

東海大學
資訊工程研究所

碩士論文

指導教授：楊朝棟 博士

使用 PCI 穿透於圖形處理器虛擬化之實作

On Implementation of GPU Virtualization Using PCI
Pass-Through

研究生：王顯義

中華民國一〇一年七月

摘要

現今，NVIDIA 公司的 CUDA 是一種為了撰寫高度平行的應用程式，所發展出來的通用型可伸縮式的平行程式設計模組。它提供了一些關鍵的抽象化概念：一個有層次的線程區塊、共享式的記憶體和屏障同步。在整個工業界和學術界的科學家們已經使用 CUDA 在生產或是研究的程式碼時，有了驚人的加速性。這模組在編寫在多核心的圖形處理器上運用的多線程程式時，已經證明是非常成功的了。內建圖形處理器的叢集環境在雲端運算中扮演一個重要的角色。因為一些高強度運算的程式需要中央處理器及圖形處理器一起運算。本篇論文中，我們以 PCI 穿透的技術，使得在虛擬環境中的虛擬機器得以使用 NVIDIA 的顯示卡，進而可以使用 CUDA 高效能運算。這將使得虛擬機器不僅只能有虛擬的中央處理器，更可以使用實體的圖形處理器來做運算，虛擬機器的效能將可大幅提升。本論文中將會量測虛擬機與實體機之間使用 CUDA 的效能差異，以及擁有不同中央處理器的虛擬機器是否會影響到 CUDA 效能。最後，我們將會比較兩套開源程式碼的虛擬環境，是否會對經過 PCI 穿透所使用的 CUDA 造成效能上的差異。透過實驗將可以知道哪個環境將會對在虛擬環境中使用 CUDA 有最佳的效能。

關鍵字：CUDA、圖形處理器虛擬化、雲端計算、PCI 穿透

Abstract

Nowadays, NVIDIA's CUDA is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions – a hierarchy of thread blocks, shared memory, and barrier synchronization. This model has proven to be quite successful at programming multithreaded many core GPUs and scales transparently to hundreds of cores: scientists throughout industry and academia are already using CUDA to achieve dramatic speedups on production and research codes. GPU-base clusters are likely to play an important role in future cloud computing centers, because some compute-intensive applications may require both CPUs and GPUs. In this thesis by using PCI pass-through technology and making the virtual machines in a virtual environment are able to use the NVIDIA graphics card, and we can use the CUDA high performance computing as well. It makes the virtual machine have not only the virtual CPU but also the real GPU for computing. The performance of virtual machine is predicted to increase dramatically. This thesis will measure the performance differences between virtual machines and physical machines by using CUDA; and how virtual machines would varyify CPU numbers under influence of CUDA performance. At last, we compare two open source virtualization environment hypervisor, whether it is after PCI pass-through CUDA performance differences or not. Through the experiment, we will be able to know which environment will reach the best efficiency in a virtual environment by using CUDA.

Keywords: CUDA, GPU virtualization, Cloud Computing, PCI pass-through

Acknowledgements

It is a pleasure to thank many people who made this thesis possible. It is difficult to overstate my gratitude to my supervisor, Prof. Chao-Tung Yang, who support my work and show me the way on this thesis, also give me a deep influence and inspiration. Prof. Yang gave me a lot of help on my study. I would like to thank Prof. Ching-Hsien Hsu, Prof. Wen-Chung Shih, Prof. Kuan-Chou Lai, and Prof. Cheng-Chung Chu for their valuable comments and advice given while serving on my reading committee. I also want to thank all HPC members who gave me lots of help, and to accompany me these days. And they share their knowledge with me selfless. Finally, thanks to my family who support me whole my life.

Table of Contents

摘要.....	I
Abstract.....	II
Acknowledgements	III
Table of Contents	IV
List of Tables.....	VI
List of Figures.....	VII
Chapter 1 Introduction.....	1
1.1. Motivations	1
1.2. Goal and Contribution.....	2
1.3. Thesis Organization	3
Chapter 2 Background Review	4
2.1. Cloud Computing.....	4
2.2. Virtualization.....	6
2.2.1. Full-Virtualization.....	8
2.2.2. Para-Virtualization	10
2.2.3. Xen.....	11
2.2.4. KVM	12
2.3. CUDA	13
2.4. Virtualization on GPU.....	15
2.5. Green Computing.....	18
2.6. Related Work.....	19
Chapter 3 System Implementation.....	24
3.1. System Architecture	24
3.2. Tesla C1060 Computing Processor Board	26
3.3. Tesla C2050 Computing Processor Board	28
3.4. End User’s Operating Interface.....	28
3.5. System Environment	31
Chapter 4 Experimental Methods and Results	34
4.1. Experimental Methods	34
4.2. Experimental Results	35
Chapter 5 Conclusions and Future Work.....	52

5.1. Concluding Remark	52
5.2. Future Work	53
Bibliography	54
Appendix.....	58
A. Setup Xen on CentOS	58
B. Setup PCI passthrough.....	59
C. CUDA Installation	60

List of Tables

Table 3-1. Hardware/Software Specification	31
Table 3-2. Hardware/Software Specification of Virtual Machine	32
Table 3-3. GPU Software Environments	33
Table 4-1. Data transfers of Benchmarks	34

List of Figures

Figure 2-1. Architecture of Cloud Computing.....	5
Figure 2-2. Virtualization Diagram	6
Figure 2-3. The General Operation System	7
Figure 2-4. The Virtualization Operation System	8
Figure 2-5. Full-Virtualization	9
Figure 2-6. Para-Virtualization.....	10
Figure 2-7. Host and Hypervisor Type.....	11
Figure 2-8. Domain0 and DomainU.....	12
Figure 2-9. Architecture of KVM	13
Figure 2-10. CUDA Programming Model from nVidia[2]	14
Figure 2-11. Processing Flow on CUDA from Wiki[16]	15
Figure 2-12. User Space Device Emulation.....	16
Figure 2-13. Hypervisor-Based Device Emulation.....	17
Figure 2-14. Pass-through within the Hypervisor	18
Figure 2-15. Front-End and Back-End.....	20
Figure 2-16. Architecture of rCUDA	21
Figure 2-17. The vCUDA Architecture	22
Figure 3-1. IOMMU On.....	24
Figure 3-2. System Architecture	25
Figure 3-3. User Architecture.....	26
Figure 3-4. Tesla T10	27
Figure 3-5. PCI Pass-through Successful.....	29
Figure 3-6. Shows in Piety	30
Figure 3-7. Shows in VNC.....	30
Figure 3-8. Using lspci.....	31
Figure 4-1. Execution Time between Native and VM with C1060.....	36
Figure 4-2. Execution Time between Native and VM with C2050.....	36
Figure 4-3. Execution Time between Native and VM with NVS295	37
Figure 4-4. Execution Time between 1 Core and 2 Core VM with C1060.....	38
Figure 4-5. Execution Time between 1 Core and 2 Core VM with C2050.....	38
Figure 4-6. Execution Time between 1 Core and 2 Core VM with NVS295	39
Figure 4-7. Execution Time between 2 Core and 4 Core VM with C1060.....	39
Figure 4-8. Execution Time between 2 Core and 4 Core VM with C2050.....	40
Figure 4-9. Execution Time between 2 Core and 4 Core VM with NVS295	40
Figure 4-10. User Time with C1060	41
Figure 4-11. User Time with C2050.....	41

Figure 4-12. User Time with NVS295	42
Figure 4-13. System Time with C1060	43
Figure 4-14. System Time with C2050	43
Figure 4-15. System Time with NVS295	44
Figure 4-16. Bandwidth Test with C1060	45
Figure 4-17. Bandwidth Test with C2050	45
Figure 4-18. Bandwidth Test with NVS295	46
Figure 4-19. Execution time of VecAdd with C1060	47
Figure 4-20. Execution time of VecAdd with C2050	47
Figure 4-21. Execution time of VecAdd with NVS295	48
Figure 4-22. Execution time of MatrixMul with C1060	49
Figure 4-23. Execution time of MatrixMul with C2050	49
Figure 4-24. Execution time of MatrixMul with NVS295	50
Figure 4-25. Compare with rCUDA	50
Figure 4-26. Compare with vCUDA	51

Chapter 1

Introduction

1.1. Motivations

GPUs are real “manycore” processors with hundreds of processing elements. A graphics processing unit (GPU) is a specialized microprocessor that offloads and accelerates graphics rendering from the central (micro-) processor. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structures make them more effective than general-purpose CPUs for a range of complex algorithms. We know that a CPU has only 8 cores at single chip currently but a GPU has grown to 448 cores. From the number of cores, GPU is appropriate to compute the programs suited massive parallel processing. Although frequency of the core on the GPU is lower than CPU’s, we believe that massively parallel can conquer the problem of lower frequency. As for GPU, it has been used on supercomputers. On top 500 sites in November 2010 [1], there are three supercomputers of the first 5 built with NVIDIA GPU [2].

In recent years, virtualization environment on Cloud [3] becomes more popular than ever. The balance between performance and cost is the most important matter we focus. In order to live up the potential of the server resource, virtualization technology is the main solution for running more virtual machines on a server and yet the resource can be used a lot more effectively. However, the performance of virtual machines has their own limitations so that users can be limited by using lots of computing on it.

Building a virtual environment in a cloud computing system for users has become an important trend in the last few years. Proper use of hardware resources and computing power to each virtual machine are the Infrastructure as a Service (IaaS) as one of the architectures of Cloud Computing. Nevertheless, virtual machine has limitation that system virtual environment does not support CUDA. A new topic-the physical GPGPU (General-Purpose computing on Graphics Processing Units) [4] is used by virtual machine in the real machine to help compute. Since GPU is real manycore processors, the computing power of virtual machines will be increasing.

1.2. Goal and Contribution

In this thesis, we introduce some hypervisor environments for virtualization and different virtualization types on cloud system. Several types of hardware virtualization are introduced as well. This thesis focuses on GPU virtualization and implements a system with virtualization environment and uses PCI pass-through [5] technology which lets the virtual machines on the system use GPU accelerator to increase the computing power. We do several experiments to compare the differences between virtual machine with GPU virtualization (PCI pass-through) and native machine with GPU. At last, we show the performance of GPU between virtual machines and native machine and compare virtual machine with native machine again. Moreover, we analyze other GPU virtualization technologies. The experiment results present the difference between PCI pass-throughs with other GPU virtualization technologies.

1.3. Thesis Organization

The rest parts of this work are organized as follows. Chapter 2 describes a background review of Cloud Computing, Virtualization technology, and CUDA (Compute Unified Device Architecture) [6]. Chapter 3 describes the system implementation, Tesla C1060's architecture and its specification and end-user's interface. Chapter 4 presents experimental environment and the methods we use and results of GPU virtualization and show that the proposed approach improved performance. Finally, conclusions are discussed in Chapter 5.

Chapter 2

Background Review Cloud Computing

Cloud Computing [3] is a computing approach based on the Internet, which is a world in recent years lively discussion. That means use software service and data storage in remote server. It is a new service architecture that brings new a selection of software or data storage service to users. Users no longer need to find out the “Cloud” in details of the infrastructure, no necessary to know the professional knowledge, without direct control the real machine.

National Institute of Standards and Technology, aka NIST, define five basic features for Cloud Computing at April 2009[7].

- On-demand Self-service
- Broad Network Access
- Resource Pooling
- Rapid Elasticity
- Measured Service

Cloud computing can be considered include the three levels of service: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [3].

- Infrastructure as a Service (IaaS): Users can follow the required level of computer and network equipment and other resources, to the service provider subscription service, and may require changes to settings, and provider by users of the CPU, memory, Disk space, network load to calculate the costs.

- Platform as a Service (PaaS): development of services vendors who rent to a computer, this computer has all the necessary hardware and software developers environment; or to provide application developers to market, in accordance with the amount of traffic with the use of resource Developer fees.
- Software as a Service (SaaS): the software stored in the data center to provide users network access services, according to provide or pay-per-order the type of charge.

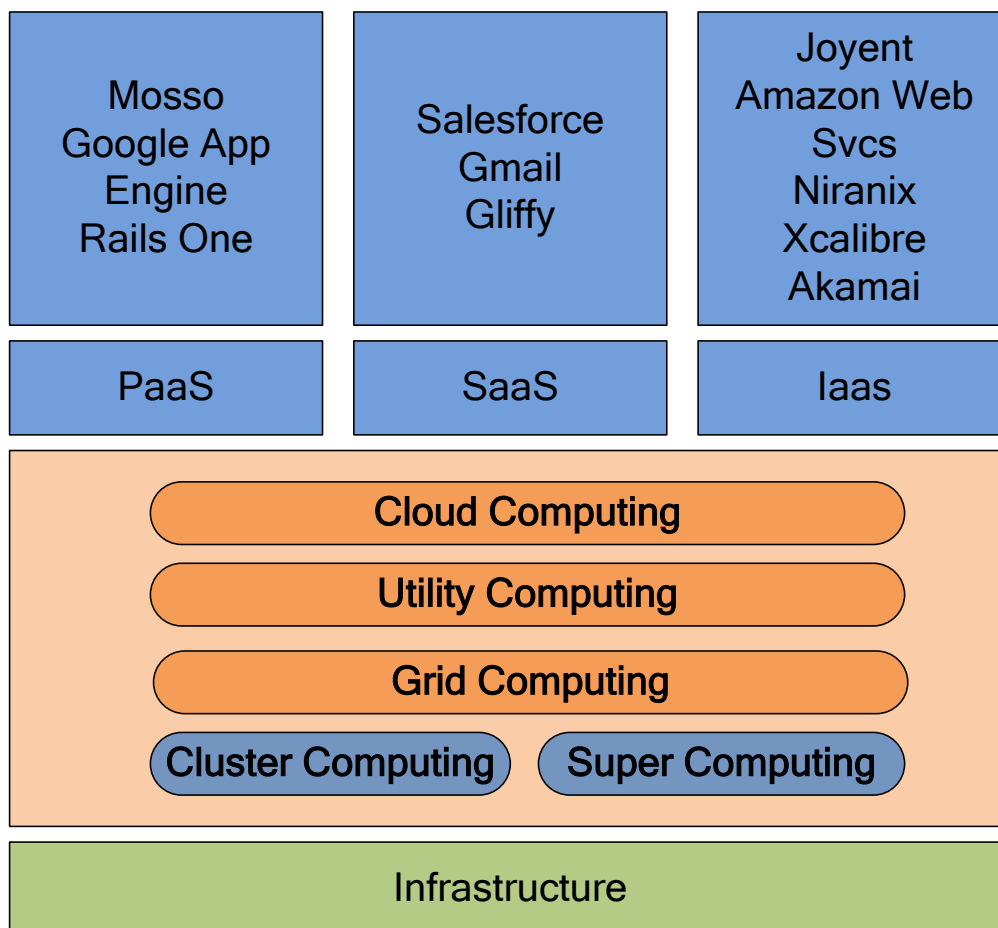


Figure 2-1. Architecture of Cloud Computing

2.2. Virtualization

Virtualization technology [8] is a technology that creation of a virtual version of something, such as a hardware platform, operation system, a storage device or network resources. The goal of virtualization is to centralize administrative tasks while improving scalability and overall hardware-resource utilization. By using virtualization, several operating systems can be run on a single powerful server without glitch in parallel. The virtualization diagram is shown on **Figure 2-2**.

