國立交通大學

資訊科學與工程研究所

碩 士 論 文

在 KVM 虛擬機器中支援 OpenCL 圖形加速裝置

Enabling OpenCL Support for GPGPU in Kernel-based Virtual Machine

研 究 生：田璨榮

指導教授：游逸平　教授

中 華 民 國 一 百 年 九 月

在 KVM 虛擬機器中支援 OpenCL 圖形加速裝置
Enabling OpenCL Support for GPGPU in Kernel-based Virtual Machine

研 究 生：田璨榮　　Student：Tsan-Rong Tian

指導教授：游逸平 博士　　Advisor：Dr. Yi-Ping You

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

September 2011

Hsinchu, Taiwan, Republic of China

中 華 民 國 一 百 年 九 月

在 KVM 虛擬機器中支援 OpenCL 圖形加速裝置

學生：田璨榮　　　　　　　　　　　　指導教授：游逸平 博士

國立交通大學資訊科學與工程所碩士班

摘　　要

現今高效能運算領域中，以異質多核心系統進行平行運算已成為一大重要發展趨勢，妥善運用不同種類核心之計算能力優勢，可大幅提高運算效能。OpenCL 即為因應愈來愈普及的異質多核心運算環境所提出的程式開發模型，幫助程式開發人員撰寫有效率、具移植性的異質多核心程式，提升計算效能，但目前於系統虛擬化環境中並不支援OpenCL，無法以系統虛擬化幫助進行更佳的OpenCL運算資源管理。在本篇論文中，我們以KVM虛擬機器為基礎提出了一個OpenCL虛擬化架構，並以API Remoting的方式達成OpenCL運算資源多工。本論文的OpenCL虛擬化架構分為：(一)適用於客戶虛擬機器(guest virtual machine)環境下的OpenCL函式庫，負責包裝OpenCL函式請求與回覆。(二)Virtio-CL，為一虛擬裝置，負責客戶虛擬機器與虛擬機器管理者(hypervisor)之間的資料傳輸。(三)一個新的執行緒(thread)，其負責真正執行OpenCL函式，稱為CL執行緒。由於API Remoting的特性，OpenCL程式在OpenCL主端與客戶端間資料傳輸量直接影響虛擬化負擔。在實驗中發現，我們選用的OpenCL運算密集型(device-intensive)測試程式僅有少量虛擬化負擔，平均為6.4%，且當客戶虛擬機器數量增加時，虛擬化負擔增幅不大，代表我們的虛擬化架構能實現有效的OpenCL運算資源管理。

關鍵詞：

OpenCL，系統虛擬化，KVM，圖形處理器虛擬化，API Remoting，Virtio

Enabling OpenCL Supports for GPGPU in Kernel-based Virtual Machine

Student：Tsan-Rong Tian　　　　Advisors：Dr. Yi-Ping You

Institute of Computer Science and Engineering

National Chiao Tung University

ABSTRACT

Heterogeneous multi-core programming has become more and more important, and OpenCL, an open industrial standard for parallel programming, provides a uniform programming model for programmers to write efficient, portable code for heterogeneous compute devices. However, OpenCL is not supported in system virtualization environment, which explores more opportunities of better resource utilization. In this thesis we propose an OpenCL virtualization framework based on Kernel-based Virtual Machine (KVM) with API Remoting to enable multiplexing of multiple guest virtual machines (guest VMs) over the underlying OpenCL resources. The framework comprises three major components: an OpenCL library implementation in guest VMs for packing/unpacking OpenCL requests/responses, a virtual device, called Virtio-CL, which is responsible for the communication between guest VMs and the hypervisor, and a new thread, called CL thread, which is dedicated for the OpenCL API invocation. Although the overhead of the proposed virtualization framework is directly affected by the amount of data to be transferred between the OpenCL host and devices because of the primitive nature of API Remoting, the experiments demonstrate that the virtualization framework has only little virtualization overhead (6.4% on average) for common device-intensive OpenCL programs and performs well when the number of guest VMs involved in the system increases, which directly infers the effective resource utilization of OpenCL devices of the framework.

Keywords:
OpenCL, System virtualization, Kernel-based Virtual Machine, GPU virtualization, API Remoting, Virtio.

致 謝

首先，誠摯感謝我的指導老師游逸平教授的指導，老師嚴謹的治學方法與細心的指導及啟發，讓我在學術研究與專業能力均有所成長，更讓我學到一絲不苟的態度，無論做人處事，老師都是令我尊敬的模範。感謝 Group 中楊武教授、徐慰中教授、單智君教授在研究方面的指引與建議，指點我在研究上疏忽的方向，使本研究得以順利完成。感謝口試委員鍾葉青教授對論文的建議，給我不同的思考與研究方向，使本研究更為完整。教授們追根究底的研究精神是我學習的榜樣。

感謝實驗室學長世融、柏瑲二年來的照顧。感謝同學羽軒、深弘的相互鼓勵與協助。感謝學弟妹們：翰融、聖偉、思捷、睦昂，有你們為實驗室帶來歡笑與活力。很高興與你們共事，使我的碩士生活更加豐富。

最重要的，我要感謝父母的關心與鼓勵，有你們的支持，讓我得以無憂慮的專注於研究，順利完成碩士學業。也感謝一路走來幫助與鼓勵我的師長朋友們，讓我更堅定前行。

誌於 辛卯季夏 風城交大

璨榮

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, heterogeneous multi-core programming has become more and more important. Programmers can leverage the computing power of different heterogeneous devices and make good use of the specific computation strength of each device to get better performance. Some programming models are proposed to provide a unified layer of heterogeneous multi-core programming for hiding the hardware-related details such as different memory organizations and synchronization between host and devices. Two well-known programming models, CUDA [2] and OpenCL [6], both are designed as host-device model. CUDA is proposed by NVIDIA, and they make CPUs as a host and GPUs as devices. OpenCL is proposed by Khronous group, and it is supported by the industry to become the standard of heterogeneous multi-core programming. OpenCL makes CPUs as host, but it can support lots of different architectures such as CPUs, GPUs, DSPs, etc as devices. These programming models can help programmers focus on high-level design and implementation.

Unified heterogeneous programming model helps programmers simplify the development process, but resource management issues still remains since an user may mot always occupys the resource of heterogeneous devices. To obtain better resource utilization, system virtualization is a good solution. System virtualization can provide an environment that supports multiple operating systems (OS) to execute simultaneously and perform hardware resource management to share physical resources between OSes for the better resource utilization. Performance of CPU and memory virtualization has been improved dramatically in the past few years thanks to the hardware assisted approach, but input/output (I/O) ability and performance are still the weakness point in system virtualization. Virtualizing the GPU devices has lots of difficulties due to the closed and generation-changed architecture, and thus the guest OSes are limited to get GPU resources, including general purpose computing on graphic processing unit (GPGPU). As for now, there is no standard solution for using CUDA/OpenCL in system virtualization, and there has been a little research [15] [28] about enabling CUDA supports.

Because of the quickly gaining demand of heterogeneous multi-core programming and the need of better resource utilization for heterogeneous devices, we believe it is useful to combine the OpenCL programming model with system virtualization. The benefit of enabling OpenCL support in system virtualization is to provide an automatic resource management in the hypervisor, not only letting programmers not worry about the resource utilization issues but ensuring fair resource allocation. Such approach also acquires other benefits such as cost reducing and easier migration of executing environments due to the management scheme provided by virtual machine. Hence, we are going to build an OpenCL support in system virtual machine (system VM) and to prepare an environment for further studies on how to share hardware resources of heterogeneous devices fairly and efficiency.

1.2 Thesis Overview

In this thesis we present our methodologies for enabling OpenCL support in system virtual machine. To provide the ability of running OpenCL programs in a virtualized environment, we develop an OpenCL virtualization framework in the hypervisor and build an VM-specific OpenCL runtime. We will present Virtio-CL, an OpenCL virtualization implementation in Kernel-based Virtual Machine (KVM) [21]. We will evaluate the semantic-correctness and effectiveness of our approach by comparing with native execution environments.

The remainder of this thesis is organized as follows. Chapter 2 introduces the basic of OpenCL and system virtualization, and Chapter 3 introduces the system design and implementation of OpenCL support in KVM. The performance evaluation are presented in Chapter 4. Chapter 5 introduces the related work. The conclusions and future work are drawn in Chapter 6.

Chapter 2

Background

In this chapter, basic materials for comprehending the thesis are explained, including the fundamentals of OpenCL, system virtualization overview, I/O virtualization, and the underlying framework of this work, Kernel-based Virtual Machine (KVM). For I/O virtualization, we will focus on the current mechanisms about how to virtualize GPU functionalities. The related work will be discussed in Chapter 5.

2.1 Introduction to OpenCL

OpenCL (Open Computing Language) is an open industry standard for general-purpose parallel programming of heterogeneous systems. OpenCL is a framework that includes a language, API, libraries and a runtime system to provide a unified programming environment for software developers to leverage the computing power of heterogeneous processing devices such as CPUs, GPUs, DSPs, and Cell/B.E. processors. Using OpenCL, programmers can write portable code efficiently with the hardware-related details being exposed by the OpenCL runtime environment. The Khronos group proposed OpenCL 1.0 specification in December, 2008. The current version, OpenCL 1.1 [18], was announced in June, 2010.

2.1.1 OpenCL Hierarchy Models

The architecture of OpenCL is divided into four hierarchy models: platform model, memory model, execution model, and programming model. In this section, we will briefly introduce the definition and relation between each hierarchy. Detail information of OpenCL can be obtained from the OpenCL Specification [18].

2.1.2 Platform Model

Figure 2.1 defines the platform model of OpenCL. The model includes a host connected to one or more OpenCL devices. An OpenCL device is consisted of one or more compute units (CUs) which are further composed of one or more processing elements (PEs). The processing elements are the smallest unit of computation on a device.

An OpenCL application is designed under such the host-device model. The application dispatches **jobs** (the workloads that will be processed by devices) from the host to devices, and the jobs are executed by processing elements within a device. The computation result will be transferred back to the host after execution completed. The processing elements within a compute element execute a single stream of instructions in a single instruction, multiple data (SIMD) or a single program, multiple data (SPMD) manner. SIMD and SPMD which are related to OpenCL Programming Model will be discussed in Section 2.1.5.

2.1.3 Execution Model

Execution of an OpenCL program is composed of two parts: the host part that executes on the host and the kernels that execute on one or more OpenCL devices. The host defines a context for the execution of kernels and creates command-queues to operate the execution. When the host assigns a kernel to a specific device, an index space is defined to help the kernel locate the resources of the device. We will introduce these terms in the following paragraphs.

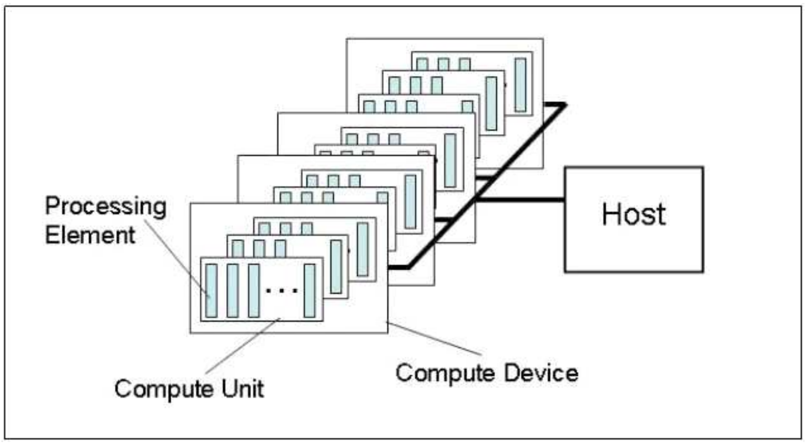


Figure 2.1: OpenCL platform model (adapted from [18])

Context

The host defines a context for the execution of kernels. The context includes resources such as devices, kernels, program objects, and memory objects. Devices are the collection of objects of OpenCL devices. Kernels are the OpenCL functions that run on OpenCL devices. Program objects reference to the kernel program source code or executables. Memory objects are visible to both host and the OpenCL devices. Data manipulation by host and OpenCL devices is done under the operation of memory objects. The context is created and manipulated using functions from the OpenCL API by the host.

Command Queue

The host creates command-queues to operate the execution of the kernels. Types of commands include kernel execution commands, memory commands, and synchronization commands. The host inserts commands into the command-queue which will be then scheduled by the OpenCL runtime. Commands relative to each other are in either in-order execution of out-of-order execution mode. It is possible to define multiple command-queues within a single context. These queues can execute commands concurrently, so prorammers should use synchronization commands to make sure the correctness of concurrent execution of multiple kernels.

Index Space

The index space supported in OpenCL is divided into a three-level hierarchy: **NDRange**, **work-group**, and **work-item**. An NDRange is an N-dimensional index space, where N is one, two, or three. An NDRange is composed of work-groups. Each work-group contains several work-items, which are the most fundamental executing element of a kernel. The work-items in a given work group execute concurrently on the processing elements of a single compute unit.

A work-item is identified by a unique global identifier (ID). Each Work-group is assigned a unique work-group ID and each work-item is assigned a unique local ID within a work-group. Work-groups are assigned IDs using the similar approach to that used for work-item global IDs. According to these identifiers, work-items can inentify themselves based on the global ID or by a combination of its local ID and work-group ID.

An example of NDRange index space relationships adapted from OpenCL Specification is showed in Figure 2.2. This is a two-dimensional index space in which we define the size of NDRange (G_x, G_y), the size of each work-groups (S_x, S_y) and the global ID offset (F_x, F_y). The total number of work-groups is the product of G_x and G_y. The size of each work-groups is the product of S_x and S_y. The global ID (g_x, g_y) is defined as the combination of the work-group ID (w_x, w_y), the local ID (s_x, s_y) and the global ID offset (F_x, F_y):

$$(g_x, g_y) = (w_x \times S_x + s_x + F_x, w_y \times S_y + s_y + F_y)$$

The number of work-groups can be computed as:

$$(W_x, W_y) = (G_x/S_x, G_y/S_y)$$

The work-group ID can be computed by a global ID and the work-group size as:

$$(w_x, w_y) = ((g_x - s_x - F_x)/S_x, (g_y - s_y - F_y)/S_y)$$

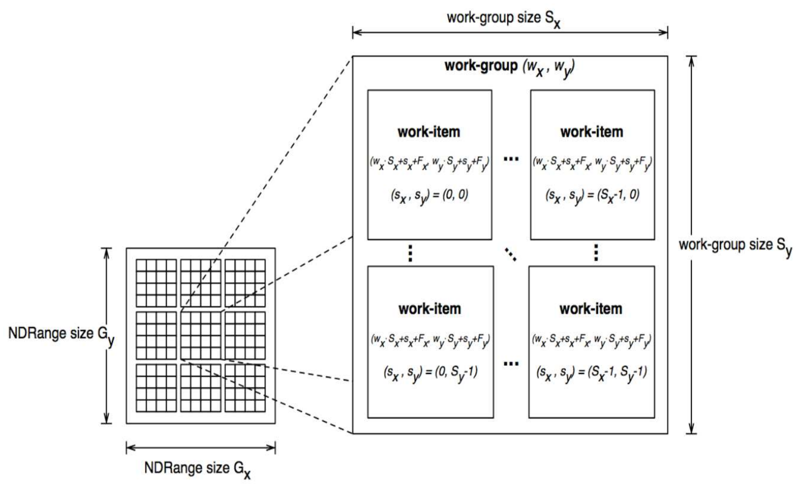

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 2.2: An example of NDRange index space (adapted from [18]).

