The Pennsylvania State University
The Graduate School

VAR: VULKAN API REMOTING FOR GPU-ACCELERATED RENDERING AND COMPUTATION IN VIRTUAL MACHINES

A Thesis in
Computer Science and Engineering
by
Yunju Lee

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2020

Abstract

While virtualization technologies have been widely adopted, virtualizing GPUs remains challenging. In this thesis, we present a GPU virtualization solution based on API remoting, forwarding API calls from guests to the host for execution. Our proposed solution provides guest applications with the Vulkan API, which has been increasingly popular recently, for utilizing GPUs for both rendering and computation. It enables dynamic resource sharing, and is not bound to specific devices. In addition, no modification on the guest OS and applications is required. We also propose several custom optimizations, with the goal of reducing the potential overheads of such a software-based virtualization layer. Our solution is evaluated on workloads with a wide range of native performance. The collected experimental results indicate that our solution achieves up to 98% of native performance on workloads with higher computation load per frame. The results also show that our solution is satisfactory in a variety of use cases.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Virtualization technologies have enabled fast deployment of software stacks by providing flexible control of hardware environments [1–3]. When employing virtualization, essential hardware resources on a host machine, such as CPU, memory, and storage, can all be shared among several guest virtual machines. We can create virtual machines on demand, take snapshots of a VM's execution state, suspend and resume a VM, or port a VM from one physical machine to another.

While GPUs have been exploited in various applications for accelerating graphics rendering and computation, it remains challenging to enable VMs to use GPUs. Without practical GPU virtualization solutions, we can only assign a whole physical GPU to a VM, which does not allow resource sharing, or resort to pure software rendering, which does not use the GPU at all, leading to extremely low performance. Thus, this missing component in virtualization is worth studying.

One major challenge of virtualizing GPUs is that the specifications of hardware interfaces are not public. Without the knowledge of how to interact with the hardware interface, it is difficult to manage resource allocation and provide isolation, which are essential to virtualization. The rapid product release cycles further raise the difficulty. Developing a new virtualization solution for each and every newly-released physical device is almost infeasible. On the other hand, though it is possible to move the virtualization layer toward the software end and prevent interacting directly with the hardware interface, software solutions usually incur large overheads. Careful design and optimization are required for satisfactory performance.

In this thesis, we take a software-based approach, called *API remoting*, to GPU virtualization. Instead of virtualizing the exact hardware interface of a specific physical GPU device, we focus on Vulkan [4], a standardized GPU API, through which an application in a VM can access virtualized GPU resources. Our proposed solution is composed of a VAR server (Virtual GPU) and VAR driver (ICD). The VAR server is a process running in the host OS that can access the physical GPU via the Vulkan API. The VAR driver, which also provides the Vulkan API, can be loaded by guest applications. It forwards rendering and computation commands to the VAR server, while handling presentation commands within the guest.

Despite the existence of API remoting implementations for common GPU APIs such as OpenGL [5] and CUDA [6], providing virtualized GPU through the Vulkan API is a topic worth investigating because Vulkan aims at enabling higher performance by allowing explicit memory management and multi-threading. These features of Vulkan, however, lead to new challenges as far as GPU virtualization is concerned. Specifically, we need to synchronize the memory content between the host and the guest, and handle the commands sent from different application threads appropriately. In this thesis, we propose and evaluate our solution to these challenges, along with more general GPU virtualization issues such as communication and remote presentation. By evaluating the proposed solution, we show its effectiveness and broad applicability.

The main contribution of this work is three-fold. First, we propose a Vulkan API virtualization solution based on API remoting, which provides a guest OS with all commonly-used Vulkan APIs. Second, we address the implementation challenges specific to Vulkan and propose effective solutions, handling communication overhead, memory mapping, remote presentation and multi-threading. Third, we present detailed experimental results, indicating that our solution achieves satisfactory frame rates in a variety of workloads, with particularly low overheads when the computation load per frame is high, leading to up to 98% of native performance.

The rest of the thesis is organized as follows. Chapter 2 provides background on virtualization and GPUs. Chapter 3 reviews previous approaches to GPU virtualization. Chapter 4 illustrates the overall architecture of our solution and its major components. Chapter 5 discusses the major challenges and how we overcome them in our Vulkan API remoting solution. Chapter 6 evaluates our virtualization solution using various GPU workloads. Finally, Chapter 7 concludes the thesis with a summary, and points out potential directions for future work.

Chapter 2 | Background

In this chapter, we first give a review of the core concepts in virtualization. We then introduce GPU and its APIs, through which applications communicate with GPU devices. Finally, we focus on the Vulkan API and discuss its advantages, demonstrating the need of an approach for its virtualization.

2.1 Virtualization

Virtualization technology enables running an entire machine, including both its hardware and software, virtually within a software environment. The machine having direct access to the real, physical hardware is called the *host*, in which multiple virtual machines, or *guests* can be hosted. A piece of software running in a guest OS has no or minimal awareness that it is in a virtualized environment. Typically, a *hypervisor* is installed in the host, facilitating the management of guests and the communication between the host and guests.

One of the major motivations for developing and employing virtualization technology is that it allows resource sharing. Any hardware device, such as a CPU or a disk, has a baseline power consumption even when being idle. Therefore, better resource utilization efficiency can be achieved if we can distribute a powerful computation resource or a large storage to multiple virtual machines. Virtualization enables us to do so while each of the virtual machines still has the *illusion* that it is accessing its own private resources. Virtualization also enables dynamic distribution of the resources. For example, when there is a burst of demand of a specific web service, to fulfill the need, adding virtual machines to a cluster is much more easier than adding physical machines. We can also add more disk space or memory to a specific virtual machine, scaling it up with the rapidly increasing demand. Similarly, when the demand decreases, resources can be released and assigned to other virtual machines.

Another scenario where virtualization may be useful is for packing an entire environment for running specific programs. Perfect platform independence is difficult to achieve, and both programmers and application users more or less suffer from setting up the prerequisites, including libraries, operating systems, or even hardware, for running a certain program and getting the expected, previously-observed outcome. With virtualization, we can turn the entire platform into a virtual machine that can be delivered and ported to various host platforms. In addition, virtual machines are isolated from each other. As a result, even if an application "breaks" a guest OS completely, neither the host nor other guests would be affected. That is, each guest machine is secured even though it shares the underlying hardware with others. This nice property also leads to the use of virtual machines as sandboxes.

2.1.1 CPU Virtualization

The typical typology of virtualization is based on how the CPU, the most essential computational resource, is shared by the host across the guests. In full virtualization, the complete set of instructions is provided to the guest OS. However, some instructions are privileged, and cannot be executed from the user space, where a virtual machine runs. Therefore, the hypervisor is responsible for identifying such privileged instructions from the guest in advance and convert them into safe instructions that can be executed in the user space while leading to the equivalent behavior expected by the guest. This process is called *binary translation*. Note that, in this kind of virtualization, there is no need to modify the guest OS.

Full virtualization can be hardware-assisted, in which the CPU provides special instructions that can be utilized by the hypervisor to enter the vm mode before handing over control to a guest OS. The CPU is responsible for ensuring that these privileged instructions are executed safely without interfering other processes in the host OS. With the elimination of software-based binary translation, this approach boosts the performance. However, it requires hardware support, which must be developed by CPU vendors.