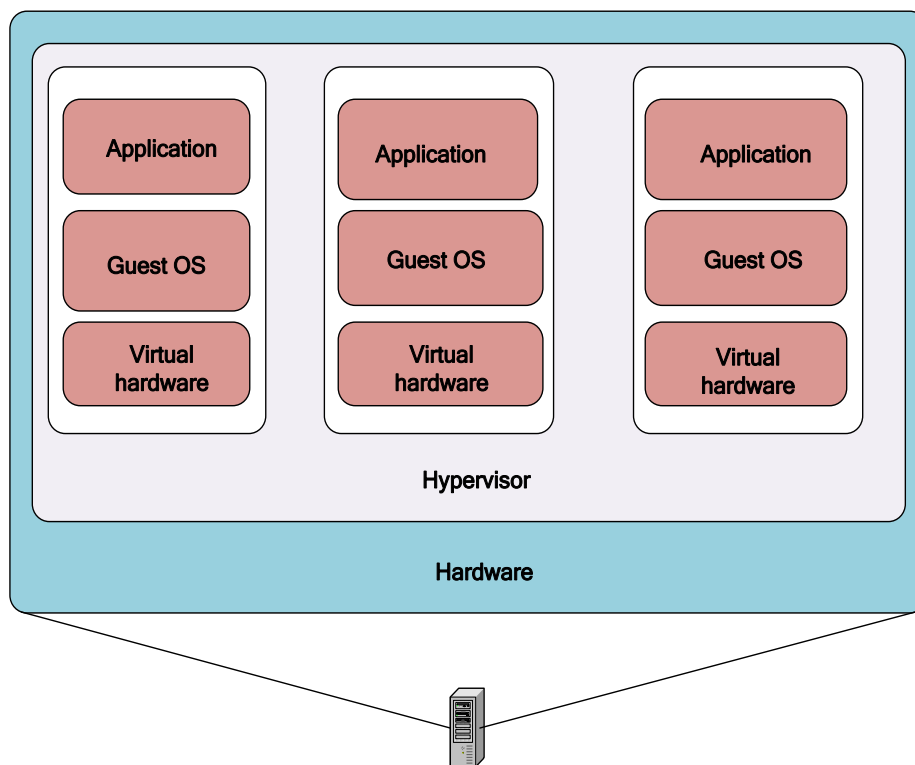


Figure 2-2. Virtualization Diagram

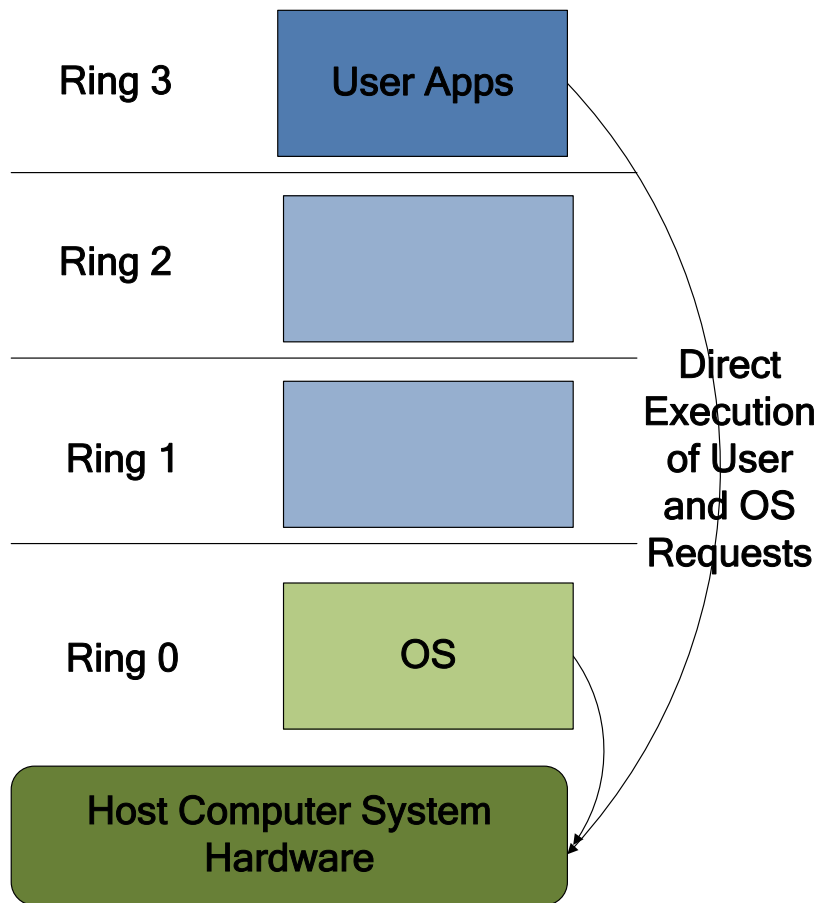


Figure 2-3. The General Operation System

The case of the general operation of the operation system is shown on **Figure 2-3**, the protection of instruction, there are four different levels of permissions. The user's applications are the implementation of the Ring 3 in the CPU part, and the implementation of the operating system and then operate in Ring 0 in the control of CPU and hardware, the hardware is in direct implementation by the operating system and user application are to the instructions.

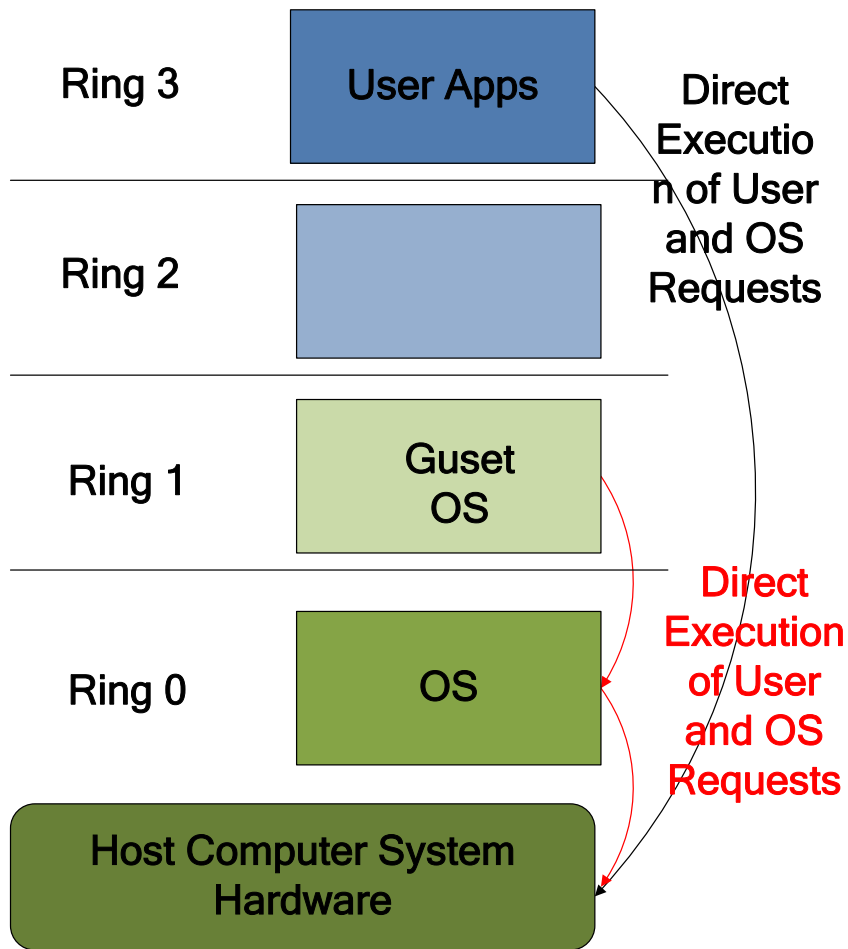


Figure 2-4. The Virtualization Operation System

Figure 2-4 shows the virtualization operation system. User's applications are still the implementation of the Ring 3, and the virtual operation system (Guest OS) is the implementation of the Ring 1. The original operation system becomes a Virtual Machine Manager (VMM). Guest OS is not to be executed directly by the CPU, but to use VMM making a translation to CPU and other hardware for the implementation..

2.2.1. Full-Virtualization

Unlike the traditional way that put the operation system kernel to Ring 0 level, full-virtualization use hypervisor instead of that. Hypervisor manage all instructions

to Ring 0 from Guest OS. Full-virtualization [9] uses the Binary Translation technology to translate all instructions to Ring 0 from Guest OS and then send the requirement to hardware. Hypervisor virtualized all hardware until, Guest OS access the hardware just like a real machine. It has highly independence. But Binary Translation technology reduces the performance of virtual machine.

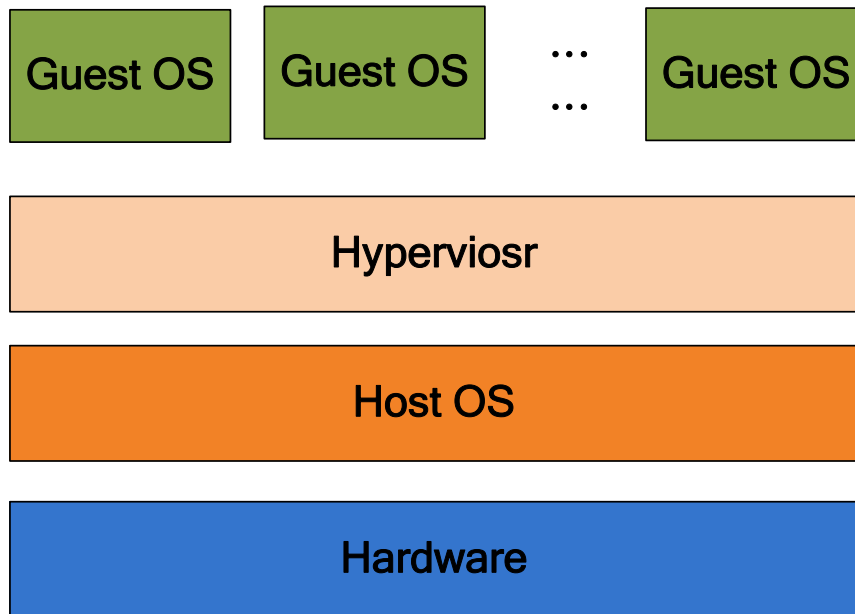


Figure 2-5. Full-Virtualization

2.2.2. Para-Virtualization

In para-virtualization [10], does not virtualize all hardware until. There is a unique Host OS called Domain0. Domain0 is parallel with other Guest OS and use Native Operating System to manage hardware driver. Guest OS accessing the real hardware by calling the driver in Domain0 through hypervisor. The requirement sent by Guest OS to hypervisor is called Hypercall. To make the Guest OS sending the hypercall instead of sending requirement directly to hardware, the Guest OS's kernel needs to rewrite, so that some non-open-sourced operation systems can not support.

Unlike full-virtualization using Binary Translation, para-virtualization let the Guest OS using hardware through Domain0. Although the performance of virtual machines enhance obviously, but the driver of hardware is binding on Domain0, and the kernel on Guest OS needs to rewrite, the independence is lower than full-virtualization.

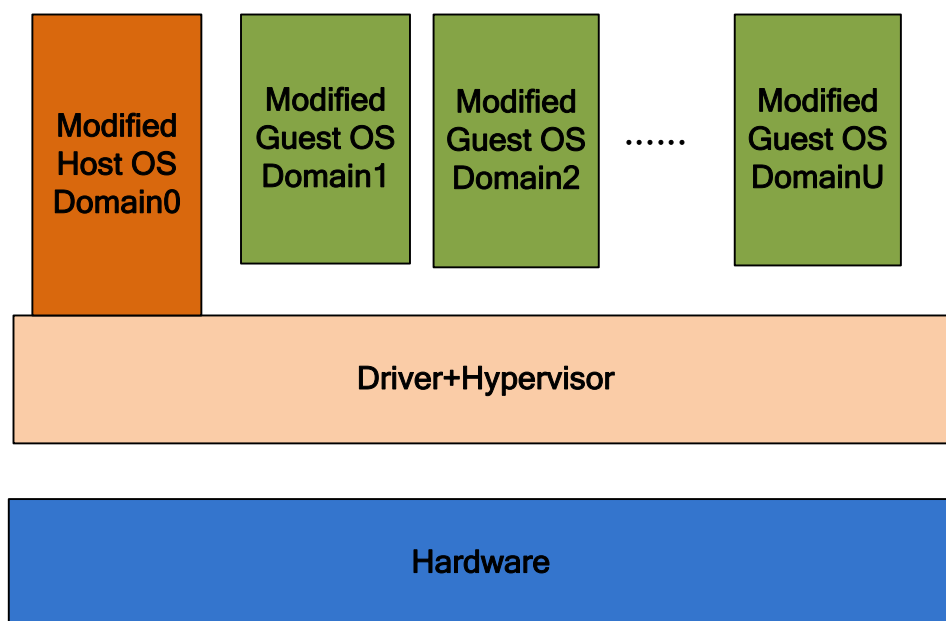


Figure 2-6. Para-Virtualization

2.2.3. Xen

There are two types of host virtualization software shown in **Figure 2-7**, Host OS type and Hypervisor type. The VM Layer of Host OS type deploys on Host OS, such like Windows or Linux, and then install other operation system on top of VM Layer. The operation systems on top the VM Layer are called Guest OS. Xen's hypervisor is installed directly in the host, and the other operation systems we want deploy are on top of it. It is easier to manage CPU, Memory, Network, Storage and other resource. The main purpose of Xen [11] uses hypervisor type and its VMM (Virtual Machine Monitor) is more efficient and safety to control the host CPU, Memory and other resource.

