing in parent pipeline.

It is probably best not to use the `VK_PIPELINE_CREATE_DERIVATIVES_BIT` unless you are creating a parent pipeline, because there is some overhead with using it.

4.11 Synchronization

An overall goal of Vulkan Synchronization is keeping the GPU busy and minimizing its idle time and avoiding any stalls both on the GPU and GPU waiting for each other. The strategy is to pipeline the work that the GPU does, over several frames. Overlapping CPU and GPU during rendering is generally a good approach. Also overlapping GPU vertex and fragment work is generally advised. Generally, try to provide the exact amount of synchronization you need, not too much or too little.

One concept to remember is that resources that the GPU needs (uniform and vertex buffers, textures, etc.) should not be modified or deleted until the GPU is finished accessing them.

Pipeline Barriers are considered “fine grained” (precise) synchronization primitive. They represent a single point in time in which all the processing done before it, is completed before proceeding with the processing done after it. Pipeline Barriers are useful for changing image layouts. One tip when using pipeline barriers is to submit multiple barriers into a single call to `vkCmdPipelineBarrier`, when possible.

Events are also “fine grained” synchronization primitives and useful for GPU/CPU synchronization. Instead of defining a single point in time, Events operate over a range in that you set an event and wait for it later, and activities in between are not interfered with by the synchronization. Rather the activities before the event are synchronized with the activities after the event. Events allow you to overlap work in between when the event is set and when it is triggered.

Renderpass Subpass Dependencies are also a “fine grained” synchronization feature built into to how subpasses are implemented by the GPU. Many pipeline barrier operations can alternatively and more efficiently be implemented by defining them in a render pass as a subpass dependency.

You can also use subpass dependences to wait on external operations such as waiting for a layout transition from a presentable to renderable layout, instead of using a pipeline barrier to do the same transition. This is a more efficient way to do this transition as well.

Semaphores are a “course grained” synchronization primitive useful for synchronizing queues. One common use case is to use acquire and submit semaphores to synchronize the graphics rendering and presenting the swap chain images.

Fences are also a “course grained” synchronization primitive useful for checking for frame completion, particularly if using a multibuffer rendering scheme. Fences are used to synchronize queue completion with execution on the CPU. You pass fences with the `vkQueueSubmit` call and wait for them to complete with `vkWaitForFences`. Fences are useful when doing multibuffering as you can easily check that a previously submitted frame to a work or present queue has been completed.

Avoid using `vkQueueWaitIdle` and `vkDeviceWaitIdle` as they are not useful towards the goal of always keeping the GPU busy. These functions are very heavyweight and should not be used in the middle of frame. Only use these just during the app’s teardown phase.

Although synchronization is confusing, using validation layers will help catch issues with the above-mentioned synchronization objects.

4.12 Renderpass

A Vulkan Renderpass is an alternative technique for rendering in multiple passes. You can think of it as an optimized way of implementing many standard rendering algorithms such as using g-buffers to implement deferred lighting using a gbuffer pass and a lighting pass. Typically, you would implement these algorithms with multiple render targets, but Vulkan Renderpass allows for a more concise and efficient implementation.

Setting up a renderpass requires defining subpasses, attachments, and dependencies between the subpasses. For tiled architectures such as Adreno, using the Renderpass object allows the graphics driver to perform significant optimization. Information about the flow of data, how it is produced and consumes allows a driver to optimally manage the rendering of each tile individually, potentially eliminating unneeded steps and minimizing bandwidth. Thus, the memory traffic for that tile is limited to on chip memory associated with that tile without the need to flush to main memory. Main memory is only written to when the tile is completely rendered. By using the information defined in the renderpass object, the driver can essentially fuse together the passes into a single actual render pass.

For optimal efficiency, Renderpass does require a dependency by region, where the sample pixel on the previous pass is feed into the same pixel on the current pass. This is specified with the `VK_DEPENDENCY_BY_REGION_BIT` flag in the `VkSubpassDependency` structure. If not set by region, the pass will wait for all pixels to be completed in the prior pass.

Given the advantages of renderpasses in a tiled based architecture such as Adreno, the more passes you can utilize in your rendering algorithm, the better the results will be. For example, encoding post processing operations in the same renderpass that generates the pre-post processed frame is advantageous.

Renderpasses are a fully connected graph of the rendering process. Make sure all dependencies between the passes are fully described.

Shaders need to be modified to work in renderpasses. In fragment shaders, you'll need to use the `subpassInput` type when defining uniforms used in the shader. You will also need to call `subpassLoad` when reading the subpass input.

Also, `VkPipelineLayout` objects need to be created with knowledge about both the renderpass and the subpass they'll be used with.

When defining the attachments used in a renderpass, be sure to set the `loadOp` and `storeOp` values appropriately. Using `VK_ATTACHMENT_LOAD_OP_CLEAR` or `VK_ATTACHMENT_LOAD_OP_DONT_CARE` tells the driver not to worry about loading the attachment from main memory. Likewise using `VK_ATTACHMENT_STORE_OP_DONT_CARE` tells the driver not to worry about storing the attachment to main memory. Don't over specify these values and only set the ones you need.

Use the subpass state transitions in dependencies rather than use pipeline barriers. For example, transition the swap buffer image for presenting and rendering by using 2 dependencies when creating the renderpass:

```
acquire.srcSubpass = VK_SUBPASS_EXTERNAL
acquire.dstSubpass = 0; // first
acquire.srcStageMask = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
acquire.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
acquire.srcAccessMask = 0;
acquire.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT - access type
```

```

present.srcSubpass = 2 // last subpass
present.dstSubpass = VK_SUBPASS_EXTERNAL
present.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_
present.dstStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
present.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
present.dstAccessMask = 0;

```

Although you can create a Renderpass using a compatible framebuffer and attachment images, using the actual framebuffer and attachment images will allow for better performance.

4.13 Framebuffers

Framebuffers are collections of attachments used in a render pass. A frame buffer object references all the image views that represent the attachments. You'll need to create one framebuffer for each color buffer image in the swap chain.

You can share a single depth buffer among the framebuffers because you are only rendering one frame at a time, and the depth buffer would not be needed for presentation (as the color buffers would)

4.14 Images

When creating images, only set the usage flags that you need.

ASTC is an excellent choice for compressing textures, which we strongly recommend. The family of ETC2 formats are also an excellent choice to consider.

An image layout refers to how the image is organized in memory. Depending on current usage of the image, a different layout would be more efficient for memory accessing and rendering needs. Conversions from one layout to another are referred to as layout transitions.

The image layout of each image subresource must be well defined at each point in the image subresources lifetime.

```

VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_GENERAL
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
VK_IMAGE_LAYOUT_PREINITIALIZED
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR

```

Image layout transitions are accomplished by either executing a command buffer with a memory barrier command using the `vkCmdPipelineBarrier` command, specifying a final layout in the render pass, or specifying a layout in a render pass subpass.

On Adreno, not all layout transitions are needed, but for future compatibility, cross platform, and device compatibility as well as a best practice, it is recommended to use them correctly. These transitions provide the driver with more information to potentially do optimization in the future. Validation layers are very useful for verifying images are in the correct layout.

You can review the framework code in the Adreno SDK for Vulkan for examples on how to implement the transitions using submitted command buffers. Example code shows how to use swapchain images for presenting and as color attachments, reading images that will be used as textures, and how to use images for render targets which are subsequently read by a shader.

Note that swapchain image layout transitions may also be done more efficiently in renderpasses. These transitions involve preparing the image for rendering before the render pass begins and preparing the image for presentation after the render pass is finished. A general recommendation for layouts with swap chains is:

Clear with `VK_IMAGE_LAYOUT_GENERAL`

Render with `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`

Present with `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`

Textures are best uploaded to a `VK_TILING_OPTIMAL` layout which requires uploading the texture in a staging buffer (refer to the framework class in the Adreno SDK for Vulkan). This layout is the most cache efficient for use with shaders.

4.15 Descriptors/push constants

We recommend the use of dynamic UBOs over push constants. Push constants have a limit size limit (128 bytes) and don't perform as well as UBOs.

Descriptor sets are a set of references to resources that are used by shaders and bound into the pipeline. These resources are textures, samplers, images, uniform buffers, etc. You can have multiple descriptor sets bound at one time, so shaders need to indicate which set the resources are to be bound from.

A descriptor set layout describes the quantity and order of resources in a set. It's like a blueprint that describes the organization, quantity, and placement of descriptors, but without referring to any specific data. This layout is used both when defining descriptor sets and when defining pipelines that will utilize such sets.

4.15.1 Descriptor set frequency

Descriptor sets should be designed with some consideration to frequency as you can bind descriptor sets independently of one another. One example of a design strategy is to have a descriptor set dedicated for scene data, one for mesh data, and one for instance data. The instance data descriptor set should change much more frequently than the scene descriptor set.

4.15.2 Descriptor pools

You should allocate one descriptor pool per thread that creates descriptor sets. Size the pool appropriately for what the pool will need to provide.