In para-virtualization, a guest OS is slightly modified so that whenever a privileged instruction is to be executed, the OS makes a *hypercall* to the hypervisor instead. That is, the guest OS needs to be built with the API provided by the hypervisor. The hypervisor, which can execute instructions in the privileged mode, handles the requested instructions for the guest. By doing so, the need of binary translation is also eliminated, so para-virtualization also brings performance improvement over pure software-based full virtualization. More importantly, para-virtualization is a pure software solution that can be developed without any hardware support.

2.1.2 I/O Device Virtualization

I/O devices exhibit quite different properties from CPUs, thus leading to different strategies for virtualizing them. First, some devices such as traditional disks and network cards are slow compared to CPUs. As a result, the virtualization layer usually does not become a major bottleneck. Second, some I/O devices have a set of essential operations much smaller than the set of instructions of a CPU. Thus, in some cases, full virtualization can be realized. The hypervisor emulates a device with exactly the same interface, using the resource in the host. For example, qemu [7] implements various storage interfaces such as SATA and SCSI, and network adapters such as rtl8139.

However, implementing the full interface may not be necessary and, more importantly, sub-optimal in terms of efficiency. Alternatively, a special interface can be implemented – for example, Virtio [8] is implemented by qemu. Virtio is a layer between the hypervisor and the guest OS. This generic layer can be specialized into disk, network, or other interfaces. That is, the hypervisor usually only exposes a virtualized device with a customized interface to the guest OS. This is similar to para-virtualization in the sense that the guest does not see exactly the same hardware interface as a physical machine, but usually modifying the guest OS is not required; instead, only a customized driver is needed. It is in essence similar to installing a new driver when a new device is attached. In other cases, difficulty for full virtualization may be experienced. For instance, PCIe SSDs have very fast performance, and performance may be the bottleneck for full virtualization. Observing this, mediated pass-through is exploited [9].

Different from most devices, interfaces of GPUs are vendor-specific and proprietary. A variety of GPU-centric virtualization techniques have been investigated in the past and we will discuss them in Chapter 3.

2.2 Application Programming Interface for Graphics Processing Unit

Given that GPU interfaces and drivers are vendor-dependent and proprietary, applications utilizing GPUs usually do not interact with GPU hardware interfaces directly. Instead, they send commands to GPUs through an API implemented by the library provided by various vendors. The library knows how to communicate with the driver, and an application only needs to know how to exploit the API.

Open Graphics Library (OpenGL [5]) has been a widely used API for rendering with GPUs. Direct3D [10] is a similar API provided by Windows, but it is not cross-platform. Enabled by a huge number of cores, the computation power of GPUs has also been utilized to accelerate computation tasks, especially highly parallelizable ones. This usage scenario is termed as general-purpose computing on GPUs (GPGPU). Currently, Nvidia's Compute Unified Device Architecture (CUDA [6]) is the most popular GPGPU API.

The separation of rendering and computation APIs, however, makes it complicated to develop an application utilizing the GPU for both purposes. Additionally, switching between API libraries inevitably increases the windows of time during which the application is not utilizing the GPU, due to the need of specifying the addresses of data in both APIs and copying data through CPU memory. This is one of the main reasons that leads to the proposal of Vulkan, a next generation GPU API, which unifies the interface for exploiting the computational power of GPUs.

2.3 Vulkan API

Vulkan [4] is a new GPU API proposed in 2016. It is rather low-level compared to previous APIs, allowing explicit memory management by applications. Because the application has knowledge about the task to perform and the resulting memory access pattern, it is expected that it would manage memory in a more efficient way. Vulkan provides a unified GPU interface for both graphic rendering and general purpose computation. This leads to a major advantage that when a program utilizes the GPU for both rendering and computation, there is no need to move data between different libraries. In addition, Vulkan APIs are *thread-safe*, allowing one application making multiple API calls simultaneously. The application thus can potentially take advantage of *parallelism* by having multiple threads accessing the GPU at the same time. This, along with other previously-mentioned characteristics of Vulkan, enables higher performance and flexibility. As a result, Vulkan has been increasingly popular. Moreover, there have been translation layers to Vulkan API calls from OpenGL [11] and Direct3D [12] ones, which means that existing applications can run in an environment with this new API.

2.3.1 Related Concepts in Vulkan

In this section, we briefly introduce Vulkan concepts that are necessary for understanding our GPU virtualization solution. Readers interested in more details may refer to Vulkan reference [13]. In fact, the implementation of our solution does not depends on specific details of the API. As a result, in the future, if a newer API is proposed, the high-level framework of our solution can still be adapted easily.

The application can query available devices (GPUs) and create a logical device (Vulkan context) on a device (GPU). The application requests resource for a logical device. To do so, it needs to specify details about its tasks, including shader programs, graphic pipelines, etc. A task is a command buffer. It records a command buffer by calling `vkBeginCommandBuffer`, a series of APIs indicating the command, and `vkEndCommandBuffer`. It can request the GPU to perform the task by submitting the command buffer to a queue. Note that the command buffer is executed asynchronously.

An application can allocate GPU-accessible memory by Vulkan API calls. GPU-accessible memory can be device-only memory or host-visible memory. Device-only memory can only be accessed by the GPU. It is more efficient for the GPU to access device-only memory than host-visible memory. An application may transfer the frequently-used data to device-only memory via a command buffer including commands to transfer data. Note that applications are responsible for managing the memory by themselves.

Synchronization is necessary among different command buffers or between the CPU and the GPU. When multiple command buffers are submitted, one may need the data produced by the other. The CPU may want the computation result or the rendered image. Semaphores are used for synchronization among commands in different submissions. Applications can specify for which semaphore to wait and which semaphore to signal when command buffers are submitted. Also, fences are used for synchronization between the CPU and the GPU. When command buffers are submitted, a fence may be attached. The application can then call the API to wait until a fence is reached.

If an application is designed to display a rendered image on the screen, it uses the `VK_KHR_surface` extension. This extension includes APIs to manage the resource related to the presentation and submit the presentation request. We refer to the functions in this extension as *present-related APIs* and other functions as *non-present-related APIs*. Where the rendered image is displayed on the screen is called a surface. Note that, a surface may contain multiple images, which are called a swapchain. An application calls the API to create a surface to display the image and a swapchain of images. It requests an available image in the swapchain, submits command buffers to render the image, and submits the image to the present queue for presentation.

2.3.2 Vulkan Loader

The operating system provides Vulkan API by Vulkan loader. This is a user-space library lying between the application and the actual GPU driver provided by the vendor. The actual GPU driver is called an Installable Client Driver (ICD). It exposes the same Vulkan API and some functions for communication with the loader. The Vulkan loader loads all ICDs and enumerates available GPUs for the application. A logical device created by an application is bound to a specific physical GPU. The application calls the API provided by the loader, and the loader forwards it to the corresponding GPU driver. This architecture allows applications to dynamically determine which available GPU device to use at run time. During its execution, an application may switch from using one device to another.

Chapter 3 Related Works

GPU virtualization solutions can be put on a spectrum from purely hardware-based to purely software-based. Hardware-based solutions usually achieve the best performance but allow less flexibility in resource allocation. Also, these solutions are usually bound to specific physical devices. On the other hand, software-based implementations are more portable, with less reliance on specific devices. However, the software layer might limit the performance. Solutions combining hardware- and software-based approaches have also been proposed, where different trade-off decisions are made to favor performance or flexibility. No matter which position a solution lies in, the main design choice is which component(s) along the stack of the physical device, the host, the hypervisor and the guest, to be tailored to accomplish virtualization.