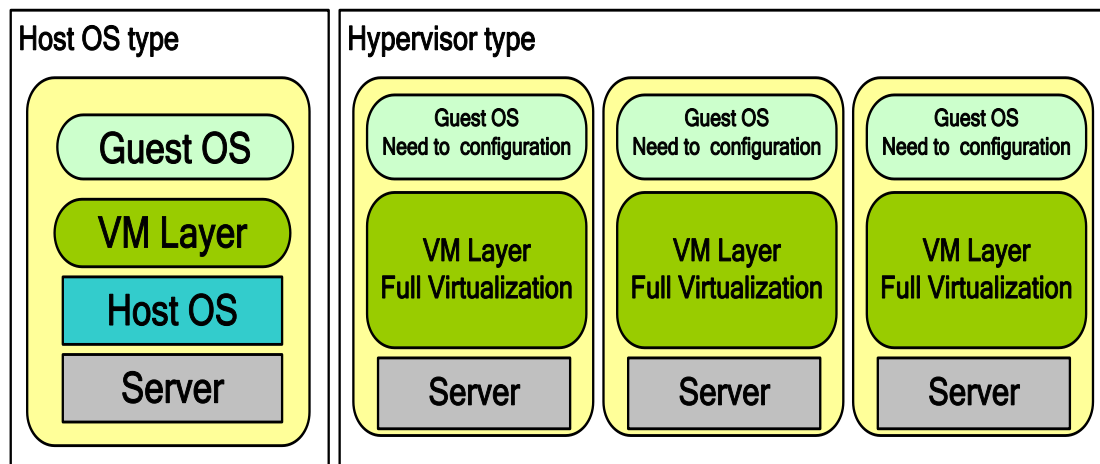


Figure 2-7. Host and Hypervisor Type

There are two types of hypervisor that Xen uses, Para-Virtualization and Full-Virtualization. The feature of these two types virtualization is at the section 2.2.1 and section 2.2.2.

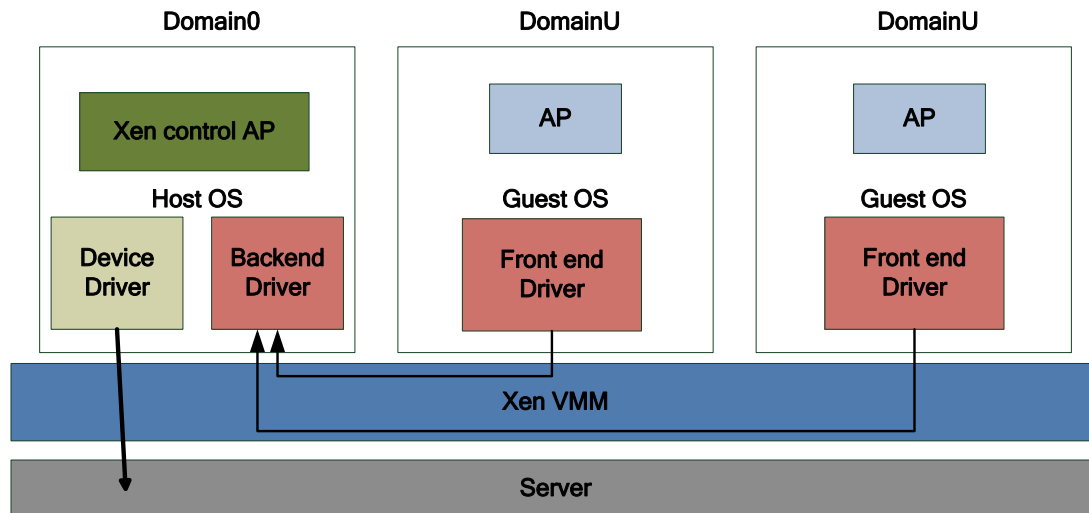


Figure 2-8. Domain0 and DomainU

Xen uses an unit called Domain to manage virtual machines. Domain is divided into two types as shown in **Figure 2-8**, one of them is called Domain0, played like Host OS, has control AP of Xen, used for management. Another type called DomainU is a field that Guest OS installed on it. When using physical resource, DomainU can not call the hardware driver directly, it must be through Domain0 to deal with.

In industry, Xen have been used in SUSE Linux Server (SLES) by Novell company and in Red Hat Enterprise Linux (RHEL) and in other commercial Linux version. In addition, Oracle also introduced a virtualization product called Oracle VM, and xVM Server released by Sun Microsystem, all base on Xen. That is shown that Xen have been supported by the system vendors widely in many virtualization software.

2.2.4. KVM

Kernel-based Virtual Machine (KVM) [12] is a part of architecture in Linux core. For now, KVM support native virtualization architecture, and hardware-assisted

virtualization is supported by CPU. This virtualization technology in Intel is called VT-x, and in AMD is called AMD-V. These two CPUs use different module to support KVM, kvm-intel.ko and kvm-amd.ko in Linux.

Currently KVM is running only on i386/x86_64 CPU in the system. Running on PowerPC and IA64 are still in development. Linux kernel have been include KVM since 2.6.20 and later. FreeBSD uses the way that kernel module to support KVM.

KVM's architecture consists of two parts:

- Kernel Device Driver – Used to manage and simulation virtual machine hardware.
- User Space Process – QEMU is a PC hardware emulator, become kqemu after modified by KVM.

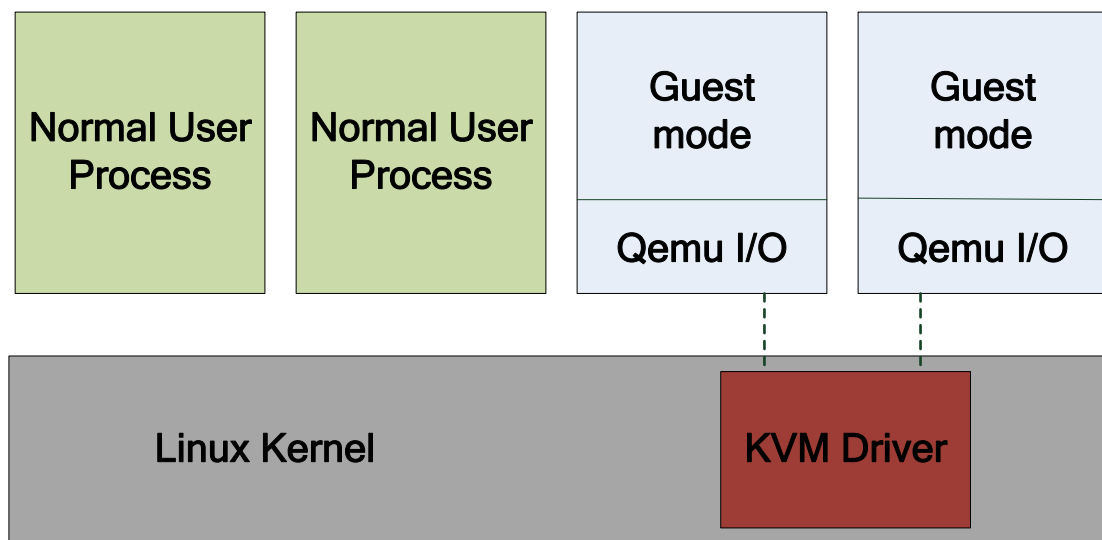


Figure 2-9. Architecture of KVM

2.3. CUDA

CUDA (Compute Unified Device Architecture) [6][13][14][15][16][17][18][19] is

architecture of parallel computing developed by NVIDIA. It is an official name that NVIDIA face to GPGPU. It is the first time that use C-complier as a develop environment for GPU. CUDA's programming model maintains a low learning curve for programmer familiar with standard programming languages such as C and FORTRAN shown in **Figure 2-10**. The architecture of CUDA is compatible with OpenCL [20][21][22] and C-complier from its own. The instructions are transformed into PTX code by driver no matter they come from CUDA C-language or OpenCL, and then calculate by graphics cores.

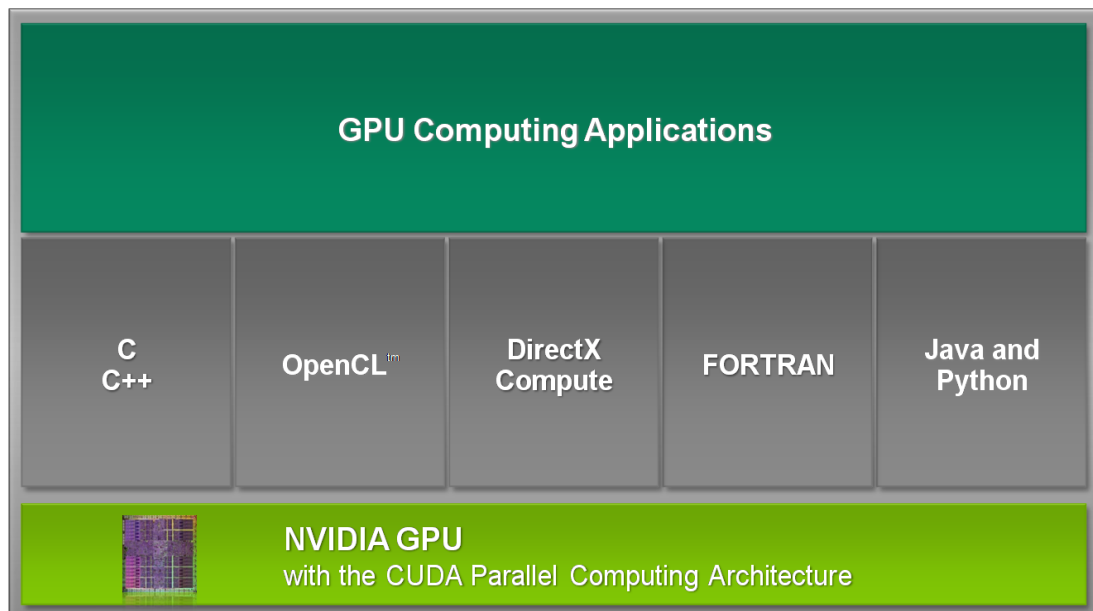


Figure 2-10. CUDA Programming Model from nVidia[2]

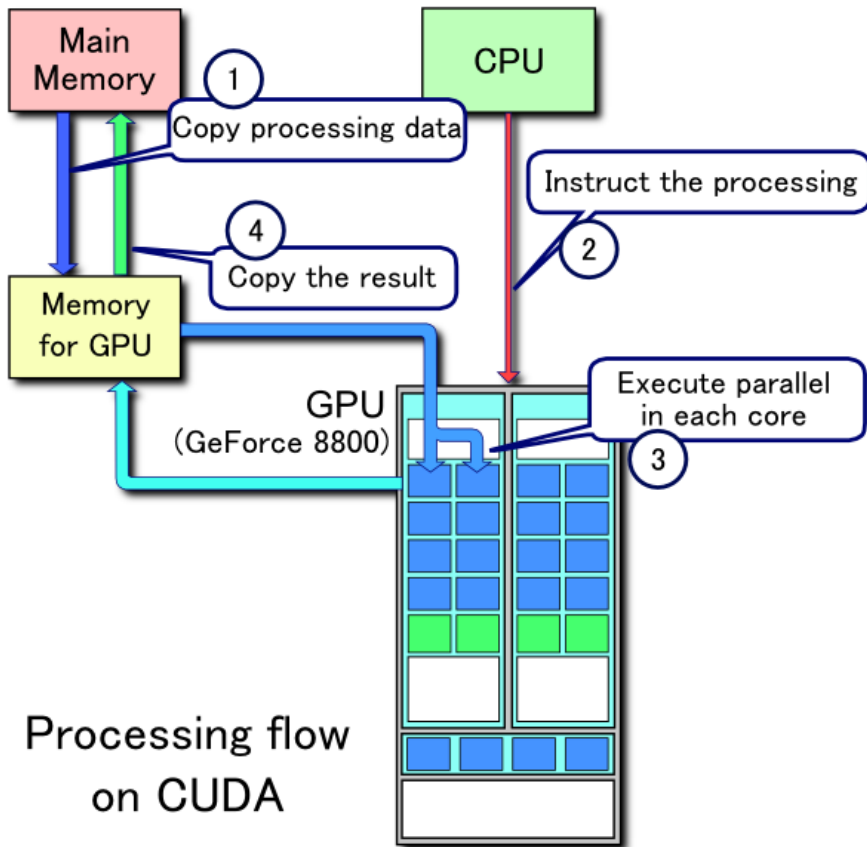


Figure 2-11. Processing Flow on CUDA from Wiki[16]

CUDA's processing flow is described in **Figure 2-11**. The first step: to copy the data which are on the main memory of CPU to the memory of GPU. The second: to instruct the process to GPU by CPU. The third: to parallel execute in each core on GPU. The last: to copy the result from the memory of GPU to the main memory of CPU.

2.4. Virtualization on GPU

Virtualization is more and more popular in recent days. The requirement of virtualization is also increase. Common virtual machines are inadequate for our use,

because the environment of the virtual machines is through virtualization after all. **Figure 2-12** and **Figure 2-13** show the two common virtual types to emulate devices and how to support I/O.

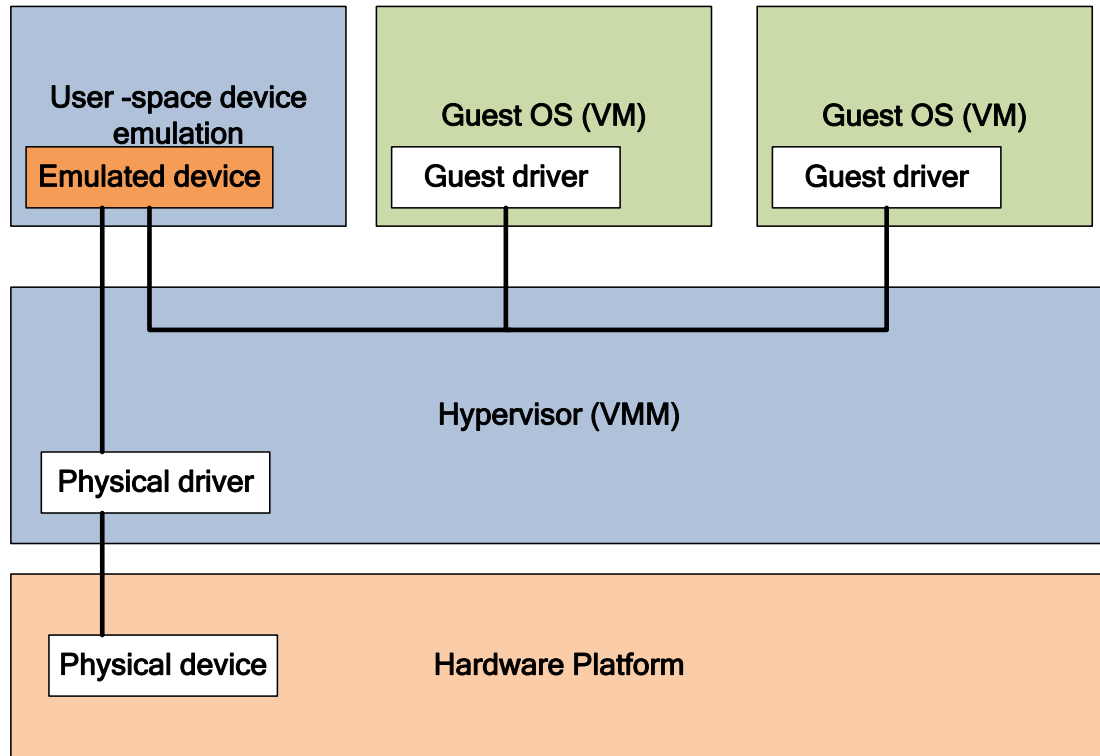


Figure 2-12. User Space Device Emulation

Figure 2-12 shows that the virtualization of user-space device emulation. Guest OS most through the emulated device created in Host OS to communicate with physical device. Rather than the device emulation being embedded within the hypervisor, it is instead implemented in user space. QEMU [23] which provides not only device emulation but a hypervisor as well, provides for device emulation and is used by a large number of independent hypervisors such as Kernel –based Virtual Machine (KVM) and VirtulaBox [24].

