Vulkan and OpenGL Interoperability

Internship Report 2020

Wolfgang Rumpler
01526299
wolfgang.rumpler@tuwien.ac.at

January 14, 2021

1 Introduction

The Vulkan specification was released by the Khronos Group as a new graphics API in 2016. It was designed to provide direct control over the GPU and to allow for better performance on modern systems [8]. With its first release in 1992, OpenGL has been widely used as a cross-platform graphics API for almost three decades [10]. Today a massive portfolio of OpenGL code, frameworks, and projects exists. While the industry is slowly adopting the newer standard, rewriting every application from scratch in Vulkan is impractical. Therefore, it is of great interest to combine legacy software written in OpenGL with newer solutions written in Vulkan. For this purpose, multiple interoperability mechanisms exist and are discussed in Section 2.

In this work, interoperability between the two APIs refers to sharing and synchronizing access to the same physical resources. It allows efficient utilization of resources in both APIs without passing the data through host memory. Sharing arbitrary data is required in general. However, in many cases, it is sufficient to share and synchronize access to textures. For example, overlaying a user interface that is developed in OpenGL onto a Vulkan based application is such a use case.

The main goal of the internship at Snap Inc. is to implement a working solution for the interoperability between new Vulkan applications and an existing OpenGL framework for mobile platforms. For brevity, this existing OpenGL framework is also referred to as *the framework* in this work. A working prototype for specific hardware with a particular driver has to be developed and integrated into the framework without breaking its interface for existing applications. Furthermore, this prototype should be as platform-independent as possible and point to limitations and problems regarding the interoperability under the existing constraints.

2 Overview of the Approaches

There are multiple possible approaches to achieve interoperability between Vulkan and OpenGL. While it would be possible to copy the data from the source API to the host memory and then back to the target API, such a solution is unfavourable due to the additional time and power consumption involved in copying the data. All discussed approaches utilize some API extensions which allow exporting and importing images and synchronization objects to and from some

common representation. The efficient implementation of these extensions is the hardware and driver developers' responsibility who optimize this process for various platforms. As a result, it is not necessary to consider the platform's exact memory architecture, and the implementation becomes more platform-independent.

2.1 Nvidia Approach

The first possible solution is to use a single OpenGL extension developed by the Nvidia Corporation called `NV_draw_vulkan_image` [2]. It provides an additional draw call that allows drawing a Vulkan texture directly to screen space coordinates of the currently bound frame buffer. However, there are some obvious downsides to this extension. First, the extension API does not support programmable shaders. The texture has to be drawn into another OpenGL texture via the provided draw call before using it in a programmable manner. Additionally, it is a vendor-specific extension that is typically not implemented by other hardware vendors [11]. Therefore, the extension is not a good fit for the implementation, which has to run on mobile devices where NVIDIA hardware is rare.

2.2 Android Approach

An alternative approach is to use Android specific extensions for Vulkan and EGL that allow importing native buffers represented as `AHardwareBuffer`[1] in both APIs [20, 1, 3]. From EGL the images can then be used in OpenGL via the extension `GL_OES_EGL_image_external_essl3` [6].

The synchronization of the application and the framework can be implemented manually by using the Android specific extension `EGL_ANDROID_native_fence_sync`. This extension allows creating and importing native fence objects which are referenced using file descriptors [7].

This approach will work on most Android devices, as long as the extensions are available. However, it has the obvious downside that it will not be supported on, for example, desktop computers.

2.3 Direct Interoperability Approach

The third and probably the most portable solution is to use a set of extensions developed directly for OpenGL and Vulkan. Together these extensions provide a clean and platform-independent interoperability interface. In contrast to the other mentioned approaches, this method is also capable of sharing arbitrary buffers. In the context of this work, however, sharing and synchronizing access to textures is sufficient.

In Vulkan the required extensions are `VK_KHR_external_memory`, `VK_KHR_external_fence`, and `VK_KHR_external_semaphores` [13, 4, 5]. They provide the basic definitions for exporting buffers, fences, and semaphores to other representations. An additional set of extensions provides the specification for the platform-specific representations. In our case we are interested in `VK_KHR_external_memory_fd`, `VK_KHR_external_fence_fd`, and `VK_KHR_external_semaphores_fd`, which allow exporting the Vulkan objects into POSIX file descriptors [14, 16, 12].