A wide range of programming models can be mapped onto this execution model. OpenCL explicitly supports data- and task-parallel programming models.

2.1.4 Memory Model

There are four distinct memory regions: global, constant, local and private memory. Global memory can be used by all work-items, constant memory is a region of global memory that remains constant during kernel execution, local memory can be shared by all work-items in a work-group, and private memory can only be adapted by a single work-item. Table 2.1 describes the access abilities and limites among the host and kernels.

The host uses OpenCL APIs to create memory objects in global memory and to enqueue memory commands for manipulating these memory objects. Data transfers between the host and devices are done by explicitly copying data or by mapping and unmapping regions of a memory object. The relationship between memory regions and the platform model are described in Figure 2.3.

OpenCL uses relaxed consistency memory model. There are no guarantees of memory consistency between different work-groups. The consistency for memory objects shared between enqueued commands is promised at a synchronous point.

Table 2.1: Memory region—allocation and memory access capabilities (adapted from [18]).

	Global	Constant	Local	Private
Host	Dynamic allocation Read/Write access	Dynamic allocation Read/Write access	Dynamic allocation No access	Dynamic allocation No access
Kernel	No allocation Read/Write access	Static allocation Read-only access	Static allocation Read/Write access	Static allocation Read/Write access

2.1.5 Programming Model

The OpenCL execution model supports data-parallel and task-parallel programming models. Data-parallel programming model means a sequence of instructions being applied to multiple elements of data. The index space defined in the OpenCL execution model is used to indicate a work-item where to fetch the data for the computation. Programmers can define the total number of work-items along with the number of work-items to form a work-group or only the total number of work-items to specify how to access data by a unique work-item.

The OpenCL task-parallel programming model defines a model that a single instance of a kernel is executed independent of any index space. Users can exploit parallelism via the following three methods: using vector data types implemented by the device, enqueueing multiple tasks, or enqueueing native kernels developed by a programming model orthogonal to OpenCL.

The synchronization occurs in OpenCL in two situations. For work-items in a single work-group, a work-group barrier is useful to permit the consistency. For commands in the same context but enqueued in different command-queues, programmers can use command-queue barrier and/or events to perform synchronization.

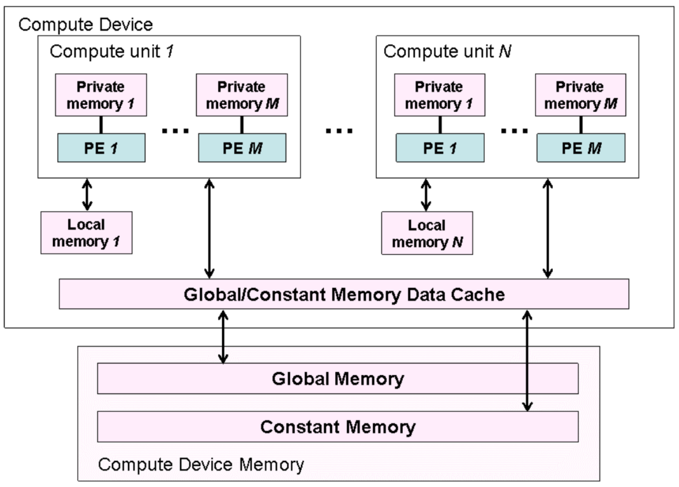


Figure 2.3: Conceptual OpenCL device architecture with processing elements, compute units and devices (adapted from [18]).

2.2 System Virtual Machine

System virtualization supports multiple operating systems (OSes) executing on a single hardware platform simultaneously and shares the hardware resources between each OS. System VMs ensure the benefits such as work isolation, server consolidation, operating debugging, dynamic load balancing, etc. Thanks to the evolution of multi-core CPUs, system virtualization has become more and more useful.

In order to support multiple OSes (called guest OSes) in a system virtual machine (System VM) running on a single hardware, a hypervisor (also called virtual machine monitor, VMM) is responsible for managing and allocating the underlying hardware resources between guest OSes and promises that each guest OS won't be affected by each other. Resource sharing of a system VM is in time-sharing manners similar to the time-sharing mechanisms in OS. When the control switches from one guest to another, the hypervisor has to save the current guest system state and restore the system state of the incoming guest. The guest system state contains program counter (PC), general-purpose registers, control registers, etc. In this work we will focus on system virtualization on Intel x86 (IA-32) architectures.

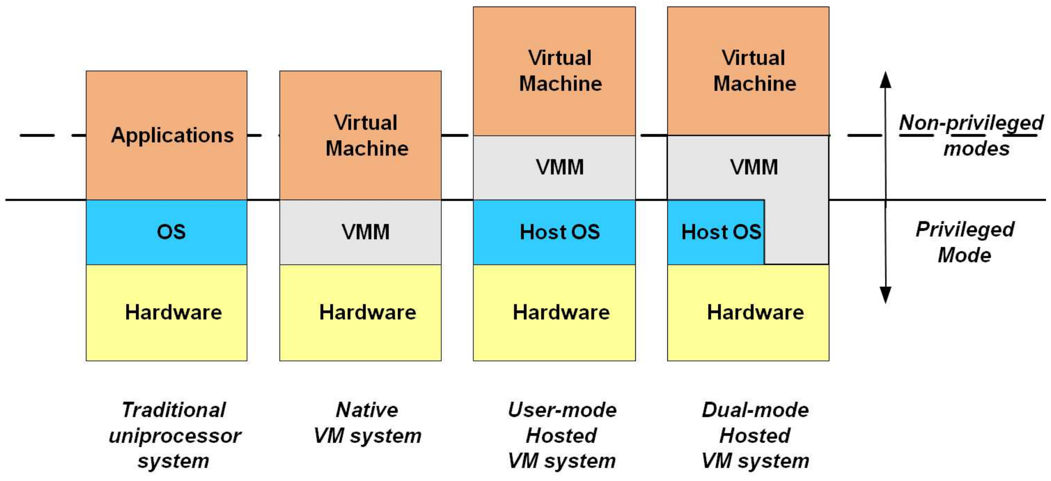


Figure 2.4: Native and hosted VM systems (adapted from [23]).

Native and Hosted Virtual Machines

Traditionally system virtual machines can be divided into three categories: native VMs, user-mode hosted VMs and dual-mode hosted VMs as shown in Figure 2.4 [23]. In a native VM, only the hypervisor executes in the highest privilege level defined by the system architecture. In a user-mode hosted VM, the hypervisor is constructed upon a host platform that running an existing OS called host OS. The hypervisor can take advantage of the functionalities, such as device drivers and memory management, which are provided by host OS, and thus the implementation is simplified. However, the combination of native and hosted VMs can achieve better performance than hosted VMs and also can adapt the features provided by the existing OS. This can often be achieved by entending the host OS with extra kernel modules or device drivers. Such a system is called a dual-mode hosted VM.

2.3 CPU Virtualization

To virtualize a CPU and share the resources of the processor, the hypervisor needs to intercept and handle the execution of special instructions of guest OSes. The types of instructions in a hardware architecture are divided into innocuous instructions and sensitive instructions [26]. Sensitive instructions are those that should be intercepted and handled by the hypervisor when they are executed by a guest OS while innocuous instructions are those other than sensitive instructions. Sensitive instructions can be further divided into control- and behavior-sensitive. Control-sensitive instructions are those that provide control of resources while behavior-sensitive instructions are those whose behavior or results depend on the configuration of resources. The control should be transferred to the hypervisor when a guest system executes sensitive instuctions to avoid directly accessing the resources or change the system configuration of other guests. Such mechanism is called trap and emulation.

Popek and Goldberg defined a set of conditions sufficient for a computer architecture to support system virtualization in 1974 [26]. They intorduced privileged instructions which are defined as those that trap if the machine is in user mode and do not trap if the machine is in privileged mode. In Popek and Goldberg's theories, an effctive system VM is constructed if the set of sensitive instructions is a subset of the set of privileged instructions in a specific hardware architecture. If a hardware architecture meets the condition, the architecture can be fully virtualized. The relationship between privileged and sensitive instructions is illustrated in Figure 2.5. In x86 architectures, there is a set of instructions called critical instructions which are sensitive instructions but not belong to privileged instructions. For example, `POPF` instruction pops the flag registers from a stack held in memory, but the interrupt-enable flag is not affected since it can only be modified in privilieged mode. Such instructions in x86 architectures can not be trapped and emulated efficiently in a system virtualization environment.

There are three methods to handle critical instructions in x86 architectures: software emulation, para-virtualization and hardware-assisted virtualization. The three methods will be discussed as follows.

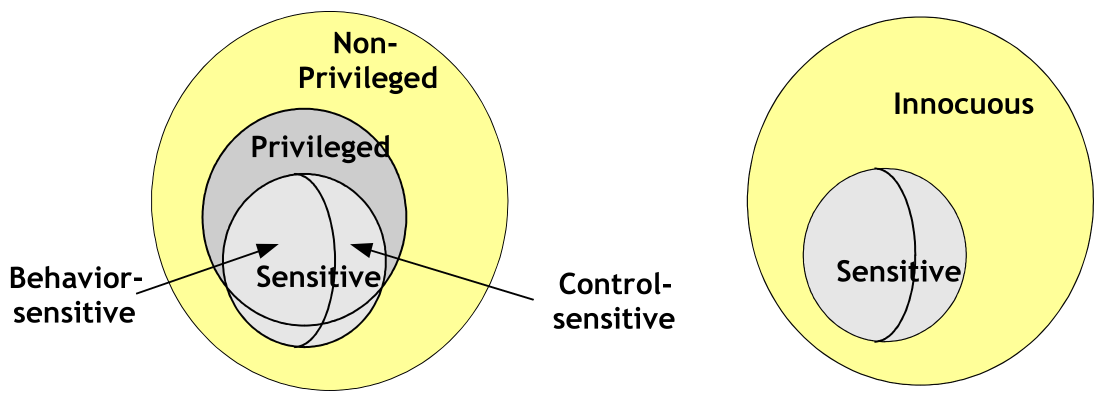


Figure 2.5: Types of insturctions and their relationship with respect to CPU virtualization.

Software Emulation

With software emulation, the hypervisor emulates all of the execution of instructions so the hypervisor can handle the execution of critical instructions. The guest VM can run unmodified OS but such mechanism has significant performance degradation because the emulation processes have lots of overheads. To reduce the performance impact, dynamic binary translation (DBT) is introduced to increase the speed of the hot-path and decrease the performance impact of emulation. QEMU [11] is an example of CPU emulation.

Para-virtualization

For the efficiency concern, para-virtualization requires the critical instructions in the guest OSes to be substituted by hypercalls which generate a trap so that the hypervisor can receive the notification and perform suitable actions, and it can execute innocuous instructions as in the native environment. Para-virtualization can achieve significant performance improvement, but the requirement of modifying guest OSes is the major disadvantage. Xen [10] is an example which uses para-virtualization.

Hardware-assisted virtualization

Hardware-assisted virtualization is a hardware extension that enables efficient full virtualization using the help from hardware capabilities and allows the hypervisor to execute un-

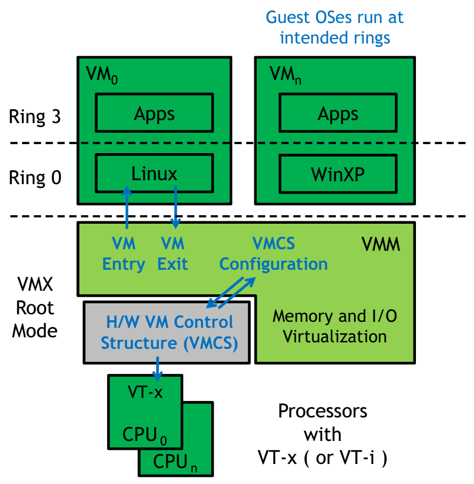

Guest OSes run at intended rings
VM_0
VM_n
Apps
Apps
Ring 3
Ring 0
Linux
WinXP
VM Entry
VM Exit
VMCS Configuration
VMM
VMX Root Mode
H/W VM Control Structure (VMCS)
Memory and I/O Virtualization
VT-x
CPU_0
CPU_n
Processors with VT-x (or VT-i)

Figure 2.6: The relationship among guest OSes, the hypervisor, and hardware-assisted virtualization, using Intel VT-x as an example (adapted from [3]).

modified OSes in complete isolation. Intel and AMD proposed their x86 hardware-assisted virtualization implementations (Intel VT-x and AMD-V) in 2006. Multiple system VMs, such as KVM and Xen hardware virtual machine (Xen HVM), have added the hardware-assisted support for better performance.

Both the virtualization support provided by Intel and AMD are conceptually similar. Intel VT-x introduced two new execution modes, VMX-root-mode and non-root mode, which are orthogonal to the existing x86 provileged modes. VMX-root-mode and non-root mode are also known as root mode and guest mode, respectively. The hypervisor lies in root mode while guest OSes execute in guest mode, and thus guest OSes do not need to be modified. When a CPU executed in guest mode encounters a critical instruction, the CPU will switch to root mode which is called a VM exit and pass the execution to a pre-registered routine of the hypervisor, i.e. trap and emulation by hardware extension. After the emulation processed by the hypervisor, the control is switched back to a specific guest OS which is called a VM entry. A new register called virtual machine control structure (VMCS) is added to record the system configuration of a specific guest OS, which is maintained by the hypervisor. The relationship among guest OSes, the hpervisor, and

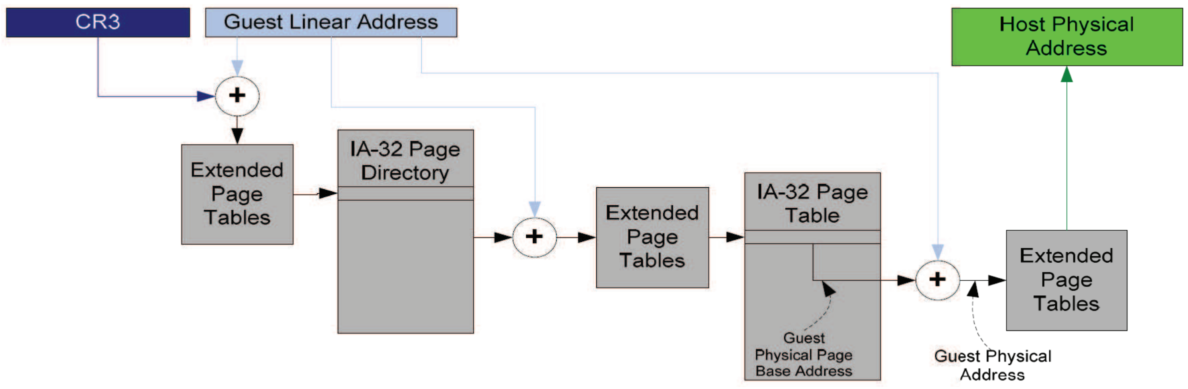


Figure 2.7: Intel EPT translation details (adapted from [3]).

hardware-assisted virtualization support is shown in Figure 2.6.