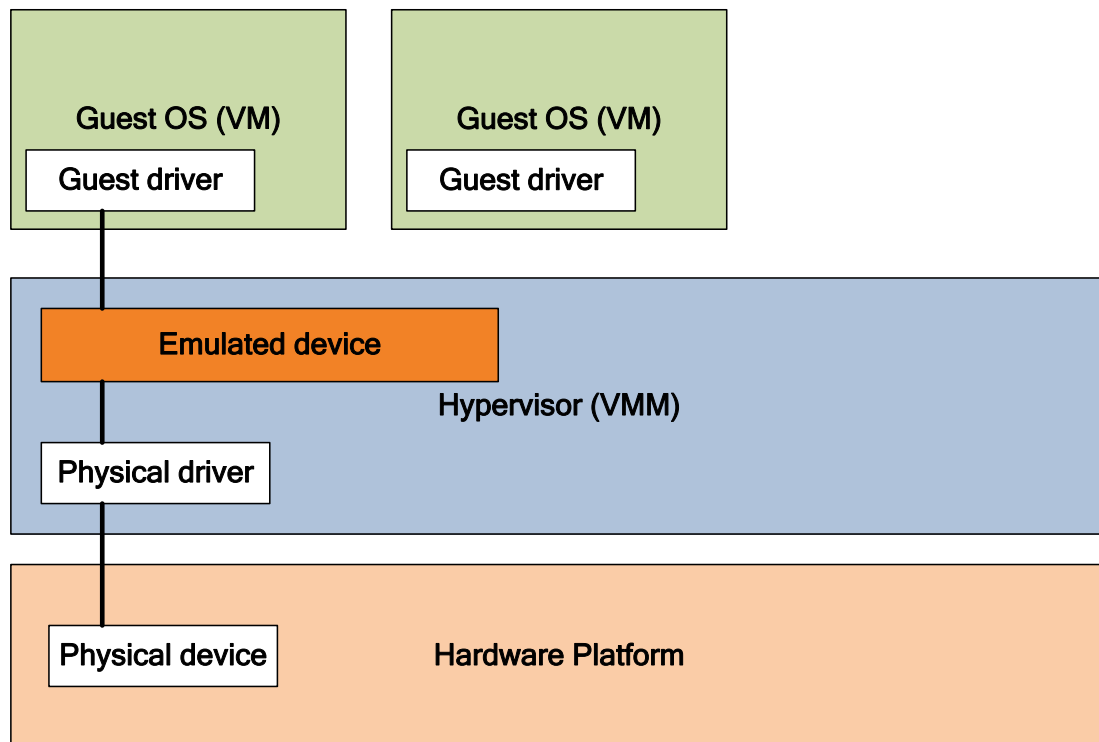


Figure 2-13. Hypervisor-Based Device Emulation

Figure 2-13 shows the other way to emulate devices. All devices or I/O in the virtual machine are emulated by hypervisor. This is a common method implemented within an operation system-based hypervisor. In this model, the hypervisor includes emulations of common devices that the various guest operating systems can share, including virtual disks, virtual network adapters, and other necessary platform elements.

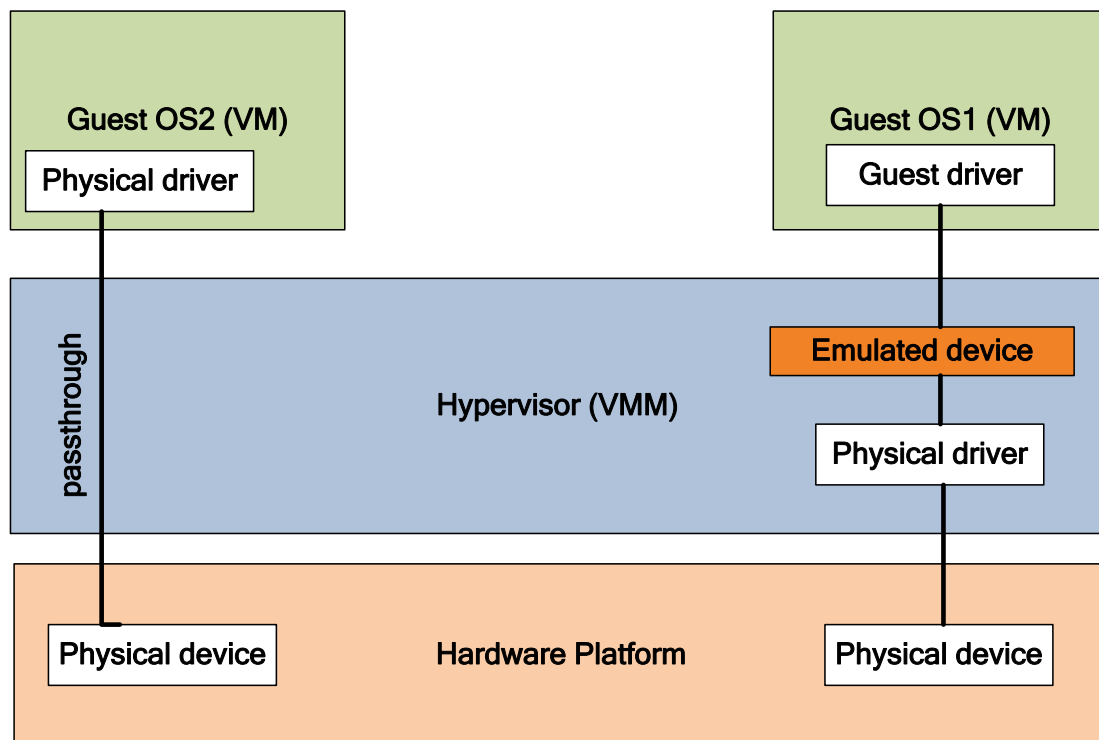


Figure 2-14. Pass-through within the Hypervisor

Unlike the two kinds of emulation of devices before, device pass-through is about providing an isolation of devices to a given guest operating system shown in **Figure 2-14**. Assigning devices to specific guests is useful when those devices can not be shared. For performance, near-native performance can be achieved using device pass-through.

2.5. Green Computing

Green computing [25][26] is to effectively use the resources such as implementation of energy-efficient CPU, servers and peripherals as well as reduce resource consumption. Green computing use virtualization technology and power management to reach energy saving and carbon emission reduction. Virtualization is one of the

most effective tools for more cost-effective, greener-energy efficient computing where each server is divided into multiple virtual machines that run different applications.

2.6. Related Work

In recent years, virtualization environment on Cloud become more and more popular. The balance between performance and cost is the most important point that everybody focused. For more effective to use the resource on the server, virtualization technology is the solution. Running many virtual machines on a server, the resource can be more effective to use. But the performance of virtual machines has their own limit. Users will limited by using a lot of computing on virtual machine. Therefore, there is a new topic that let the virtual machines using the physical GPGPU (General-Purpose computing on Graphics Processing Units) in the real machine to help computing.

There are some approaches that pursue the virtualization of the CUDA Runtime API for VMs such as rCUDA[27][28][29], vCUDA[30], GViM[31] and gVirtuS[32]. The solutions feature a distributed middleware comprised of two parts, the front-end and the back-end[33].

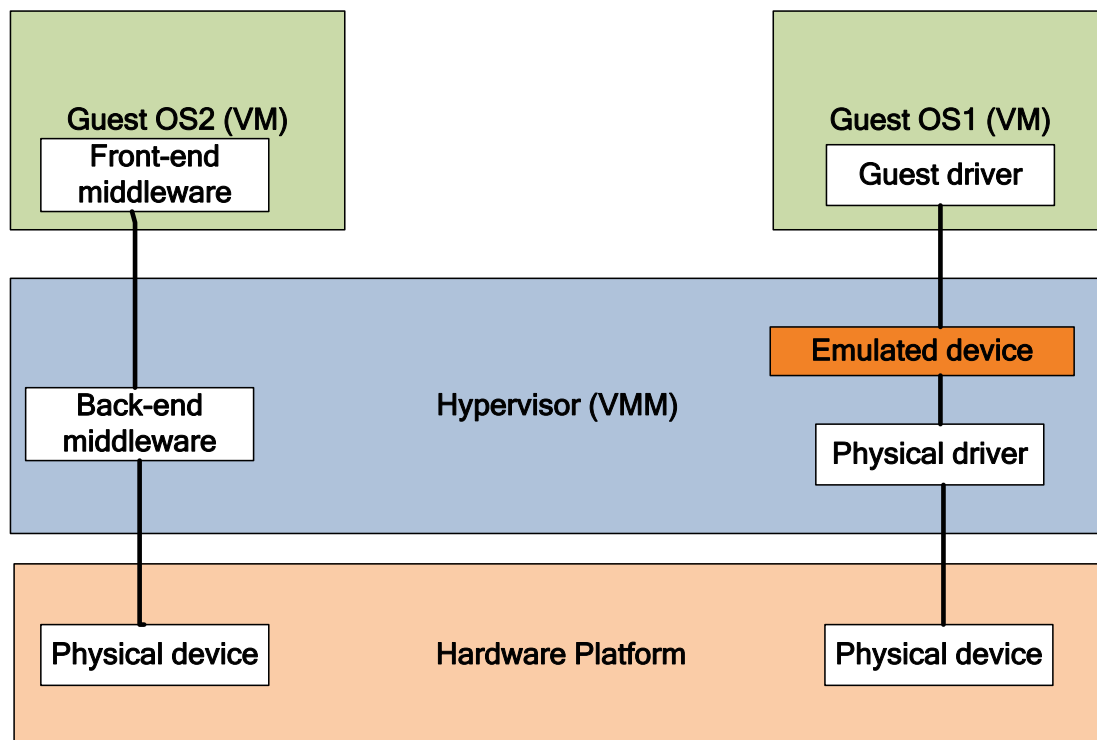


Figure 2-15. Front-End and Back-End

Figure 2-15 shows that the front-end middleware is installed in the virtual machine, and the back-end middleware with direct access to the acceleration hardware, is running by host OS with executing the VMM.

rCUDA using Sockets API to let the client and server have communication with each other. And client can use the GPU on server through that. It is a production-ready framework to run CUDA applications from VMs, based in a recent CUDA API version. We can use this middleware to make a customized communications protocol and is independent [27]. The architecture is shown in **Figure 2-16**.

Unlike rCUDA, GViM and vCUDA are not at expense of losing VMM independence.

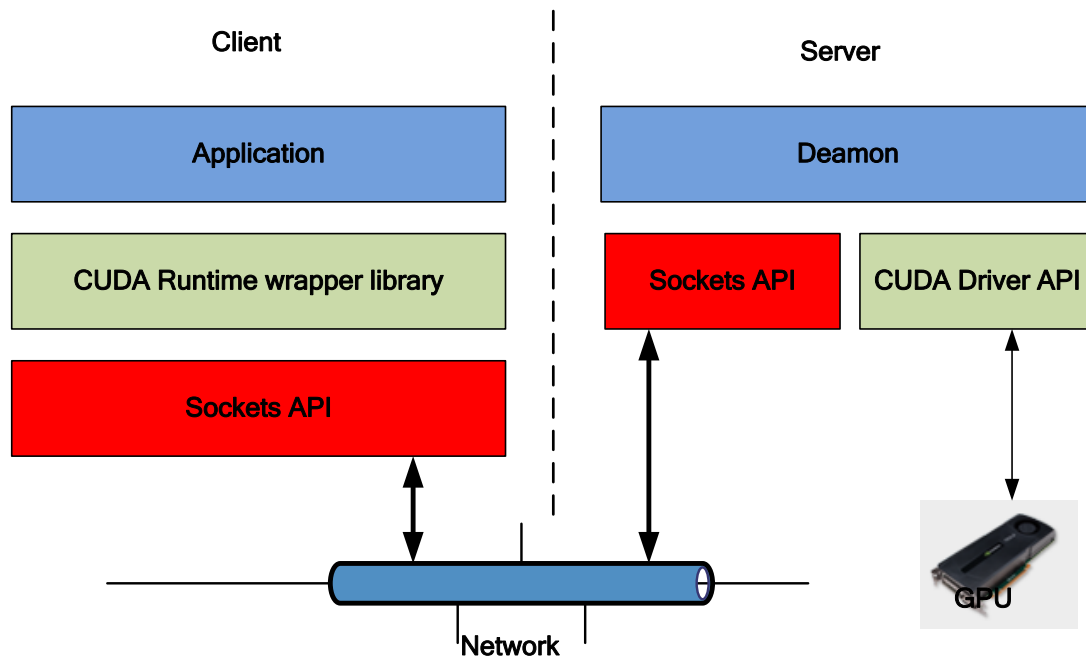


Figure 2-16. Architecture of rCUDA

The key idea in vCUDA is: API call interception and redirection. With API interception and redirection, applications in VMs can access graphics hardware device and achieve high performance computing applications. It allows the application executing within virtual machines to leverage hardware acceleration. They explained how to access graphics hardware in VMs transparently by API call interception and redirection. Their evaluation showed that GPU acceleration for HPC applications in VMs is feasible and competitive with those running in a native, non-virtualized environment [30]. The architecture is shown in **Figure 2-17**.