[1] `https://developer.android.com/ndk/reference/group/a-hardware-buffer`

The OpenGL side is designed very similar and the extensions `EXT_external_objects`, and `EXT_external_objects_fd` are specified [18, 19]. These OpenGL extensions map cleanly onto the concepts of Vulkan that are not present in the older API. They separate the memory allocation and binding of memory to an image and add semaphores as a synchronization primitive in OpenGL.

The file descriptors, generated during exporting from Vulkan, can also be passed between different processes of Vulkan and OpenGL applications. However, there is no routine for importing fences in the `EXT_external_objects` extension. One has to fall back on the extension `EGL_ANDROID_native_fence_sync` mentioned earlier for handling exported fences. Another downside is that the resources have to be created with Vulkan since there is no support for creating semaphores or exporting textures in OpenGL [18].

3 Implementation

3.1 Overview

The preferred implementation approach is the last of the presented approaches in Section 2, mainly due to its flexibility and portability. During setup, the Vulkan application creates the textures which are used for rendering. The memory allocations of these textures are exported as file descriptors using the `VK_KHR_external_memory_fd` extension. These file descriptors are passed to the OpenGL framework, where they are imported as OpenGL textures using the `EXT_external_objects` extension.

For synchronization, the framework already uses fences to orchestrate internal processes. The implemented solution taps directly into these existing synchronization mechanisms to minimize the required changes and avoid breaking the interface. Every frame, the Vulkan application has to export a file descriptor referring to the fence used in the recent `vkQueueSubmit` call. This file descriptor is again passed to the OpenGL framework, where it is used for further synchronization. For doing so, it is imported as an EGL fence sync object using the `EGL_ANDROID_native_fence_sync` extension.

In summary, only two extensions to the already existing interface have to be made in the framework to support the interoperability approach. First, accepting and importing the Vulkan textures and second, receiving and utilizing a file descriptor every frame for the synchronization.

3.2 Challenges and Findings

3.2.1 Integration into an existing OpenGL Framework

The interface of the framework has to remain backward compatible with existing applications. However, it is designed with OpenGL applications in mind only. This assumption, unfortunately, introduces some incompatibilities with the Vulkan API.

The essential difference between OpenGL and Vulkan regarding the interface design is that resources have to be passed explicitly to all functions in Vulkan. An example of this is the command buffer recording, where the command buffer has to be provided to every `vkCmd*` function call [9]. In OpenGL, command buffers are collected and submitted implicitly through

its stateful API. Therefore, the existing OpenGL framework did not need to model APIs representing this state, other than holding an OpenGL context. In Vulkan, however, this state needs to be modelled explicitly, which has to be addressed in the prototype.

For example, the framework manages some resources internally for the convenience of the user. The creation, binding, clearing, and destruction of framebuffers are done internally. Implementing similar resource management for Vulkan applications would require incorporating Vulkan into the framework and would also imply further extensions to the interface. Rewriting the framework to be API agnostic is significant work and is not a goal of the interoperability prototype. Additionally, such fundamental changes can be challenging to integrate without breaking backward compatibility. So it is desired to only extend the interface with the absolute minimal set of necessary functions for the Vulkan support.

The solution for the resource management problem is to internally disable all unnecessary OpenGL code paths when Vulkan is used. This way, the Vulkan application can manage all resources independently, while the framework's interface remains unchanged for existing applications.

3.2.2 Separating Debugging Symbols

When developing native code for a mobile device, the application must be installed on some physical device to test the code in a real environment. This installation process can take a notable amount of time if the application is large enough. Especially builds containing debugging symbols for debuggers can be significantly larger than builds without them. This can impact the development efficiency, as the time for transferring, installing, and running the application in debug mode is slow.

In order to speed up the development workflow, it is possible to separate the debugging symbols from the executable binary file using programs like `objcpy`[2]. The stripped symbols are not transmitted to the device during installation. The debugger runs on the development machine while the target device runs the application as a remote debugging process. Debuggers connected to this remote process can use the locally stored symbols to complete the debugging information of the running application.

Custom CMake targets copy and strip the debugging symbols from all built libraries and store them to a fixed location before packaging the app for installation. The packaged app is then installed and run on the target device without the overhead of transferring the additional symbols, while the debugging workflow is not impaired. For debugging in GDB, the solib-search-path[3] is set accordingly to the location of the separated debugging symbols so that GDB can resolve the symbols locally.