4.16 Shaders

The Vulkan API only supports shader modules which are in a format called SPIR-V. This format is used both with Vulkan and OpenCL APIs. These SPIR-V modules are created by calling `vkCreateShaderModule` which specifies the SPIR-V code in a `VkShaderModuleCreateInfo` parameter. The modules are then used in a `VkPipelineShaderStageCreateInfo` structure when creating a graphics or compute pipeline.

It is generally not advised to author SPIR-V byte code directly. Generally, a tool such as glslangValidator included in the [LunarG SDK](#) is the most convenient tool to use if your shaders are authored in GLSL. Refer to the spirv_toolchain.html document in the LunarG sdk for detailed usage instructions. Note that we don't have experience with the spirv-remap tool so we currently don't recommend it. There are also other tools available to create SPIR-V byte code from HLSL code. Also, Android Studio has an automatic method for compiling shaders into SPIR-V format (ahead of time or at runtime). Refer to <https://developer.android.com/ndk/guides/graphics/shader-compilers.html> for details.

After downloading the glslangValidator tool, you'll run once per shader (e.g. glslangValidator -V shader.vert and glslangValidator -V shader.frag) to create the SPIR-V byte code files (.SPV). These files should then be included in your assets folder when building your apk.

Rather than having your application read in the SPV file from disk, you may opt to store the SPV byte code in a const uint array and pass a pointer to that structure when creating the shader modules for the pipeline. The triangle sample in the Adreno SDK for Vulkan uses that approach.

You may also consider using glslang in a library form that you can call from your program at runtime to compile glsl shaders. Although this defeats the purpose of having shaders compiled to an optimal SPV format pre-prepared to speed up application start up time, there may be appropriate use cases for doing so. Refer to the [Khronos](#) site for details on glslang. Official and latest Khronos source code for glslang can be found on [github](#).

Notice that glslang applies std140 layout to your shaders by default. This could alter existing shaders that didn't use a layout because the memory offsets in the structure will now be different.

Adreno Vulkan drivers currently do not support for geometry or tessellation shader

Multiple entry points can be defined in a single SPIR-V shader-module which prevents redundant code

The KHR_vulkan_glsl extension defines changes to glsl to support these shaders in Vulkan. Some of the keep points are:

Removed:

- default uniforms (uniform variables not inside a uniform block), except for opaque types
- atomic-counters (those based on atomic_uint)
- subroutines
- shared and packed block layouts
- the already deprecated texturing functions (e.g., texture2D())
- compatibility-mode-only features
- gl_DepthRangeParameters and gl_NumSamples
- gl_VertexID and gl_InstanceID

Added:

- push-constant buffers
- shader-combining of separate textures and samplers
- descriptor sets
- specialization constants

- `gl_VertexIndex` and `gl_InstanceIndex`
- subpass inputs
- 'offset' and 'align' layout qualifiers for uniform/buffer blocks for versions that did not support them

Changed:

- precision qualifiers (mediump and lowp) will be respected for all versions, not dropped for desktop versions (default precision for desktop versions is highp for all types)
- `gl_FragColor` will no longer indicate an implicit broadcast
- arrays of opaque uniforms take only one binding number for the entire object, not one per array element
- the default origin is `origin_upper_left` instead of `origin_lower_left`

GLSLangValidator automatically enables this extension, so you don't need to explicit enable it in your shaders.

As mentioned the default origin has changed which is common overlook when debugging Vulkan shaders rendering differently (e.g. upside-down) from GL. Your vertex shader may need to invert the y coordinate. Also, the depth range is `[0,1]` rather than `[-1,1]` in GL.

Vertex data now needs to explicitly set using layout specifies with the "in" keyword, (e.g.)

```
layout (location = 0) in vec4 a_vPosition;
```

Vertex varyings use a similar layout, (e.g.):

```
layout (location = 0) out vec2 v_TexCoord;
```

`gl_Position` needs to be wrapped in a structure, (e.g.):

```
out gl_PerVertex
{
    vec4 gl_Position;
};
```

All uniform data should be stored in UBOs and referenced with explicit set and binding numbers that match up with the corresponding pipeline used, (e.g.):

```
layout (std140, set = 0, binding = 0) uniform UniformBufferObject
{
    mat4 u_MVP;
    mat4 u_matModel;
    mat4 u_matNormal;
    vec4 u_LightPos;
    vec4 u_EyePos;
} clothVertexUniforms;
```

In a fragment shader specify the output color with a layout command instead of just assigning `gl_FragColor`, (e.g.):

```
layout (location = 0) out vec4 outColor;
```

One item to pay attention to is the `minUniformBufferOffsetAlignment` value in the physical device limits returned by `vkGetPhysicalDeviceProperties`. If using a single uniform buffer object (a good strategy to minimize multiple buffer creation) for multiple draws, that alignment needs to be applied between the data.

Also make sure that signed and unsigned values in UBOs (e.g. `uint32x4`, `int32x4`) match the exactly variables used in your shaders (e.g. `vec4`, `uvec4`). Type conversion is not automatic in Vulkan as with OpenGL.

4.17 Multi-buffer rendering

You may want to consider a ring-buffer rendering scheme, where you have multiple sets of Vulkan objects in flight during your rendering. As part of an overall multithreading strategy this will allow your CPU's to be busy creating command buffers for future frames, as the GPU is rendering the current frame. Matching the number of sets with the number of images in the swap chain makes sense as it is simpler to keep track of frames.

Each of these sets will contain at a minimum multiple dedicated command pools (`VkCommandPool`) from which you can allocate command buffers to fill and subsequently submit to render only for that set's frame. The set could also contain other Vulkan objects which are need for rendering (e.g. `VkFence`, `VkDescriptorSetPool`, `VkDescriptorSet`, buffer objects).

When submitting the set's rendering to the queue, you can use the set's `VkFence` object which can be waited on when that set is needed again. When you are ready to use the set for a new frame, you can simply call `vkResetCommandPool` to have command buffer resources discarded and thus return to the pool and begin allocating and recording command buffers from the pool again.

You might want to use a different uniform buffer per frame buffer so that the appropriate data is set for each frame. Another option would be to use a single uniform buffer for all frame buffers, but use offsets based on the buffer index.

5 Understanding the developer tools

5.1 Snapdragon profiler

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

Rarely do developers have access to a raw list of the graphics API commands that the application issues. This makes the optimization process tricky, since the developer is likely to know and understand what their application does in general, but may 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 and it is difficult to identify specifically which of these draw calls is responsible for rendering a broken mesh. It is helpful to have the ability to highlight the geometry that was drawn because of any specific draw call.

These are some of the situations where the Snapdragon Profiler can help, resulting in better rendering performance and shorter application development times. As shown in the following figures, the tool works in three different modes, each suiting a different purpose.

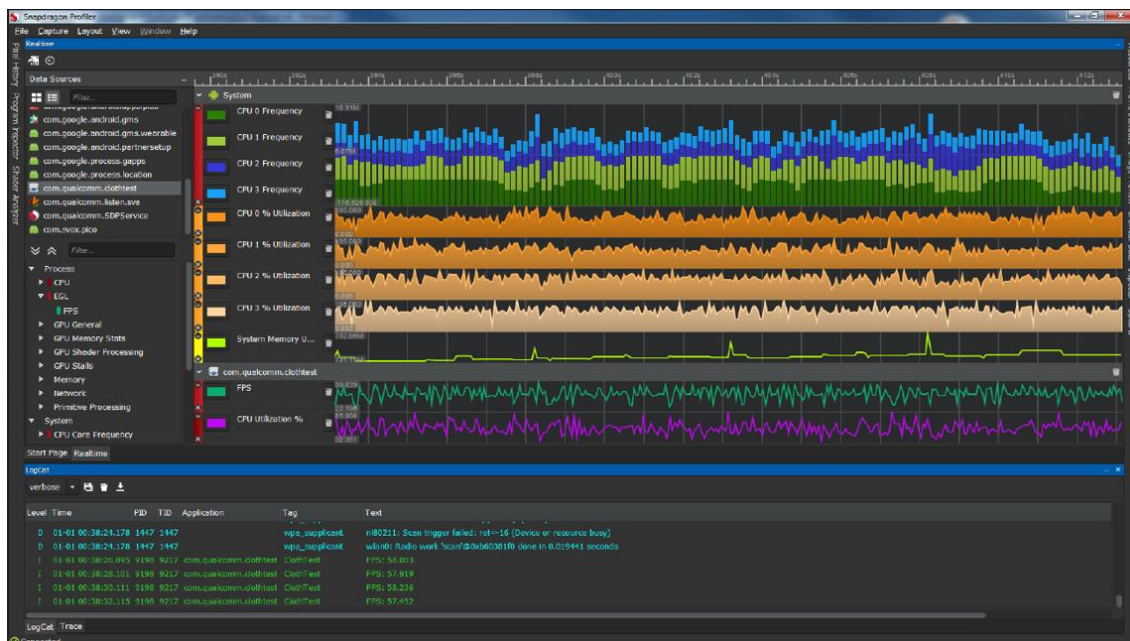


Figure 5-1 Snapdragon profiler – realtime performance

As shown in the previous [Figure 5-1](#), realtime performance visualization allows for a wide range of both system and per-process metrics and counters to be graphed in in real time. Categories of metrics include CPU, EGL, GPU, memory, network, power, primitive processing, system memory, and thermal