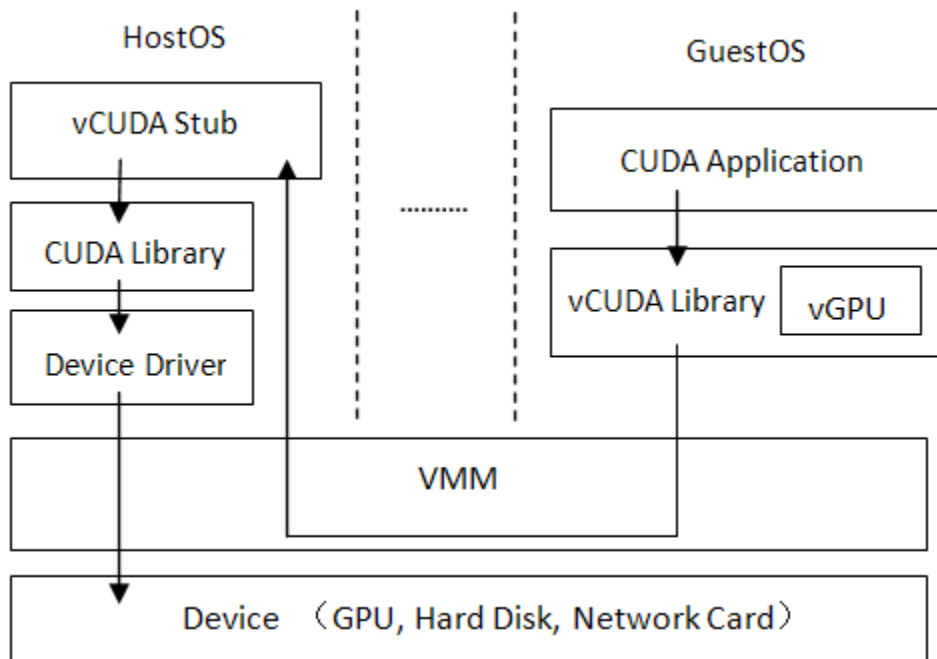


Figure 2-17. The vCUDA Architecture

GViM is a system designed for virtualization and managing the resources of a general purpose system accelerated by graphics processors. GViM uses Xen-specific mechanisms for the communication between front-end and back-end middleware. The GViM virtualization infrastructure for a GPGPU platform enables the sharing and consolidation of graphics processors. Their experimental measurements of a Xen-based GViM implementation on a multicore platform with multiple attached NVIDIA graphics accelerators demonstrate small performance penalties for virtualized vs. non-virtualized settings, coupled with substantial improvements concerning fairness in accelerator use by multiple VMs [31].

VMGL [34] is the OpenGL hardware 3D acceleration for virtual machines, OpenGL apps can run inside a virtual machine through VMGL. VMGL can be used on VMware guests, Xen HVM domains (depending on hardware virtualization extensions) and Xen paravirtual domains, using XVnc or the virtual frame buffer.

VMGL is available for X11-based guest OS's: Linux, FreeBSD and OpenSolaris. Finally, VMGL is GPU-independent: we support ATI, NVidia and Intel GPUs.

In J. Duato's work, he uses remote GPU for virtual machine. Although his virtualization technique noticeably increases execution time when using a 1 Gbps Ethernet network, it performs almost as efficiently as a local GPU when higher performance interconnects are used. Therefore, the small overhead incurred by our proposal because of the remote use of GPUs is worth the savings that a cluster configuration with less GPUs than nodes reports [29].

Atsushi Kawai and Kenji Yasuoka proposed DS-CUDA, a middleware to virtualize a GPU cluster as a distributed shared GPU system. It simplifies development of a code that uses multiple GPUs distributed on a network. Results with good scalability were shown in their paper. Also the usefulness of the redundant calculation mechanism is confirmed.

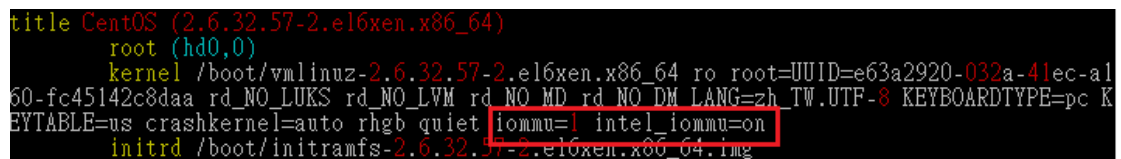
Chapter 3

System Implementation

3.1. System Architecture

To use GPU accelerator on virtual machines, this thesis plans using PCI-pass-through to implement the system for better performance. For performance, near-native performance can be achieved using device pass-through. This technology is perfect for networking applications or those that have high disk I/O or like using hardware accelerator that have not adopted virtualization because of contention and performance degradation through the hypervisor. But assigning devices to specific guests is also useful when those devices can not be shared. For example, if a system included multiple video adapters, those adapters could be passed through to unique guest domains.

VT-d Pass-Through is a technique to give a DomU exclusive access to a PCI function using the IOMMU [35] provided by VT-d. It is primarily targeted at HVM (fully virtualized) guests because Para-Virtualized pass-through does not require VT-d. There is an important thing that your hardware must support that. In addition to the motherboard chipset and BIOS also your CPU must have support for IOMMU IO virtualization (VT-d). VT-d is disabled by default, to enable it, need 'iommu' parameter to enable it.



```
title CentOS (2.6.32.57-2.el6xen.x86_64)
  root (hd0,0)
  kernel /boot/vmlinuz-2.6.32.57-2.el6xen.x86_64 ro root=UUID=e63a2920-032a-41ec-a1
60-fc45142c8daa rd_NO_LUKS rd_NO_LVM rd_NO_MD rd_NO_DM LANG=zh_TW.UTF-8 KEYBOARDTYPE=pc K
EYTABLE=us crashkernel=auto rhgb quiet iommu=1 intel_iommu=on
  initrd /boot/initramfs-2.6.32.57-2.el6xen.x86_64.img
```

Figure 3-1. IOMMU On

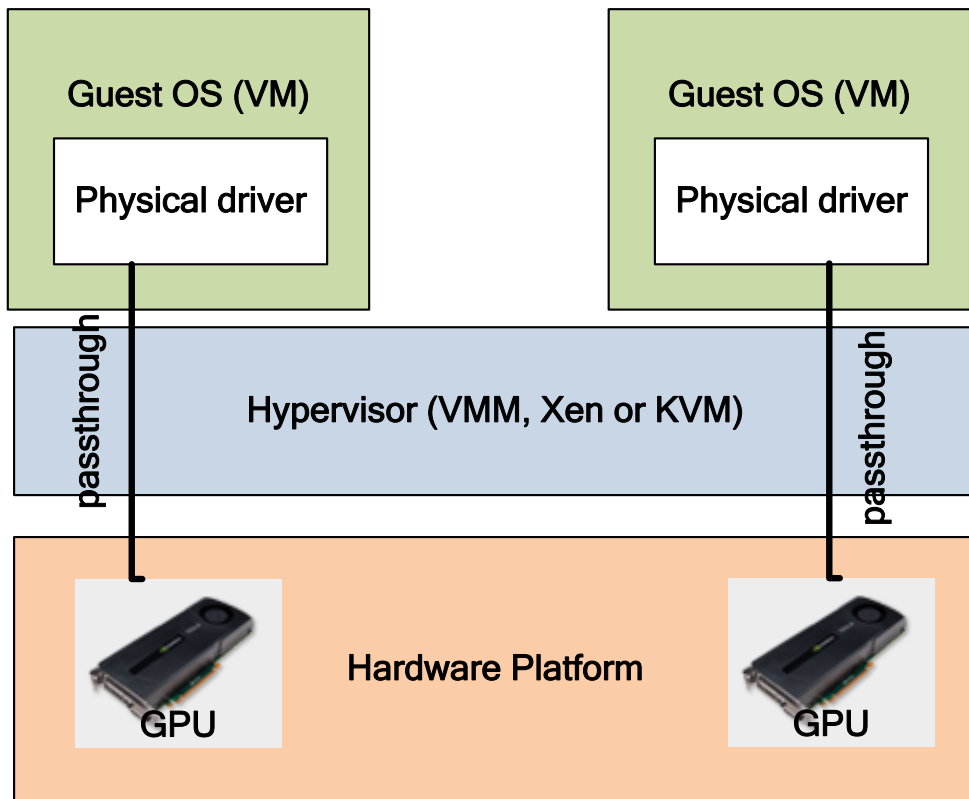


Figure 3-2. System Architecture

This thesis using Xen or KVM as a hypervisor. And implement PCI passthrough passing through the GPUs to those virtual machines on the hypervisor in the whole system. **Figure 3-3** shows the user's architecture. Users can through the internet to use the GPU accelerator when the GPU virtualization environment is setting up.

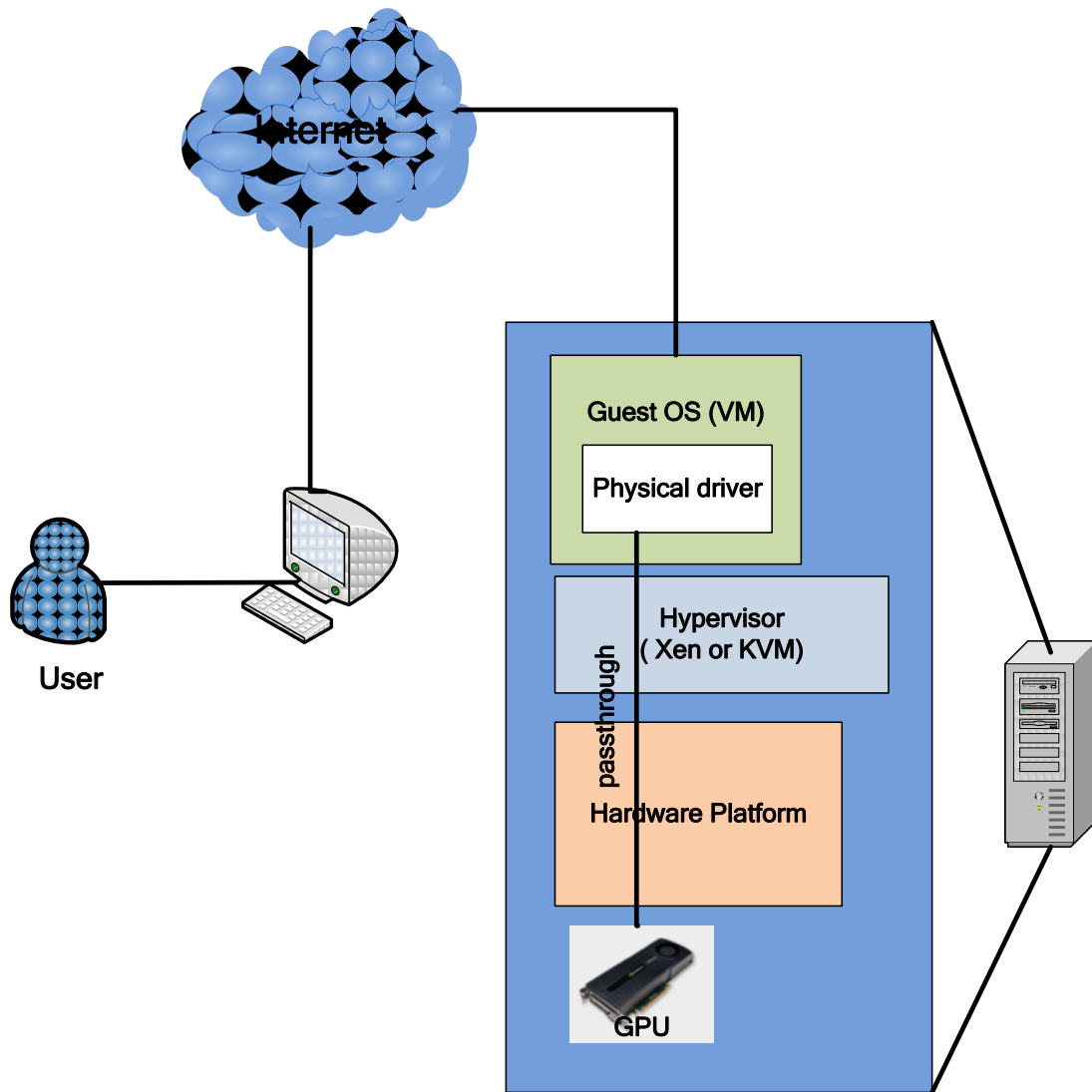


Figure 3-3. User Architecture

3.2. Tesla C1060 Computing Processor Board

The NVIDIA Tesla™ C1060 [36] transforms a workstation into a high-performance computer that outperforms a small cluster. This gives technical professionals a dedicated computing resource at their desk-side that is much faster and more energy-efficient than a shared cluster in the data center. The details of NVIDIA Tesla™ C1060 computing processor board's specification is shown below.

- One Tesla T10

- 240 CUDA cores
- 1.296 GHz core frequency
- 933 Gflops Single Precision
- 78 Gflops Double Precision
- 4 GB GDDR3 memory at 102 GB/s bandwidth
- 800 MHz memory frequency

A computer system with an available PCI Express $\times 16$ slot is required for the Tesla C1060. For the best system bandwidth between the host processor and the Tesla C1060, it is recommended (but not required) that the Tesla C1060 be installed in a PCI Express $\times 16$ Gen2 slot. The Tesla C1060 is based on the massively parallel, many-core Tesla processor, which is coupled with the standard CUDA C programming [15] environment to simplify many-core programming. The architecture of Tesla T10 is shown in **Figure 3-4**.

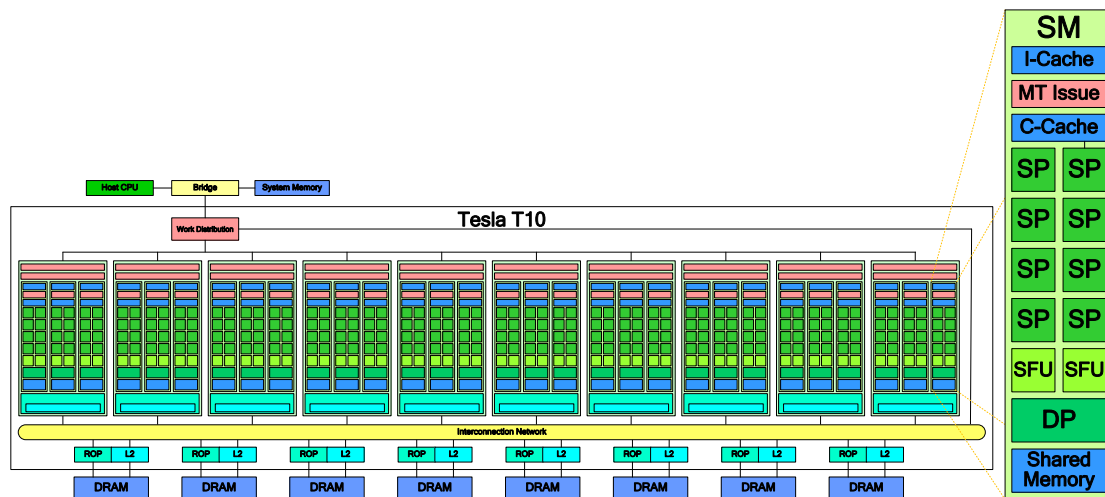


Figure 3-4. Tesla T10

3.3. Tesla C2050 Computing Processor Board

The NVIDIA Tesla™ C2050[38] is based on the next-generation CUDA™ architecture codenamed “Fermi”, the 20-series family of Tesla GPUs support many “must have” features for technical and enterprise computing including c++ support, Ecc memory for uncompromised accuracy and scalability, and a 7X increase in double precision performance compared Tesla 10-series GPUs.