3.1 Hardware Solution

Virtualization Technology for Directed I/O (VT-d) [14] allows an I/O device to be passed-through to a virtual machine. VT-d employs an IOMMU, which automatically translates Guest Physical Address to Host Physical Address for I/O accesses. It works as if the GPU is attached to the virtual machine. The guest OS loads the original GPU driver. In this scenario, performance is near native; however, GPUs cannot be shared among different virtual machines. This solution is popular across cloud providers such as Amazon Elastic Compute Cloud (EC2). Strictly speaking, VT-d itself is not a complete virtualization solution. It allows GPU resource to be utilized by virtual machines, but it does not turn a physical device into a virtualized, shareable resource.

Single Root I/O Virtualization (SR-IOV) [15] is another hardware solution for virtualizing I/O devices. An SR-IOV-enabled device serves as several logical devices, including a Physical Function (PF) and multiple Virtual Functions (VFs), available to the host OS.

The GPU vendor provides both a PF driver and a VF driver. The host OS loads the PF driver, and configures how many VFs it needs. Each VF can be passed-through to a VM. A guest OS loads the VF driver in order to access the GPU. Though this is a full virtualization solution allowing resource sharing, it relies on vendor support; so, it is only available in some GPUs. For example, SR-IOV support is provided through AMD MxGPU [16] and Nvidia GRID [17].

3.2 Mediated Pass-through

Mediated pass-through is a solution with performance close to pass-through that allows resource sharing. By loading a special piece of software, either in the GPU driver or the hypervisor, host OS can provide multiple virtual GPUs having the same hardware interface as the physical GPU. The hypervisor attaches a virtual GPU, which works exactly the same as the physical GPU does, to a guest. Therefore, it is another way to achieve full virtualization. The guest can thus use the native GPU driver, while the resource can be shared among multiple guests. With the help of the special software, the host OS schedules the resource, fulfilling guest isolation and other security requirements. Note that, this solution can be regarded as a software solution, while having a strong reliance on the hardware side. Though hardware support is not required, implementations are highly dependent on hardware interfaces. Consequently, several solutions have been proposed for different devices.

gVirt [18] is a mediated pass-through solution based on Intel Processor Graphics (integrated GPU). The architecture includes a mediator driver in DOM0 (the privileged virtual machine) and a stub in the Zen hypervisor. Address space translation for memory access is performed by the stub, while the mediator driver maintains a scheduler that is responsible for managing GPU context switches. The native GPU driver running in a guest can access the GPU directly for normal rendering commands. Software intervention only occurs on privileged operations, so the performance is only slightly lower than that of pure hardware-based solutions such as VT-d.

GPUvm [19] is a similar solution based on Nvidia's GPUs and CUDA. Since the official drivers are proprietary, an open-source GPU driver is adopted by the guest. Virtualization is provided mainly through special designs implemented in the hypervisor. More specifically, they made the GPU shareable by handling memory space partitioning and GPU time scheduling. With an aim to support computationally intensive jobs, the fairness of scheduling is emphasized to prevent a single VM from occupying all the resources throughout large periods of time. However, since the performance of the open-source driver is limited, the overall performance of this virtualization is limited, despite the potential of this kind of approaches to exploit device-specific knowledge for optimization.

Mediated pass-through is a kind of solution that balances hardware-level performance and software-level configurability. However, this technique relies on the knowledge of hardware interfaces. Unless the specification of the hardware interface is public (which is not the case for most GPUs), GPU vendors are much more suited than other researchers or developers in devising this kind of solutions. That is, if our understanding about the hardware interface is limited, we are prevented from making full use of the performance benefits brought about by adopting a more hardware-dependent solution. Also, due to the necessity for the virtualization solution to interact with the hardware interfaces directly, it takes substantial amount of effort to support newly released hardware, which is especially an overwhelming development load given the rapid product release cycles. Moreover, the dependence on proprietary hardware interfaces and/or drivers makes these solutions hard to be analyzed and compared with each other on a standardized basis. It might also be impractical to propose an “improvement” of an existing mediated pass-through solution, because switching to a newer GPU architecture usually leads to a much larger performance gain.

3.3 API Remoting

Usually, the application only needs functionalities provided by API, so replicating the exact hardware interface is not required. API remoting is a type of solution that allows API functions to be called remotely from a different machine. In the context of virtualization, the guests are the machines making remote API calls. API remoting is usually implemented via para-virtualization, with a customized GPU driver loaded by applications in the guest OS. The host OS provides the API to guests as an interface to the virtual GPU, allowing them to enjoy the hardware acceleration by the physical GPU. Typically, a process running in the host OS handles guest API calls and passes them to the same API provided by the native GPU driver.

The implementation of such a solution is independent of the technical details of the physical GPU. Instead of trying to understand complex hardware interfaces given the usual absence of public specifications, a solution based on API remoting only requires the implementation to follow the API standard. Once the solution is implemented, any GPU compatible with the API can be virtualized. Therefore, VMs utilizing GPUs under an API remoting framework are portable across different physical GPUs. Several solutions have been proposed for both the rendering API and computation API.

VMGL [20] forwards OpenGL API calls to the host through VNC connection. A major drawback is that all OpenGL API calls, including those for presentation, are redirected to the host, so the rendered image is also displayed on the host's screen. Thus, it does not work in the cloud environment. AWS Elastic Graphics [21] provides a special driver, which also forwards OpenGL API calls through TCP/IP. The rendered images are also transmitted for display, thus solving the remote presentation problem, but with the drawback of a limited performance, supporting only 25fps at maximum.

VMware [22] adopted a solution which is slightly different from API remoting. More specifically, SVGA3D, a simplification of Direct3D, was designed. The driver in the guest OS translates a Direct3D API call to a SVGA3D call and passes the call to its emulated GPU. The host OS receives such calls from the emulated GPU, and processes them by either Direct3D or OpenGL.

rCUDA [23] and vCUDA [24] redirect CUDA API calls through TCP/IP and the hypervisor respectively, allowing CUDA to be used in in-host virtual machines and remote machines. qCUDA [25] further improves the bandwidth.

Modern GPU architectures allow multiple "contexts" to access one GPU device at the same time. The functions for creation and management of contexts are provided by the API to applications. API remoting can thus take advantage of the natural context-level isolation, which has been originally designed for accommodating multiple applications on a single machine in a safe fashion. Context-level isolation almost eliminates the need of additional protection mechanisms, which are emphasized in mediated pass-through solutions. Security can be achieved by minimal checking of resource ownership. The simplicity of such an isolation approach also potentially prevents the occurrences of software bugs.

The main limitation of API remoting is the cost of performance introduced by a software forwarding layer. However, it is the most flexible type of solution with respect to hardware variety and evolution. The performance enhancements of a newly-released GPU can directly be exploited by existing VMs as long as the same API is followed. Further, given that rendering images takes time (33ms for 30fps), the potential performance overheads can usually be hidden by pipelining the rendering task and other tasks. With the use of Vulkan, since command buffers are processed asynchronously, such pipelining techniques can be more easily exploited by applications having high performance requirements. That is, the overheads induced by API remoting can become a minor issue with more recent applications adopting Vulkan. We will discuss this in more detail in Section 6.2.2.

3.4 Software Rendering

The GPU API can also be served completely by software, in which case no GPU is used. For instance, OpenGL is implemented in Mesa 3D Graphics Library [26], and Vulkan is implemented by Google SwiftShader [27]. This solution has the worst performance, but it is commonly used in virtualized environment where better GPU virtualization solutions are not available.