3.2.3 Retrieving Function Pointers

Getting the pointers to the extension functions turns out to be a challenge as well. Glad 2.0[4] is used to generate the header files and a function loader for Vulkan. This function loader queries the device driver's function pointers during runtime to build a dispatch table for all

[2] `https://sourceware.org/binutils/docs/binutils/objcopy.html`
[3] `https://sourceware.org/gdb/onlinedocs/gdb/Files.html`
[4] `https://github.com/Dav1dde/glad/tree/glad2`

functions to optimize the cost of API calls. This table bypasses an additional indirection through a trampoline function, which is a small function that manages a similar dispatch table and the call-chain in the system library [17].

Glad 2.0 also supports generating headers and loading functions for device extensions. However, retrieving the function pointers for the required extensions fails with the implementation of Glad. It turns out that, unfortunately, the driver does not officially support the required extensions. The listing of the available extensions with `vkEnumerateDeviceExtensionProperties` does not contain, for example, `VK_KHR_external_fence_fd`. This fact would prevent any implementation entirely from working because the function `vkGetDeviceProcAddr`, which is used to get the function pointers from the device driver, has to return `NULL` if the extension is not enabled during device creation [9]. The implementation of Glad relies on this specification and checks whether the extensions are listed as available by the driver and, in our case, figures that it can not retrieve the function pointers to the extensions. However, it is still possible to use the extension functions by manually querying them directly with `vkGetDeviceProcAddr` and without enabling them during the device creation. Therefore, the driver's implementation does not correspond to the specification, as it is supposed to return `NULL` in this case.

3.2.4 Vulkan Layers Breaking the Extensions

After manually retrieving the function pointers for the extensions, testing whether the functions are fully functional is the next step. Exporting a Vulkan fence as file descriptor with the function `vkGetFenceFdKHR` returned the Vulkan success code `VK_SUCCESS`. However, the returned file descriptor was always -1, which is not a valid descriptor, as they are defined as non-negative integers in the POSIX standard [15].

Investigating the program flow in GDB revealed the intermediate function calls to the activated layers. Vulkan layers can intercept, evaluate, and modify functions before they are invoked in the device driver [17]. The activated layers are suspected to interfere with the extension call since the extension is not enabled in a standard way, as described in Section 3.2.3. Deactivating all Vulkan layers resolved the issue, and the file descriptor returned by `vkGetFenceFdKHR` was valid.

Unfortunately, enabling any of the Vulkan layers provided in the Android NDK 21.3.6528147 resulted in the described issue. To still use the validation layers during development, the workaround is to entirely decouple the Vulkan implementation from the OpenGL framework and only use the interoperability features with deactivated validation layers when necessary for testing. Further investigation of the validation layer's source code is required to understand the exact origin of the problem and provide a clean fix.

3.2.5 Warning: Command Buffer in Use

Checking the output of validation layers during development is desirable to prevent typical Vulkan API misuse. However, the validation layers keep complaining about the command buffer being still used when a new recording of an old command buffer is started. This definitely should not be an issue, as the fence used in the `vkQueueSubmit` call is waited on in the OpenGL framework after importing it from the file descriptor. Waiting on this fence ensures all command buffers have finished executing [9].

Explicitly waiting a second time on the same fence in Vulkan before starting the recording removes the warning. It is important to note that it does not matter how long a timeout we specify for waiting on the fence to be signaled. A timeout of zero nanoseconds in the `vkWaitForFences` call, which results in an immediate return from the call, works as well.

A plausible explanation is that the validation layers cannot track the usage of the submit fence after it is exported. The validation layer implementation might assume that no synchronization happened and that the command buffer, therefore, might still be in use. It would be required to investigate the validation layer implementation in detail to confirm this. However, the evidence was sufficient to prevent the warning with the additional wait call and a timeout of zero nanoseconds.

3.2.6 OpenGL and the Image Layout

Vulkan has a concept called image layouts [9]. The layout of an image dictates which operations on the image are valid. The user has to transition images between layouts to achieve the best performance or when an operation requires a different layout. OpenGL does not provide a similar concept out of the box. However, passing the memory layout to OpenGL in an interoperability implementation is desirable so that OpenGL can interpret the memory correctly. The extension `EXT_external_objects`, therefore, provides an interface for doing so. Providing the current layout and performing layout transitions via this interface is possible with semaphores.

