Vulkan-ize VirGL

Proposal for GSoC 18
Nathan Gauër

Name:
Email:
Phone
Programming languages: C / C++
Past open source contributions: PoC for VirGL on Windows
Sample source code, hobby projects: https://github.com/Keenuts
Website: www.studiopixl.com/blog

Summary

Summary **2**
Introduction 3
Motivation 3
Understanding 4
Vulkan on Linux 4
QEMU Consoles 5
VirGL Integration 6
VirGL and Vulkan 6
Project presentation 7
QEMU and VirGL 7
VirGL and Vulkan 7
Timeline 8
GSOC 12 weeks planning 8
First period 8
Second period 8
Third period 8

Introduction

I'm Nathan, a French student in Computer Science. I expect to graduate in 2019. Since last year, I joined a laboratory in my school. In this lab I worked on several subjects like VirGL, CAN bus reversing, non-biased rendering algorithms, and a small 32 bit kernel.
These projects taught me a lot. However, I always go back to one main subject: graphics.
This fascination takes its roots from my early exposure to 3D software and drawings. It started as a simple hobby, and slowly became my favorite subject.
You can see some subjects I worked on on either my blog, of GitHub.
Last year, I worker on a PoC to provide 3D acceleration on Windows guests running on QEMU.
This year, I had two opportunities, either continue working on this subject, or try this other one.
Both closely related to my subject of interest.

Motivation

Vulkan is a low-level, low-overhead API. Validation layers are removed, state-tracking is simplified, and the user is free to use all the resources the hardware provides.
This API is getting more popular each year, thus, adding a decent support for it on QEMU will be a mandatory step.

Last year, I had the opportunity to work on VirGL support for Windows. Through this project, I learned a lot about VirGL, Windows driver structure, and the linux graphic stack itself.
Along these skills, I also the gaps I still have.
To continue working on this subject is logical. However, switching lines to focus on the host part may be a better way to do it regarding my abilities and the current state of the project.

Both Windows and VirGL subjects are huge, and both cannot fit in one GSoC. But this subject has the advantage take place on the Linux part, which is open source. This allows me to understand the behavior of surrounding components, and not only rely on a publicly disclosed documentation.

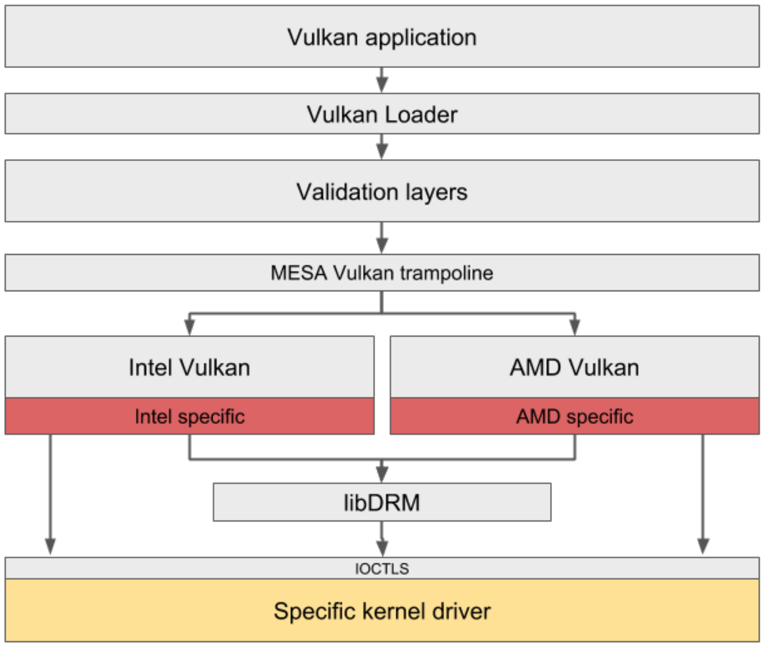
I do not have a extensive knowledge of Vulkan. I worked on a small object viewer using it (REPO), and understood available code on MESA and AMD's open source loader.
It's sure not a lot, but it's more than what I know about D3D subsystem.

Understanding

This project will interact with several components of QEMU. Before starting anything, we must be sure to understand every parts, and see how they interact together. Here is an exposé of my current understanding of the subject. Followed by a proposal on a VirGL API.

Vulkan on Linux

The Vulkan stack is composed of several parts resumed in the following schema.