Chapter 4 Overview

We implement VAR, Vulkan API remoting. The overall architecture is shown in Figure 4.1. GPU virtualization is provided under a client-server framework, in which the server serves as the virtual GPU and the client is a GPU driver. The driver, which is called VAR driver, is a user-space driver that will be loaded by a Vulkan application in the guest OS. The virtual GPU, or VAR server, is a process running in the host OS, providing Vulkan API to applications through the driver. The driver can communicate with the virtual GPU via TCP/IP or the hypervisor. We discuss the details of communication in Section 5.1.

The main advantages of API remoting over other GPU virtualization solutions include shareable device and dynamic allocation. Multiple guests can utilize the same physical GPU device. We can also dynamically detach a virtual GPU from a guest when it is no longer needed, and assign it to another newly-created guest, thereby potentially increasing the utilization of the underlying hardware. API remoting also allows extensions to other virtualization features such as suspension and taking snapshots because the VAR server in cooperation with the VAR driver knows the complete state of the virtualized GPU provided to the guest. More specifically, all non-present-related API calls are sent to the server (virtual GPU), and all present-related requests are handled within the guest OS. Therefore, to store a virtual machine state, the only thing we need to save, besides the guest OS state, is the state of the VAR server. The VAR server may save the context, including logical device and its resource, and restore it later.

Moreover, API remoting is *vendor-independent*. As long as the GPU driver provides the same API to applications, the details of how it communicates with the physical device does not change how API remoting should be implemented. This offers both *abstraction* and *transparency*, enabling the addition and/or replacement of underlying physical devices without affecting the entire solution, including the VAR server, the

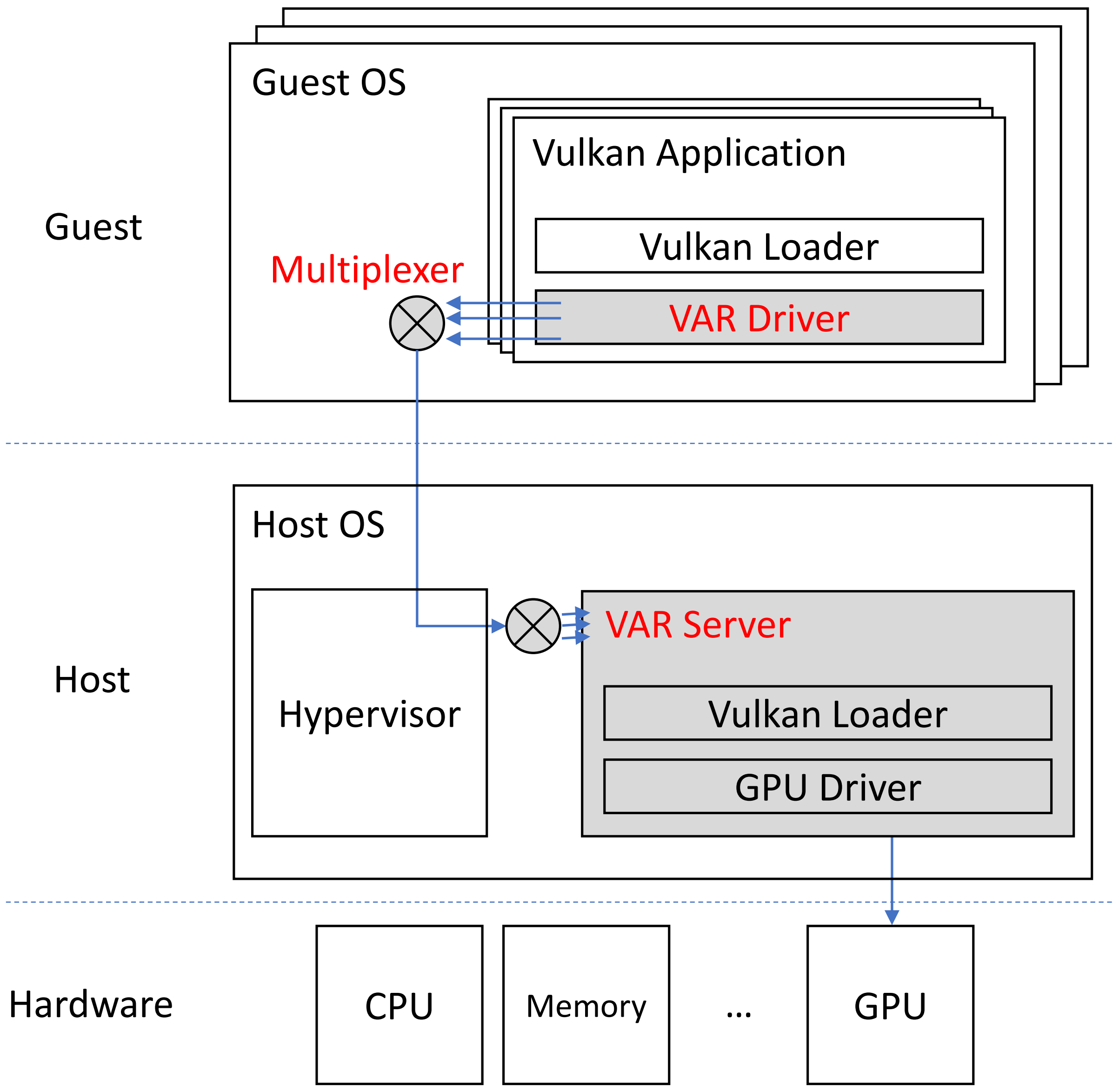


Figure 4.1: The Architecture of Vulkan API Remoting (VAR)

hypervisor, and the existing guests.

Last but not least, unlike some previous approaches such as VMGL which shows the results of a rendering task specified by guest on host's display, preventing use cases where the guest has a display different from the host's, our approach renders images on guest's display.

4.1 VAR Driver (ICD)

To provide access to the virtualized GPU, we implement an ICD, called VAR driver. An application needs to load VAR driver in order to utilize the Vulkan API provided through the virtualized GPU. VAR driver is different from other GPU drivers. Drivers for physical GPUs process Vulkan API calls by native GPU interfaces, while VAR driver processes Vulkan API calls through VAR server. VAR driver is mainly responsible for forwarding API requests to VAR server with some work required to be done in the guest.

The APIs provided by Vulkan can be categorized into two types: *present-related APIs* and *non-present-related APIs*. For the present-related APIs, since presenting in the guest's display cannot be done outside the guest, it needs to be handled by the driver. Therefore, VAR driver includes an implementation of these APIs. We elaborate more on this in Section 5.3. On the other hand, for the non-present-related APIs, VAR driver forwards the API calls to the virtual GPU. More specifically, it serializes an API call, including which function is called as well as the parameters, and sends the byte stream to the virtual GPU via a connection. If the API has any return value, it waits for the virtual GPU returns, deserializes the return value and sends it to the application. If the API does not have any return value, it returns immediately.

Since VAR driver provides all commonly-used Vulkan APIs, including the present-related and non-present-related ones as described above, the application loading it can utilize the virtual GPU in the same way as using a physical GPU device. In other words, no modifications on existing applications are needed. We only require a specialized user-space GPU driver, VAR driver, to be installed in the guest OS. VAR driver, just like normal ICDs provided by GPU vendors, can be discovered by the Vulkan Loader, which facilitates applications to load the required driver for any available device. This solution can be regarded as para-virtualization. It is worth noting that, different from CPU virtualization, there is no need to pursue full virtualization for GPU. The Vulkan API already serves as an abstraction layer that hides the details of the GPU interfaces from different vendors. Typical applications utilize the GPU via the API, instead of interacting with the vendor-specific interface directly. Therefore, it is unnecessary to build the full interface of a physical GPU and expose such details to the guest.