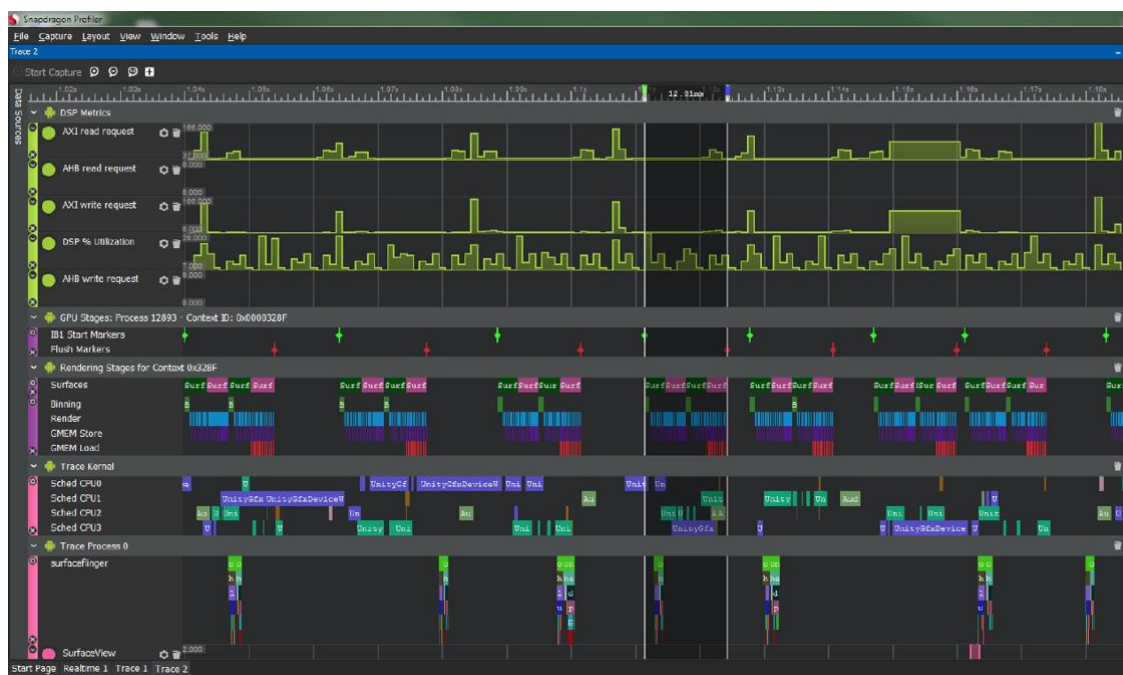


Figure 5-2 Snapdragon profiler – trace

As shown in [Figure 5-2](#), trace allows for a specific time performance capture providing a detailed visualization of the system and driver work being performed. After capture, the user can zoom in, inspect, and measure many detailed attributes. Traces can be capture 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, Synchorizaion Manager, Video, View, WebView, and Window manager.

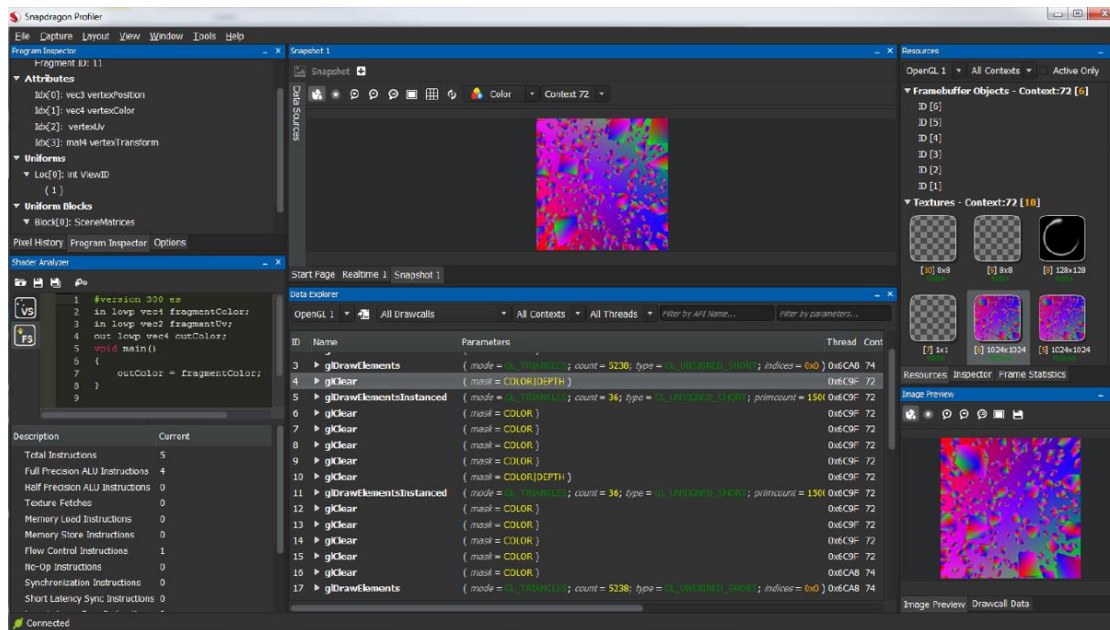


Figure 5-3 Snapdragon profiler – snapshot

Snapshot, shown in [Figure 5-3](#), 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.

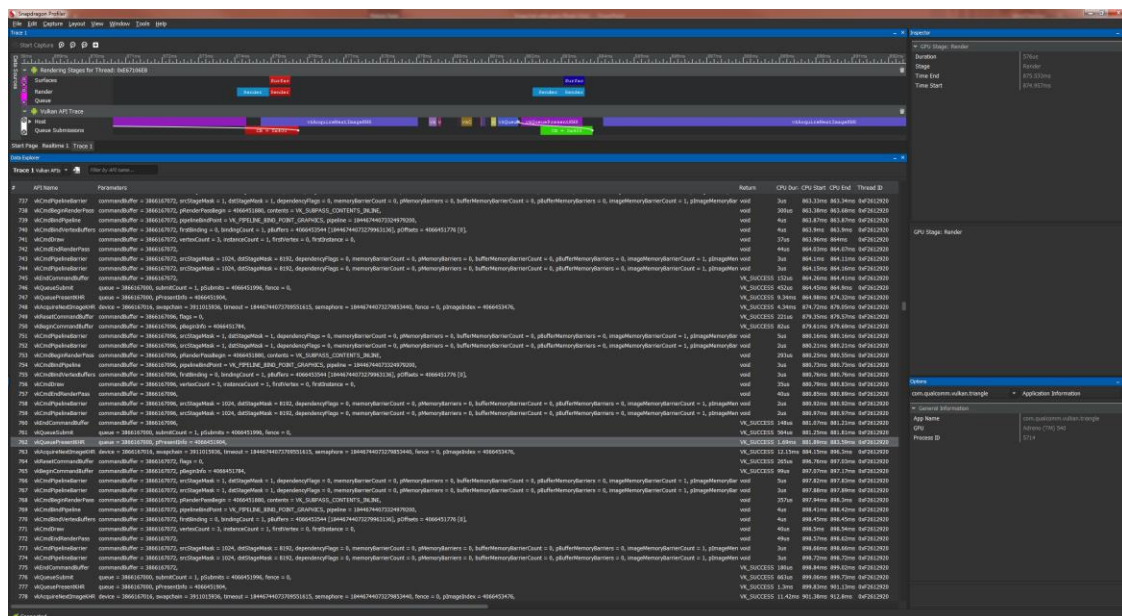


Figure 5-4 Snapdragon profiler – Vulkan trace

[Figure 5-4](#) shows that the 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 as well as download a copy from the Qualcomm Developer Network website at <https://developer.qualcomm.com/software/snapdragon-profiler>.