Unfortunately, since semaphores are not used in the presented approach for synchronization, it is impossible to provide this additional information to OpenGL. Luckily, everything works correctly with a layout transition to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` as the last operation in the command buffer. This is done because the OpenGL framework only reads data from the provided texture. This solution suggests that either the specified layout matches the one used by OpenGL, that the driver does not care about the layout, or that there is some internal mechanism for detecting and handling the layout in the OpenGL driver. This, of course, is not guaranteed to work, and a better implementation would be to utilize semaphores to pass the layout specification.

3.2.7 Staging Buffers Not Required

Initializing data in host visible memory before transferring it to device local memory for optimal performance is a typical practice known from previous experience with Vulkan development on desktop computers. However, querying the memory flags for all available memory types on the testing device revealed that they all have the device local bit set. This makes sense since mobile GPUs typically do not provide dedicated memory and instead use the system memory as shared memory between CPU and GPU. For example, the memory flags of an Adreno 630 can be viewed on the website "gpuinfo.org"[5]. This fact is also mentioned in the Vulkan specification and can be found in the section "Device Memory" under the term "Unified Memory Architecture" [9]. Therefore, intermediate staging buffers are not required for uploading data to the GPU on the hardware used and will not be beneficial to the performance.

[5] `https://vulkan.gpuinfo.org/displayreport.php?id=8585`

	CPU Time	GPU Time
OpenGL Cube	$1.682 \pm 0.214ms$	$0.991 \pm 0.160ms$
Vulkan Cube	$1.580 \pm 0.236ms$	$1.147 \pm 0.206ms$

Table 1: The mean and standard deviation of the time spent on GPU and CPU of 800 frames for rendering a cube in Vulkan with the interoperability implementation, and in OpenGL without it.

4 Performance Evaluation

It is crucial to verify that the implemented solution does not perform worse than an OpenGL app when Vulkan is used. An existing tracing routine based on GPUVis[6] is already implemented in the OpenGL framework. This routine is used to measure the amount of time spent on rendering on the GPU and processing on the CPU. Thereby, the CPU time is mainly of interest because the raw render time is expected to be equal in both APIs for this simple benchmark.

An existing OpenGL sample that renders a single cube was replicated in Vulkan to match the required workload in both APIs approximately. Furthermore, the GPU and CPU frequency must be set to a fixed value to prevent measurement errors from dynamic frequency scaling.

Unfortunately, the render time is not directly comparable in the implementation because the OpenGL sample is rendered with MSAA and the Vulkan sample is not. Additionally, the Vulkan sample renders into a separate texture and copies the image content over to the target texture, which is then used by the OpenGL framework. It should be possible to optimize this and directly render the image into the target texture. Therefore, the results are expected to differ for the GPU time. The times should match very closely on the CPU side because the only additional work on a frame basis is exporting the submit fence. The synchronization is done internally equally for both samples, and the rest of the interoperability implementation is done solely during setup.

Table 1 lists the result of 800 frames for both implementations and provides evidence that no time is wasted on the CPU when using the interoperability extensions. On one hand, the CPU time appears to be about $100\mu s$ faster in the Vulkan case. It is important to note that the Vulkan implementation did record the complete command buffer every frame, which is unnecessary and optimizing this would allow reducing the CPU time further. On the other hand, the additional $150\mu s$ required on the GPU for Vulkan is most likely due to the additional texture copy. Figure 1 additionally shows the same results as a stacked bar chart.

5 Conclusion

This work summarizes the design, implementation, and issues of a Vulkan to OpenGL interoperability implementation in an existing OpenGL framework on a mobile platform. There are multiple approaches to implement interoperability between those APIs. The discussed solution is based on a set of Vulkan and OpenGL extensions that are not vendor-proprietary and allow for a platform-independent implementation as long as the driver supports the required extensions.