Intel and AMD also proposed their virtualization support for memory management unit (MMU) to accelerate the address translation from guest virtual address to physical address with much less overhead than maintaining a shadow page table by the hypervisor. The MMU virtualization techiniques in Intel is named extended page table (Intel EPT) and AMD names it as nested page table (AMD NPT). When a guest OS tries to maintain its page table by accessing the x86 CR3 register, the hypervisor will intercept this action and substitute the page table entry with the extended/nested page table. The Intel EPT translation scheme is shown in Figure 2.7.

2.4 I/O Virtualization

Virtualization of I/O devices is more difficult than virtualization of processors or memory subsystems in a system VM. The difficulty of virtualizing I/O devices is that there are many kinds of I/O devices and the characteristics of I/O devices are much different. There are two key points of virtualizing an I/O device, including building the virtual version of the device and virtualizing the I/O activities of the device. We will briefly describe the two issues as follows.

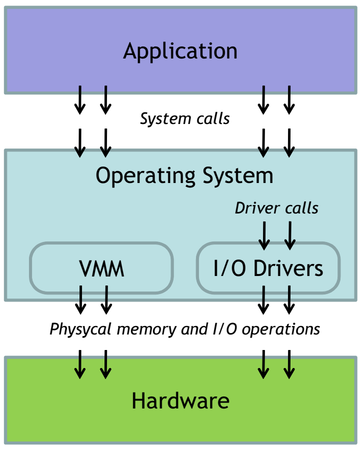


Figure 2.8: Major interfaces in performing an I/O action (adapted from [23]).

2.4.1 Virtualizing Devices

There are different virtualization strategies for different kinds of I/O devices. Some I/O devices such as keyboards, mouses, speakers must be dedicated to a specific guest VM or be switched between guest VMs for a long period. Such devices are called dedicated devices. For devices such as disks, it is suitable to partition the resources for multiple guest VMs, which are called partitioned devices. Some devices such as a network interface card (NIC) can be shared among guest VMs. Such devices are called shared devices.

For the different types of devices, the hypervisor has not only to maintain the virtual states for each virtual device but to intercept the interactions between physical and virtual devices. The requests from different guest VMs should be dispatched by the hypervisor in a fairly-sharing manner. The results of I/O devices should be routed by the hypervisor, and the interrupts from physical devices should be first handled by the hypervisor directly and routed to the destination guest VM.

2.4.2 Virtualizing I/O Activities

The actions of I/O processes are divided into three levels: I/O operation level, device driver level, and system call level, which are illustrated in Figure 2.8.

Virtualizing at the I/O Operation Level

The x86 architectures provide both memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) for signaling the device controller or transferring the data. The hypervisor can intercept such I/O operations due to the nature of x86 privilege level. When a guest VM executes a PMIO instruction or access an MMIO space, the operation will trap into the hypervisor, and the hypervisor then performs corresponding emulation. Since a high-level I/O action may takes several I/O operations, it is extremely difficult for the hypervisor to "reverse engineer" the individual I/O operations to infer the complete I/O action. On the other hand, too much trap-and-emulation for I/O actions would cause dramatic performance degradation.

Virtualizing at the Device Driver Level

System calls such as `read()` or `write()` are converted by the OS into corresponding device driver calls. If the hypervisor can intercept the invocation of these driver calls, it can directly get the information of high-level I/O action of a virtual device and redirect the calls to the corresponding physical device. This scheme requires the guest VMs to execute a modified version of a device driver which is designed for a specific hypervisor and an OS, and the virtual device driver would deliver the I/O actions actively to the hypervisor. Although the modification of the device driver results in the guest OS being aware of itself in the virtualized environment, it can extremely reduce the overhead of virtualizing I/O actions. This approach can be regarded as an I/O para-virtualization scheme at device driver level.

Virtualizing at the System Call Level

Virtualizing at the system call level means the hypervisor will handle the whole system call requests of guest VMs. To accomplish this, however, the guest OSes are required to be modified to add a mechanism to transfer the requests of guest VMs or the emulation results by the hypervisor, typically by adding new routines at the application binary interface (ABI) level. Comparing with virtualizing at the device driver level, this scheme requires more knowledges about the internals of different guest OS kernels and has much more difficulty for implementation.

2.5 GPU Virtualization

To support the functionalities of OpenCL in virtual machine environments, it is important to virtualize graphic processing units (GPUs). Vitrualizing a GPU has unique challenges with several reasons. Firstly, GPUs are extremely complicated devices. Secondly, the hardware specification of GPUs is closed. Thirdly, GPU architectures change rapidly and dramatically across generations. Because of these challenges, it is nearly intractable to virtualize a GPU corresponding to a real modern design. Genearlly, there are three main approaches to virtualize GPUs: software emulation, API Remoting and I/O pass-through.

Software Emulation

One way to virtualize a GPU is to emulate the functionalities of GPUs and to provide a virtual device and driver for guest OSes, which is used as the interface between guest OSes and the hypervisor. The architecture of the virtual GPU could remain unchanged, and the hypervisor would synthesize host graphics operations in response of the requests from virtual GPUs. VMware has proposed VMware SVGA II—a para-virtualized solution of emulating a GPU on the hosted I/O architecture [13]. The VMware SVGA II defines its common graphics stack and provides 2D and 3D rendering with the supports of OpenGL.

Since OpenCL supports multiple different kinds of heterogeneous devices, it is extremely complicated to emulate such devices with different hardware architectures and provides a unified architecture for OpenCL virtualization. Thus, software emulation is not appropriate in this work.

API Remoting

Graphics APIs such as OpenGL and Direct 3D with heterogeneous programming APIs such as CUDA and OpenCL are standard, common interfaces. It is appropriate to target the virtualization layer at API level. The API call requests from guest VMs would be forwarded to the hypervisor which performs the actual invocation. Such mechanism acts like a guest VM invoking a remote procedure call (RPC) to the hypervisor.

I/O Pass-through

As described in 2.4.2, a GPU has its own I/O ports and MMIO spaces. I/O pass-through is to assign a GPU to a dedicated guest VM so that the guest VM can access the GPU as in the native environment. The hypervisor has to handle the address mapping of PMIO or MMIO spaces between virtual and physical devices, which can be done by either software mapping or hardware-assisted mechanisms such as Intel virtualization technology for directed I/O (Intel VT-d) [16]. With the hardware support, the performance of I/O pass-through has rapidly improvement.

Comparisons Between API Remoting and I/O Pass-through

According to VMware's technical report [13], there are four primary criteria for assessing GPU virtualization approaches: performance, fidelity, multiplexing and interposition. Fidelity implies the consistency between virtualized and native environments, and multiplexing and interposition imply the abilities of GPU resources sharing. Table 2.2 summarizes the comparisons between API Remoting and I/O pass-through based on these four criteria. I/O pass-through has better performance and fidelity than API Remoting because of the ability of its direct accessing to physical GPU and thus can adopt the device driver and runtime library which are used for the native environment. On the other hand, API Remoting requires a modified version of device driver and runtime library for transferring the API call requests/responses, and the data volume of API calls is the key point of virtu-

Table 2.2: Comparisons between API Remoting and I/O pass-through based on the four criteria.

	API Remoting	I/O pass-through
Performance	Δ	$\vee$
Fidelity	Δ	$\vee$
Multiplexing	$\vee$	$\times$
Interposition	$\vee$	$\times$

($\vee$: best supported ; Δ: supported ; $\times$: not supported)

alization overhead. Althrough I/O pass-through is considered better in the first two criteria, the fatal weakness of I/O pass-through is that it can not share GPU resources across guest VMs—resource sharing is an essential characteristic of system virtualization. API Remoting, however, can share GPU resources based on the concurrent abilities at the API level.

In this work, we chooses API Remoting for our OpenCL virtualization solution according to the comparisons summarized in Table 2.2, which not only enables the OpenCL support in a virtual machine environment but ensures resource sharing across guest VMs. The design and implementation issues will be discussed in Chapter 3.

Chapter 3

System Design and Implementation

In this chapter, the software architecture and details of the design for enabling OpenCL support in KVM are described. The virtualization framework adopted in this work is described in Section 3.1. The details of design and implementation issues are introduced in Sections 3.2 and 3.3.

3.1 KVM Introduction

Kernel-based Virtual Machine (KVM) is a full virtualization framework for Linux on x86 platform with the help of hardware-assisted virtualization. The key concept of KVM is "Linux as a hypervisor", that is, Linux is turned into a hypervisor by adding the KVM kernel module. The comprehensive functionalities in a system VM can be adapted from the Linux kernel such as scheduler, memory management, I/O subsystems, etc. KVM leverages hardware-assisted virtualization to ensure a pure trap-and-emulation scheme of system virtualization in x86 architectures, not only allowing the execution of unmodified OSes but increasing the performance in virtualizing CPUs and MMU. KVM kernel component has been included in mainline Linux since version 2.6.20 and became the main virtualization solution in Linux kernel.

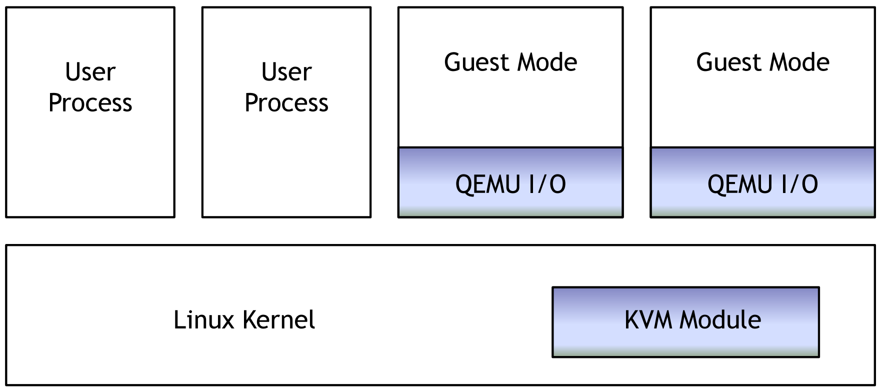


Figure 3.1: KVM overview.

3.1.1 Basic Concepts of KVM

KVM is divided into two components: a KVM kernel module which provides an abstract interface (/dev/kvm) as an entry point of accessing the functionalities of Intel VT-x or AMD-V and a process called hypervisor process that executes a guest OS and emulates I/O actions by QEMU. The hypervisor process is regarded as a normal process in the point of view of host Linux kernel. The overview of KVM is shown in Figure 3.1.

Process Model

The KVM process model is illustrated in Figure 3.2. In KVM, a guest VM is executed within the hypervisor process which provides the necessary resource virtualization for a guest OS such as CPUs, memory spaces and device modules. The hypervisor process contains N threads ($N \geq 1$) for virtualizing CPUs with a dedicated thread for emulating asynchronous I/O actions, which are also known as vCPU threads and an I/O thread. The physical memory space of a guest OS is a part of the virtual memory space in the hypervisor process.

Execution Flow in vCPU View

The execution flow of vCPU threads is illustrated in Figure 3.3 which is divided into three execution modes: guest mode, kernel mode and user mode. In Intel VT, the guest mode is

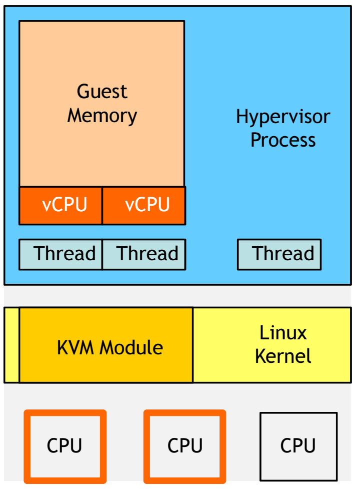


Figure 3.2: KVM process model (adapted from [20]).

mapped into VMX-non-root mode and both kernel mode with user mode are mapped into VMX-root mode.

A vCPU thread in guest mode could execute guest instructions as in native environments unless encountering a privileged instruction. When the vCPU thread executes a privileged instruction, the control will transfer to the KVM kernel module. The KVM kernel module first maintains the VMCS of the guest VM and then decides how to handle such instruction. Only a small amount of actions are processed by the kernel module, including virtual MMU management and in-kernel I/O emulation. In other cases, the control will further transfer to user mode. In user mode, the vCPU thread performs the corresponding I/O emulation or signal handling by QEMU. After the emulated operation completed, the context held by user space is updated, and then the vCPU thread switches back to guest mode.

The control transferring from guest mode to kernel mode is called a light-weight VM exit, and that from guest mode to user mode is called a heavy-weight VM exit. The performance of I/O emulation is highly related to the number of heavy-weight VM exits since the cost of heavy-weight VM exits is much more than that of light-weight VM exits.

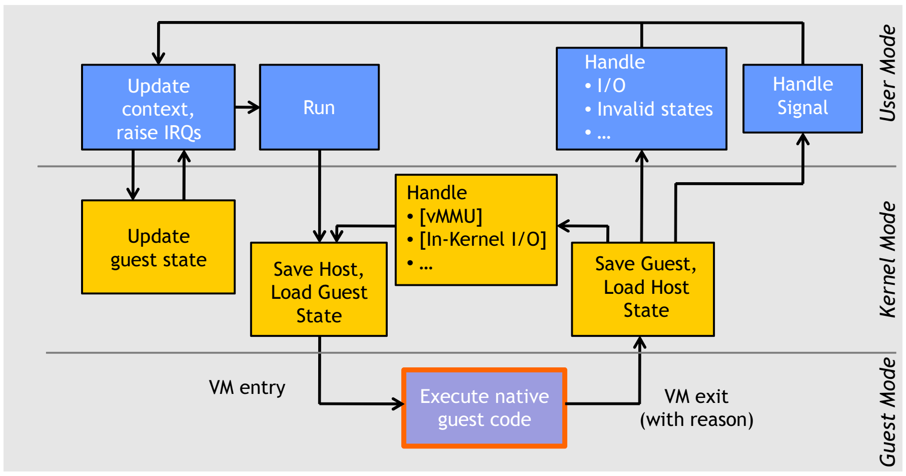


Figure 3.3: KVM execution flow in vCPU view (adapted from [20]).

I/O Virtualization

Each virtual I/O device in KVM has a set of virtual components such as I/O ports, MMIO spaces and device memory. Emulation of I/O actions is to maintain the access or event of such virtual component. Each virtual device will register its I/O ports and MMIO space handler routine when the device starts. When PMIO or MMIO actions are executed in the guest OS, the vCPU thread will trap from guest mode to user mode and lookup the record of the allocation of I/O ports or MMIO spaces to choose the corresponding I/O emulation routines. For asynchronous I/O actions such as response network packets arriving or keyboard signals occurring, it should be processed with the help of the I/O thread. The I/O thread is blocked waiting for the new incoming I/O events and handles them by sending virtual interrupts to the target guest OS or emulates direct memory access (DMA) between virtual devices and the main memory space of the guest OS.