Compared to the latest quad-core CPUs, Tesla C2050 computing Processors deliver equivalent supercomputing performance at 1/10th the cost and 1/20th the power consumption. The specification is shown below.

- One Tesla core
- 448 CUDA cores
- 1.15 GHz core frequency
- 1.03 Tflops Single Precision
- 515 Gflops Double Precision
- 3GB GDDR5 memory at 144 GB/s bandwidth
- 1.5 GHz memory frequency

3.4. End User’s Operating Interface

When users create a virtual machine and pass the GPU through to virtual machine successful, users can through an application called “virtual machine manager” in Linux to see the result. In **Figure 3-5**, the GPU pass-through is successful and virtual is running.

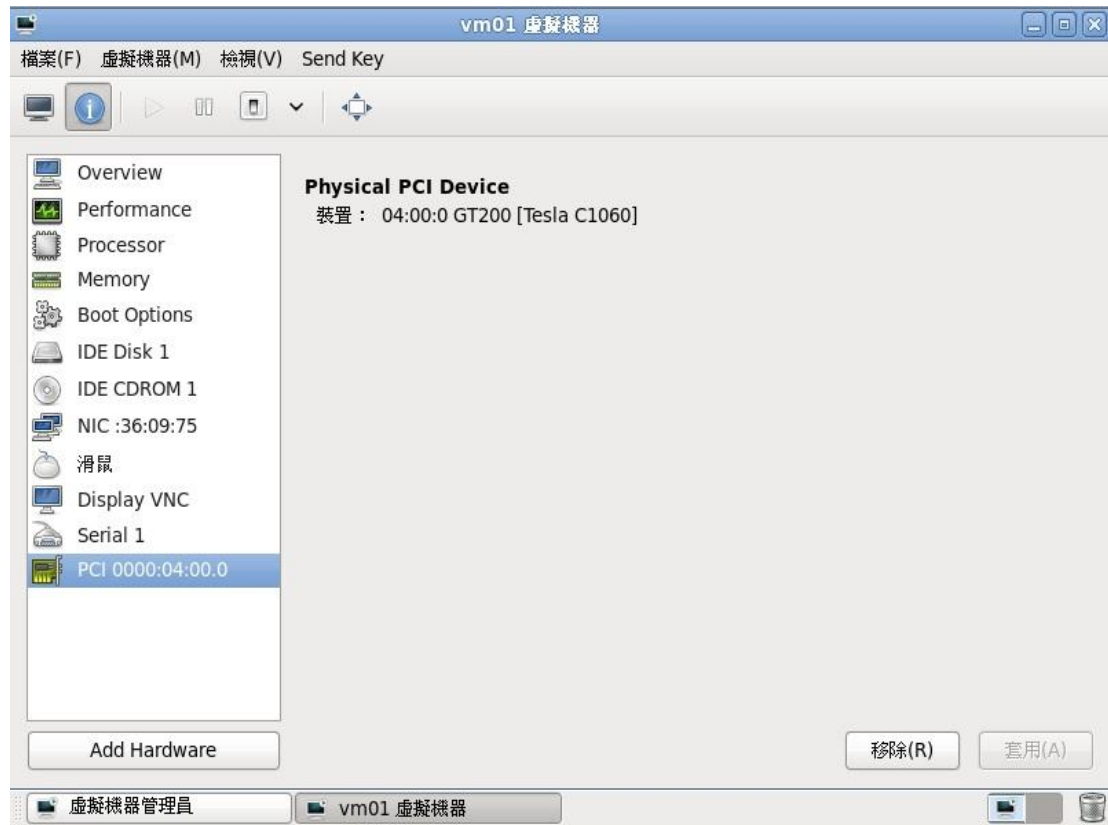


Figure 3-5. PCI Pass-through Successful

On the other way, users can also use pietty or VNC. Users must be prepared to internet connection and VNC connection, and then set the IP and port as long as you can connect to the virtual machine. In the console, users can use the command “lspci” to see the PCI pass-through is working or not. The setup is shown in **Figure 3-6**, **Figure 3-7** and **Figure 3-8**.

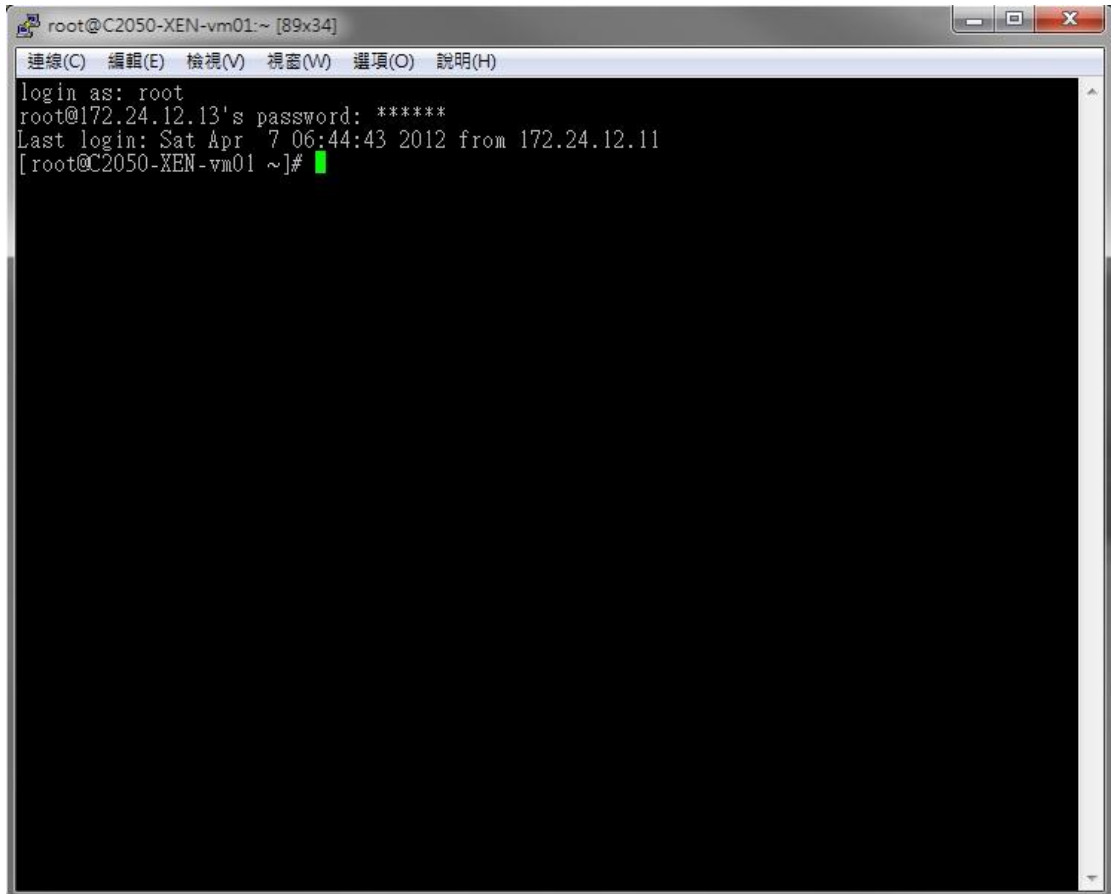


Figure 3-6. Shows in Pietty

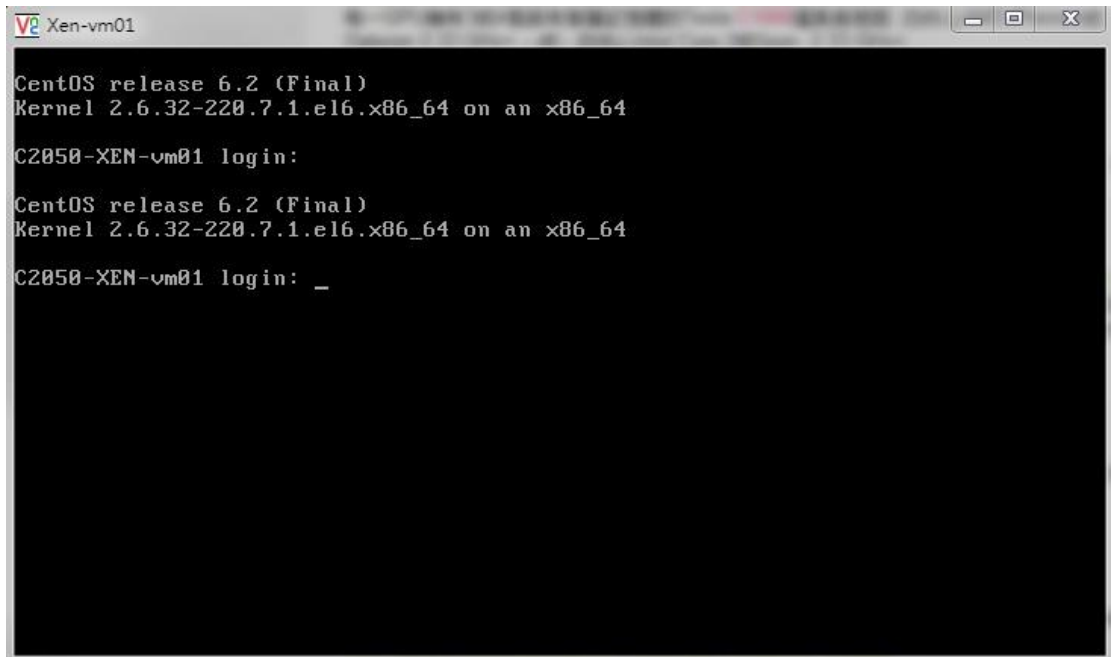


Figure 3-7. Shows in VNC

```

root@C2050-XEN-vm01:~ [89x34]
連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)
login as: root
root@172.24.12.13's password: *****
Last login: Tue Apr  3 05:36:09 2012 from 172.24.12.11
[root@C2050-XEN-vm01 ~]# lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.2 USB controller: Intel Corporation 82371SB PIIX3 USB [Natoma/Triton II] (rev 01)
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 01)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Unassigned class [ff80]: XenSource, Inc. Xen Platform Device (rev 01)
00:05.0 3D controller: nVidia Corporation GT200 [Tesla C1060] (rev a1)
[root@C2050-XEN-vm01 ~]#

```

Figure 3-8. Using lspci

3.5. System Environment

Previously, we have conducted the design principle and implementation methods. We present here several experiment conducts on two machines. The node's hardware specification is listing in **Table 3-1**.

Table 3-1. Hardware/Software Specification

Hardware/Software Specification						
	CPU	Memory	Disk	OS	Hypervisor	GPU
Node1	Xeon E5506	12GB	1TB	CentOS 6.2	Xen	Quadro NVS 295/ Tesla C1060/ Tesla C2050

In **Table 3-1**, we use two machines with the same hardware specification. And the hypervisor which one is Xen, another is KVM. The purpose is compare the performance between these two hypervisor using PCI pass-through with the same GPU. Quadro NVS 295 [37] is using for primary graphics card. Tesla C1060 is the one we using for computing and passing through to virtual machine.

Table 3-2. Hardware/Software Specification of Virtual Machine

Hardware/Software Specification of Virtual Machine							
	CPU	Memory	Disk	OS	Hypervisor	GPU	Virtualization
VM1	1,2,4	1GB	12GB	CentOS6.2	Xen	Quadro NVS 295	Full
VM2	1,2,4	1GB	12GB	CentOS6.2	Xen	Tesla C1060	Full
VM3	1,2,4	1GB	12GB	CentOS6.2	Xen	Tesla C2050	Full

Table 3-2 is the hardware/software specification of virtual machines. We create three virtual machines with the same specification but in different CPU number and different GPU. We want discuss that the CPU number will or not affect the performance of virtual machine in PCI pass-through. So we will use 1, 2 or 4 CPUs in our virtual machine to see the difference between each other. These of two virtual machines' virtualization type are full, because we found out that PCI pass-through is not working in para-virtualization in our research. **Table 3-3** shows the GPU software

environment.

Table 3-3. GPU Software Environments

GPU Software Environments	
Driver	285.05.33
Cuda toolkit	4.1.28
CUDA SDK	4.1.28

Chapter 4

Experimental Methods and Results

4.1. Experimental Methods

We set up ten comparison benchmarks: alignedTypes, asyncAPI, BlackScholes, clock, convolutionSeparable, fastWalshTransform, matrixMul, Bandwidthtest, matrixmul-sizeable and VecAdd. The first seven benchmarks are ports of CUDA SDK [13]. From benchmarks in the suite, we select 7 representative SDK benchmarks of varying computation loads and data size which use different CUDA features. These benchmarks are executed with the default. Another two benchmarks are selected as matrixmul-sizeable and VecAdd which the problem size can be set with high computation loading. All SDK benchmarks' execution time is measured by the command 'time' in the CentOS [39]. Table 4-1 shows the data transferring size of each benchmark.

Table 4-1. Data transfers of Benchmarks

SDK name	Data transfers
Aligned Types	413.26MB
Async API	128.00MB
Black Scholes	76.29MB
Clock	2.50KB
Convolution Separable	36.00MB
Fast Walsh Transform	64.00MB

Matrix Mul	79.00KB
------------	---------

The first experiment is the comparison between native and virtual machine performance. We present how much GPU performance reduced with PCI pass-through. The second experiment is the comparison between virtual machines with 1 CPU, 2 CPUs and 4 CPUs performance whether presenting CPU numbers in virtual machines would affect the GPU performance or not. The final experiment is the comparison between the two common virtualization hypervisors performance. We will display which one has better GPU performance with PCI pass-through than the others.

4.2. Experimental Results

We first analyze the performance of the CUDA SDK benchmarks running in a VM using PCI pass-through, and compare their execution times with those in a native environment —i.e., using the regular CUDA Runtime library in a non-virtualized environment. The results of these experiments are reported as below.