[6] `https://github.com/mikesart/gpuvis`

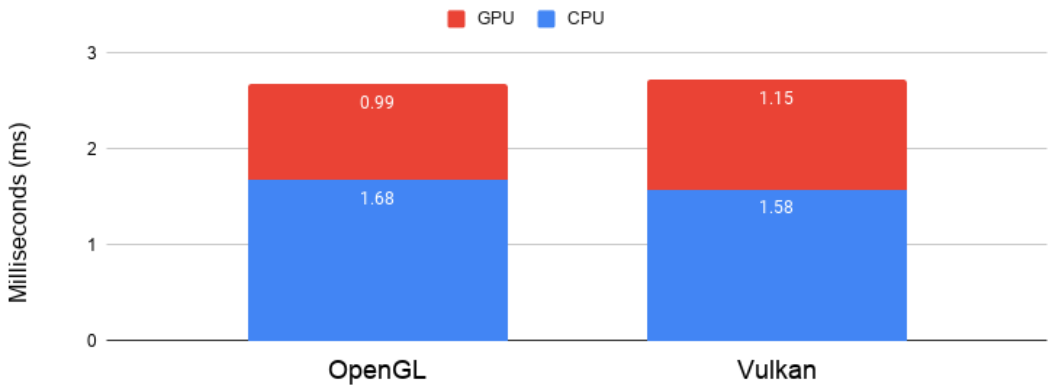

Figure 1: The values from Table 1 visualized as stacked bar chart.

Unfortunately, the presented implementation has a few shortcomings regarding the development workflow with Vulkan layers, synchronization, and Vulkan image layouts. These shortcomings are mainly due to the requirement of not breaking the framework's interface, keeping the changes inside the framework to a minimum, and a presumably incomplete implementation of the required extensions in the driver. They would be preventable in a second iteration of the implementation but require significant changes to the OpenGL framework and possibly breaking the interface. Finally, the achieved performance using the interoperability implementation is comparable with an OpenGL only reference sample.

References

[1] Mathias Agopian, Jamie Gennis, and Jesse Hall. `https://www.khronos.org/registry/EGL/extensions/ANDROID/EGL_ANDROID_image_native_buffer.txt`. Accessed on October 6th 2020. Nov. 2012.

[2] Jeff Bolz. `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_draw_vulkan_image.txt`. Accessed on October 6th 2020. Feb. 2016.

[3] Craig Donner. `https://www.khronos.org/registry/EGL/extensions/ANDROID/EGL_ANDROID_get_native_client_buffer.txt`. Accessed on October 6th 2020. Jan. 2017.

[4] Jason Ekstrand et al. `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_external_memory.html`. Accessed on October 6th 2020. Oct. 2016.

[5] Jason Ekstrand et al. `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_external_semaphore.html`. Accessed on October 6th 2020. Oct. 2016.

[6] Jan-Harald Fredriksen, James Helferty, and Mark Callow. `https://www.khronos.org/registry/OpenGL/extensions/OES/OES_EGL_image_external_essl3.txt`. Accessed on October 6th 2020. July 2014.

[7] Jamie Gennis. `https://www.khronos.org/registry/EGL/extensions/ANDROID/EGL_ANDROID_native_fence_sync.txt`. Accessed on October 6th 2020. May 2012.

[8] Khronos Group. `https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification`. Accessed on October 6th 2020. Feb. 2016.

[9] Khronos Group. https://www.khronos.org/registry/vulkan/specs/1.2/html/index.html. Accessed on November 2th 2020 – Version 1.2.159. Aug. 2020.

[10] Khronos Group. https://www.khronos.org/opengl/wiki/History_of_OpenGL. Accessed on October 6th 2020.

[11] Khronos Group. https://www.khronos.org/opengl/wiki/OpenGL_Extension#Extension_Types. Accessed on October 6th 2020.

[12] Jesse Hall et al. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_external_semaphore_fd.html. Accessed on October 6th 2020. Oct. 2016.

[13] Jesse Hall et al. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_external_fence.html. Accessed on October 6th 2020. May 2017.

[14] Jesse Hall et al. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_external_fence_fd.html. Accessed on October 6th 2020. May 2017.

[15] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), p. 60.

[16] James Jones and Jeff Juliano. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_external_memory_fd.html. Accessed on October 6th 2020. Oct. 2016.

[17] KhronosGroup. https://github.com/KhronosGroup/Vulkan-Loader/blob/master/loader/LoaderAndLayerInterface.md. Accessed on October 13th 2020.

[18] Carsten Rohde et al. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_external_objects.txt. Accessed on October 6th 2020. Aug. 2016.

[19] Carsten Rohde et al. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_external_objects_fd.txt. Accessed on October 6th 2020. Aug. 2016.

[20] Ray Smith et al. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_ANDROID_external_memory_android_hardware_buffer.html. Accessed on October 6th 2020. Mar. 2018.