3.1.2 Virtio Framework

Virtio framework [27] is an abstraction layer over para-virtualized I/O devices in a hypervisor. Virtio was developed by Rusty Russel to support his own hypervisor called `lguest` and adopted by KVM for its I/O para-virtualization solution. With virtio, it is easy to im-

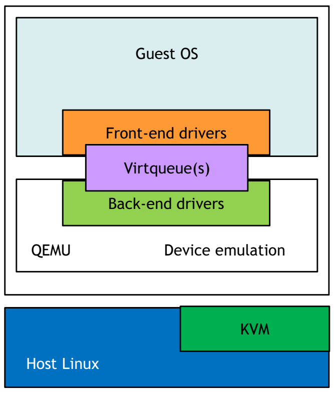


Figure 3.4: Virtio architecture in KVM.

plement new para-virtualized devices by extending the common abstraction layer. Virtio framework has been included in Linux kernel since version 2.6.30.

Virtio conceptually abstracts an I/O device as front-end drivers, back-end drivers and one or more virtqueues as shown in Figure 3.4. Front-end drivers are implemented as device drivers of virtual I/O devices and use virtqueues to communicate with the hypervisor. Virtqueues can be regarded as shared memory spaces between guest OSes and the hypervisor. There is a set of functions for operating virtqueues including adding/retrieving data to/from a virtqueue (`add_buf`/`get_guf`), generating a trap to switch the control to the back-end driver (`kick`), and enabling/disabling call-back functions (`enable_cb`/`disable_cb`) which are the interrupt handling routines of the virtio device. The back-end driver in the hypervisor retrieves the data from virtqueues and then emulates the corresponding I/O emulation based on the data from guest OSes.

The high-level architecture of virtio in Linux kernel is illustrated in Figure 3.5. The virtqueue and its transmission are implemented in `virtio.c` and `virtio_ring.c`, and there are a series of virtio devices such as `virtio-blk`, `virtio-net`, `virtio-pci`, etc. The object hierarchy of the virtio front-end is shown in Figure 3.6, which illustrates the fields and methods of each virtio object with the relationships between them. A virtqueue object contains the description of available operations, a pointer to the call-back function,

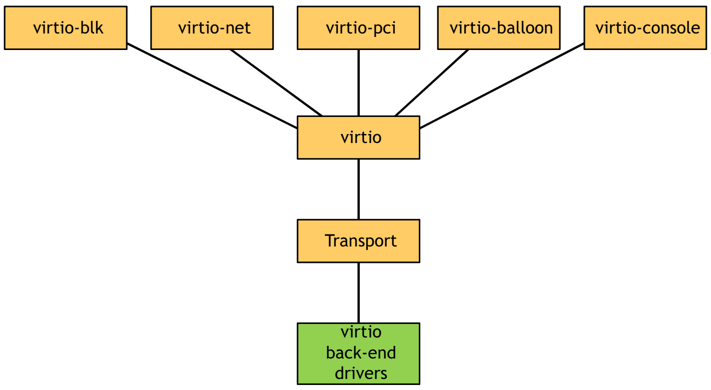
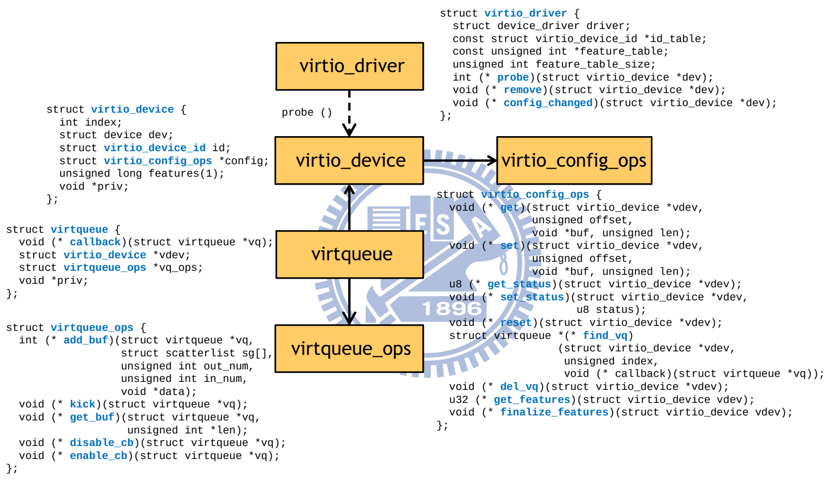


Figure 3.5: High-level architecture of virtio (adapted from [17]).



```
struct virtio_driver {
  struct device_driver driver;
  const struct virtio_device_id *id_table;
  const unsigned int *feature_table;
  unsigned int feature_table_size;
  int (* probe)(struct virtio_device *dev);
  void (* remove)(struct virtio_device *dev);
  void (* config_changed)(struct virtio_device *dev);
};
```

```
struct virtio_device {
  int index;
  struct device dev;
  struct virtio_device_id id;
  struct virtio_config_ops *config;
  unsigned long features(1);
  void *priv;
};
```

```
struct virtqueue {
  void (* callback)(struct virtqueue *vq);
  struct virtio_device *vdev;
  struct virtqueue_ops *vq_ops;
  void *priv;
};
```

```
struct virtqueue_ops {
  int (* add_buf)(struct virtqueue *vq,
                  struct scatterlist sg[],
                  unsigned int out_num,
                  unsigned int in_num,
                  void *data);
  void (* kick)(struct virtqueue *vq);
  void (* get_buf)(struct virtqueue *vq,
                   unsigned int *len);
  void (* disable_cb)(struct virtqueue *vq);
  void (* enable_cb)(struct virtqueue *vq);
};
```

```
struct virtio_config_ops {
  void (* get)(struct virtio_device *vdev,
               unsigned offset,
               void *buf, unsigned len);
  void (* set)(struct virtio_device *vdev,
               unsigned offset,
               void *buf, unsigned len);
  u8 (* get_status)(struct virtio_device *vdev);
  void (* set_status)(struct virtio_device *vdev,
                      u8 status);
  void (* reset)(struct virtio_device *vdev);
  struct virtqueue *(* find_vq)
                    (struct virtio_device *vdev,
                     unsigned index,
                     void (* callback)(struct virtqueue *vq));
  void (* del_vq)(struct virtio_device *vdev);
  u32 (* get_features)(struct virtio_device vdev);
  void (* finalize_features)(struct virtio_device vdev);
};
```

Figure 3.6: Object hierarchy of the virtio front-end (adapted from [17]).

and a pointer to the virtio_device which owns this virtqueue. A virtio_device object contains the fields used to describe the features and a pointer to a virtio_config_ops object, which is used to describe the operations that configures the device. In the device initialization phase, the virtio driver would invoke the `probe` method to setup and new an instance of virtio_device.

In this work, we implement our API Remoting mechanism based on the virtio framework to perform the data communication for OpenCL virtualization. The design and implementation details will be discussed in the following sections.

3.2 OpenCL API Remoting in KVM

In this section, the framework of OpenCL API Remoting in KVM is described, including the software architecture, execution flow and the relationship among each software components.

3.2.1 Software Architecture

Figure 3.7 presents the architecture of OpenCL API Remoting in this work, which includes an OpenCL library specific to guest OSes, a virtual device called Virtio-CL and a thread called CL thread. The functionalities of each component are described as follows:

- Guest OpenCL library

  The guest OpenCL library is response for packing OpenCL requests of user applications from the guest and unpacking the results from the hypervisor. In our current implementation, the guest OpenCL library is designed as a wrapper library and performs basic verifications according to the OpenCL specifications such as null pointer or integer value range checking.

- Virtio-CL device

  The Virtio-CL virtual device is response for data communication between the guest OS and the hypervisor. The main component of Virtio-CL is two virtqueues: one for data transmission from the guest OS to the hypervisor and the other vice versa. The Virtio-CL device can be further divided into **front-end** (residing in the guest OS) and **back-end** (residing in the hypervisor). The guest OS accesses the Virtio-CL device by the front-end driver and writes/reads OpenCL API requests/responses via virtqueues using the corresponding driver calls. The Virtio-CL back-end driver accepts the requests from the guest OS and passes them to the CL thread. The actual invocation of OpenCL API calls is done by the CL thread. The virtqueues can

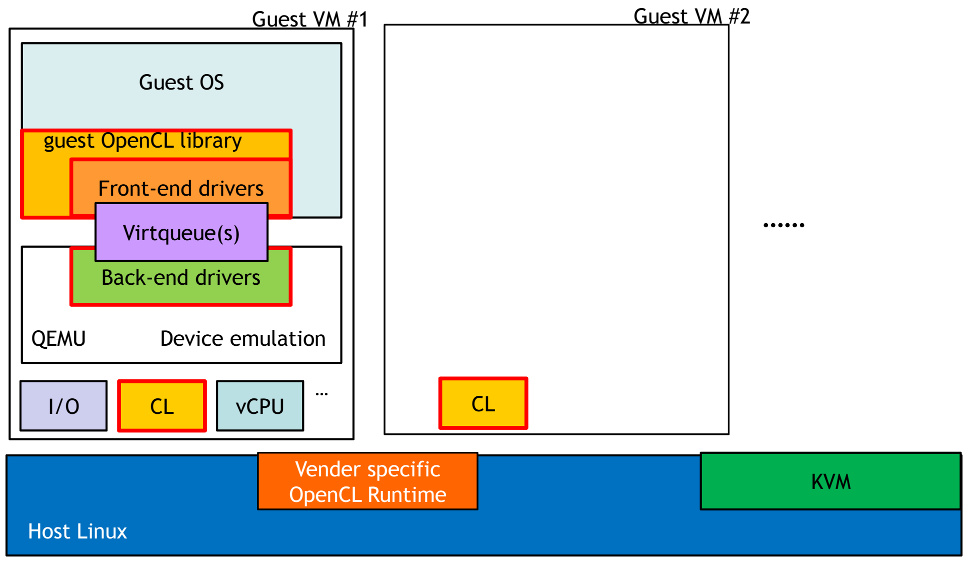


Figure 3.7: Architecture of OpenCL API Remoting in KVM.

be regarded as shared memory spaces which can be modeled as device memory of Virtio-CL in the point of view of the guest OS.

- CL thread

  The CL thread is dedicated for accessing vendor-specific OpenCL runtimes in user mode. The CL thread reconstructs the requests, performs the actual invocation of OpenCL API calls, and then passes the results back to the guest OS via the virtqueue used for response transmission. Since the processing time for each OpenCL API call is different, it is appropriate to handle OpenCL requests by an individual thread instead of extending the functionalities of existing I/O thread in the hypervisor process in order to let the execution of OpenCL APIs be independent from the functionalities of the I/O thread.

  An alternative approach instead of creating an CL thread to process OpenCL requests is to implement the actual invocation of OpenCL API calls in the handler of the vCPU thread and configure as multiple vCPUs for each guest VM. However, both of the two approaches require a buffer (CLRequestQueue as shown in Figure 3.8) to store segmented OpenCL requests in case the size of an OpenCL request is larger than the size of the virtqueue for requrests (VQ_REQ in Figure 3.8). In addition, the latter approach has to handle the synchronization of the virtuqueue for response (VQ_RESP as shown in Figure 3.8) between different vCPU threads, while the former approach handles the synchronization of CLRequestQueue between the vCPU thread and the CL thread.

As in Figure 3.7, the architecture of our OpenCL virtualization framework can be modeled as multiple processes accessing the OpenCL resources in the native environment. Behaviors of the execution depend on the implementation of the vendor-specific OpenCL runtime.

3.2.2 Execution Flow

Figure 3.8 illustrates the execution flow of OpenCL API calls, and the processing steps are as follows:

1. A process running in a guest OS invokes an OpenCL API function.
2. The guest OpenCL library (libOpenCL.a) first performs basic verifications of the parameters according to the OpenCL API specifications. If the verifications failed, it returns the corresponding error code to the user-space process.
3. After the parameter verification, the guest OpenCL library sets up the data needed to be transferred and executes system call `fsync()`.
4. The `fsync()` system call adds the data to the virtqueue for requests (VQ_REQ) and then invokes the `kick()` method of VQ_REQ to generate a VM exit. The control of current vCPU thread is transferred to a corresponding handler routine of VM exit in user-mode.

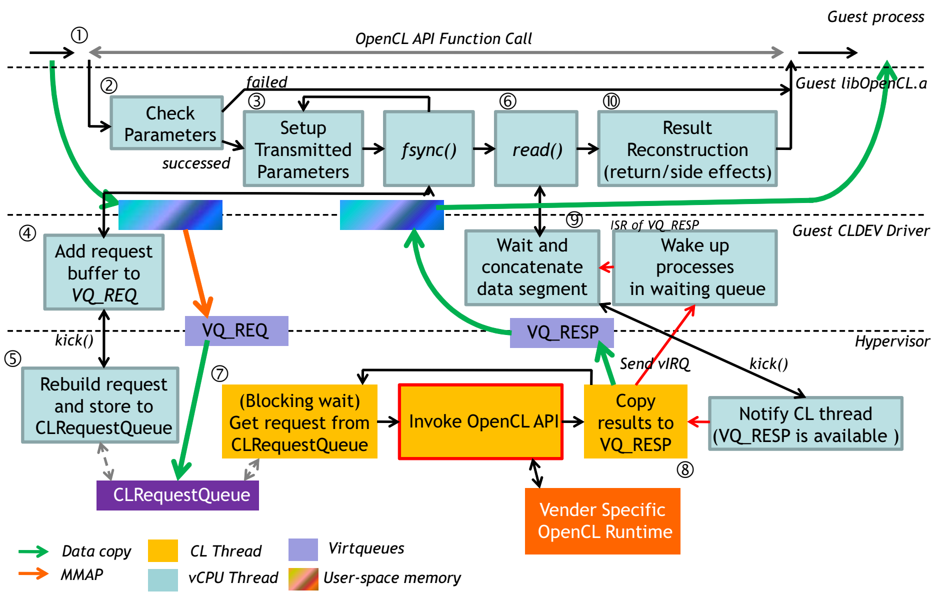

Guest process
OpenCL API Function Call
Guest libOpenCL.a
failed
Check Parameters
successed
Setup Transmitted Parameters
fsync()
read()
Result Reconstruction (return/side effects)
Guest CLDEV Driver
Add request buffer to VQ_REQ
Wait and concatenate data segment
Wake up processes in waiting queue
ISR of VQ_RESP
kick()
VQ_REQ
VQ_RESP
Hypervisor
Send vIRQ
kick()
Rebuild request and store to CLRequestQueue
(Blocking wait) Get request from CLRequestQueue
Invoke OpenCL API
Copy results to VQ_RESP
Notify CL thread (VQ_RESP is available)
CLRequestQueue
Vender Specific OpenCL Runtime
Data copy
MMAP
CL Thread
vCPU Thread
Virtqueues
User-space memory

Figure 3.8: Execution flow of OpenCL API Remoting.

5. In user-mode, the handler routine copies the OpenCL API request to CLRequestQueue, which is a shared memory space between vCPU threads and the CL thread. After the copy process completes, the control is transferred back to the guest OpenCL library. If the data size of the request is larger than the size of VQ_REQ, the request is divided into segments and the execution jumps to step 3 and repeats step 3–5 until all the segments are transferred. If the whole data segments of the OpenCL request are transferred, the handler will signal the CL thread there is an incoming OpenCL API request and the CL thread starts processing (see step 7).

6. After the request is passed to the hypervisor, the guest OpenCL library invokes `read()` system call which is blocked waiting for the result data and returns after the whole result data have been transferred.

