Using the Vulkan API on Android & NVIDIA SHIELD

Vulkan is an industry standard, cross-platform 3D API and once loaded, the core Vulkan API works on Android as it does on all other platforms. As such, the Vulkan API itself is not discussed here; here we only deal with setting up and using Vulkan on Android and NVIDIA SHIELD.

Requirements

NVIDIA is currently rolling out Vulkan support across the range of its desktop graphics cards and NVIDIA SHIELD devices. Please visit the Vulkan developer hub for the latest information on drivers and on OTA updates.


To use the Vulkan API in Android, 32-bit applications must be compiled with “hardfp” for the ARMv7 ABI.

Loading Vulkan Functions

Vulkan is not yet included in an Android API level, so applications cannot rely on the Vulkan library being present on Android Marshmallow devices. Even when it is included in an official Android API level, applications that want to run on earlier platform versions (e.g. with a fallback to OpenGL ES) are not able to directly link against the API. In both cases, applications need to load Vulkan dynamically, and handle the possibility that it might not be present:

```c
void* vulkan_so = dlopen("libvulkan.so", RTLD_NOW | RTLD_LOCAL);
if (!vulkan_so) {
    LOGD("Vulkan not available: %s", dlerror());
    return false;
}
```

Function pointers for the global Vulkan commands (those that do not take a dispatchable object as their first parameter) and vkGetInstanceProcAddr must be loaded dynamically:

```c
PFN_vkEnumerateInstanceExtensionProperties
    vkEnumerateInstanceExtensionProperties =
        reinterpret_cast<PFN_vkEnumerateInstanceExtensionProperties>(
            dlsym(vulkan_so, "vkEnumerateInstanceExtensionProperties"));

PFN_vkEnumerateInstanceLayerProperties
    vkEnumerateInstanceLayerProperties =
        reinterpret_cast<PFN_vkEnumerateInstanceLayerProperties>(
            dlsym(vulkan_so, "vkEnumerateInstanceLayerProperties"));

PFN_vkCreateInstance
    vkCreateInstance =
```
reinterpret_cast<PFN_vkCreateInstance>(
    dlsym(vulkan_so, "vkCreateInstance"));

PFN_vkGetInstanceProcAddr vkGetInstanceProcAddr =
    reinterpret_cast<PFN_vkGetInstanceProcAddr>(
        dlsym(vulkan_so, "vkGetInstanceProcAddr"));

All other Vulkan commands and the commands in the VK_KHR_swapchain and
VK_KHR_device_swapchain extensions can be obtained in the same way: function pointers obtained
this way can be used with any Vulkan instance.

Alternately, vkGetDeviceProcAddr, and any commands that take VkInstance or
VkPhysicalDevice as their first parameter can be obtained by calling vkGetInstanceProcAddr.
The function pointers returned are specific to the instance used to retrieve them, and avoid a dispatch
indirection. Similarly, commands that take a VkDevice, VkQueue, or VkCommandBuffer as their
first parameter (except vkGetDeviceProcAddr) can be obtained from vkGetDeviceProcAddr,
are specific to a particular device, and avoid a dispatch indirection.

Compatibility Note: This alternative process reflects an earlier version of the Vulkan
specification: the Android implementation should now have updated to the finaly
required behavior. Developers should be able to use dlsym to obtain
vkGetInstanceProcAddr, and through that obtain function pointers for all other
core and extension commands. vkGetDeviceProcAddr will continue to be
available and will return device specific function pointers that avoid dispatch
overhead.

Window System Integration

Android uses the VK_KHR_surface, VK_KHR_swapchain, and VK_KHR_android_surface
extensions to allow Vulkan to render to onscreen windows represented by an ANativeWindow.
This document doesn’t describe how to obtain an ANativeWindow representing an Android
window: see the NDK nativeactivity or gles3jni samples or NVIDIA’s GameWorks OpenGL ES samples.

There should be no need to call any ANativeWindow * functions on the ANativeWindow
directly when using Vulkan: all queries and configuration can be done through the VkSurfaceKHR
object and VkSwapchainKHR creation. The vkCreateAndroidSurfaceKHR function in the
VK_KHR_android_surface extension is used to create the VkSurfaceKHR from an
ANativeWindow.

Surface properties and swapchain creation have some platform specific behaviors on Android:

▪ VkSurfacePropertiesKHR::currentExtent is the default size of the window: a
  swapchain with this size will not be scaled during presentation.
▪ VkSwapchainCreateInfoKHR::minImageCount should be set to 3 for best
  performance on current Android devices when attempting to render at the display refresh
  rate.
▪ If VkSwapchainCreateInfoKHR::imageExtent is not the same as
  VkSurfacePropertiesKHR::currentExtent, the swapchain images will be scaled to
  the window size during presentation. The scaling filter is not specified, but is bilinear or
better. If the image and surface aspect ratios are different, images will be scaled non-uniformly rather than letterboxed.

- On Android there is no performance advantage to setting VkSwapchainCreateInfoKHR::clipped to VK_TRUE, though there may be on other platforms.