5.2 Adreno SDK for Vulkan

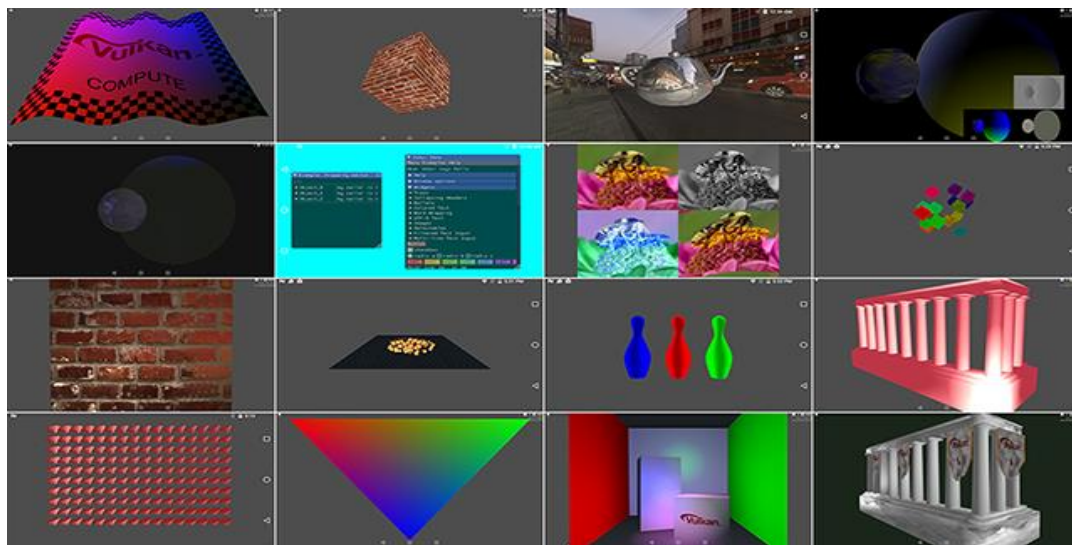


Figure 5-5 Snapdragon profiler – Adreno SDK for Vulkan

Figure 5-5 shows that the Adreno SDK for Vulkan contains several working samples covering Vulkan features. Regardless of your level of graphics experience, this SDK is the perfect introduction to the Vulkan API. You'll find instructive and commented code samples on all the basic topics:

- Initializing Vulkan
- Setting up swap chains
- Creating Vulkan devices and queues
- Synchronizing the rendering steps
- Drawing a simple triangle
- Creating image processing effects
- Rendering in multiple passes
- Compiling SPIR-V shaders
- Using compute shaders
- Making use of push constants
- Rendering user interfaces
- Threading
- Memory management

The SDK also includes easy-to-read guides in areas like writing a simple triangle app and compiling shaders. The documentation will help you get up to speed quickly on the complete Vulkan development cycle.

Of course, you need to have a few things in place first, like Android Studio, the Android SDK and the Android NDK. We've covered these requirements in a few videos on our developer website. The [videos](#) walk you through installation of the Android NDK and our Vulkan SDK, the complete contents of the SDK and step-by-step instructions on building and running your first Vulkan app on a mobile device using Android Studio. You'll also find links to useful information from Khronos, the standards body behind Vulkan.

You can download the Adreno SDK for Vulkan from the Qualcomm Developer Network website at <https://developer.qualcomm.com/software/adreno-gpu-sdk>.

6 Optimizing applications

6.1 Shader tips

This section presents various tips and tricks to help optimize an OpenGL ES application on Adreno architectures.

6.1.1 Build pipelines during initialization

The compilation and linking of shaders is a time-consuming process as is building the Vulkan pipeline. Make use of the Pipeline Cache and create derivative pipelines.

6.1.2 Use built-ins

Built-in functions are an important part of the OpenGL ES Shading Language specification and should always be used in preference 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. As a result, they will usually be faster than any other implementation.

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

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

Refer to the [GL_KHR_vulkan](#) extension for details on changes to GLSL built-ins in Vulkan.

6.1.3 Use the appropriate data type

Using the most appropriate data type in code can enable the compiler and driver to optimize code, including the pairing of shader instructions.

Using a `vec4` data type instead of `float` could prevent the compiler from performing optimizations. Small mistakes can have a significant impact on performance.

Another example is the following code should take a single instruction slot:

```
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 consume eight instruction slots. 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`.

6.1.4 Reduce type casting

It is also recommended to reduce the number of type cast operations performed. 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 one instruction slot. Changing the code as follows might reduce the instruction numbering by one:

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

6.1.5 Pack scalar constants

Packing scalar constants into vectors consisting of four channels substantially improves the hardware fetch effectiveness. In the case of an animation system, this increases the number of available bones for skinning.

Consider the following code:

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

By changing the code as follows it might take one less instruction, because the compiler can optimize the line to a more efficient instruction (mad):

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

6.1.6 Keep shader length reasonable

Excessively long shaders can be inefficient. If there is a need to include many instruction slots in a shader relative to the number of texture fetches, consider splitting the algorithm into several parts. Values that are generated by one part of the algorithm for later reuse by another part can be stored into a texture and later retrieved via a texture fetch. However, this approach could be expensive in terms of memory bandwidth. Usage of trilinear, anisotropic filtering, wide texture formats, 3D and cube map textures, texture projection, texture lookup with gradients of different Lod, or gradients across a pixel quad may also increase texture sampling time and reduce the overall benefit.

6.1.7 Sample textures in an efficient way

To avoid texture stalls, follow these rules:

- Avoid random access – Hardware operates on blocks of 2x2 fragments, so the shaders are more efficient if they access neighboring texels within a single block.
- Avoid 3D textures – Fetching data from volume textures is expensive owing to the complex filtering that needs to be performed to compute the result value.
- Limit the number of textures sampled from shaders – Usage of four samplers in a single shader is acceptable, but accessing more textures in a single shader stage could lead to performance bottlenecks.
- Compress all textures – This allows better memory usage, translating to a lower number of texture stalls in the rendering pipeline.
- Consider using mipmaps – Mipmaps 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 developer might see a benefit in using bilinear or nearest filtering over trilinear or anisotropic in certain architectures. Mipmap clamping may reduce the cost of using trilinear filtering, so the average cost might be lower in real-world cases. Adding anisotropic filtering multiplies with the degree of anisotropy; that means, 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. It could be only a few fragments in all. A rule of a thumb for real world cases is that anisotropic filtering is, on average, less than double the cost of isotropic.

Cube map texture and projected texture lookups do not incur any extra cost, while shader-specific gradients, based on the dFdx and dFdy functions, cost extra. These shader-specific gradients cannot be stored across lookups. If a texture lookup is done again with the same gradients in the same sampler, it will incur the cost again.

6.1.8 Threads in flight/dynamic branching

Branching is crucial for the performance of the shader. 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 with predicates using NULL out operations for the elements that do not take a given branch. This is true only if the data is aligned in a way that follows those conditions, which is rarely the case for fragment shaders.

There are three types of branches, listed in order from best performance to worst on Adreno GPUs:

- 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.

6.1.9 Pack shader interpolators

Shader-interpolated values or varyings 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 varyings 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.

6.1.10 Minimize usage of shader GPRs

Minimizing the usage of GPRs can be an important means of optimizing performance. Inputting simpler shaders to the compiler helps guarantee optimal results. Modifying GLSL to save even a single instruction can save a GPR sometimes. 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]);
```

6.1.11 Minimize shader instruction count

The compiler optimizes specific instructions, but it is not automatically efficient. Analyze shaders to save instructions wherever possible. Saving even a single instruction is worth the effort.

6.1.12 Avoid uber-shaders

Uber-shaders 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 has an impact on performance.

6.1.13 Avoid math on shader constants

Almost every shipped game since the advent of shaders has spent instructions performing unnecessary math on shader constants. Identify these instructions in shaders and move those calculations off to the CPU. It may be easier to identify math on shader constants in the postcompiled microcode.

6.1.14 Avoid discarding pixels in the fragment shader

Some developers believe that manually discarding, also known as *killing*, pixels in the fragment shader boosts performance. The rules are not that simple for two reasons:

- If some pixels in a thread are killed and others are not, the shader still executes.
- It depends on how the shader compiler generates microcode.

In theory, if all pixels in a thread are killed, the GPU will stop processing that thread as soon as possible. In practice, discard operations can disable hardware optimizations.

If a shader cannot avoid discard operations, attempt to render geometry, which depends on them after opaque draw calls.

6.1.15 Avoid modifying depth in fragment shaders

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

6.1.16 Avoid texture fetches in vertex shaders

Adreno is based on a unified shader architecture, which means the vertex processing performance is similar to the fragment processing performance. However, for optimal performance, it is important to ensure that texture fetches in vertex shaders are localized and always operate on compressed texture data.

6.1.17 Break up draw calls

If a shader is heavy on GPRs and/or heavy on texture cache demands, increased performance can result from breaking up the draw calls into multiple passes. Whether the results will be positive is hard to predict, so using real-world measurements both ways is the best method to decide. Ideally, a two-pass draw call would combine its results with simple alpha blending, which is not heavy on Adreno GPUs because of the graphics memory (GMEM).

Some developers may consider using a true deferred rendering algorithm, but that approach has many drawbacks, e.g., the GMEM must be resolved for a previous pass to be used as input to a successive pass. Because resolves are not free, it is a performance cost that must be recouped elsewhere in the algorithm.

Ideally the use of Vulkan's RenderPass will help minimize GMEM resolves, so restructuring the rendering algorithm to use as many subpasses as possible will be the optimal approach

6.1.18 Use medium precision where possible

On Adreno, 16-bit operations tend to be faster and more power-efficient than 32-bit operations. QTI recommends setting the default precision to mediump and promoting only those values that require higher precision.