7. The CL thread waits on a blocking queue until receiving the signal from the handler routine of VM exit in step 5. The CL thread then unpacks the request and performs the actual invocation.

8. After the completion of the OpenCL API invocation, the CL thread packs the result data, copies them to the virtqueue for responses (VQ_RESP), and then notifies the guest OS by sending a *virtual interrupt* of the Virtio-CL device.

9. Once the guest OS receives the virtual interrupt from the Virtio-CL device, the corresponding interrupt service routine (ISR) wakes up the process waiting for response data of the OpenCL API call, and the result data are copied from VQ_RESP to the user space memory. Steps 8 and 9 repeat until all of result data are transferred.

10. Once the `read()` system call returns, the guest OpenCL library can unpack and rebuild the return value and/or side effects of parameters. The execution of OpenCL API function has been completed.

3.2.3 Implementation Details

In this section, we present the implementation details of the proposed virtualization framework, including the guest OpenCL library, device driver, data transmissions and synchronization points.

Guest OpenCL Library and Device Driver

The device driver of Virtio-CL implements `open()`, `close()`, `mmap()`, `fsync()` and `read()` system calls. `mmap()` and `fsync()` are used for transferring OpenCL requests, and `read()` system call is used for retrieving the response data. The guest OpenCL library uses those system calls to communicate with the hypervisor.

Before a user process start using the resources of OpenCL, the process has to explicitly invoke a specific function—`clEnvInit()` to perform resource initialization such as opening the Virtio-CL device and preparing the memory storages for data transmissions. Another additional function—`clEnvExit()` has to be invoked to release the resources of OpenCL before the process finishes execution. Therefore, OpenCL programs

```
pthread_mutex_t clReqMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  clReqCond  = PTHREAD_COND_INITIALIZER;
...

pthread_mutex_lock( &clReqMutex );
if( /* OpenCL request is not ready */ ) {
  pthread_cond_wait( &clReqCond, &clReqMutex );
}

/* pop an request node from CLRequestQueue */
...
```

Figure 3.9: 1^{st} synchronization point (CL thread).

```
pthread_mutex_lock( &clReqMutex );

/* Copy the request data segment to CLRequestQueue */

if( /* OpenCL request is ready */ ) {
  pthread_cond_signal( &clReqCond );
}

pthread_mutex_unlock( &clReqMutex );
...
```

Figure 3.10: 1^{st} synchronization point (VM exit handler).

to be executed in the virtualized platform have to be revised with minor changes—adding `clEnvInit()` and `clEnvExit()` at the beginning and the end of an OpenCL program, respectively. Nevertheless, the revisions can be done automatically by a simple OpenCL parser. An alternative approach that avoids such revisions is possible, but it involves a complex virtualization architecture and results in much more overhead. Further discussions about the alternative implementation will be made in Section 3.3.5.

Synchronization Points

There are two synchronization points in the execution flow: one occurs in the CL thread which waits for a completed OpenCL request, and the other occurs in the implementation of the `read()` system call which waits for the result data processed by the CL thread. The

```
ssize_t cldev_read( struct file *filp, char __user *buf, size_t count, loff_t *f_pos )
{
  do {
    /* put the current process into waiting queue */
    schedule();

    /* the process will be blocked until response data segment is ready */

    copy_to_user( buf, kernel_buffer, size_data_seg );

    /* notify the hypervisor that VQ_RESP is available */

  } while( /* transmission not completed */ )
}
```

Figure 3.11: 2^{nd} synchronization point (`read` system call).

1^{st} synchronization point is handled by pthread mutexes and condition variables. When the data segments of an OpenCL request are not completely transmitted to CLRequestQueue, the CL thread invokes `pthread_cond_wait()` to wait for the signal from the handler routine of VQ_REQ. The pseudo code of the 1^{st} synchronization point is described in Figures 3.9 and 3.10. The 2^{nd} synchronization point works as follows. The `read()` system call first puts itself to the waiting queue in the kernel space to be blocked. The blocked process will resume execution and retrieve the response data segment from VQ_RESP after the virtual interrupt of Virtio-CL is raised. The pseudo code of the 2^{nd} synchronous point is described in Figure 3.11.

The two synchronization mechanisms not only ensure the data correctness among the transmission processes but also allow the vCPU resource to be used by other processes in the guest OS.

Data Transimssions

In Figure 3.8, the green and orange arrows represent the paths of data transmissions. The green arrows indicate the data copies among the guest user-space, the guest kernel-space and the host user-space. Since the data transmission of OpenCL requests is processed actively by the guest OpenCL library, the data copy from the guest user-space to the guest kernel-space can be bypassed by preparing a memory-mapped space (indicated by the orange arrow). The data copy can not be eliminated in the opposite direction because the guest OpenCL library is waiting for results from the hypervisor passively. Thus, there are two data copies for a request and three copies for a response.

3.3 Related Issues of Implementation

In this section, the related issues of design and implementation of supporting OpenCL in KVM, including the size of virtqueues, data coherence between guest OSes and the hypervisor, etc., will be discussed.

3.3.1 Size of Virtqueues

Virtqueues are shared memory space between guest OSes and the hypervisor, and there are a series of address translation mechanisms for both the guest OS and the QEMU part to access the data from virtqueues. On one hand, the size of virtqueues directly affects the virtualization overhead because larger size of virtqueues results in fewer number of the heavyweight VM exits. On the other hand, since the total virtual memory space of the hypervisor process is limited (4 Gigabytes in 32-bit host Linux), the size of virtqueue is also limited.

In our framework, the size of VQ_REQ and VQ_RESP are both 256 Kilobytes (64 pages) according to the configurations of the existing Virtio devices: *virtio-blk* and *virtio-net*. *virtio-blk* has one virtqueue which is 512 Kilobytes (128 pages), and *virtio-net* has two virtqueues and both of them are 1024 Kilobytes (256 pages). A request (or a response) will be partitioned into multiple 256-Kilobyte segments and then transferred sequentially if the data size of the request (or the response) exceeds the virtuqueue size.

3.3.2 Signal Handling

A user OpenCL process may suffers a segmentation fault while calling OpenCL APIs. In the native environments, it will cause the end of process execution. In our virtualization framework, the situation should be handled by the CL thread carefully, or the hypervisor process will crash. The CL thread has to build handler routines for signals like `SIGSEGV` to recover the thread from such signal and return the corresponding error messages to the guest OpenCL library.

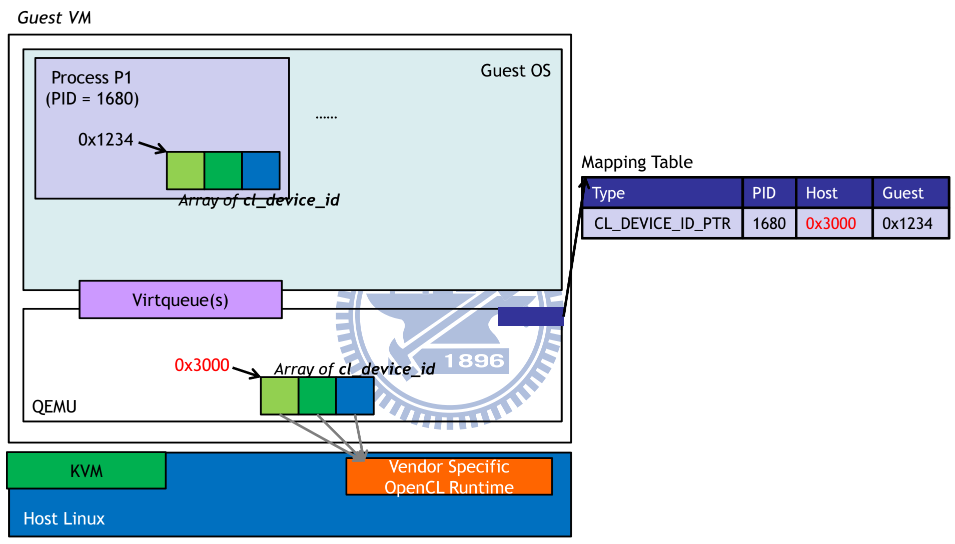
3.3.3 Data Structures Related to Runtime Implementation

Figure 3.12 lists the data structures which are related to the runtime implementation in OpenCL. The data structures are maintained by the vendor-specific OpenCL runtime, and the users only can access them by the corresponding OpenCL functions. For example, when a process invokes `clGetDeviceIDs()` with a pointer of an array of `cl_device_id` as a parameter, the function fills each entry of the array with a pointer to the object of an OpenCL device. In our framework, the CL thread is used to access the actual OpenCL runtime in user-mode and consume the parameters provided by the guest process. However, the CL thread can not directly access the array since it is in a virtual address space of the guest process. Thus, we have to construct a "shadow mapping" of the guest array: the CL thread allocates a "shadow array" and maintains a mapping table between the array in the guest and the shadow array. Figure 3.13 illustrates an example of the shadow mapping mechanism. When a guest process invokes `clGetDeviceIDs()`, the CL thread allocates a "shadow array" and creates a new entry in the mapping table. The entry records the pointer type, the process identifier (PID) of the guest process and the mapping between the host address and the guest address. The CL thread then performs the actual OpenCL function invocation with the shadow array and transfers all the contents of the shadow array to the guest process after the function invocation completes to ensure that the array of `cl_device_id` in the guest has the same contents with the shadow array.

```
/* /usr/local/cuda/include/CL/cl.h (NVIDIA OpenCL SDK) */

typedef struct _cl_platform_id *    cl_platform_id;
typedef struct _cl_device_id *      cl_device_id;
typedef struct _cl_context *        cl_context;
typedef struct _cl_command_queue *  cl_command_queue;
typedef struct _cl_mem *            cl_mem;
typedef struct _cl_program *        cl_program;
typedef struct _cl_kernel *         cl_kernel;
typedef struct _cl_event *          cl_event;
typedef struct _cl_sampler *        cl_sampler;
```

Figure 3.12: The data structures related to the OpenCL runtime implementation.



Type	PID	Host	Guest
CL_DEVICE_ID_PTR	1680	0x3000	0x1234

Figure 3.13: Shadow mapping mechanism.

When the guest process invokes an OpenCL function that uses a `cl_device_id` object, value of the `cl_device_id` object can be directly used because it is a pointer to the host address space. When the guest process invokes an OpenCL function that uses an array of `cl_device_id` objects, the CL thread will lookup the mapping table to find the address of the shadow array for the actual function invocation. After the guest process invokes `clEnvExit()`, the CL thread deletes the entries related to the process and release the memory spaces recorded in the shadow mapping entries.

```
cl_mem clCreateBuffer( cl_context   context,
                       cl_mem_flags flags,
                       size_t       size,
                       void *       host_ptr,
                       cl_int *     errcode_ret );
```

Figure 3.14: Prototype of clCreateuffer().

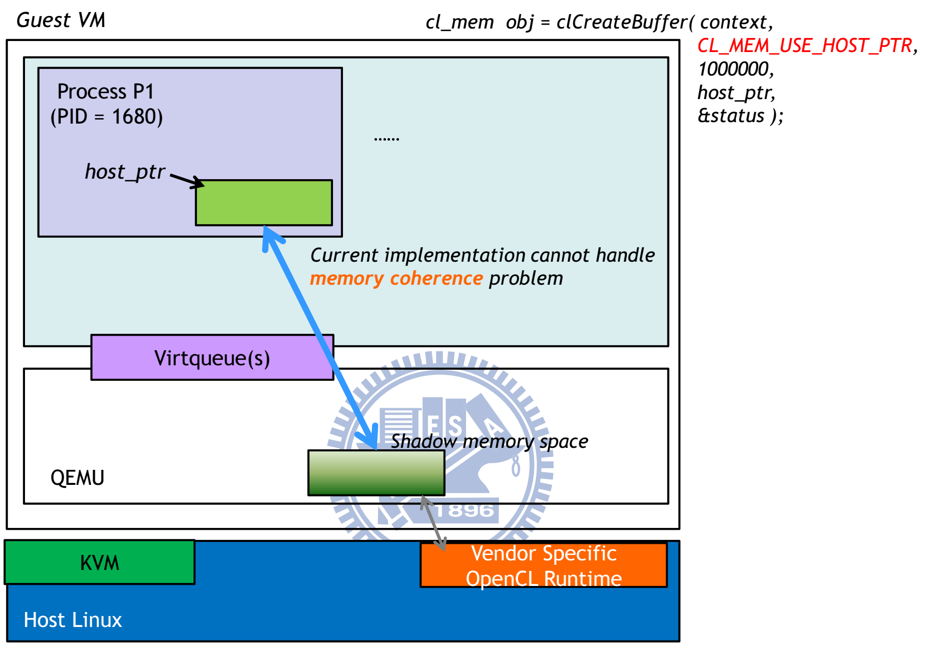


Figure 3.15: Memory coherence problem in clCreateBuffer().

3.3.4 OpenCL Memory Objects

OpenCL memory objects (cl_mem) is an abstraction of global device memory that can serve as data containers for computation. A process can use `clCreateBuffer()` to create a memory object which can be regarded as an one-dimensional buffer. Figure 3.14 shows the prototype of `clCreateBuffer()`, where `context` is a valid OpenCL context associated with this memory object, `flags` is used to specify allocation and usage information—Table 3.1 describes the possible values for `flags`, `size` indicates the size of the memory object in bytes, `host_ptr` is a pointer to the buffer data that may already be allocated, and `errcode_ret` is the field used to store the information of error code.

The buffer object can be allocated in either the OpenCL device or the OpenCL host memory, which is decided by the `flags` parameter. If the **CL_MEM_USE_HOST_PTR** or **CL_MEM_ALLOC_HOST_PTR** option is selected, the buffer object would use the memory spaces pointed by `host_ptr` which resides in the virtual address space of the guest process. Although the memory spaces pointed by `host_ptr` belong to the OpenCL host memory, it can only be accessed by a set of corresponding OpenCL functions such as `clEnqueueReadBuffer()` and `clEnqueueWriteBuffer()`, and the behavior is undefined when directly accessing the region of the memory spaces. There are two difficulties to support **CL_MEM_USE_HOST_PTR** or **CL_MEM_ALLOC_HOST_PTR** options in our virtualization framework, which is illustrated in Figure 3.15. Firstly, the CL thread can not access this memory region directly. Secondly, even through creating a shadow memory space in the CL thread, the memory coherence between the guest process and the CL thread is still a complicated problem. The possible solution of the memory coherence problem will be discussed in Section 3.3.5.

3.3.5 Enhancement of the Guest OpenCL Library

The current implementation of our OpenCL virtualization framework only fully supports the **CL_MEM_COPY_HOST_PTR** option which allocates storages in the device memory and copies the data pointed by `host_ptr` to the device memory. **CL_MEM_USE_HOST_PTR** and **CL_MEM_ALLOC_HOST_PTR** options are supported only if users follow the OpenCL specification to access the memory object by the corresponding OpenCL functions. The memory coherence can not be guaranteed if the memory region was accessed by the OpenCL host program directly. Although the functionalities of `clCreateBuffer` are not fully supported in the current virtualization framework, the framework is capable to virtualize most of the OpenCL operations on a CPU-GPU platform since the accesses of buffer objects are generally done by OpenCL functions which are response for operating buffer objects. The second drawback in our framework is that programmers have to add a `clEnvInit()` and a `clEnvExit()` function call at the beginning and the end of an OpenCL program, as mentioned in Section 3.2.3, makes OpenCL programs not natively portable in our framework.