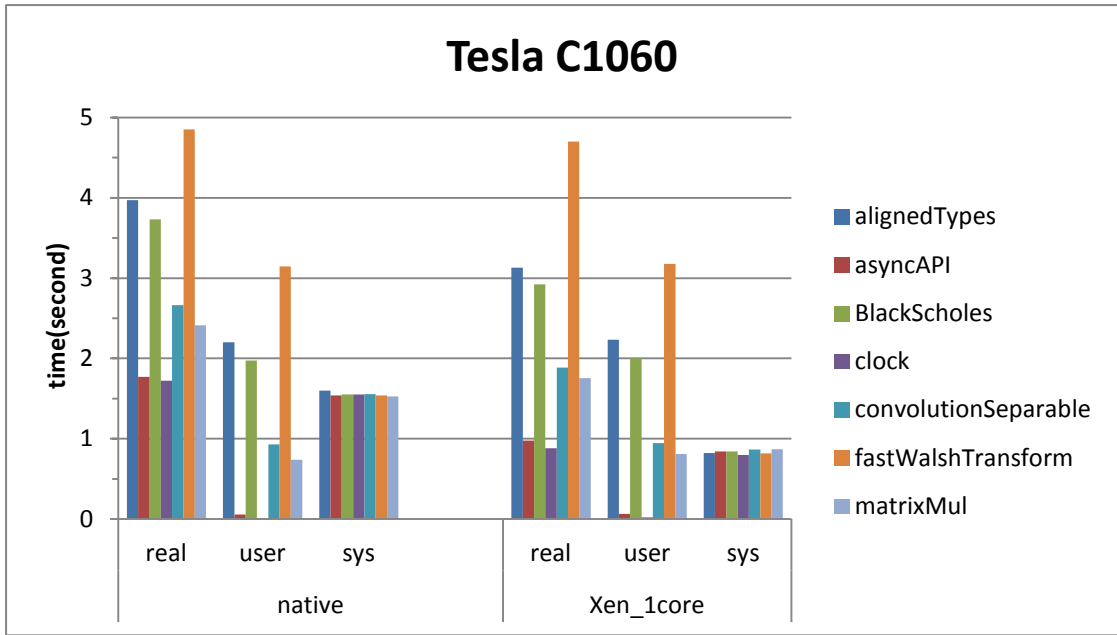


Figure 4-1. Execution Time between Native and VM with C1060

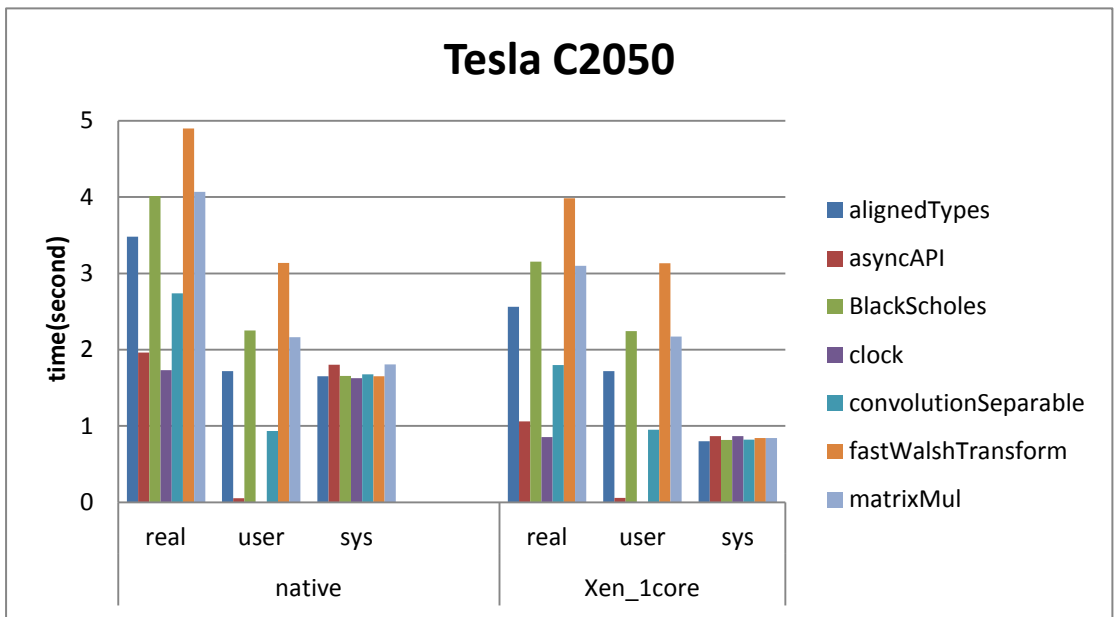


Figure 4-2. Execution Time between Native and VM with C2050

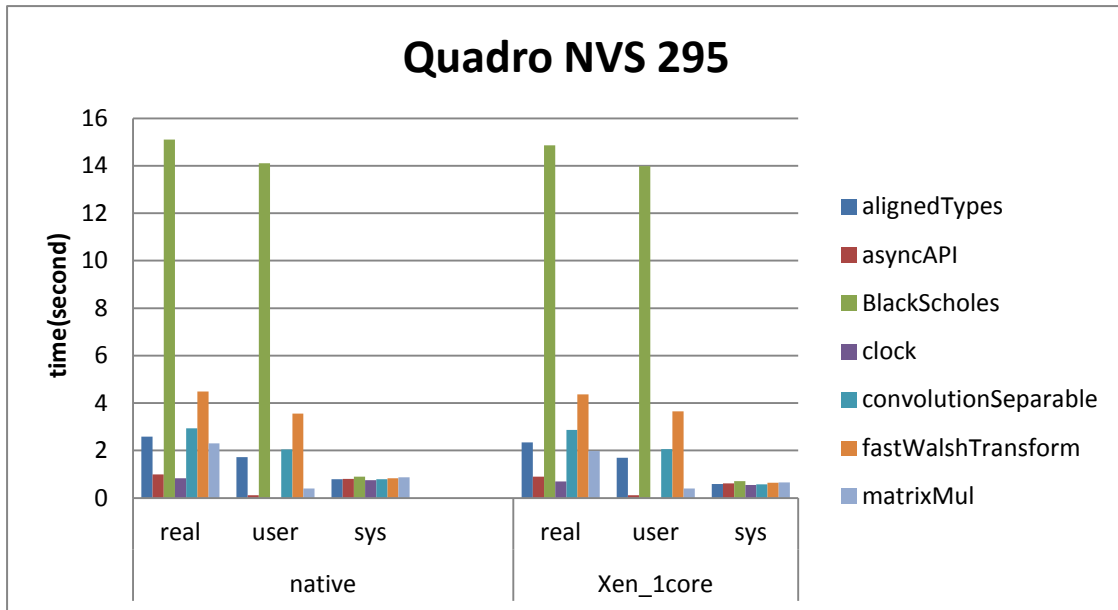


Figure 4-3. Execution Time between Native and VM with NVS295

Figure 4-1, Figure 4-2 and **Figure 4-3** show that the execution time on processing the SDK benchmark in native and virtual machine using one CPU on Xen. We can see the real time of these benchmarks on virtual machine is less than native machine.

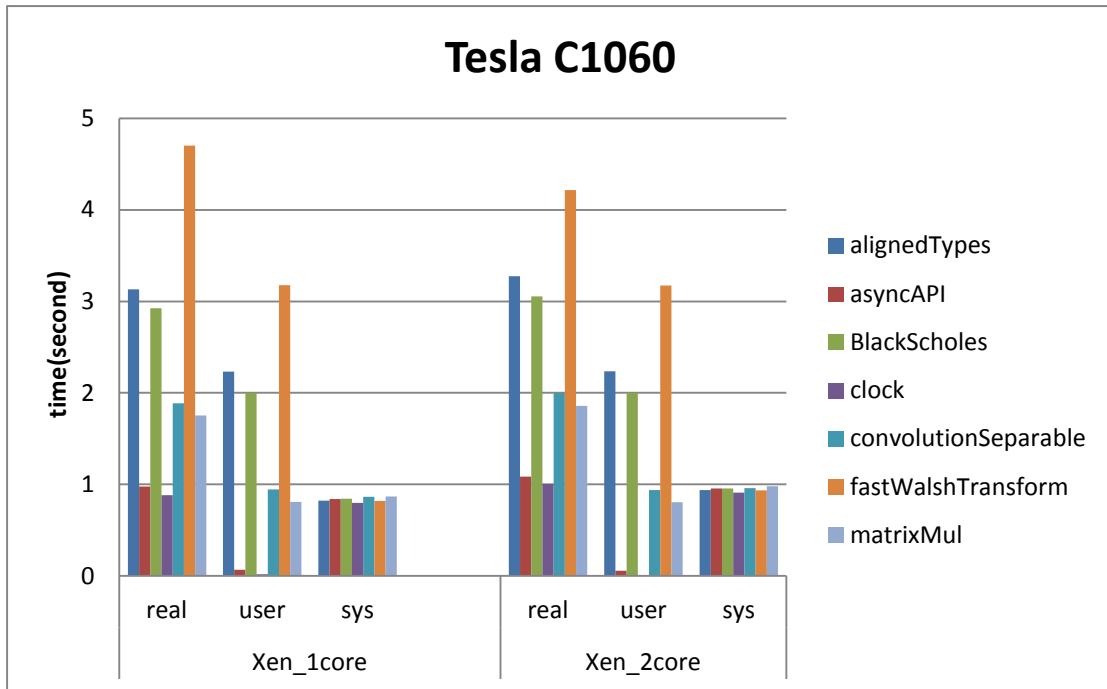


Figure 4-4. Execution Time between 1 Core and 2 Core VM with C1060

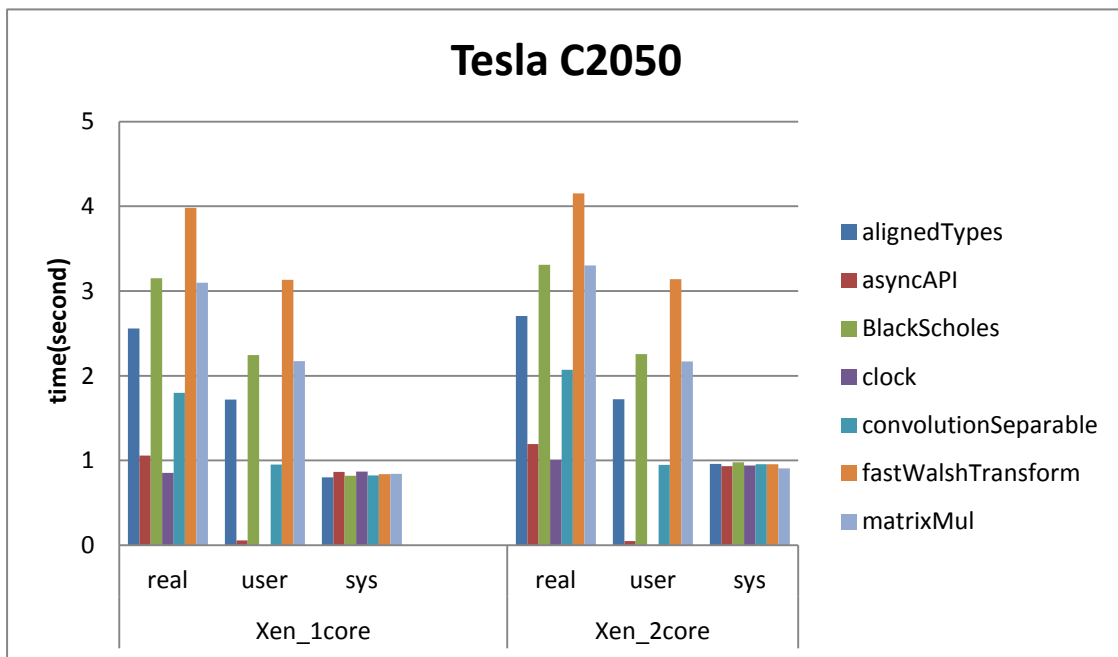


Figure 4-5. Execution Time between 1 Core and 2 Core VM with C2050

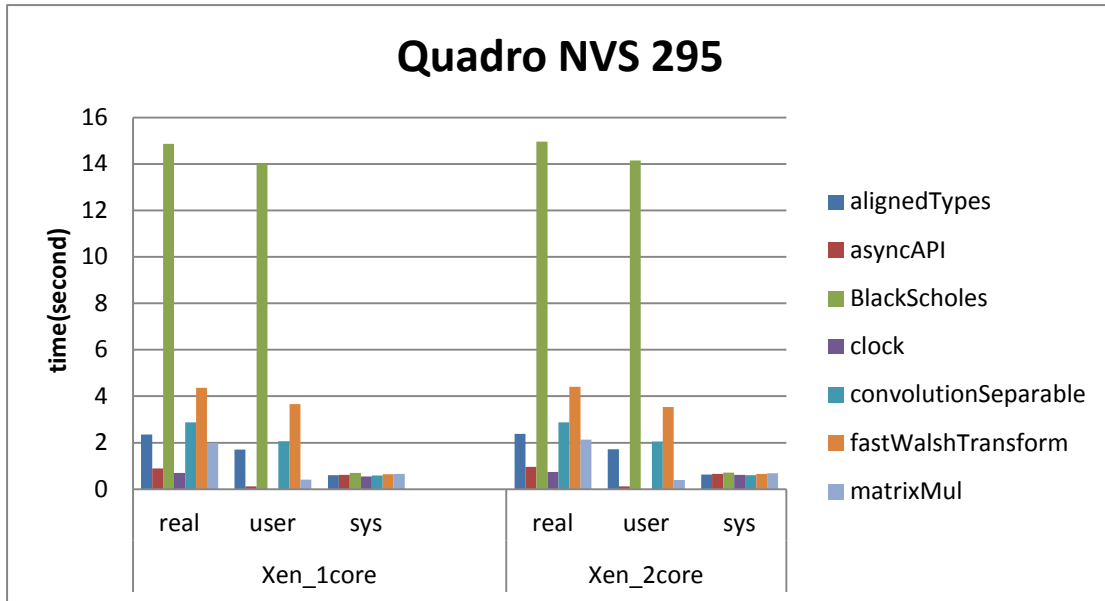


Figure 4-6. Execution Time between 1 Core and 2 Core VM with NVS295

Figure 4-4, **Figure 4-5** and **Figure 4-6** show that the different execution time between virtual machines which has one CPU, and the other has two CPUs. In this figure, we can see that the number of CPUs does not affect the user time, which means the GPU computing time is not changed when CPU increases from one to two.