However, there may be situations when highp must be used for certain varyings, e.g., texture coordinates, in shaders. These situations can be handled with a conditional statement and a preprocessor-based macro definition, as follows:

```
precision mediump float;
#ifdef GL_FRAGMENT_PRECISION_HIGH
    #define NEED_HIGHP highp
#else
    #define NEED_HIGHP mediump
#endif
varying                vec2 vSmallTexCoord;
varying NEED_HIGHP vec2 vLargeTexCoord;
```

6.1.19 Favor vertex shader calculations over fragment shader calculations

Typically, vertex count is significantly less than fragment count. It is possible to reduce GPU workload by moving calculations from the fragment shader to the vertex shader. This helps to eliminate redundant computations.

6.1.20 Measure, test, and verify results

Finding bottlenecks is necessary for optimization, whether the application is vertex bound, fragment bound, or texture fetch bound. Measure performance before attempting to make the code faster. Use tools to take these measurements, e.g., the Snapdragon Profiler or even software timers.

Do not assume something runs faster based solely on intuition. When code is modified to perform better, it can disable compiler/hardware optimizations that are more beneficial. Always measure timing before and after changes to assess the impact of modifications performed for the sake of optimization.

6.1.21 Prefer uniform buffers over shader storage buffers

As long as read-only access is sufficient for needs and the amount of space Uniform Buffers offer is enough, always prefer them over Shader Storage Buffers. They are likely to perform better under the Adreno architecture. This is true if the Uniform Buffer Objects are statically indexed in GLSL and are small enough that the driver or compiler can map them into the same hardware constant RAM that is used for the default uniform block uniforms.

Also prefer uniform buffers over push constants on Adreno hardware for performance reasons.

6.2 Optimize vertex processing

This section describes tips and tricks that can help optimize the way Vulkan applications organize vertex data, so that the rendering process can run efficiently on Adreno architectures.

6.2.1 Use interleaved, compressed vertices

For vertex fetch performance, interleaved vertex attributes (“xyz uv | xyz uv | ...”, rather than “xyz | xyz | .. | uv | uv | ..”), work the most efficiently. The throughput is better with interleaved vertices and compressing vertices improves it further. Deferred rendering gives these optimizations an advantage.

For binning pass optimization, consider one array with vertex attributes and other attributes needed to compute position, and another interleaved array with other attributes.

Currently there is no support for half floats in Vulkan shaders.

6.2.2 Consider geometry instancing

Geometry instancing is the practice of rendering 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.

Although vertex data is duplicated across all instanced meshes, each instance could have other differentiating parameters, e.g., color, transforms, or lighting, changed to reduce the appearance of repetition.

As shown in [Figure 6-1](#), 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.



Figure 6-1 Geometry instancing for drawing barrels

Geometry instancing offers a significant savings in memory usage. 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.

In Vulkan, `vkCmdDraw`, `vkCmdDrawIndexed`, `vkDrawIndirectCommand`, and `vkDrawIndexedIndirectCommand` support instanced rendering,

6.3 Texture compression strategies

Compressing textures can significantly improve the performance and load time for graphics applications, since it reduces texture memory and bus bandwidth use. 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.

Significant texture compression formats supported by Adreno GPUs are:

- ATC – Proprietary Adreno texture compression format supporting RGB and RGBA component layouts
- ETC – Standard OpenGL ES 2.0 texture compression format supporting the RGB component layout only
- ETC2 – Standard OpenGL ES 3.0 texture compression format supporting R, RG, RGB, and RGBA component layouts, as well as sRGB texture data
- ASTC – Texture compression format offering remarkably low levels of image degradation given the compression ratios achieved

Tip

QTI recommends the following strategy for selecting a texture compression format:

1. Use ASTC compression if it is available.
2. Otherwise, use ETC2 compression if available.
3. Otherwise, select the compression format as follows:
 - a. ATC if using alpha
 - b. ETC if not using alpha

ASTC is a newer format and may not be supported or optimized on all content creation pipelines. Also, the sRGB formats for ASTC are more efficiently handled on Adreno hardware than the RGBA formats.

The following are popular methods for converting textures to Adreno hardware:

- On-device conversion
- Prepackaging
- Downloading

On-device conversion involves a time-consuming one-time conversion of texture assets that occurs when the game starts up. Prepackaging the correct textures results in 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 either case, the first step is to create the compressed textures.

Texture data can be compressed to any of these texture compression formats using the Adreno Texture Compression and Visualization Tool or Adreno Texture Converter Tool, both included in the Adreno SDK for OpenGL. There is a Compressed Texture tutorial in the SDK, which presents how to use compressed textures in OpenGL ES applications.

The effectiveness of these compression formats is throughout [Figure 6-2](#) through [Figure 6-15](#), consisting of one diffuse and one object-space normal RGB texture, compressed using the 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.

6.3.1 Diffuse texture tests

The diffuse texture used for this test is shown in [Figure 6-2](#).



Figure 6-2 Diffuse texture used for the test

6.3.1.1 ATC compression

Figure 6-3 and Figure 6-4 show the use of the GL_ATC_RGB_AMD internal format.



Figure 6-3 ATC compression result for GL_ATC_RGB_AMD



Figure 6-4 Noncompressed vs. ATC-compressed versions for GL_ATC_RGB_AMD

6.3.1.2 ETC1 compression

Figure 6-5 and Figure 6-6 show the use of the GL_ETC1_RGB8_OES internal format.



Figure 6-5 ETC1 compression result for GL_ETC1_RGB8_OES



Figure 6-6 Noncompressed vs. ETC1-compressed versions for GL_ETC1_RGB8_OES

6.3.1.3 ETC2 compression

Figure 6-7 and Figure 6-8 show the use of the GL_COMPRESSED_RGBA8_ETC2 internal format.



Figure 6-7 ETC2 compression result for GL_COMPRESSED_RGBA8_ETC2

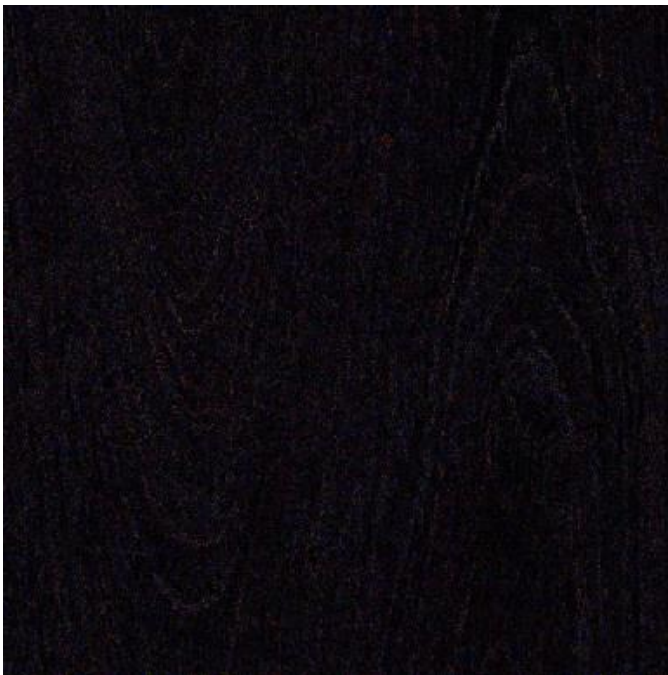


Figure 6-8 Noncompressed vs. ETC2-compressed versions for GL_COMPRESSED_RGBA8_ETC2

6.3.2 Normal texture tests

Texture data can be compressed to any of these texture compression formats using the Adreno Texture Tool (QCompress.exe) included in the Adreno [SDK for Vulkan](#). There is a Textures tutorial in the SDK, which demonstrates how to use compressed textures in Vulkan applications.

The normal texture used for this test is shown in [Figure 6-9](#).

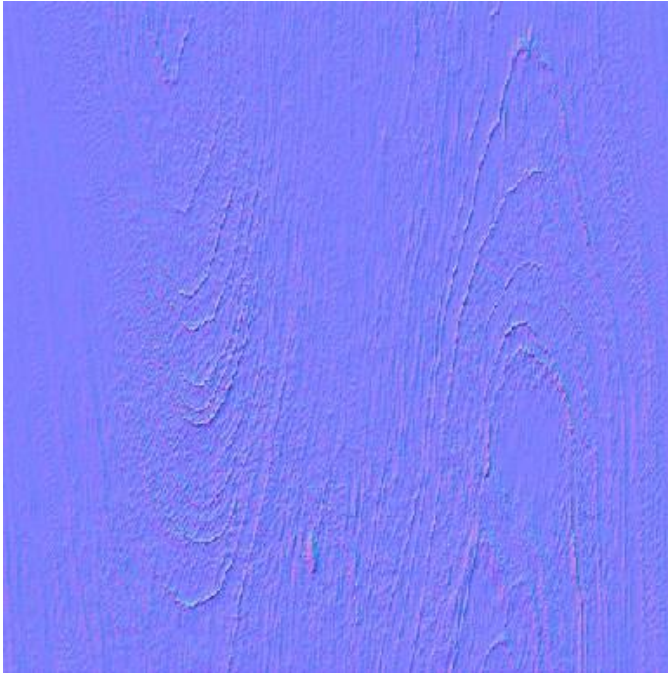


Figure 6-9 Normal texture used for the test

6.3.2.1 ATC compression

Figure 6-10 and Figure 6-11 show the use of the GL_ATC_RGB_AMD internal format.