4.2 VAR Server (Virtual GPU)

To provide a virtual GPU to a client application in the guest, VAR server uses the Vulkan API provided on the host to access the physical GPU. VAR server handles the non-present-related API calls sent from the driver one by one by deserializing the request byte stream and calling the corresponding API function with the client-specified parameters. If the function has a return value, VAR server serializes the return value and sends the byte stream back to the driver. The present-related APIs are handled by the driver, so those calls are not sent to VAR server. However, to serve those calls, VAR driver may need to call some non-present-related APIs, such as `vkCreateImage`, which are sent to the server.

Our architecture allows various possibilities for the correspondence between clients (applications) and servers. One server can provide virtualized GPU access to one application, one guest, or even multiple guests. Generally, one server per application is dis-preferred because one server requires one connection accepting API requests. In addition to the potentially large number of connections, another issue is that a server needs to be started and listen to a connection before a guest application can make a request to it. But, we do not know how many applications will be launched before the guest OS boots.

One server per guest provides the nice property of guest isolation brought by virtualization. Every server can be viewed as a virtual GPU device for a guest. A GPU error encountered by one guest does not affect any other guests. Dynamically creating a guest machine equipped with a virtual GPU device is easy to accomplish. To prepare for booting the guest OS, we simply start a new VAR server, which is ready for accepting a connection. In our experiments, this one-guest-one-server implementation is adopted.

On the other hand, one server multiple guests, where the server have a centralized control over multiple virtual GPU devices for multiple guests, allows us to implement some quality-of-service (QoS) policies. For example, we can prioritize the API requests from a certain guest, which is equivalent to allocating more GPU computational power to that guest. However, since different virtual devices are not isolated into different server processes, more sophisticated error handling needs to be implemented in the server, to prevent security issues, given the absence of natural protection from interference of other guests connecting to the same server.

Chapter 5 | Challenges and Solutions

The introduction of a virtualization layer between the guest OS and the physical GPU brings up several challenges in communication and data transfer. Communication overhead may become performance bottleneck, which is an important factor for making design choices. The transfer of data and rendered image between host and guest also needs to be properly handled to achieve the desired GPU features with efficiency. In addition, to provide multi-threading support as Vulkan does, a special design is needed, due to the limitation of the communication channel between the host and the guest provided by the hypervisor. In this chapter, we discuss these challenges and describe how we overcome them by a custom design in VAR driver and VAR server.

5.1 Communication Overhead

While Vulkan is an asynchronous API, most of its functions have return values indicating whether they were successful or if not, what errors have occurred. The overhead incurred by the virtualization layer is one round trip time per API call. Therefore, round trip time should be minimized, or it will become the bottleneck. TCP/IP connection is not preferred because its reliability guarantee incurs unnecessary delays. It is to be noted that TCP/IP was used by VMGL. However, most OpenGL functions do not have any return values, doing so did not result in high communication cost. For our Vulkan API remoting, it is better to go through the hypervisor. To make our design simple, we adopt the virtual serial port in qemu for the connection between the VAR server and VAR driver. Although it seems that this design choice makes our solution dependent on qemu, migration to other hypervisors is fairly feasible because most hypervisors provide similar interfaces.

To set up the virtual GPU before any virtual machine can use it, we need to run

VAR server, which listens to a socket and waits for incoming connections through which API requests will be sent. When the virtual machine boots, the hypervisor connects to the socket listened by the server. The hypervisor exposes a character device to the guest OS with the help of the virtual serial port driver. Applications in the guest OS can establish a connection by opening the character device. However, the character device is a serial port, accepting only one connection at a time; so, it can only support one application with one thread. We explain how we address this restriction in Section 5.4.

5.2 Memory Mapping

In Vulkan, device memory accessible by both application and GPU can be allocated through the `vkAllocateMemory` API. The application needs to map the host-visible memory to its address space after the allocation by `vkMapMemory`, and unmap it before freeing the memory by `vkUnmapMemory`. When the application in the guest requests to map the memory, the memory is mapped by VAR server to server's address space. However, the application in the guest cannot access the address space of the server, which is a process in the host OS. Therefore, we allocate a same size of memory in the application's address space when the memory is mapped, and synchronize it with the memory in the server.

When to synchronize the memory is a design choice worth considering. On the one hand, VAR driver cannot detect modifications from either the GPU or the application. VAR driver has no host-side information (including memory access by the GPU) as it runs in the guest. In addition, there is no mechanism for VAR driver, a user-space driver, to detect the application's modifications on the allocated memory. On the other hand, the choice of the unit of synchronization is also a concern. While using larger units such as hundreds of bytes causes a large fraction of unnecessary data transfers when there are small but frequent modifications, using smaller units such as several bytes also incurs large overheads because, each time a synchronization is performed, we need to send not only the memory content but also some metadata. Given the above considerations, it is impractical to synchronize the memory whenever it is modified by the application or the GPU.

As a result, in our design, memory synchronization is triggered only by a few events. The governing rule is that the latest host-visible memory content only needs to be transferred between server's address space and guest's address space when the memory written by one side is to be read by the other side. We further assume that the GPU and the client do not access the host-visible memory at the same time. This kind of memory access patterns can cause inconsistencies so are usually avoided by typical applications. Memory that has not been mapped cannot be accessed by the application, so only the mapped memory needs to be synchronized. The memory is synchronized to the application when it is mapped, and to the server when it is unmapped. If the memory is still mapped when the application submits tasks to the GPU, it is synchronized by the following rules.

From the server's perspective, while the application may modify the memory at any time, we only need to guarantee that the modification is applied to the server before the GPU accesses the memory. Therefore, we synchronize the modification to the server whenever a command buffer is submitted, which indicates that the GPU is likely to access the host-visible memory subsequently.

From the client's perspective, memory modification made by GPU only needs to be visible after a requested task is completed. Given that the Vulkan API is asynchronous, the application does not know whether the task is completed until it checks with the API. Consequently, we only need to synchronize the memory back to the application when the synchronization-related APIs are called, including `vkDeviceWaitIdle`, `vkQueueWaitIdle`, and `vkWaitFence`. The server and the client collaboratively ensure that host-visible memory mapped in server's address space and application's memory are synchronized *before* returning these API calls to the application.

Although memory synchronization causes bandwidth consumption and latency, this approach works well with most applications. For rendering, typically, large data structures such as object models and texture are not frequently accessed because once loaded, they can be reused over many frames in a certain scene. The memory required to be mapped and synchronized frequently contains only matrices for coordinate transformation and projection from 3D worlds to 2D images. When GPUs are exploited for accelerating computationally intensive programs, the data transfer time only takes up a small fraction of the whole span of the computation time. Thus, memory synchronization does not become a major bottleneck either.

5.3 Display

In the native environment, DRI (Direct Render Infrastructure) allows GPU to put the rendered image on the screen directly. The image is not sent back to the OS, thus not costing additional CPU time and bandwidth between CPU and GPU memory. This is an efficient way to display locally. However, under a host-guest virtualization architecture, if we forward all the presentation request through the server to the GPU, the rendered image will be displayed on host's screen, instead of guest's screen. In addition, we target the cloud environment, where the rendered image should be displayed on the screen of a remote machine; so, DRI cannot be adopted. Note that this was not a challenge for VMGL because the main focus was the desktop environment. It is assumed the VNC client is on the same machine, and as a result, it can still use DRI.