Our Vulkan application will use Vk* calls. These calls are made to a loader, which will, depending of the application behaviour, either directly call the real driver, or layers called 'Validation Layers'.

Validation Layers are designed to help application developers. They can check parameters validity, dump a trace, or enable bindings for a debugging application like RenderDoc. These layers can slow down our application a lot, and are not useful in a release build. Thus, instead of embedding them in an usermod driver, we put them on top, and enable them on VkInstance creation.

From now on, we will assume we are using MESA as our current Vulkan implementation. Thus, we start with another trampoline, in MESA. This one will redirect our calls to the correct backend.

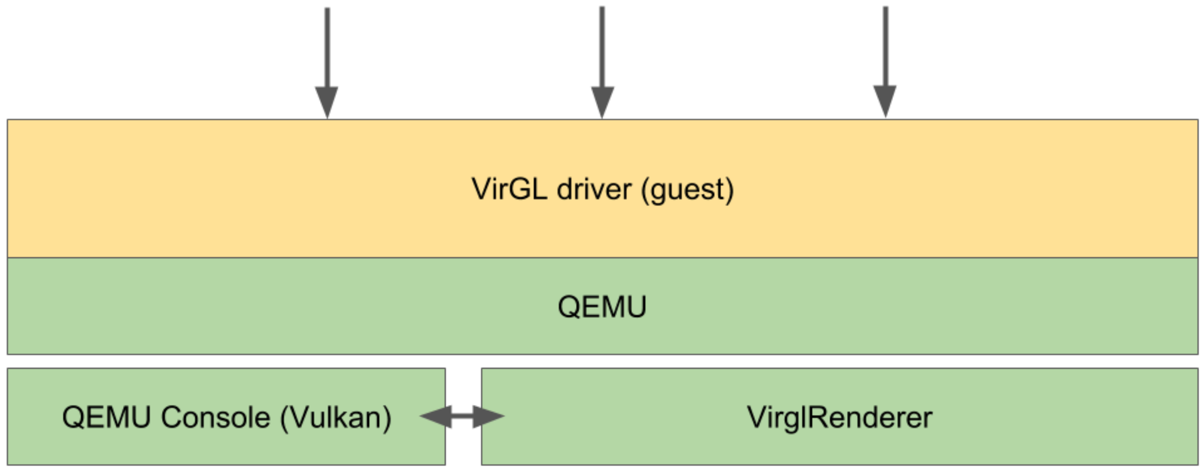
For now, we can find two implementations: Intel's and AMD's. (mesa-git/src/amd/vulkan/*, mesa-git/src/intel/vulkan/*).

By checking both implementations, we can see a lot of similarities, and even if both code bases are independant, we can find a common trunc. (using diff on some files can be fun).
To communicate with the kernel driver, our drivers will use both generic libDRM wrappers, and driver specific IOCTLs.
On the application side, a basic scenario would be:

- Create a VkInstance (with or without extensions and layers)
- Physical device enumeration
- Find correct memory/queue types, create command pool
- Logical device and queue creation
- Creating command buffer, swapchain, depth, etc
- shader loading, pipeline creation
- render loop

QEMU Consoles

To let the user interact with the VM gui, QEMU has a part called a "Console".
There is several console type available, using SDL, OpenGL, or EGL.
(located qemu-git/ui/*)



Each backend will implement a "struct DisplayChangeListenerOps". This structure is similar to a fops struct for a char-device on Linux. It contains all the basic operations the backend implements. This struct can be split in two parts. The upper half is valid for non-OpenGL consoles, the bottom one contains OpenGL related functions pointers.
This struct is a part of a DisplayChangeListener, which will be used in a QemuConsole.
Now, we can implement a Console. As in real life, we have both text only consoles, and graphical consoles. Depending of the configuration, we can use one listener, or another.

Now, we can add another layer: the QemuDisplay.
A QemuDisplay will implement two functions: init and early_init. This is basically our final display abstraction (in the SDL case for example, it's a GUI).

Now, in the main function, after ~1500 lines of main, we find a qemu_display_find_default, which will choose, depending of hardcoded priorities, a DisplayOptions. Using this result,
we will call the early_init of our QemuDisplay.

VirGL Integration

As we saw, the QEMU console is an abstraction to represent a display.
On the other side, we have VirGL, which is called by the VirtioGPU device.
How both are linked ?