In order to solve these problems, a possible approach is to substitute the guest OpenCL library with a new virtualization layer. The virtualization layer can notify the hypervisor the start/completion of a guest process and intercept the read/write events of the guest process so as to ensure memory coherence between the guest and the shadow memory space. The virtualization layer is conceptually like a *process virtual machine*. The disadvantage of introducing the new virtualization layer is the larger cost for ensuring the memory coherence. The enhancement of the guest OpenCL library is one of our future work.

Table 3.1: List of supported *cl_mem_flags* values (adapted from [18]).

cl_mem_flags	**Description**
CL_MEM_READ_WRITE	This flag specifies that the memory object will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	This flag specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with CL_MEM_WRITE_ONLY inside a kernel is undefined.
CL_MEM_READ_ONLY	This flag specifies that the memory object is a read-only memory object when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined.
CL_MEM_USE_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by *host_ptr* as the storage bits for the memory object. OpenCL implementations are allowed to cache the buffer contents pointed to by *host_ptr* in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same *host_ptr* or overlapping host regions is considered to be undefined.
CL_MEM_ALLOC_HOST_PTR	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.
CL_MEM_COPY_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by *host_ptr*. CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive. CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.

Chapter 4

Experimental Results

Evaluations of the OpenCL virtualization framework proposed in this work are discussed in this chapter, including the environments, experimental results and discussions.

4.1 Environment

4.1.1 Testbed

All of the evaluations were conducted under the environments of an Intel Core i7-930 processor (8 cores at 2.8GHz) with 8 Gigabytes of memory and two NVIDIA GeForce GTX 580 GPUs running Linux 2.6.32.21 kernel with NVIDIA CUDA SDK of version 3.2 (NVIDIA OpenCL SDK is included). Our OpenCL virtualization framework was implemented in qemu-kvm-0.14.0 operating with kvm-kmod-2.6.32.21. Each guest VM in the virtualization framework was configured with one virtual CPU, 512 Megabytes of memory, and a 10-Gigabyte disk drive running Linux 2.6.32.21 kernel. The size of virtqueues in Virtio-CL was configured as 256 Kilobytes (64 pages).

As described in Section 3.2, the architecture of our OpenCL virtualization framework can be modeled as multiple processes accessing the OpenCL resources in the native environment. In order to effectively realize the resource-sharing characteristics of our virtualization framework, we took advantage of NVIDIA *Fermi* GPUs which support "concurrent kernel execution" and "improved application context switching" [24]. Concurrent kernel

Table 4.1: Statistics of benchmark patterns.

Name	Source	Number of API Calls	Data Volume (MBytes)	Description
FastWalshTransform [FWT]	AMD	34	3.08	Generalized Fourier transformations
BlackScholes [BS]	NVIDIA	34	77.27	Modern option pricing techniques, applied areas of finance
MersenneTwister [MT]	NVIDIA	44	183.88	Mersenne Twister random number generator and Cartesian Box-Muller transformation on the GPU
MatrixManipulation [MM]	AMD	53	48.76	An implementation of matrix multiplication
ScanLargeArray [SLA]	AMD	70	0.51	An implementation of scan algorithm for large arrays
ConvolutionSeparable [CS]	NVIDIA	38	72.75	Convolution filter of a 2D image with arbitrary separable kernel

execution allows multiple OpenCL kernels in an application to execute concurrently if the GPU resources are not fully occupied, and the optimized *Fermi* pipeline significantly improves the efficiency of context switches.

4.1.2 Benchmarks

The benchmarks we used for evaluating our proposed OpenCL virtualization framework are collected from both NVIDIA OpenCL SDK [5] and AMD Accelerated Parallel Processing (APP) SDK [1], and both of them contain a set of general-purpose algorithms in various domains. Table 4.1 summarizes the statistics of these benchmarks, including the description of each patterns, the number of OpenCL API calls and the data volume which is transferred between the guest OS and the hypervisor process.

4.2 Evaluation

First we evaluate the virtualization overhead of the proposed framework by measuring the execution time of OpenCL programs for two configurations:

Table 4.2: Execution time of six OpenCL benchmarks for both the Native and 1VM configurations.

	Native (Sec.)	1VM (Sec.)	Virtualization Overhead (Sec.)	Virtualization Overhead (%)
FastWalshTransform[FWT]	1.175	1.256	0.081	6.9
BlackScholes [BS]	2.330	2.424	0.094	4.0
MersenneTwister [MT]	8.078	11.603	3.525	43.6
MatrixManipulation [MM]	0.969	1.082	0.113	11.7
ScanLargeArray [SLA]	0.939	0.976	0.037	3.9
ConvolutionSeparable [CS]	3.937	4.154	0.217	5.5

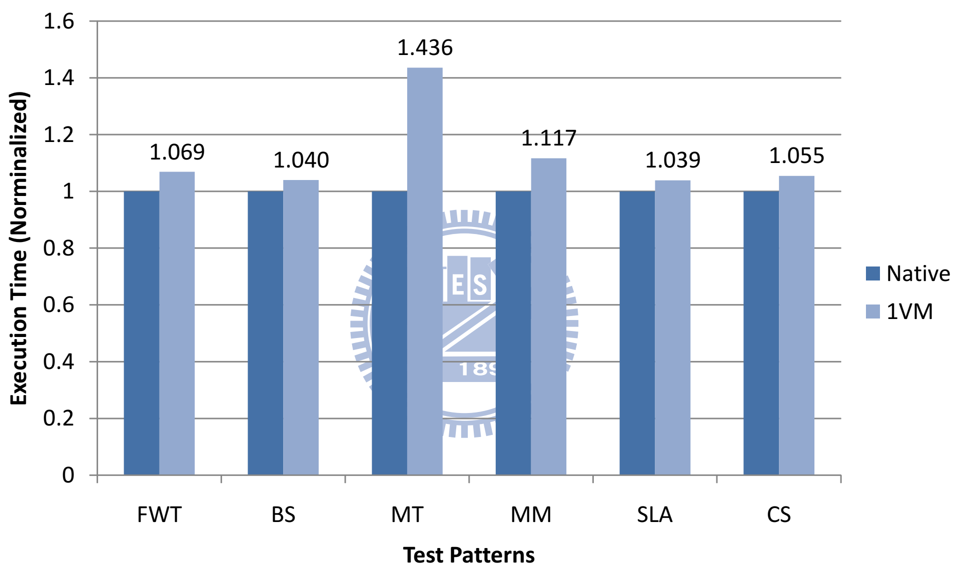


Figure 4.1: Normalized execution time for the Native and 1VM configurations.

- Native

  Benchmark programs execute in the host platform and directly access the vendor-specific OpenCL runtime. This configuration represents the baseline in all experiments.

- 1VM

  Benchmark programs execute in a guest VM and acquire the ability of OpenCL by our proposed virtualization framework. This configuration is used for evaluating the

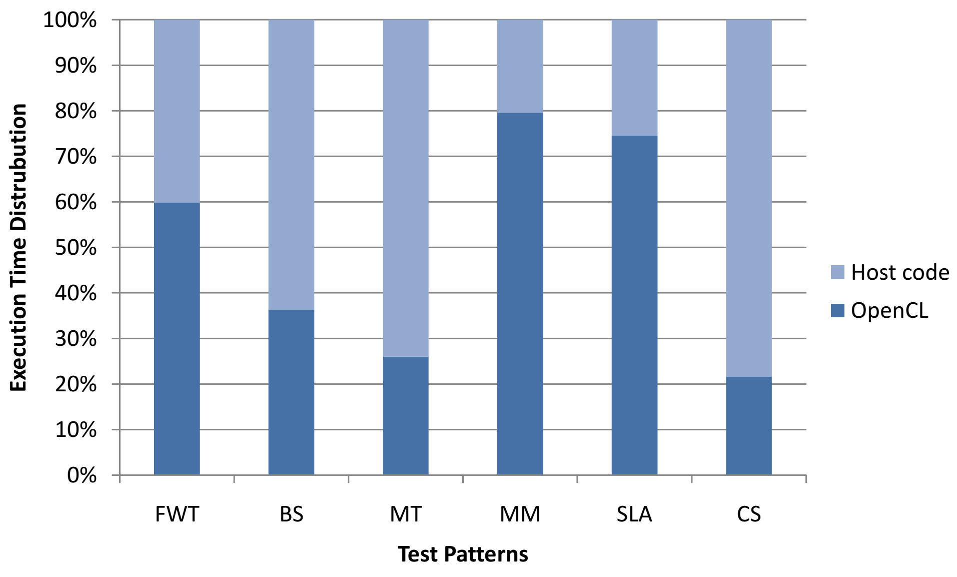


Figure 4.2: The ratio of execution time of OpenCL APIs and OpenCL host code.

overhead in the virtualized environment.

4.2.1 Virtualization Overhead

Table 4.2 summarizes the execution time of the six OpenCL programs for the **Native** and **1VM** configurations, and Figure 4.1 shows the results in which the execution time for the native environment are normalized to 1. The overhead of executing the OpenCL programs in the virtualized platform ranges from 3.9% (for $ScanLargeArray$) to 43.6% (for $MersenneTwister$), with the average value of 12.2%. It is observed that the overhead is less than 10% for most of the benchmark programs, except for $MersenneTwister$. As a matter of fact, the average overhead is 6.4% if $MersenneTwister$ is excluded.

Overhead Analysis

In order to identify the reason of varied overhead distribution, we first measured the execution time of OpenCL APIs and OpenCL host code in both configurations. Figures 4.2 and 4.3 show the ratio of execution time and virtualization overhead of OpenCL APIs and

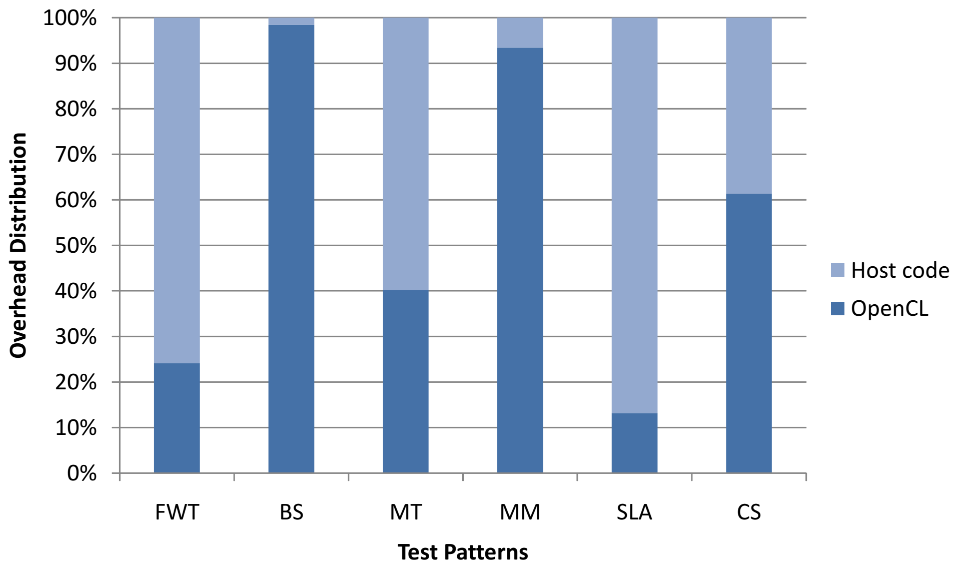


Figure 4.3: The ratio of virtualization overhead of OpenCL APIs and OpenCL host code.

OpenCL host code, respectively. According to the analysis, MT is a CPU-intensive program and the overhead of OpenCL host code (referred to the CPU virtualization in KVM) dominates the performance loss. In addition, we analyzed the virtualization overhead of the OpenCL APIs. Table 4.3 and Figure 4.4 show the detailed breakdown of the virtualization overhead. The virtualization overhead is divided into three parts in our measurement: data transmission from the guest VM to the hypervisor, data transmission from the hypervisor to the guest and others. We used the *gettimeofday()* function to measure the time of data transmissions from the guest VM to the hypervisor in the guest OpenCL library and the transfer time in the opposite direction in the CL thread. "Others" represents the rest of the time period of the virtualization overhead, which is incurred in the execution of the guest OpenCL library, the synchronization between the vCPU thread and CL thread, the execution of the CL thread, etc. The overhead distribution is as follows: the overhead caused by data transmissions ranges from 78.3% and 94.1%, and the others ranges from 5.9% to 21.7%. As expected, the data transmissions dominate the OpenCL virtualization overhead due to the nature of API Remoting.

Table 4.3: The detailed OpenCL virtualization overhead breakdown.

	Overhead (Sec.)	G to H[†] (Sec.)	H to G[‡] (Sec.)	Others (Sec.)
FastWalshTransform[FWT]	0.019548	0.008987	0.008248	0.002313
BlackScholes [BS]	0.092471	0.044952	0.042076	0.005443
MersenneTwister [MT]	1.416176	0.042463	1.066627	0.307087
MatrixManipulation [MM]	0.105522	0.058606	0.035610	0.011306
ScanLargeArray [SLA]	0.004884	0.002477	0.002062	0.000345
ConvolutionSeparable [CS]	0.133159	0.048418	0.067795	0.016946

(†: guest to hypervisor ; ‡: hypervisor to guest.)

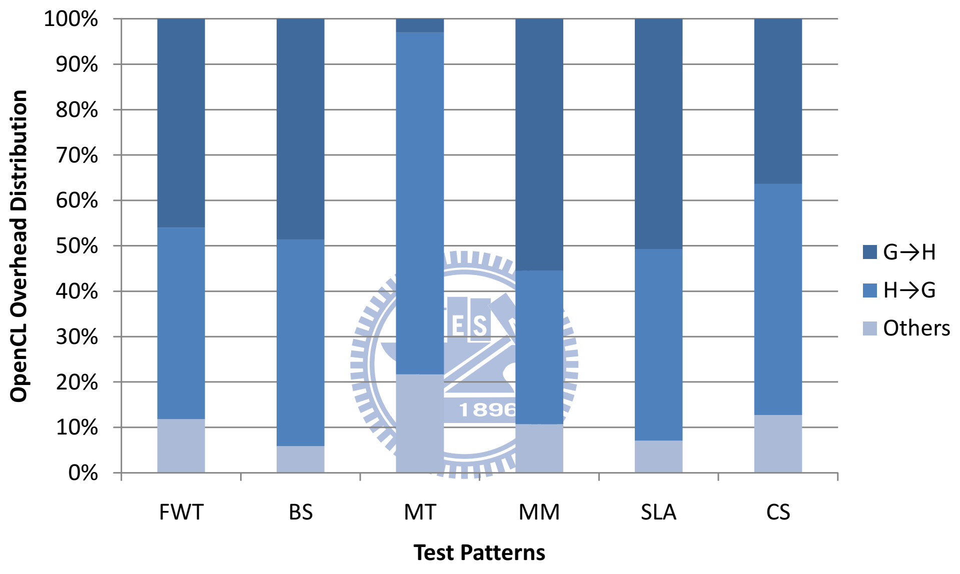


Figure 4.4: The breakdown of the virtualization overhead.