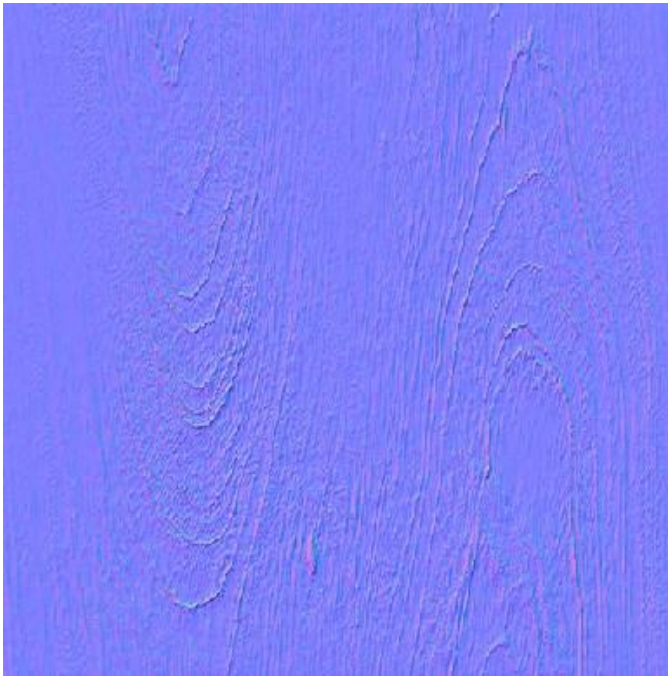


Figure 6-10 ATC compression result for GL_ATC_RGB_AMD

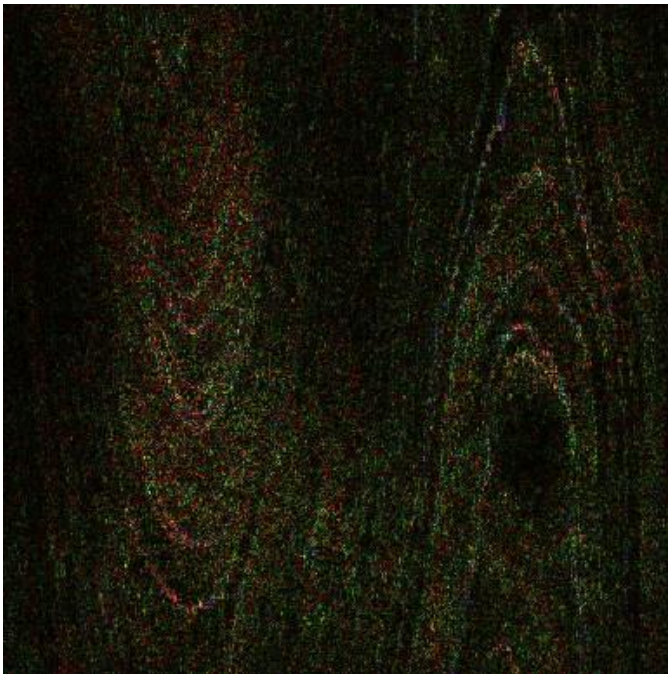


Figure 6-11 Noncompressed vs. ATC-compressed versions for GL_ATC_RGB_AMD

6.3.2.2 ETC1 compression

Figure 6-12 and Figure 6-13 show the use of the GL_ETC1_RGB8_OES internal format.

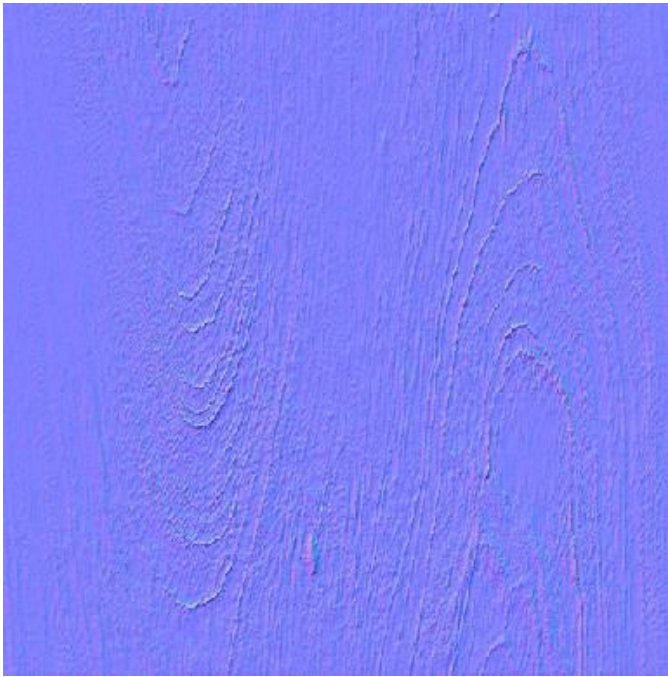


Figure 6-12 ETC1 compression result for GL_ETC1_RGB8_OES

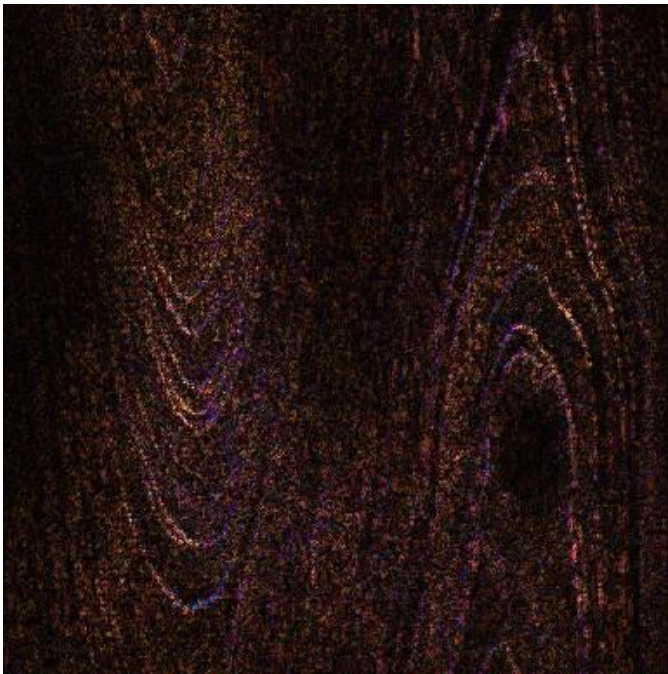


Figure 6-13 Noncompressed vs. ETC1-compressed versions for GL_ETC1_RGB8_OES

6.3.2.3 ETC2 compression

Figure 6-14 and Figure 6-15 show the use of the GL_COMPRESSED_RGBA8_ETC2 internal format.

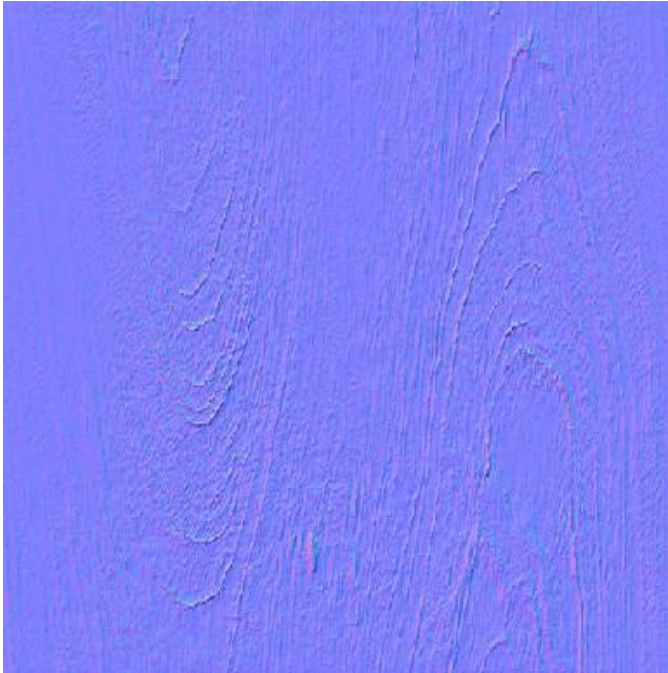


Figure 6-14 ETC2 compression result for GL_COMPRESSED_RGBA8_ETC2

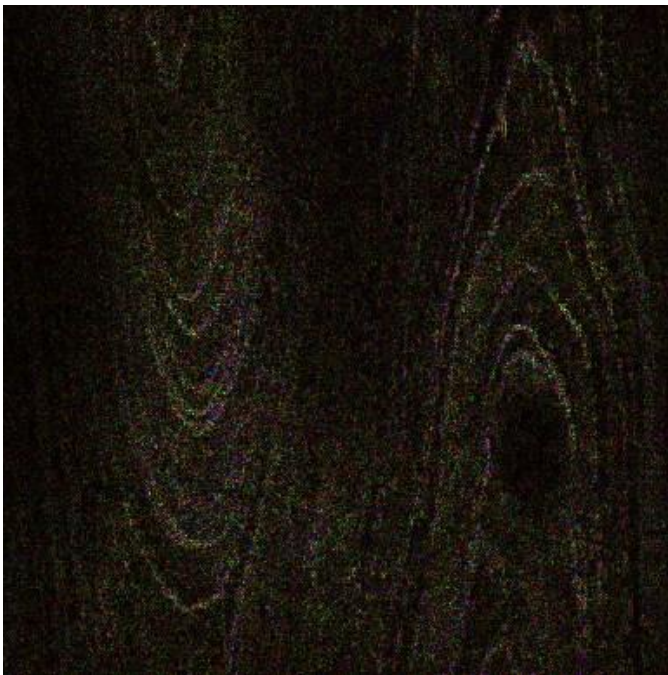


Figure 6-15 Noncompressed vs. ETC2-compressed versions for GL_COMPRESSED_RGBA8_ETC2