We handle remote displaying as follows. Upon a presentation request, VAR driver sends a command buffer with a fence to the GPU to copy the image to the host. Then, VAR server transfers the image to the guest, and the driver sends the image to the display server. We propose a simple synchronous presentation solution and an improved asynchronous presentation solution, each corresponding to a different implementation of the `vkQueuePresentKHR` API in the driver. It is worth noting that the more sophisticated goal of displaying remotely, instead of locally, has prevented us from using DRI and thus resulted in an inherent limitation on the performance of our solution. As a result, our performance cannot be directly compared to the performance of displaying locally, and the inherent gap should be considered when interpreting the evaluation results.

5.3.1 Synchronous Presentation

In this solution, the driver creates a command buffer, including the task to copy a rendered image to memory, when a surface is created. When `vkQueuePresentKHR` is called, the driver submits the command buffer with a fence. It then waits for the fence to be reached and after that, it maps the memory at which the rendered image is located to guest's address space. When memory mapping is requested, the rendered image is synchronized with the guest. Then, it copies the image to the display server. The main disadvantage of this solution is that, during the rendering and presenting period, the calling thread in the application is blocked. Therefore, the thread cannot continue to prepare the next frame and submit any command buffers until the presentation is completed. Consequently, the GPU will be idle until the image is presented and the thread submits a new command buffer. We thus improve the efficiency by making the presentation process asynchronous, which will be elaborated next.

5.3.2 Asynchronous Presentation

In this asynchronous `vkQueuePresentKHR` design, the driver creates a dedicated presentation thread to apply the rendered image to the display server. A presentation thread is created when a surface is created but the application does not have a presentation thread yet. This thread connects to the server and informs that it is a presentation thread. The server will notify this thread whenever an image is ready to be presented.

When `vkQueuePresentKHR` is called, the driver submits the same command buffer with a fence. It then sends a special command, which is recognizable by the server, indicating that a frame should be presented when the fence is reached. After doing so, the driver returns to the calling thread immediately. The rest of the present task will be handled by the presentation thread. The server maintains a presentation queue, which contains the fences corresponding to previously-submitted command buffers. Since there is a connection between the presentation thread and the server, the server can send the rendered image to the presentation thread when the next fence is reached. The presentation thread then copies the rendered image to the display server.

After forwarding the command buffer and sending the special command, the calling thread can prepare the next frame and submit the next command buffer. Since rendering and presentation do not block the calling thread, the GPU utilization is improved.

Figure 5.1 illustrates the difference between synchronous presentation and asynchronous presentation. In the asynchronous presentation implementation, the performance is improved by overlapping the execution of the rendering task and presentation task. The presentation overhead can almost be hidden when the presentation overhead is less than the rendering latency.

5.4 Multi-threading

Vulkan supports multi-threading, so an application can have more than one thread accessing the API at the same time. For each of the threads in the application (including the presentation thread), VAR driver maintains one connection. When an API is called, the driver detects whether the calling thread has a connection and automatically establishes a new connection if necessary. It also registers a function to be called at thread exit to close the connection.

To allow multiple connections to go through the virtual serial port, we set up one multiplexer in the guest OS and one between the hypervisor and VAR server. We

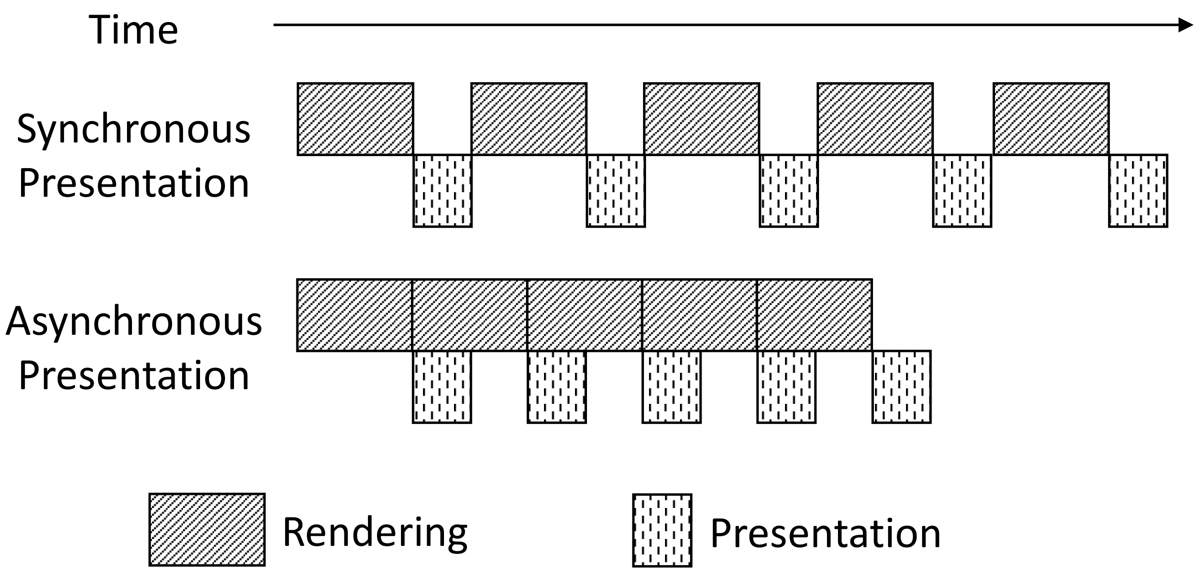


Figure 5.1: The Difference between Synchronous Presentation and Asynchronous Presentation

implemented the multiplexer with system call `select`, though other implementations are also possible.

The guest-side multiplexer, whose setup can be included in the installation process of the driver, opens the virtual serial port character device and listens to a Unix socket which can accept multiple connections. To create a new connection for an application thread, the driver connects to that Unix socket. The host-side multiplexer is set up after the server is started. For each application thread accessing the virtual GPU, there is a corresponding thread in the server responsible for the requests from that application thread. The Vulkan API itself supports multi-threading; so, it can accept calls from multiple threads in the server process. The multiplexer is responsible for passing API requests sent through the virtual serial port to the corresponding threads in the server.

Both multiplexers maintain the ID for each connection in order to forward the API calls from an application thread to the correct thread in the server or vice versa. This enables multiple threads or even multiple applications to access the server. Neither the driver nor the server is aware of the existence of these multiplexers.

While, through the hypervisor, there is only one connection, this does not prevent multiple threads from accessing the Vulkan API concurrently. More specifically, although byte streams are queued for transmission, a thread can make a request while some other threads are waiting for return values.

The effective bandwidth is limited only when multiple threads attempt to send or receive data at the same time. However, the serialized API calls take no more than tens of bytes, so each of them occupies the connection channel for a very short period. Larger byte streams are created when there is a need of memory synchronization, including the transfer of the rendered images. Nevertheless, images are displayed frame by frame, and thus, it is unlikely that multiple application threads would need to receive images at the same time. In sum, when using GPU in the virtualization environment we proposed, application developers do not have to take special care of any implications resulting from our multiplexing mechanism.

Chapter 6 Evaluation

We focus on evaluating the performance of our virtualization solution based on API remoting. We report the native performance as the "upper bound". Theoretically, the hardware-based solutions can reach this performance upper bound. For comparison, we also report the performance of SwiftShader [27], which is based on pure software rendering.

6.1 Experimental Environment

All experiments are conducted on a desktop with Intel i5-9400 CPU and B360 chipset. Two 16GB 2666MHz DDR4 RAM are installed, and we virtualize the integrated GPU.