We collected the amount of data transmission to and from the hypervisor and the number of VM exits to investigate the relationship among overhead and the above two factors. The profiling information are shown in Table 4.4 and Figure 4.5. In general, the more data being transferred between the host and the device in an OpenCL program, the more number of VM exits because the data copy process between the guest and the hypervisor requires a series of VM exits. It requires two data copies from the guest to the hypervisor and three copies from the hypervisor to the guest for a data transfer of an OpenCL API call. Thus, both VM exits and data transmission cause additional overhead. But the relationship can

Table 4.4: Profiling information of data transfer and VM exits.

	G to H (MB)†	H to G (MB)‡	Total (MB)	# of VM Exits
FastWalshTransform[FWT]	3.02	0.05	3.07	744
BlackScholes [BS]	46.75	30.52	77.27	485
MersenneTwister [MT]	0.75	183.13	183.88	1541
MatrixManipulation [MM]	32.75	16.01	48.76	482
ScanLargeArray [SLA]	0.50	0.01	0.51	132
ConvolutionSeparable [CS]	36.75	36.00	72.75	645

(†: guest to hypervisor ; ‡: hypervisor to guest)

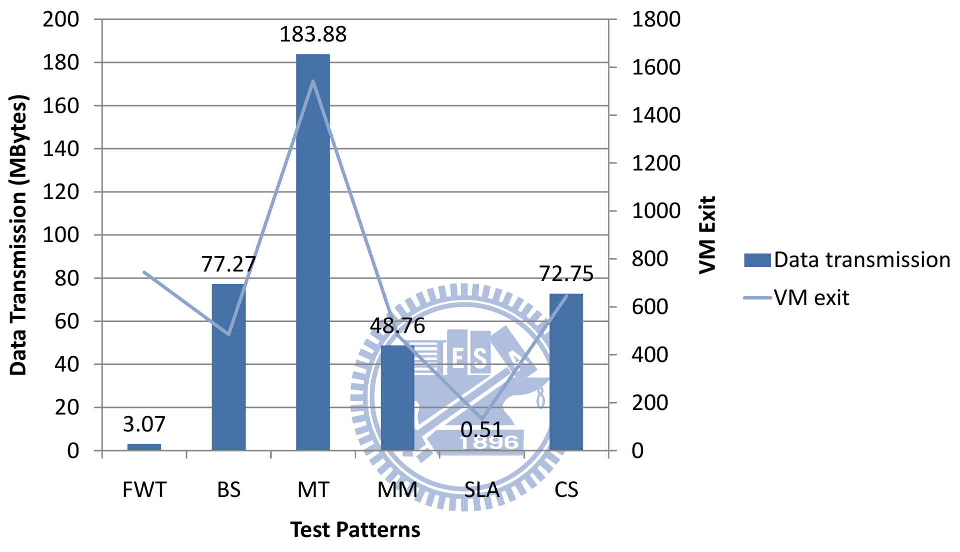


Figure 4.5: Profiling information of data transfer and VM exits.

not be clarified by the profile information because each benchmark program has its specific behavior which causes different amount of data transmission and number of VM exits.

Variable Input Size

To clarify the relationships among the amount of data transmission, the number of VM exits and the OpenCL virtualization overhead, we took $BlackScholes$, which implements a mathematical model for financial options, and investigated it with variable input sizes by adjusting the value of variable $optionCount$ from two millions (2×10^6) to sixteen millions (16×10^6). Table 4.5 and Figures 4.6 and 4.7 summarize the evaluation results and the profiling statistics including both the amount of data transmission and the number

Table 4.5: OpenCL virtualization overhead and profile information of BlackScholes [BS] with variable input size.

optionCount	Native (Sec.)	1VM (Sec.)	Overhead (Sec.)	Overhead (%)	Data Volume (MB)	# of VM Exit
2,000,000	0.737729	0.787253	0.049524	6.7	38.76	268
4,000,000	0.752795	0.869053	0.116258	15.4	77.27	485
6,000,000	0.783554	1.022023	0.238469	30.4	115.03	695
8,000,000	0.820061	1.433890	0.613829	74.9	153.54	912
10,000,000	0.852416	2.825916	1.973500	231.5	191.86	1124
12,000,000	0.877067	4.280623	3.403556	388.1	229.81	1339
14,000,000	0.908456	5.725110	4.816654	530.2	269.69	1557
16,000,000	0.946489	7.266876	6.320387	667.8	306.07	1766

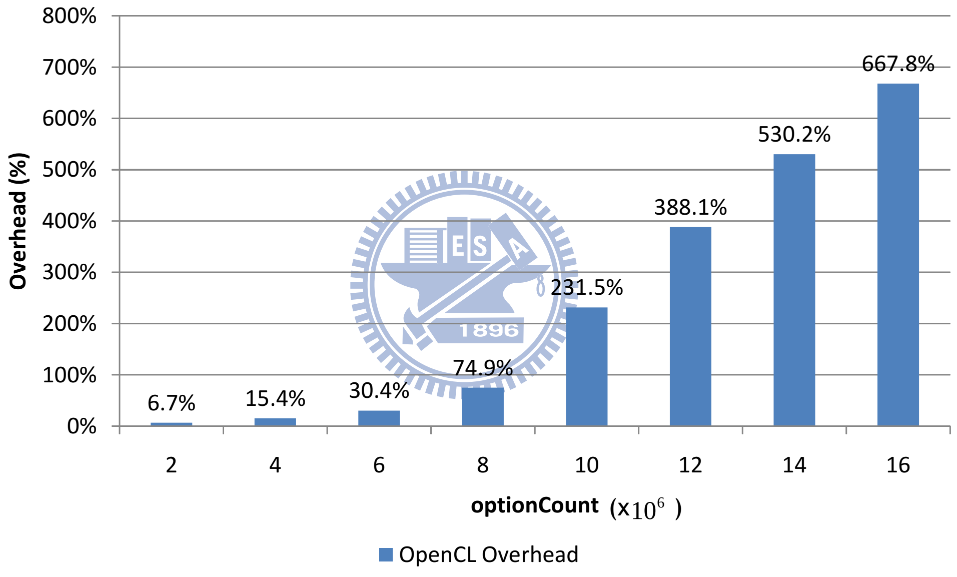


Figure 4.6: OpenCL virtualization overhead of BS with variable input size.

of VM exits. We observe that both the amount of data transmission and the number of VM exits are proportional to the input data size, and the OpenCL virtualization overhead increases in more than a proportional manner. Figure 4.8 show the detailed breakdown of the virtualization overhead of $BlackScholes$ with variable input size. We observed that the overhead contribution of "Others" part is a relatively stable factor which ranges from 12.9% to 22.0%. On the other hand, the ratio of "hypervisor to guest" increases from 43% to about 80% when the optionCount grows to 16 millions, and the ratio of "guest

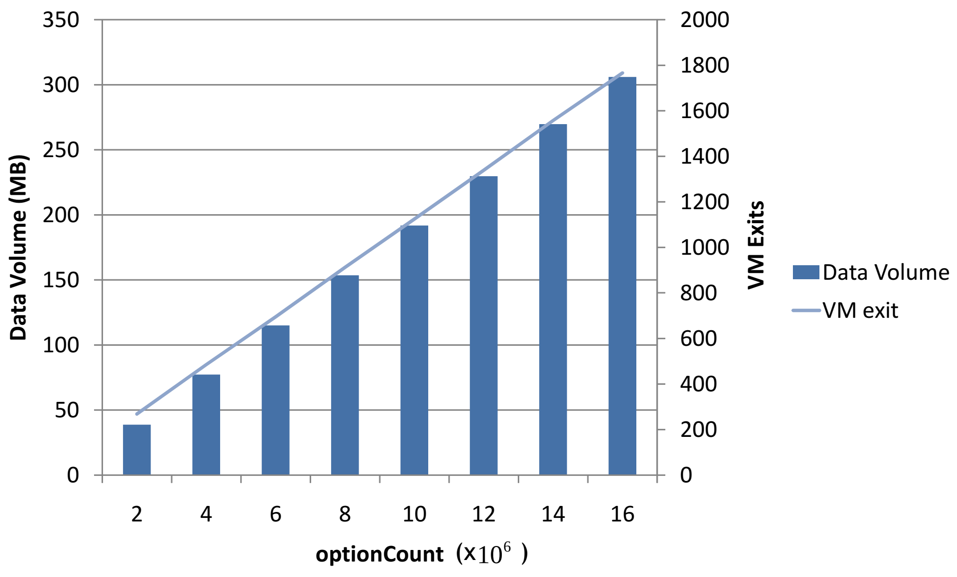


Figure 4.7: Profile information of BS with variable input size.

to hypervisor" decreases relatively. As mentioned in Section 3.2.2, the data transmission from the guest to the hypervisor is actively done by the vCPU thread and requires a series of VM exits. The data transmission from the hypervisor to the guest is actively done by the CL thread, and requires a series of VM exits and virtual interrupts. Because of the different implementation of data transmission, the overhead of "hypervisor to guest" grows faster with the increasing transmitted data.

In conclusion, the virtualization overhead in our framework is highly affected by the amount of data transmission and the number of VM exits of OpenCL programs, and the characteristic is also obvious in other API Remoting approaches.

4.2.2 Evaluation in Multiple Guest VMs

The following experiment is used to evaluate the scalability of our OpenCL virtualization framework. We executed multiple guest VMs, and each guest VM ran an arbitrary benchmark program simultaneously. There are two configurations: **2VM** and **3VM**, and they represent the scenario of concurrently executing two or three OpenCL programs by differ-

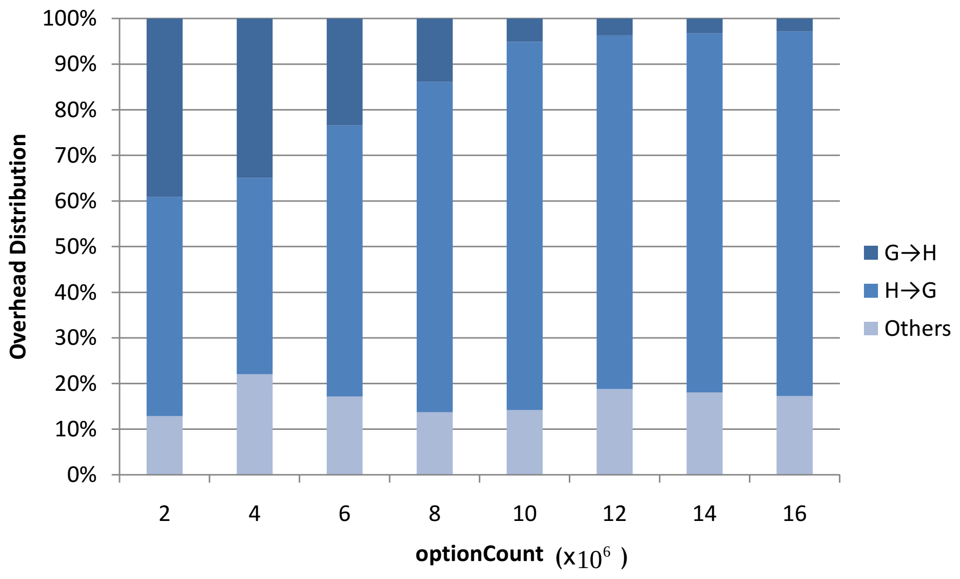


Figure 4.8: The breakdown of OpenCL virtualization overhead of variable input size for BlackScholes [BS].

ent guest VMs. Figure 4.9 shows the normalized execution time of each benchmark under the native environment, the **1VM**, the **2VM** and the **3VM** configurations. We observed that the virtualization overhead in the **2VM** configuration is smaller than in **1VM**, even smaller than in the native environments in most of the benchmarks such as FWT, BS, MM, SLA, and CS. The virtualization overhead in the **3VM** configuration is bigger than in **2VM**, but still smaller than in **1VM**.

The experimental results shown in Figure 4.9 seem not reasonable because we expect the virtualization overhead in the **2VM** and **3VM** configurations would be more than that in the **1VM** configuration. In order to clarify the reason, we collected the execution time of each OpenCL API function in the guest OpenCL library and performed in-depth analyses of the interaction among two or three OpenCL programs residing in two of three different VMs. We observed that the execution time of OpenCL API calls for resource initialization in one program would be decreased dramatically if the other program(s) had completed its own resource initialization. Moreover, the execution time difference for these API calls

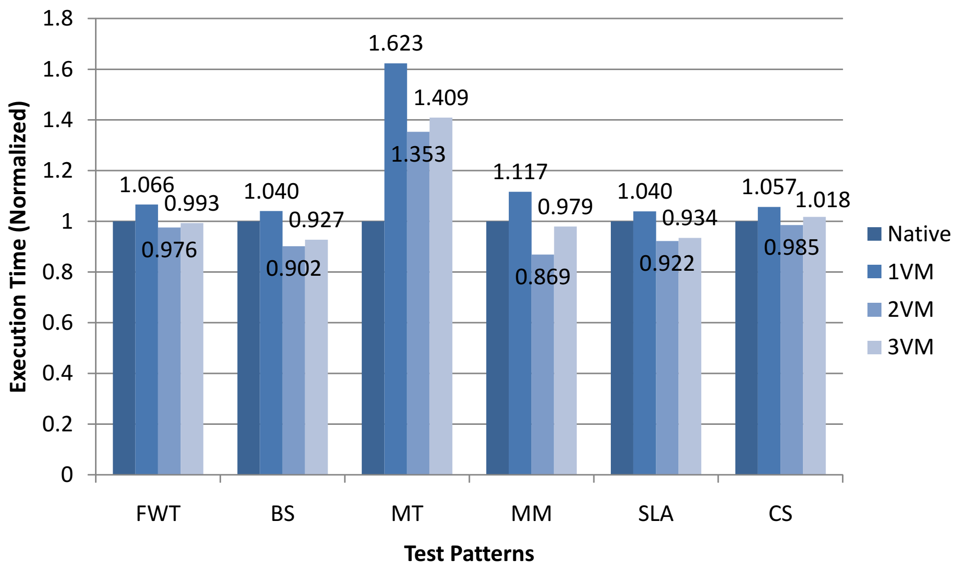


Figure 4.9: Normalized execution time for the Native, 1VM, 2VM, and 3VM configurations.

almost matches the difference in overall execution time of the OpenCL program between the **1VM** and **2VM** configurations. Therefore, we strongly believe that the abnormality of the **2VM** and **3VM** configurations outperforming the **1VM** configuration is attribute to the implementation of the vendor-specific OpenCL runtime, which may contain a resource pool for maintaining the OpenCL resource initialization.