6.4 Bandwidth optimization

OpenGL ES applications can suffer from the bottleneck of being memory-bandwidth limited. This is a manifestation of the physical limitation of how much data the GPU can access within a given timeframe. 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 causes significant performance drops. These cache misses are often caused when applications draw or generate many primitives, or when shaders need to access many locations within textures.

There are two measures to take to minimize the problem of 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.

OpenGL ES provides 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 mipmap size. Many of the compressed texture formats supported by OpenGL ES 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 it can seem to be inefficient from the point of view of data compression theory, it does have 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 with 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 use packed vertex data formats. These formats are based on the premise that many vertex data sets will not suffer greatly from a reduction in the precision of their components. It is strongly recommended to use packed vertex data formats wherever possible.

For certain assets whose range span is 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 format by normalizing the 10-bit unsigned integer data range of <0, 1024> onto the floating-point range <-1, 1>.

Table 6-1 lists the vertex data formats that can be used in OpenGL ES 2.0, ES 3.0, and ES 3.1 on Adreno hardware. When an OpenGL ES extension must be used to access a given format, it is necessary to use a different identifier for the format—this is also shown in the table.

Table 6-1 Vertex data format support in Adreno architecture

Format name	ES 2	ES 3	ES 3.1
GL_BYTE	Supported	Supported	Supported
GL_FIXED	Supported	Supported	Supported
GL_FLOAT	Supported	Supported	Supported
GL_HALF_FLOAT	Supported ^a	Supported	Supported
GL_INT	Not supported	Supported	Supported
GL_INT_2_10_10_10_REV	Supported ^b	Supported	Supported
GL_SHORT	Supported	Supported	Supported
GL_UNSIGNED_BYTE	Supported	Supported	Supported
GL_UNSIGNED_INT	Not supported	Supported	Supported
GL_UNSIGNED_INT_2_10_10_10_REV	Supported ^c	Supported	Supported
GL_UNSIGNED_SHORT	Supported	Supported	Supported

^aUse GL_OES_vertex_half_float

^bUse GL_OES_vertex_type_10_10_10_2

^cUse GL_OES_vertex_type_10_10_10_2

The third method is to always use indexed draw calls. 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.

6.5 Depth range optimization

A common artifact in games is z-fighting, where two or more primitives have similar values in the depth buffer. The artifact becomes apparent when the camera view changes slightly and fragments from different primitives fight to win the depth test and become visible.

There are several ways to eliminate z-fighting:

- Modify the scene geometry to provide more distance between near planar surfaces (preferred).
- Choose an EGL configuration with a larger depth buffer, e.g., 24 bits vs 16 bits, though performance may be impacted by using a larger buffer.
- Tighten camera depth range (near and far clipping planes) allowing for more precision in the z direction, though this will not help coplanar primitives; consider rendering the scene with multiple ranges, one for near objects and one for far objects.

6.6 Other optimizations

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 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.

Also, the busier the hardware is, the more heat it will generate. If, over an extended period of time, the GPU is not left with any idle time between frames, the device may become hot and uncomfortable to hold. If the temperature exceeds a certain safety threshold, the device may even automatically reduce the GPU clock frequency to prevent it from overheating. This will further degrade the user experience.

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 OpenGL ES provides a fix for this issue.

OpenGL ES 3.0 introduced support for frame buffer blit operations. 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. Use upscaling to scale up the reduced-size render target to match the full native display size. The best strategy depends on how intensive the computations are that the GPU must perform for an application. The upscaling can be done either at the end of the rendering process or at some point in the rendering pipeline; e.g., one approach might be to render the geometry at 1:1 resolution, but apply postprocessing effects using render targets of a slightly lower 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.

Alternatively, control scaling through the Android API:

- 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 `ANativeWindow_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.

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 be affected by the hardware constraints discussed.

A Device limits (Adreno 530)

uint32_t	maxImageDimension1D	16384
uint32_t	maxImageDimension2D	16384
uint32_t	maxImageDimension3D	2084
uint32_t	maxImageDimensionCube	16384
uint32_t	maxImageArrayLayers	2084
uint32_t	maxTexelBufferElements	65536
uint32_t	maxUniformBufferRange	65536
uint32_t	maxStorageBufferRange	2147483647
uint32_t	maxPushConstantsSize	128
uint32_t	maxMemoryAllocationCount	4096
uint32_t	maxSamplerAllocationCount	4000
VkDeviceSize	bufferImageGranularity	1
VkDeviceSize	sparseAddressSpaceSize	0
uint32_t	maxBoundDescriptorSets	4
uint32_t	maxPerStageDescriptorSamplers	16
uint32_t	maxPerStageDescriptorUniformBuffers	14
uint32_t	maxPerStageDescriptorStorageBuffers	4
uint32_t	maxPerStageDescriptorSampledImages	128
uint32_t	maxPerStageDescriptorStorageImages	4
uint32_t	maxPerStageDescriptorInputAttachments	8
uint32_t	maxPerStageResources	158
uint32_t	maxDescriptorSetSamplers	96
uint32_t	maxDescriptorSetUniformBuffers	84
uint32_t	maxDescriptorSetUniformBuffersDynamic	8
uint32_t	maxDescriptorSetStorageBuffers	24
uint32_t	maxDescriptorSetStorageBuffersDynamic	4
uint32_t	maxDescriptorSetSampledImages	768
uint32_t	maxDescriptorSetStorageImages	24
uint32_t	maxDescriptorSetInputAttachments	8
uint32_t	maxVertexInputAttributes	32
uint32_t	maxVertexInputBindings	32
uint32_t	maxVertexInputAttributeOffset	4096
uint32_t	maxVertexInputBindingStride	2048
uint32_t	maxVertexOutputComponents	128
uint32_t	maxTessellationGenerationLevel	0
uint32_t	maxTessellationPatchSize	0
uint32_t	maxTessellationControlPerVertexInputComponents	0
uint32_t	maxTessellationControlPerVertexOutputComponents	0
uint32_t	maxTessellationControlPerPatchOutputComponents	0

uint32_t	maxTessellationControlTotalOutputComponent	0
uint32_t	maxTessellationEvaluationInputComponents	0
uint32_t	maxTessellationEvaluationOutputComponents	0
uint32_t	maxGeometryShaderInvocations	0
uint32_t	maxGeometryInputComponents	0
uint32_t	maxGeometryOutputComponents	0
uint32_t	maxGeometryOutputVertices	0
uint32_t	maxGeometryTotalOutputComponents	0
uint32_t	maxFragmentInputComponents	128
uint32_t	maxFragmentOutputAttachments	8
uint32_t	maxFragmentDualSrcAttachments	0
uint32_t	maxFragmentCombinedOutputResources	71
uint32_t	maxComputeSharedMemorySize	32768
uint32_t	maxComputeWorkGroupCount	[3] [65535] [65535] [65535]
uint32_t	maxComputeWorkGroupInvocations	1024
uint32_t	maxComputeWorkGroupSize	[3] [1024] [1024] [64]
uint32_t	subPixelPrecisionBits	4
uint32_t	subTexelPrecisionBits	8
uint32_t	mipmapPrecisionBits	8
uint32_t	maxDrawIndexedIndexValue	4294967295
uint32_t	maxDrawIndirectCount	1
float	maxSamplerLodBias	15.9960938
float	maxSamplerAnisotropy	16
uint32_t	maxViewports	1
uint32_t	maxViewportDimensions	[2] [16384] [16384]
float	viewportBoundsRange	[2] [-32768] [32768]
uint32_t	viewportSubPixelBits	0
size_t	minMemoryMapAlignment	64
VkDeviceSize	minTexelBufferOffsetAlignment	64
VkDeviceSize	minUniformBufferOffsetAlignment	64
VkDeviceSize	minStorageBufferOffsetAlignment	64
int32_t	minTexelOffset	8
uint32_t	maxTexelOffset	7
int32_t	minTexelGather	Offset-32
uint32_t	maxTexelGather	Offset
float	minInterpolationOffset	0.5
float	maxInterpolationOffset	0.4375
uint32_t	subPixelInterpolationOffsetBits	4
uint32_t	maxFramebufferWidth	16384
uint32_t	maxFramebufferHeight	16384
uint32_t	maxFramebufferLayers	2048
VkSampleCount	FlagsframebufferColorSampleCounts	7
VkSampleCount	FlagsframebufferDepthSampleCounts	7
VkSampleCount	FlagsframebufferStencilSampleCounts	7
VkSampleCount	FlagsframebufferNoAttachmentsSampleCounts	7
uint32_t	maxColorAttachments	8