We evaluate an integrated GPU instead of a discrete GPU on purpose. Our solution submits the same command buffer to the GPU as in the native environment. The GPU performs the same task no matter it is in the virtualized environment or not. Only the communication and presentation overheads, which are not handled by the GPU, affect the performance. With a less powerful GPU, there will be a wider range of native performance on different workloads, giving us a higher resolution for observing the virtualization performance in different situations. Furthermore, since the communication and presentation overheads are GPU-independent, the result can be applied to other GPUs.

Both the host and the guest run Linux Ubuntu 20.04 LTS. We use qemu as the hypervisor with KVM enabled. The desktop environment is Gnome shell and X server, and the virtual machine has 4 virtual CPUs and a 4GB memory. We use the VMWare SVGA-II compatible VGA card as qemu's manual page suggests, which enables us to perform evaluations with high resolutions. Table 6.1 gives the hardware and software configuration used in our experimental evaluations.

Hardware	
CPU	Intel i5-9400
Chipset	Intel B360
GPU	Intel UHD Graphics 630
RAM	2×16GB 2666MHz DDR4
Software	
OS	Ubuntu 20.04 LTS
Hypervisor	qemu 4.2.0
Display Server	Xorg X Server 1.20.8
Desktop	Gnome 3.36.1
Vulkan	1.2.131

Table 6.1: Hardware and Software Configuration

6.2 Benchmarks

We report the performance of our solution on workloads involving both computation and rendering. The *virtualization performance* is defined to be the relative performance in a VM compared to the native performance. We run the benchmarks in the native environment and in the virtual machine, with both our solution and SwiftShader.

6.2.1 Workloads

Graphic acceleration performance is measured by fps. We evaluate 12 workloads from Vulkan C++ examples and demos [28], and run each workload for 60 seconds. While all these workloads involve rendering, they contain a wide range of computation loads. Some of our workloads render based on the results of computation; so, they have additional computational load beside the rendering task itself. Table 6.2 lists the selected workloads and the corresponding abbreviations used in the results presented.

The Vulkan examples are just for demonstration purpose and are not optimized for performance. One major drawback in programs in the example repository lies in the sequential workflow of rendering and presentation. A program submits the rendering command buffer, calls the present function, and then waits until the presentation task is completed. During the presentation, no command buffer is submitted, and consequently, the GPU computation power is not fully utilized. However, since Vulkan is an asynchronous API, a program does not need to be blocked after calling the present function. It is expected that this advantage of Vulkan will be leveraged by performance-critical real-world applications. In order to perform more realistic evaluations, we modify

Workload	Abbreviation
Advanced	
High dynamic range	HDR
Cascaded shadow mapping	SMC
Omnidirectional shadow mapping	SMO
Performance	
Indirect drawing	ID
Physically Based Rendering	
PBR image based lighting	PI
Deferred	
Deferred multi sampling	DMS
Deferred shading shadow mapping	DS
Screen space ambient occlusion	SSAO
Compute Shader	
GPU particle system	CP
N-body simulation	CN
Ray tracing	CR
Cull and LOD	CC

Table 6.2: Selected Workloads

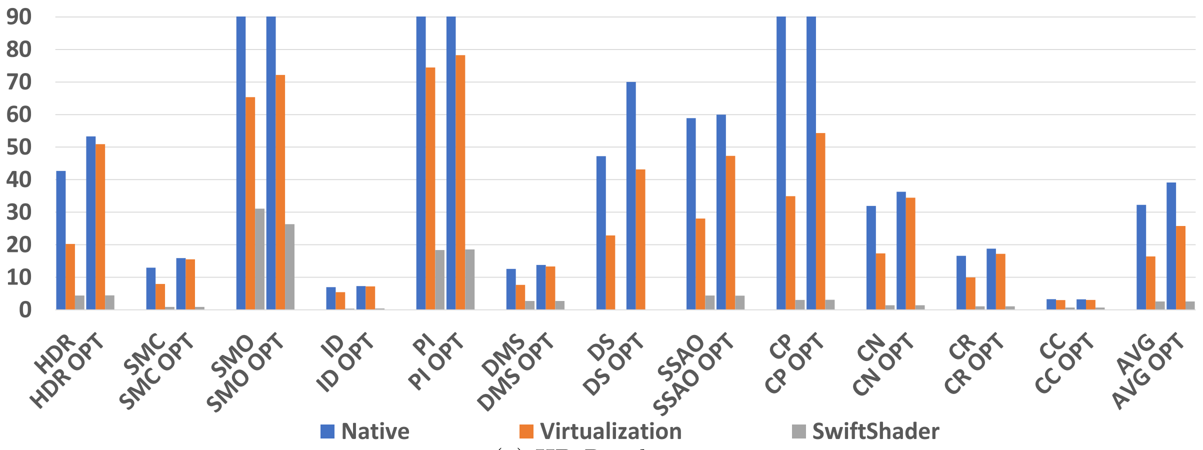
the examples as explained below.

6.2.2 Pipeline-based Optimization

By our modification, a program can submit up to two frames to be rendered by GPU. While a frame is being rendered, the program submits the command buffer for rendering the next frame. The GPU starts rendering the next frame as soon as the rendering task for the previous frame is finished, while the previously-rendered frame is being presented. By doing so, the utilization of GPU is improved, and the overhead is reduced to a negligible level if the rendering time per frame is longer than the presentation time per frame.

6.3 Discussion

The performance of the workloads under different schemes are plotted in Figure 6.1. Both original workloads and their optimized versions are evaluated in HD resolution and Full HD resolution. The rightmost ones (AVG and AVG OPT) show the geometric mean


(a) HD Resolution

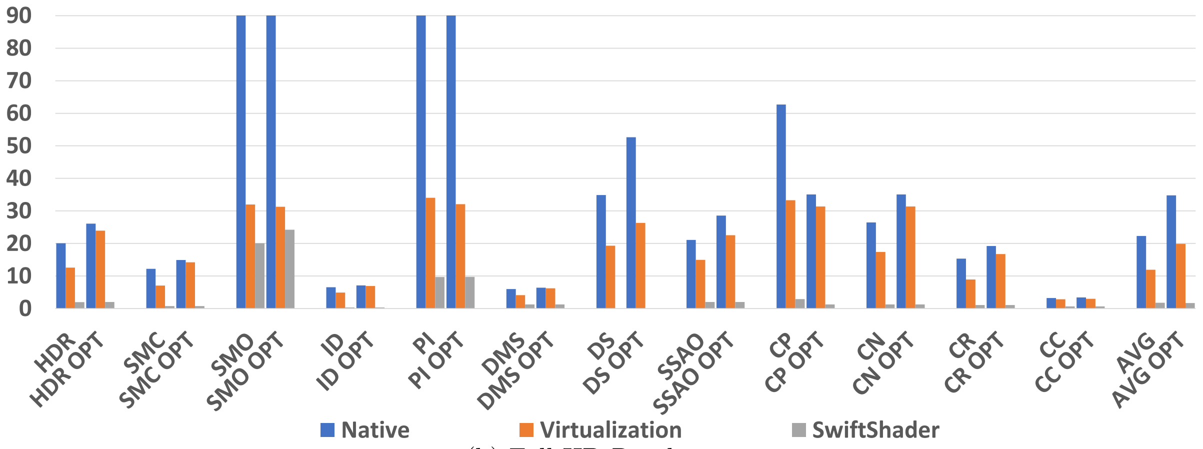

(b) Full HD Resolution

Figure 6.1: Performance on Workloads in FPS

over all selected workloads. On all workloads, our solution outperforms SwiftShader significantly. It can be seen that, the performance in the VM ranges from 12% to nearly full native performance (98%), varying from workload to workload. According to the results in the native environment, the workloads can be categorized into two types: low-fps and high-fps. Rendering time per frame is longer for low-fps workloads and shorter for high-fps ones. In other words, the low-fps workloads refer to those that require more computation power to render a frame.