Figure 4.10 illustrates a scenario of multiple guest VMs accessing the vendor-specific OpenCL runtime. As mentioned in Section 3.2.1, the lifetime of a CL thread almost spans the hypervisor process it belongs to. Once an OpenCL process in a guest VM starts execution, the CL thread in the guest VM accesses the vendor-specific OpenCL runtime and interferes the resource allocation scheme in the vendor-specific OpenCL runtime, which somehow keeps the information about the underlying OpenCL resources until the CL thread ends, i.e., the guest VM is shut down, even if the OpenCL resources are released by OpenCL programmers. Consequently, the execution time of resource initialization-related API functions is nearly eliminated if a CL thread is active. Therefore, the performance

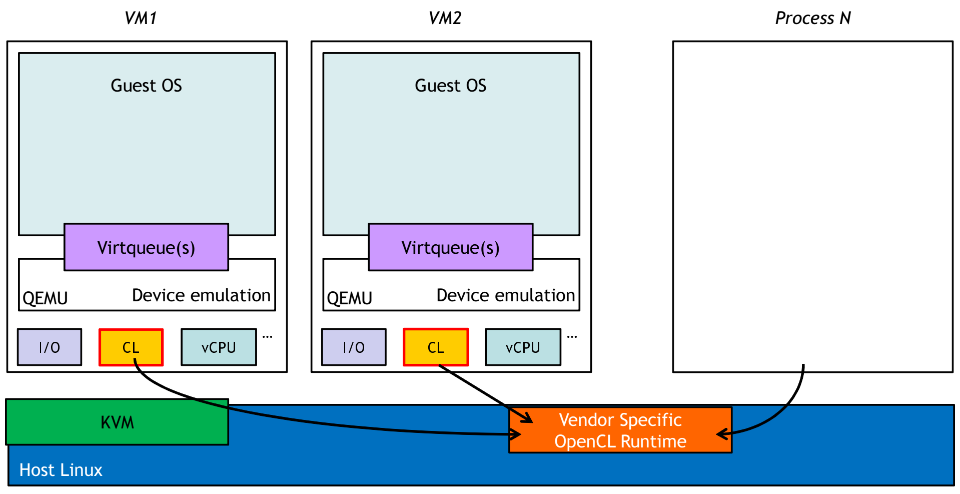


Figure 4.10: A Scenario of multiple VMs accessing the vendor-specific OpenCL runtime.

of OpenCL virtualization under the 2VM and 3VM configurations surpasses that under the 1VM configuration. In addition, without considering the influence of the resource initialization mechanism of the runtime implementation, the virtualization overhead grows slowly as the number of guest VMs increases. This implies that with the virtualization framework, the underlying OpenCL resources can be better utilized without further intervention of programmers.

4.2.3 A Brief Comparison with Related Work

In literature, there have been some researches working on API Remoting for virtualization. GViM [15] and vCUDA [28], which will be discussed in detail in Chapter 5, are two of them whose objectives are conceptually similar to this work. Briefly speaking, both GViM and vCUDA were designed to support CUDA virtualization in Xen-based virtual machines [10] for high performance computing while this work enabled OpenCL support in Kernel-based Virtual Machine. In this section, we will make a brief comparison in terms of virtualization overhead among GViM, vCUDA and this work, though the comparison might not be used to claim the superiority of our approach over the others due to the different evaluation configurations such as different underlying hardware platforms, different

Table 4.6: Configurations of input data size in this work and the related works.

	This Work	GViM	vCUDA
BlackScholes [BS]	16MB	4MB	16MB
ConvolutionSeparable [CS]	36MB	——	36MB
FastWalshTransform[FWT]	32MB	64MB	32MB
MersenneTwister [MT]	22.89MB	——	22.89MB
ScanLargeArray [SLA]	0.004MB	——	N/A
MatrixManipulation [MM]	2KB × 2KB	2KB × 2KB	——

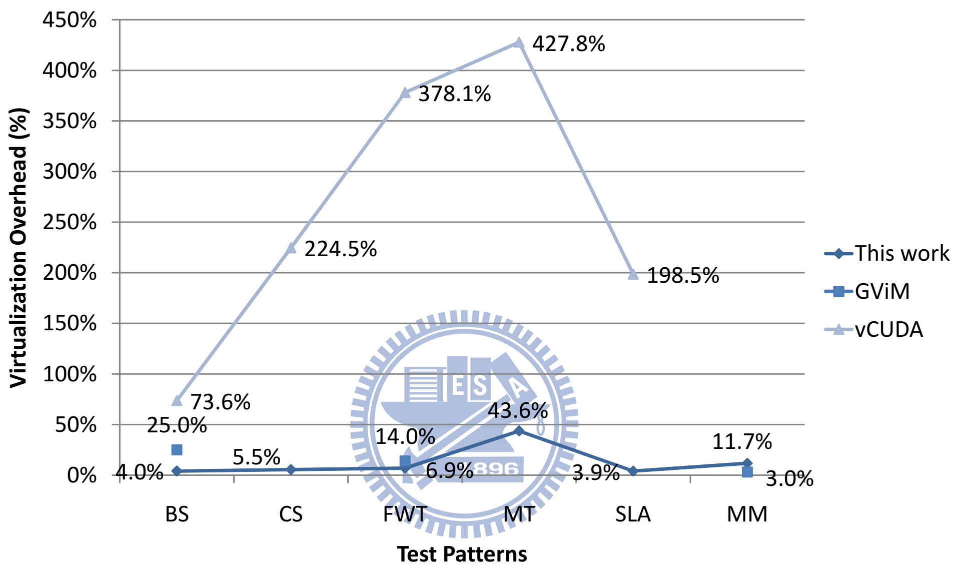


Figure 4.11: Comparison of virtualization overhead among different virtualization framework.

implementations of benchmark programs in CUDA and OpenCL, different input data size, etc. Nevertheless, we can still get a rough figure of how much virtualization overhead the proposed virtualization framework has in compared with other frameworks. Table 4.6 summarizes the input data size of benchmark programs among the three frameworks.

Figure 4.11 shows the virtualization overhead of CUDA or OpenCL programs in GViM, vCUDA and this work—some results are not available for GViM and vCUDA. The overhead of the benchmark programs in vCUDA ranges from 73.6% (BS) to 427.8% (MT), which is much more than the overhead in GViM and this work. We believe the high virtualization overhead in vCUDA is attributed to its data transmission scheme between hypervisor and guests, XML-RPC, which wraps an API call as a remote procedure call, whereas GViM and our framework use shared memory-based approaches for the communication between hypervisor and guests. However, vCUDA has the capability to redirect API requests to remote devices for coarse-grain resource management in cluster environments. The virtualization overhead in GViM and in our framework are relatively similar in terms of percentages. More specifically, our framework has 4.0%, 6.9%, and 11.7% of virtualization overhead for BS, FWT, and MM, respectively, whereas GViM has 25.0%, 14.0%, and 3.0%, respectively. More discussions about the comparison among the three research work will be given in Chapter 5.

Chapter 5

Related Work

OpenCL has become an industrial standard in the field of heterogeneous multi-core computing. Many industry-leading companies (AMD, Apple, IBM, Intel, NVIDIA, etc.) have released their OpenCL runtime implementations to help users access the abilities of OpenCL [1, 4, 5, 7, 8]. However, there are no standard solutions to support OpenCL execution in a virtualized system in either industry or academia. This work is the first research to support OpenCL in system virtualization environment based on API Remoting. Although the framework only support GPU devices currently, it is easy to extend it with API Remoting for supporting more OpenCL devices.

There were several researches adopting API Remoting for their virtualization approach. Lagar-Cavilla et al. proposed VMGL, a solution of supporting OpenGL graphics acceleration API in Xen or VMware-based virtual machines with the functionalities of GPU multiplexing and suspension and resumption of operations [22]. VMGL virtualizes OpenGL API and uses an advanced OpenGL network transmission scheme called WireGL [12] to optimize the data transmissions of OpenGL contexts. A. Weggerle et al. proposed VirtGL, an OpenGL virtualization solution for QEMU [29]. VirtGL implements OpenGL API Remoting by adding a new QEMU virtual device, and the data transmission in VirtGL is proposed via the MMIO spaces of the virtual device. Vishakha Gupta et al. proposed GViM, a CUDA virtualization solution for Xen hypervisor [15]. GViM uses the shared

Table 5.1: A Comparison of API Remoting-based virtualization frameworks.

	Target API	Target VM	Guest-host VM Communication	Main Features
VMGL	OpenGL	Xen VMware, ...	WireGL [12]	GPU Multiplexing Suspension and resumption
VirtGL	OpenGL	QEMU	QEMU virtual device (shared memory based)	None
GViM	CUDA	Xen	Shared ring for function calls	GPU Multiplexing Credit-based GPU scheduling
vCUDA	CUDA	Xen	XML-RPC Lazy RPC (by authors)	GPU Multiplexing Suspension and resumption
rCUDA	CUDA	Clusters (Not system VMs)	RPC	Remote execution in cluster environments
This Work	OpenCL	KVM	Virtio-CL (shared memory based)	GPU Multiplexing

memory-based I/O communication mechanism in Xen hypervisor to implement the data transmission between guest VMs and the dedicated VM that emulates I/O actions (Dom0 in Xen). GViM supports *GPU multiplexing* and implements a simple credit-based scheduling mechanism for CUDA API. Shi et al. proposed a CUDA API Remoting framework, called vCUDA, for Xen hypervisor [28]. vCUDA uses a remote procedure call (RPC) scheme named XML-RPC [9] to perform data transmission between hypervisor and guests. The authors introduced a “lazy RPC” mechanism to reduce the frequency of RPC and thus the virtualization overhead. José Duato et al. proposed rCUDA [14], a framework for executing CUDA applications in clusters environments, which contain CUDA devices in certain cluster nodes. rCUDA uses RPC-based mechanism to perform data transmission among different cluster nodes.

Table 5.1 summarizes a comparison among the related work and our work. In comparison with GViM and vCUDA, our virtualization framework enables OpenCL support in KVM, whereas GViM and vCUDA allow CUDA programs executing within Xen-based virtual machines. Unlike OpenCL, which supports different kinds of heterogeneous computing devices, CUDA is only supported on machines with NVIDIA’s GPUs. Our proposed OpenCL virtualization framework provides a solution for heterogeneous multi-core computing in virtual machines. Regarding to the underlying virtualization framework, both KVM and Xen are well-known frameworks in system virtualization area. Xen supports para-virtualization and hardware-assisted virtualization, and KVM supports hardware-assisted virtualization for virtualizing CPUs. The performance of CPU virtualization in KVM and Xen are both excellent against other virtualization frameworks. However, the architecture of I/O virtualization in Xen is much different from in KVM. The I/O actions of guest VMs in Xen have to be intercepted and processed by a specific guest VM called **I/O domain** (or **Dom0**), and thus the I/O domain becomes the bottleneck of I/O virtualization and results in unpredictable I/O bandwidth and latency [25] [19]. In KVM, I/O virtualization is processed in the same hypervisor process, and the architecture of I/O processing is much simpler. Therefore, we consider that KVM is a better platform for OpenCL virtualization than Xen.

Chapter 6

Conclusion and Future Work

6.1 Summary

In this thesis, we enabled OpenCL support in Kernel-based Virtual Machine by API Remoting. The proposed OpenCL virtualization framework allows multiplexing of multiple guest VMs over the underlying OpenCL resources to make better utilization of the resources. Owing to the characteristic that in KVM each VM is a process in Linux, virtualizing OpenCL programs in multiple guest VMs can be modeled as multiple processes accessing the OpenCL resources in the native environment. Although, in our current implementation, the virtualization framework only supports running OpenCL programs in GPGPU architectures, the API Remoting scheme is easy to extend for supporting other OpenCL devices.

The evaluation indicates that the virtualization overhead of the proposed framework mainly come from the data transmission and the synchronization between KVM's "vCPU thread" and our new added "CL thread", which are directly affected by the amount of data to be transferred for an API call because of the primitive nature of API Remoting and the limited size of the shared memory between hypervisor and guest VMs. The experimental results show that the virtualization overhead is only less than 10% (6.4% on average) for five common GPU-intensive OpenCL programs. Furthermore, a better utilization of OpenCL resources is achieved due to the support of OpenCL multiplexing of the framework.

6.2 Future Work

The results in this thesis provide a strong foundation for future work in OpenCL virtualization:

- As discussed in Section 3.3.5, the guest OpenCL library can be substituted with a *process virtual machine* version. The new guest OpenCL library not only eliminates the need of modifying OpenCL programs but ensures the memory coherence of OpenCL memory objects between guest VMs and the hypervisor.

- We will try to reduce the overhead of data transmissions by leveraging the MMU virtualization of x86 architectures, such as Intel EPT and AMD NPT (described in Section 2.3).

- Currently, our virtualization framework supports OpenCL for GPGPU computation which covers most of OpenCL applications. We will support more OpenCL functionalities such as operations of image object and OpenGL extensions to complement the supports.

- For better resource utilization among multiple OpenCL workloads, we will evaluate the performance impact in KVM and enhance the scheme of OpenCL resource management in system VM level.

Bibliography

[1] AMD Accelerated Parallel Processing (APP) SDK. `http://developer.amd.com/sdks/amdappsdk/pages/default.aspx`.

[2] CUDA: Compute Unified Device Architecture. `http://www.nvidia.com/object/cuda_home_new.html/`.

[3] Intel Virtualization Technology. `http://www.intel.com/technology/virtualization/`.

[4] Intel®OpenCL SDK. `http://software.intel.com/en-us/articles/opencl-sdk/`.

[5] NVIDIA OpenCL SDK. `http://developer.nvidia.com/opencl`.

[6] OpenCL—The open standard for parallal programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[7] OpenCL Development Kit for Linux on Power. `http://www.alphaworks.ibm.com/tech/opencl/`.

[8] OpenCL, taking the graphics processor beyond graphics. `http://developer.apple.com/technologies/mac/snowleopard/opencl.html`.

[9] XML-RPC Specification. `http://www.xmlrpc.com/spec`.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[11] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[12] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '00, pages 87–95, New York, NY, USA, 2000. ACM.

[13] Micah Dowty and Jeremy Sugerman. GPU virtualization on VMware's hosted I/O architecture. *SIGOPS Operating Systems Review*, 43:73–82, July 2009.

[14] J. Duato, F. D. lgual, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, and F. Silla. An efficient implementation of GPU Virtualization in High Performance Clusters. In *Euro-Par 2009*, Euro-Par '09, 2009.

[15] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[16] Intel Cooperation. *Intel® Virtualization Technology for Directed I/O*, 1.3 edition, 02 2011.

[17] M. Tim Jones. Virtio: An I/O virtualization framework for Linux. `http://www.ibm.com/developerworks/linux/library/l-virtio/`.

[18] Khronos Group. *The OpenCL Specification*, 1.1 edition, 06 2011.

[19] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 101–110, New York, NY, USA, 2009. ACM.

[20] Jan Kiszka. Architecture of the Kernel-based Virtual Machine (KVM). Technical Sessions of Linux Kongress, 2010.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*, OLS '07, pages 225–230, 2007.

[22] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. VMM-independent graphics acceleration. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 33–43, New York, NY, USA, 2007. ACM.

[23] Ravi Nair. *Virtual machines : versatile platforms for systems and processes*. Morgan Kaufmann Publishers, 2006.

[24] NVIDIA Corporation. *NVIDIA's Next Generation CUDA^{TM}Compute Architecture: Fermi^{TM}*, 1.1 edition, 2009.

[25] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling I/O in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[26] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17:412–421, July 1974.

[27] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review*, 42:95–103, July 2008.

[28] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –11, May 2009.

[29] A. Weggerle, T. Schmitt, C. Low, C. Himpel, and P. Schulthess. VirtGL - a lean approach to accelerated 3D graphics virtualization. In *Cloud Computing and Virtualization 2010*, CCV '10, 2010.