VkSampleCount	FlagssampledImageColorSampleCounts	7
VkSampleCount	FlagssampledImageIntegerSampleCounts	7
VkSampleCount	FlagssampledImageDepthSampleCounts	7
VkSampleCount	FlagssampledImageStencilSampleCounts	7
VkSampleCount	FlagsstorageImageSampleCounts	1
uint32_t	maxSampleMaskWords	1
VkBool32	timestamp ComputeAndGraphics	1
Floattimestamp	Period	52.0833321
uint32_t	maxClipDistances	8
uint32_t	maxCullDistances	8
uint32_t	maxCombinedClipAndCullDistances	8
uint32_t	discreteQueuePriorities	3
floatpoint	SizeRange	[2][1][1]
floatline	WidthRange	[2][1][1]
floatpoint	SizeGranularity	0
floatline	WidthGranularity	0
VkBool32	strictLines	1
VkBool32	standardSampleLocations	1
VkDevice	SizeoptimalBufferCopyOffsetAlignment	64
VkDevice	SizeoptimalBufferCopyRowPitchAlignment	64
VkDevice	SizeonCoherentAtomSize	1

B Sparse properties (Adreno 530)

VkBool32	residencyStandard2DblockShape	0
VkBool32	residencyStandard2DmultisampleBlockShape	0
VkBool32	residencyStandard3DblockShape	0
VkBool32	residencyAlignedMipSize	0
VkBool32	residencyNonResidentStrict	0

C Features (Adreno 530)

VkBool32	robustBufferAccess	1
VkBool32	fullDrawIndexUint32	1
VkBool32	imageCubeArray	1
VkBool32	independentBlend	1
VkBool32	geometryShader	0
VkBool32	tessellationShader	0
VkBool32	sampleRateShading	1
VkBool32	dualSrcBlend	0
VkBool32	logicOp	0
VkBool32	multiDrawIndirect	0
VkBool32	drawIndirectFirstInstance	0
VkBool32	depthClamp	0
VkBool32	depthBiasClamp	0
VkBool32	fillModeNonSolid	0
VkBool32	depthBounds	0
VkBool32	wideLines	0
VkBool32	largePoints	0
VkBool32	alphaToOne	0
VkBool32	multiViewport	0
VkBool32	samplerAnisotropy	1
VkBool32	textureCompressionETC2	1
VkBool32	textureCompressionASTC_LDR	1
VkBool32	textureCompressionBC	0
VkBool32	occlusionQueryPrecise	0
VkBool32	pipelineStatisticsQuery	0
VkBool32	vertexPipelineStoresAndAtomics	0
VkBool32	fragmentStoresAndAtomics	1
VkBool32	shaderTessellationAndGeometryPointSize	0
VkBool32	shaderImageGatherExtended	1
VkBool32	shaderStorageImageExtendedFormats	0
VkBool32	shaderStorageImageMultisample	0
VkBool32	shaderStorageImageReadWithoutFormat	0
VkBool32	shaderStorageImageWriteWithoutFormat	0
VkBool32	shaderUniformBufferArrayDynamicIndexing	1
VkBool32	shaderSampledImageArrayDynamicIndexing	1
VkBool32	shaderStorageBufferArrayDynamicIndexing	0
VkBool32	shaderStorageImageArrayDynamicIndexing	0
VkBool32	shaderClipDistance	1
VkBool32	shaderCullDistance	1
VkBool32	shaderFloat64	0

VkBool32	shaderInt64	0
VkBool32	shaderInt16	0
VkBool32	shaderResourceResidency	0
VkBool32	shaderResourceMinLod	0
VkBool32	sparseBinding	0
VkBool32	sparseResidencyBuffer	0
VkBool32	sparseResidencyImage2D	0
VkBool32	sparseResidencyImage3D	0
VkBool32	sparseResidency2Samples	0
VkBool32	sparseResidency4Samples	0
VkBool32	sparseResidency8Samples	0
VkBool32	sparseResidency16Samples	0
VkBool32	sparseResidencyAliased	0
VkBool32	variableMultisampleRate	0
VkBool32	inheritedQueries	1

D References

Documents and Links	
Qualcomm Technologies, Inc.	
<i>Qualcomm Adreno GPU Vulkan Overview</i>	80-NB295-10
<i>Qualcomm Adreno OpenGL ES</i>	80-NU141-1
<i>MSM8998 Linux Android Graphics Overview</i>	80-P2484-12
<i>Adreno SDK download</i>	https://developer.qualcomm.com/download/
<i>Vulkan tutorial videos</i>	https://developer.qualcomm.com/software/adreno-gpu-sdk/tutorial-videos
<i>Walkthrough of sample applications</i>	https://developer.qualcomm.com/software/adreno-gpu-sdk/tutorial-videos
Standards	
<i>Vulkan 1.0 Specification</i>	https://www.khronos.org/vulkan/
Resources	
<i>Android NDK Toolset</i>	https://developer.android.com/ndk/downloads/index.html
<i>Android Studio</i>	https://developer.android.com/studio/index.html
<i>Android Surface Texture API</i>	https://developer.android.com/reference/android/graphics/SurfaceTexture.html
<i>Android Windows 64-bit NDK</i>	http://developer.android.com/tools/sdk/ndk/index.html
<i>Library of Graphics Rendering Algorithms – Sasha Willems</i>	https://github.com/SaschaWillems/Vulkan
<i>Library of Graphics Rendering Algorithms – Norbery McNopper</i>	https://github.com/McNopper/Vulkan
<i>NDK validation errors source code</i>	https://vulkan.lunarg.com/
<i>OpenGL Extensions</i>	http://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image.txt
<i>Oracle – Windows x64 JDK download</i>	http://www.oracle.com/technetwork/java/javase/downloads/
<i>SPIR-V byte code authoring tool</i>	https://www.lunarg.com/vulkan-sdk/
<i>SPIR-V graphics shader compiler</i>	https://developer.android.com/ndk/guides/graphics/shader-compilers.html

Documents and Links	
<i>Swapchain image size mismatch with surface size (Googleblog)</i>	https://android-developers.googleblog.com/2013/09/using-hardware-scaler-for-performance.html
<i>Vulkan API</i>	https://www.khronos.org/api/vulkan
<i>Vulkan changes to GLSL built-ins</i>	https://www.khronos.org/registry/vulkan/specs/misc/GL_KHR_vulkan_gsl.txt

E Glossary

Term	Definition
2D array texture	A two-dimensional mipmapped texture type, where each mipmap level can hold multiple 2D images
2D texture	A two-dimensional mipmapped texture type, where each mipmap level can hold a single 2D image
3D texture	A three-dimensional mipmapped texture type, where each mipmap level can hold a set of texture slices defining 3D image data
Alpha blending	The process of blending incoming fragment values with the data already stored in the color buffer, controlled by the alpha component value of the source or the destination, or by both values
Alpha test	A rendering pipeline stage specific to OpenGL ES 1.1, which discards fragments, depending on the outcome of a comparison between the incoming fragment value and a constant reference value; this behavior can easily be reproduced with fragment shaders if using OpenGL ES 2.0 or later
Atomic counter	A special OpenGL ES Shader Language integer variable type; any read/write operations performed on the atomic counter are serialized
Atomic function	A thread safe function, guaranteeing that the value the function operates on will be correctly read from/written to, even when more than one thread is using the function at the same time
Back buffer	Off-screen color buffer used for rendering purposes
Back face culling	An optimization performed by OpenGL ES; suppresses fragment shader invocations for primitives that are not facing the camera
Binding point	An abstract concept with the usage where an OpenGL ES object can be attached to a specific binding point, allowing it to be subsequently accessed through an OpenGL ES function operating on that binding point, or to be used in the rendering process
Blend equation	Rendering context state, which specifies which equation (from a small set defined in the OpenGL ES specification) should be used for the blending process
Blend mode	A rendering mode, which when toggled on, causes incoming fragments to be blended with the existing color data, instead of simply overriding it
Blit operation	Memory copy operation that takes color/depth/stencil attachments of a read frame buffer and transfers their contents to the corresponding attachments of a draw frame buffer; input data can be optionally resized, prior to being transferred
Buffer object	A data container that can be managed by the OpenGL ES API; usually hosted in VRAM
Buffer subrange mapping	The process of mapping a region of storage of a buffer object into process space for read and/or write operations
Buffer swap operation	The act of flipping the back buffer of the default frame buffer with the front buffer