The first step is to look at the VirtIOGPU device declaration.
Two important things. First, the virtio_gpu_class_init function, which setup all the VirtioDeviceClass struct.
This struct contains basic functions any VirtIO device must implement.
The “realize” function is called when the device is created. Thus, it’s the place where our display will be linked to the GPU framebuffer.
We can now find a graphic_console_init(...) and a dpy_gfx_replace_surface(...) call.
This is the place we were looking for.

VirGL and Vulkan

Tweaking the SDL2 console to add the ability to display Vulkan based commands is one little thing. The real deal is to add Vulkan support to VirGL itself.
Thanks to it’s independent nature, VirGL can easily be tested as a standalone OpenGL/Vulkan application. Just replacing the guest by an userland application using a strange API.
Anyway, we are getting to the real things, time to present the project itself !

Project presentation

QEMU and VirGL

The first rule is: we cannot break everything.
A first step will be to implement the Vulkan backend itself. A simple approach would be to use the current SDL GUI. SDL 2.0.6 provides support for Vulkan. Thus, we could add a support for Vulkan issued frame buffers.

Now, how can we generate those frame buffers ? Are we not gonna break all display devices in QEMU ?

A method could be to keep the GL console, and use VK_KHR_external_memory/VK_KHR_external_memory_fd extensions to share our buffer with OpenGL. Thus, we could mix OpenGL and Vk on VirGL, and keep the actual console model.

VirGL and Vulkan

As we saw, we could keep the Vulkan addition transparent to QEMU Consoles.
As you suggested, we shouldn't try to do any OpenGL -> Vulkan conversion. Instead, we should focus on Vulkan-guest to Vulkan-host interfaces. And if needed, find bridges.
The first step is to try to find what Vulkan needs, and add new API entries to VirGL.
Globally we'll need to expose memory regions, queues, and accept SPIR-V, and have sync primitives.

Commands we may need involve:

- Resource creations / Deletion
- Data transfer
- Command submission
- State fetch

On VkInstance creation, we create our internal state.
Then, for all discovery functions, we expose host devices, to let the guest use the best device for his use case. So basically, we mainly forward enumerations calls to the host.
Regarding the VirGL API, it could be a single call to get all informations about the devices.

Then, we will create a lot of different objects, from simple uniform buffers to more complex objects like pipelines.
Thus, we need to add two API points, one to create VK-objects, and one to delete them.

Once objects have been created, we need to initialize the concerned memory regions, and send commands.
Memory mapping related functions need a transfer function, or shared mappings.

Later, for command buffers, one last command to send a guest mapped buffer. This buffer will also contains informations about IN and OUT semaphores.
Finally, we will need some calls to get a fence state. Be able to let the guest wait for a fence.
More specific parts will be missing, like extensions to support OpenGL/Vulkan interop, and WSI. But this can be added later.

So globally, we will have the same kind of API Virgl already propose. For now, we could start a new set of commands, and thus, be sure to avoid regressions.
Once the API is more defines, we may find common grounds, and re-factorize the code.

Timeline

This project can definitely not fit in a single GSoC. However, we could try to find some subtasks we could do. First thing would be to only work on the VirGL side. No Guest/Host.
We take a vulkan application, plug our 'userland driver' to it, and directly call VirGL functions.
No QEMU, no VMs, no complex setup, easy debug.
Once we can execute some Vulkan application, we will need to add a support for our new API in QEMU, and write real drivers.

GSOC 12 weeks planning

The global project aim to provide a coherent API for VK-Virgl. Also a sample "ICD model". (userland - userland, no VM involved).

First period

Vulkan application starts, we should be able to get device state, and initialize correctly.
This implies forwarding host vulkan specs on the guest, and simulating Vulkan ICD's behavior.
This step could be faster or longer than expected, and is here just as an indicator.

Second period

Objects can be created, and deleted on instance deletion/on request.
We should start to get some command buffers from the guest reconverted to Vulkan calls.
This step will require to serialize Vulkan object creation (which iglobaly done by the VkStruct format of the commands).
We will also need to track these objects, and manage semaphores.

Third period

It's hard to determine what problems we may encounter in the process, and how many times the API will have to evolve. But the goal of the GSoC is to be able to run a basic Vulkan application using VirGL (still on the same machine, no paravirtualization).
If by any chance, the VirGL part works, and there is time left, next step will be to add the command parsing on QEMU.