For the original versions of the low-fps workloads, the virtualization performance ranges between 47% and 98%. Communication and presentation overheads account for the performance drop compared to the native case. The overhead can mostly be hidden by pipelining. Therefore, the performance is improved by 78%–98% by such optimization. The benefits of the pipeline-based implementation are more pronounced in the virtualized

Workload	Native	1 VM	2 VMs
SMO	172	65	43
SMO OPT	272	72	41
PI	341	74	47
PI OPT	613	78	48

Table 6.3: Performance for Multiple VMs in FPS per Application

environment, because the presentation takes longer, for the large image to be transmitted through the hypervisor.

For high-fps workloads, the communication and presentation become the main bottleneck. We can only achieve between 31 and 34 fps for Full HD and 54 to 78 fps for HD in the VM. The performance gain from the pipeline-based implementation is limited, primarily because, for each frame, the rendering latency usually is not long enough to hide the latency induced by communication. Note that the virtualization performance should not be compared with the native performance directly in this case. Presenting images on guest's screen requires rendered images to be transferred from the GPU to host memory, resulting in the bottleneck at image transfer. This result is satisfactory in a cloud environment, where the client accesses the virtual machine remotely, since the screen is streamed to the client through TCP/IP with a much larger communication overhead.

We further evaluate the high-fps workloads with HD resolution in 2 VMs simultaneously. The result in Table 6.3 shows that the total performance is improved since the GPU utilization is increased. While the program in one VM is not utilizing the GPU due to presentation latency, the GPU can be utilized by other VMs.

In conclusion, our solution is fairly effective in general. It achieves high virtualization performance when the computation load per frame is high. Most latency is induced by computation; so, the low overhead incurred by our solution does not lead to any performance bottleneck. When the computation load is not heavy, our solution still provides satisfactory frame rates. In either case, our solution achieves decent results, indicating that it is a practical option.

Chapter 7 Concluding Remarks and Planned Future Work

In this thesis, we propose a GPU virtualization solution based on Vulkan API remoting. Compared to the hardware-based solutions, the advantages of API remoting include resource allocation flexibility and portability. The implementation is API-specific but independent of the underlying physical device. Our solution is composed of a VAR server running on the host and a VAR driver loaded by an application in a guest. Presentation-related API calls are handled by the driver. All other commands are forwarded to the host GPU without incurring additional commands (or computational overhead).

While Vulkan improves previous GPU APIs by an asynchronous command processing model and explicit memory management available to the application, these properties pose challenges on developing an API remoting solution for it. By addressing communication overhead, memory mapping, remote presentation and multi-threading, our solution provides "exactly the same Vulkan API" to a guest application. Experimental results show that our solution has performance closer to native on heavier workloads and satisfactory frame rates for others. Since the virtualization overhead mainly arises from communication and presentation – not taking additional GPU computation power – the GPU can be fully utilized when being shared among multiple guests. Our solution is suitable for GPU virtualization in both desktop and cloud environments, with negligible overheads, especially in the latter because the latency is dominated by streaming rendered images over a network.

Our current implementation has included commonly-used APIs. In the future, we plan to complete the whole set of standard Vulkan APIs. In addition, we are also planning to follow the same paradigm to implement Vulkan extensions such as the ray tracing API. On the other hand, we would like to extend our support of different display servers, so that our solutions can be utilized in other operating systems as well.

Memory synchronization in our solution can be further improved by mapping the address spaces of VAR server and guest application to shared memory pages. By doing so, one can allow the application to read the rendered images and send to the display server in the guest. The major bottleneck of transferring images from the host to the guest through the hypervisor can be eliminated.

In sum, our virtualization solution already provides fairly comprehensive Vulkan API support and can serve as the basis for further extensions. With the increasing popularity of Vulkan, our solution has the potential to be widely deployed. We will make the source code publicly available.

Bibliography

[1] Amazon, "Amazon Web Services," `https://aws.amazon.com/`.

[2] Google, "Cloud Computing Services | Google Cloud," `https://cloud.google.com/`.

[3] Microsoft, "Cloud Computing Services | Microsoft Azure," `https://azure.microsoft.com/`.

[4] Khronos Group, "Vulkan," `https://www.khronos.org/vulkan/`.

[5] ———, "OpenGL," `https://www.opengl.org/`.

[6] Nvidia, "About CUDA," `https://developer.nvidia.com/about-cuda`.

[7] QEMU, "QEMU," `https://www.qemu.org/`.

[8] Russell, R. (2008) "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, **42**(5), pp. 95–103.

[9] Peng, B., H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan (2018) "MDev-NVMe: a NVMe storage virtualization solution with mediated pass-through," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 665–676.

[10] Microsoft, "Direct3D," `https://docs.microsoft.com/en-us/windows/win32/direct3d`.

[11] Think-Silicon, "GLOVE (GL Over Vulkan)," `https://github.com/Think-Silicon/GLOVE`.

[12] doitsujin, "Vulkan-based implementation of D3D9, D3D10 and D3D11 for Linux / Wine," `https://github.com/doitsujin/dxvk`.

[13] Khronos Group, "Vulkan Specification," `https://www.khronos.org/registry/vulkan/specs/1.2-khr-extensions/html/vkspec.html`.

[14] Intel, "Intel Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices," `https://software.intel.com/en-us/articles/intel-virtualization-`

technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices.

[15] Intel LAN Access Division, "PCI-SIG SR-IOV Primer," https://www.intel.com/content/www/us/en/pci-express%2Fpci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[16] AMD, "GPU Virtualization Solution," https://www.amd.com/en/graphics/workstation-virtual-graphics.

[17] Nvidia, "Virtual GPU Software Documentation," https://docs.nvidia.com/grid/5.0/grid-vgpu-user-guide/index.html.

[18] Tian, K., Y. Dong, and D. Cowperthwaite (2014) "A Full GPU Virtualization Solution with Mediated Pass-Through," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 121–132.

[19] Suzuki, Y., S. Kato, H. Yamada, and K. Kono (2014) "GPUvm: Why not virtualizing GPUs at the hypervisor?" in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 109–120.

[20] Lagar-Cavilla, H. A., N. Tolia, M. Satyanarayanan, and E. De Lara (2007) "VMM-independent graphics acceleration," in *Proceedings of the 3rd international conference on Virtual execution environments*, pp. 33–43.

[21] Amazon, "Amazon Elastic Graphics," https://aws.amazon.com/ec2/elastic-graphics/.

[22] Dowty, M. and J. Sugerman (2009) "GPU virtualization on VMware's hosted I/O architecture," *ACM SIGOPS Operating Systems Review*, **43**(3), pp. 73–82.

[23] Duato, J., A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí (2010) "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing & Simulation*, IEEE, pp. 224–231.

[24] Shi, L., H. Chen, J. Sun, and K. Li (2011) "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, **61**(6), pp. 804–816.

[25] Lin, Y.-S., C.-Y. Lin, C.-R. Lee, and Y.-C. Chung (2019) "qCUDA: GPGPU Virtualization for High Bandwidth Efficiency," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, pp. 95–102.

[26] Mesa, "The Mesa 3D Graphics Library," https://www.mesa3d.org/intro.html.

[27] Google, "SwiftShader," https://github.com/google/swiftshader.

[28] SaschaWillems, “Vulkan C++ exmaples and demos,” `https://github.com/SaschaWillems/Vulkan`.