# Qualcomm® Adreno™ SDK for Vulkan™- Developer Guide

## Introduction to Vulkan

Vulkan is the latest graphics and compute API from Khronos, and while the basic rendering algorithms are the same as OpenGL ES it is a major change about how to express modern GPU hardware to developers. It addresses the growing demand for low overhead APIs that provide developers with explicit control of the graphics and compute power on GPUs like the Adreno GPU.

Vulkan supports many important features, such as:

- Explicit control over GPU operation, with minimized driver overhead for improved performance;
- Multi-threading-friendly architecture to increase overall system performance;
- Optimal API design that can be used in a wide variety of devices including mobile, desktop, consoles, and embedded platforms;
- Use of Khronos' new SPIR-V™ intermediate representation for shading language flexibility and more predictable implementation behavior;
- Extensible layered architecture that enables innovative tools without impacting production performance while validating, debugging, and profiling;
- Simple drivers for low-overhead efficiency and cross vendor portability.

Further details about the Vulkan API can be found at the official portal on the Khronos Group web site at https://www.khronos.org/vulkan/ .

## Comparison of Vulkan and OpenGL ES

If you've worked in OpenGL ES, you've seen the CPU overhead you can incur when programming the GPU. Your code calls APIs, then a graphics driver performs a significant amount of work on the CPU before the GPU gets commands. This can cause unexpected lags and high CPU usage, especially when many draw calls are encountered.

Vulkan makes that driver translation layer much thinner and gives developers more control of the execution of the code on the GPU. This also means the developer needs to manage resources and tasks that were previously handled automatically. This includes:

- Memory allocation;
- Thread and other resource management;
- Command buffer generation;
- Work submission to device queues;
- Device synchronization;
- Shader compilation to SPIR-V.

With Vulkan, graphics commands such as binding buffers, setting shader inputs and draw commands are built into command buffers. These command buffers can be built in parallel and Vulkan allows for custom memory allocators to allow for true multithreaded buffer generation. These buffers can be stored and re-submitted to device queues each frame, removing per-frame overhead, and only need to be regenerated when the developer explicitly requires it.

Additionally, Vulkan introduces multi-pass rendering. This allows for multiple sub passes to be defined in the API, allowing for optimization of any local storage available on the GPU. This is especially important on tiled GPUs such as Adreno, and allows for large increases in performance. An example of where this will give significant impact is in traditional deferred rendering, in which a pass over geometry produces G-buffers which are then resolved and used as input to further screen-space passes. With multi-pass, these passes can be implemented and dispatched in a per-tile fashion, allowing G-buffer output to remain local for the subsequent passes that produce a final scene image.

# Development Environment

## Android Studio

Android Studio can be used to develop NDK applications for Android utilizing the Vulkan API and driver. All samples included in the Development Kit are set up as Android Studio projects. To open a sample, simply point Android Studio at the relevant sample root directory.

## Gradle Scripts

The samples include Gradle build scripts for building Android apps. The `android.ndk` section contains the important definitions for linking with the Vulkan library, and ensuring the Vulkan API header files are available to the source tree.

```
// Add Vulkan header file location to cppFlags
cppFlags.add("-I${file("../../../include")}".toString())
// Add Vulkan library file location, and add “vulkan” to ldLibs
ldFlags.add("-L${file("../../../lib/")}".toString())
ldLibs.addAll(["log", "android", "vulkan"])
```

There is additional Gradle task for building SPIR-V binaries from source shaders. If you find that the spv shader files are not being created (in the sample’s asset folder), be sure to delete both the Gradle cache for the sample (e.g. deferred/.gradle/2.10/taskArtifacts/cache*) and the Gradle user cache (e.g. C:/users/username/.gradle/caches). More information on this is available in the “Compiling Shaders to SPIR-V” document.

## Shader compilation

Traditional graphics and compute shaders must be compiled to SPIR-V before use in Vulkan applications.

The shaders can be compiled into SPIR-V binary blobs using glslang, provided by the Khronos Group on github in source form. It can be found at https://github.com/KhronosGroup/glslang .

More detail about this off-line process is available in the “Compiling Shaders to SPIR-V” document.

# SDK Layout

The SDK consists of common framework files, headers, libraries, documentation and samples.

## Samples

Samples are available in the `samples` directory.

Documentation is available for two tutorial samples, `tut_cornelllights` and `tut_monument`. They are walkthrough tutorials of how the different techniques used are implemented via the Vulkan API.

The Sample Guide describes all tutorials available in the SDK, with short descriptions and screenshots of each.

## Documentation

A Getting Started guide is provided, in which rendering your first triangle using Vulkan is explained in detail. It also explains how to open and build the samples in Android Studio.

A further document is provided explaining how to compile shaders to SPIR-V using the Khronos reference frontend, glslang.

# Vulkan Concepts

## Pipelines

Vulkan Pipelines contain all state required for rendering. They contain;

- Vertex Input state such as vertex layouts for attributes.
- Assembly state (defining topology, such as `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`).
- Rasterization state (cull modes, polygon mode, winding)
- Color Blending state (blend enables, color masks)
- Viewport and Scissor states
- DepthStencil state (depth testing definitions)
- MultiSample state
- Shader stage state
- Layouts of shader stage inputs and bindings.

The pipelines can then be bound in a command buffer, and used to issue draw commands.

### Render Passes

Render passes define attachments to the framebuffer objects that are then used in the graphics pipeline. They also can define sub-passes, to enable multi-pass rendering. This is one of the new features that Vulkan brings, allowing for optimal use of tiled rendering architectures.

### Descriptor Sets

Descriptor Sets attach the bindings described in pipeline layouts to actual objects, such as uniform buffers and samplers. Many descriptor sets can be created, such that they point to different resources in the system. These then bind objects in shaders to Vulkan resources.

For example, there may be a descriptor set for defining the various matrices passed to a vertex shader as a uniform. There could be a descriptor set for the current camera view, but also another for rendering another kind of view. The different descriptor sets are bound in the command buffers when necessary. A single descriptor set could also be used, and updated to reflect changes to buffers each frame. However that would incur additional overheads. If the contents of a uniform buffer needs modification, changing it directly using vkMapMemory/vkUnmapMemory is appropriate.

An example of updating the descriptor set is provided in the ImageEffects sample. The texture for all effects changes periodically. This is done by updating the descriptor set by calling vkUpdateDescriptorSets with a VkWriteDescriptorSet structure set to change the bound VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER. When you change a descriptor set, you may need to re-record command buffers that use that set.

### Command Buffers

Command buffers contain commands that are sent to the GPU device via queues. Multiple buffers can be created, and their contents can be re-issued to the GPU each frame. Additionally, the buffers can have their contents reset and re-recorded. For maximum benefit, command buffers should be created outside of the main rendering loop.

The command buffer architecture allows for multi-threaded GPU command building, and also multi-threaded dispatch to multiple queues, allowing for efficient use of the CPU when rendering.

## Guidelines for Optimizing applications

There are some general guidelines for ensuring your application is efficient.

1. Use a custom parallel allocator for allocating resources when using multithreading. If standard allocators are used, they can incur synchronization overheads internally. When building command buffers and descriptor sets in a multithreaded way, it is better to provide custom allocators and pools which are able to perform lockless allocation within the same thread.
2. Use multiple pools so multiple threads can use them in a multithreaded environment.
3. Organize command buffers with barriers on dependencies, and don’t rely on queue submit fences for synchronization.
4. Use the multi-pass features of Render Passes to reduce bandwidth usage when rendering.
5. Do not re-record command buffers when the contents do not need to change.

6. Update uniform buffers at appropriate times during the frame, when the GPU is not using the buffers.

## Validation

Vulkan provides the mechanism for using an optional set of layers that help provide Vulkan syntax and usage checking. There are currently 8 layers that are provided in the Android NDK:

VK_LAYER_GOOGLE_threading

VK_LAYER_LUNARG_parameter_validation

VK_LAYER_LUNARG_object_tracker

VK_LAYER_LUNARG_core_validation

VK_LAYER_LUNARG_device_limits (not included in Android NDK 13.1)

VK_LAYER_LUNARG_image

VK_LAYER_LUNARG_swapchain

VK_LAYER_GOOGLE_unique_objects

Each checks a unique aspect of properly using Vulkan in your application including parameter validation, thread-safety, and lifetime usage of objects.

We recommend the use of validation layers during development only. They should be disabled to allow for highly optimized production quality app execution.

Validation layers are loaded when the Vulkan Instance is created. A verification step needs to be done to make sure each of these layers are detected by Vulkan and are available to be loaded. A callback function is defined which will process the various warnings and errors. This function can log the messages and is a suitable place to have break points during development to catch the errors when they occur.

The order of these layers makes a difference for proper reporting. We recommend keeping the order as shown above.

On Android these validation layers need to be included in your app as they are not included in Android distributions. These layers are implemented in several libraries provided in the Android NDK (/sources/third_party/vulkan/src/build-android/jniLibs)
). Refer to the module's build.gradle file to see how they are included in the build process.

Make sure you have the ANDROID_NDK environmental variable set to point to the Android SDK ndk-bundle folder. This variable is used by the Gradle scripts to locate these prebuilt Vulkan validation libraries provided in the NDK.

## Release History

Release History

V1.05 - Feb 22, 2017

- Added Gui sample which shows how to add an immediate mode Gui in Vulkan
- Added Cubemap sample which shows how to sample a cubemap
- Updated framework files to use KTX file format.
- Modified several samples to use KTX file format instead of TGA
- Added center option to mesh object loader to offset objects to origin
- Added depth stencil state parameter to InitPipeline utility function
- Updated support for Android Studio 2.2.3, Android NDK r13b, Gradle plugin 0.8.3, and Gradle 2.14.1
- Verified on Android N

V1.04 - Nov 10, 2016

- Added Pipeline sample which shows how to derive and cache pipelines
- Added Multithreading sample which shows how to use secondary command buffers to render different objects in the scene
- Added Particle sample which shows how to create and update particles rendered as animated screen facing alpha blended quads.
- Moved framework files to a common folder so that all samples can share the same framework files.
- Updated support for Android Studio 2.2, Android NDK 13.0, Gradle plugin 0.8.2, and Gradle 2.14.1
- Verified on Android N

V1.03 – Sept 6, 2016

- Added Validation Layer support to all samples both in the framework code and non-framework implemented samples
- Added Validation sample to demonstrate how validation can be enabled.
- Added Memory Allocator feature to the framework which uses a simple memory sub-allocation technique.
- Added Suballocation sample which uses the framework's memory allocation system when loading a set of objects.
- Numerous changes to clear validation errors.
- Centralized the logic in the framework for changing the image layout
- Modified the loading process for tga and astc files to use vkCopyImage and appropriate image layouts.
- Code cleanup
- Verified on both Android M and N

V1.02 – July 11, 2016

- Updated Vulkan libraries
- Updated Vulkan include files (changed VK_API_VERSION to VK_API_VERSION_1_0)
- Updated support for Android Studio 2.1.2
- Deferred Multipass cleanup and fixes for correct second subpass
- Verified on Android N Preview

V1.01 – May 5, 2016

- Added 64bit Vulkan library (libVulkan.so)
- Updated support for Android Studio 2.0
- Added deferred multipass sample

V1.0 – March 14, 2016

- Initial Release

## Known Issues

- The deferred rendering sample may not work, on some devices as it might require a more current Adreno Vulkan driver.
- There are several Validation errors that we are choosing to ignore at the current time. After investigating, our expectation is that these will go away as the validation layer logic matures.

## License acknowledgements

### OpenGL Mathematics (GLM)

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## TinyObjLoader

Copyright 2012-2015, Syoyo Fujita.

Licensed under 2-clause BSD license.

## ImGui (Immediate Mode Gui)

The MIT License (MIT)

Copyright (c) 2014-2015 Omar Cornut and ImGui contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.