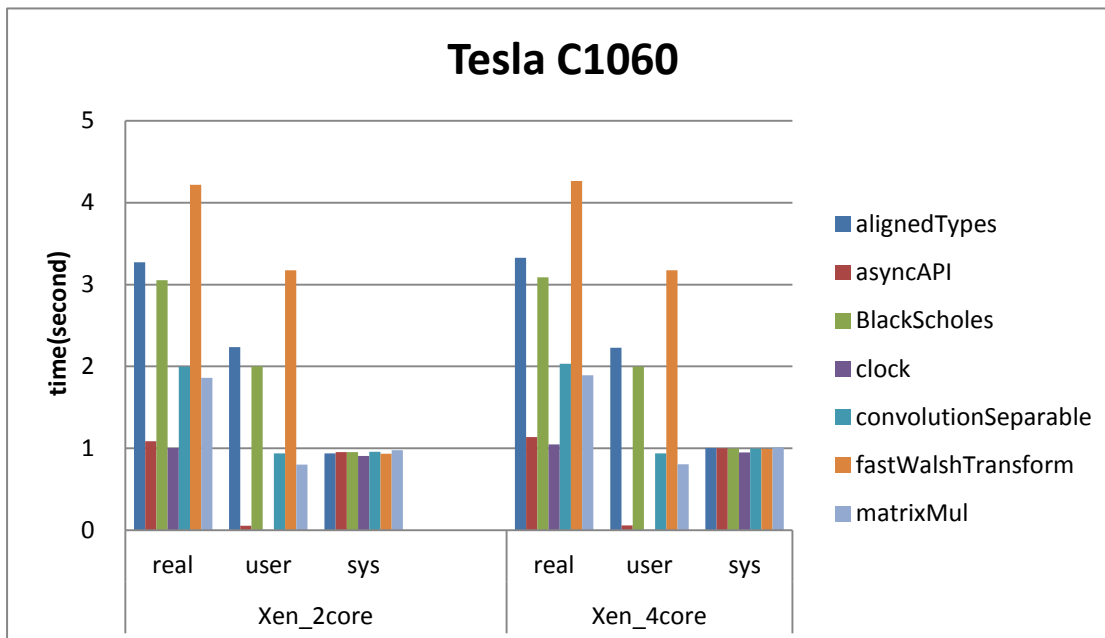


Figure 4-7. Execution Time between 2 Core and 4 Core VM with C1060

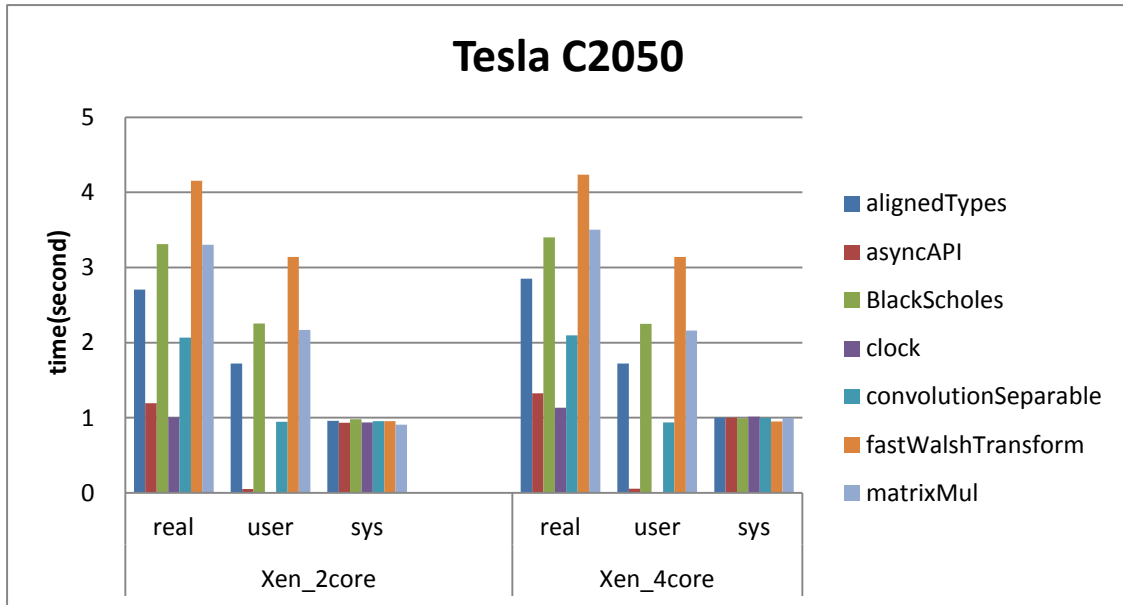


Figure 4-8. Execution Time between 2 Core and 4 Core VM with C2050

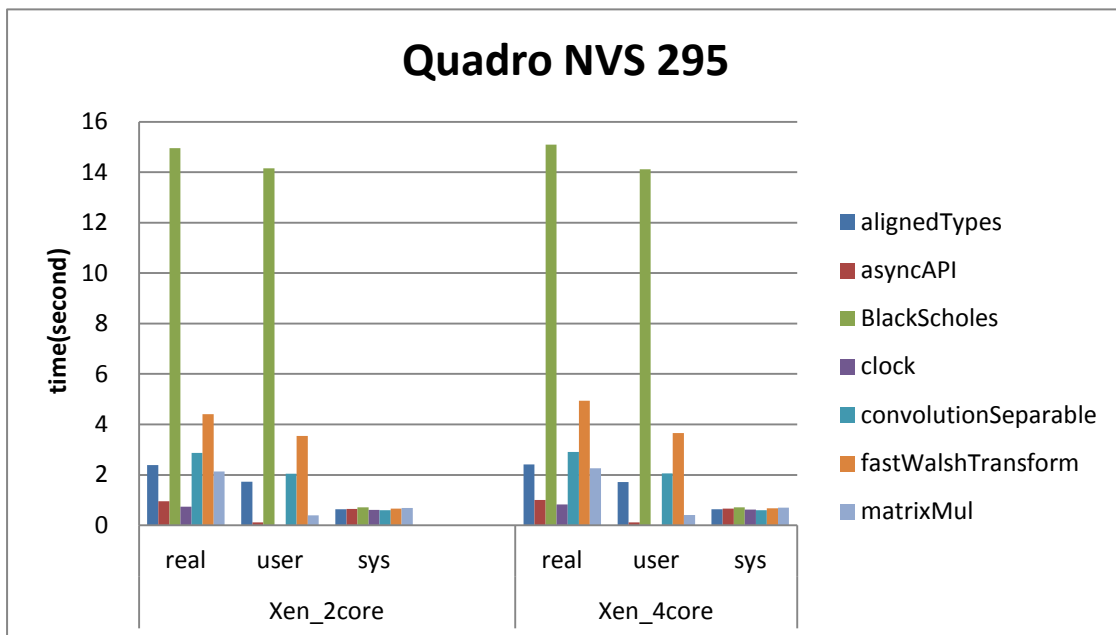


Figure 4-9. Execution Time between 2 Core and 4 Core VM with NVS295

Figure 4-7, **Figure 4-8** and **Figure 4-9** show that the execution time between two CPUs and four CPUs in virtual machines based on Xen. In this figure, it is obvious to see that the number of CPU does not affect the performance of GPU.

From **Figure 4-1** to **Figure 4-9**, we can demonstrate the execution time is very

close; only the system time is different significantly.

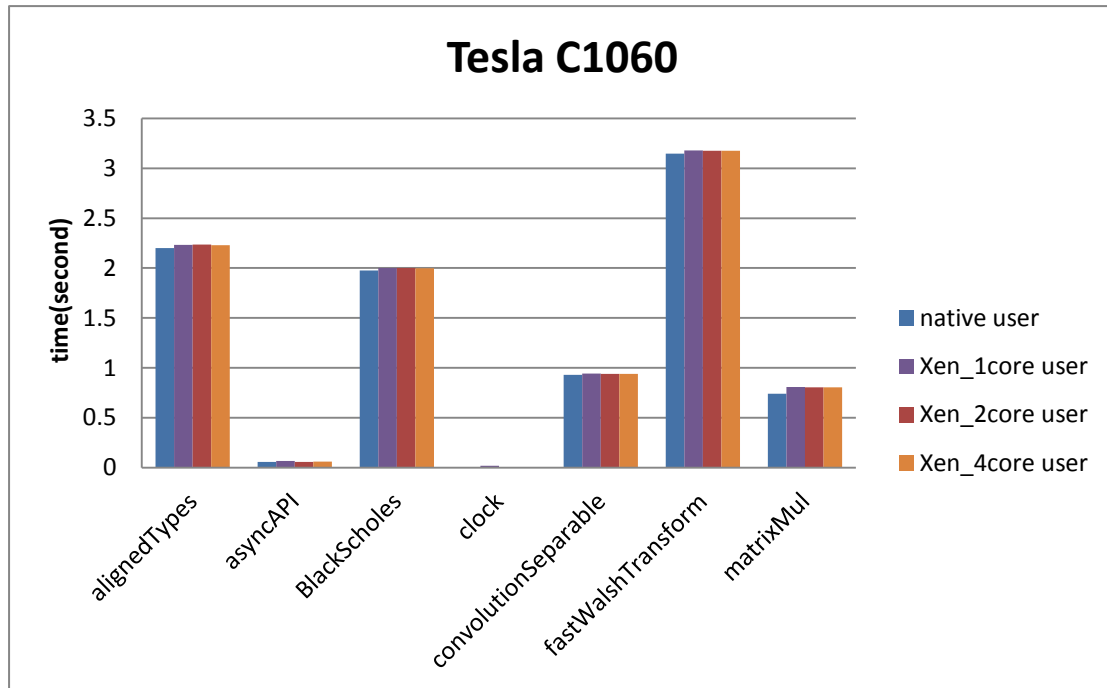


Figure 4-10. User Time with C1060

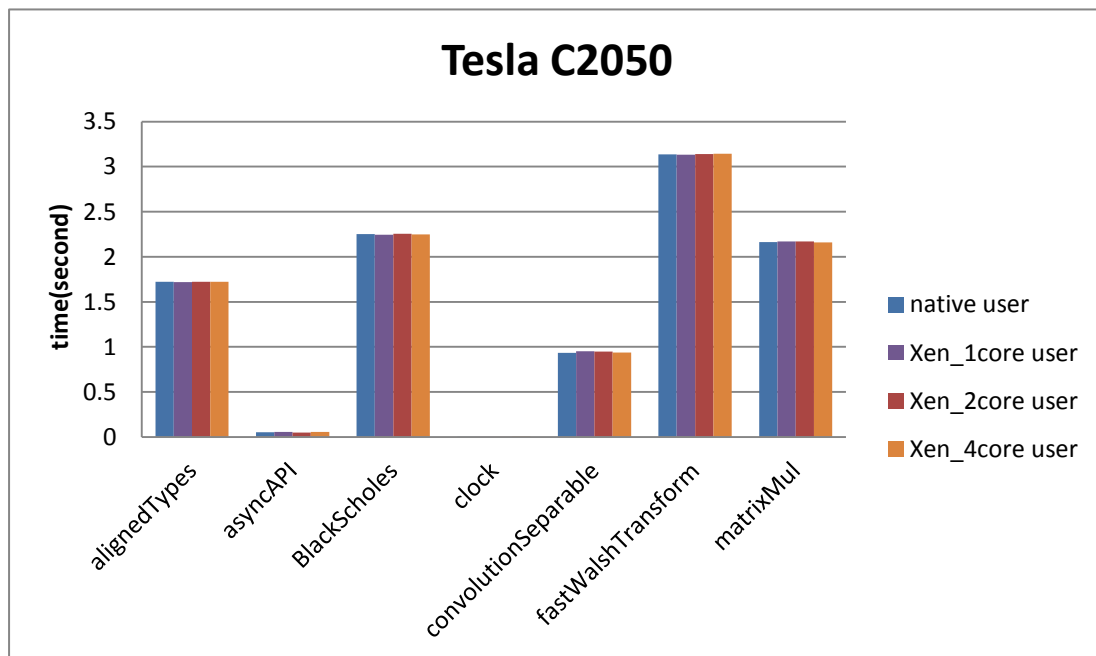


Figure 4-11. User Time with C2050

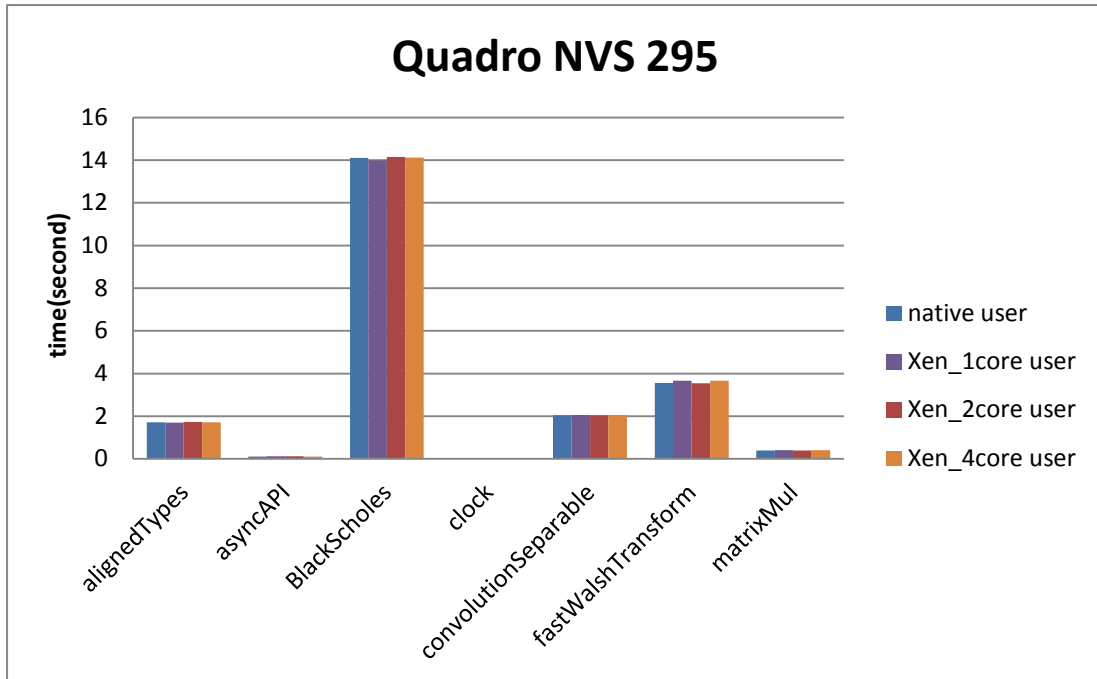


Figure 4-12. User Time with NVS295

In **Figure 4-10**, **Figure 4-11** and **Figure 4-12**, we pick the user time in each SDK benchmark execution time. The user time in this figure means the GPU computing time. No matter the native or virtual machines, the performance of GPU is the same even through the PCI pass-through. There is only a slight different between native and virtual machines -simply 0.001 second. User time of benchmark called “clock” are all under 0.001 second; therefore, it is not obvious to tell in this figure.

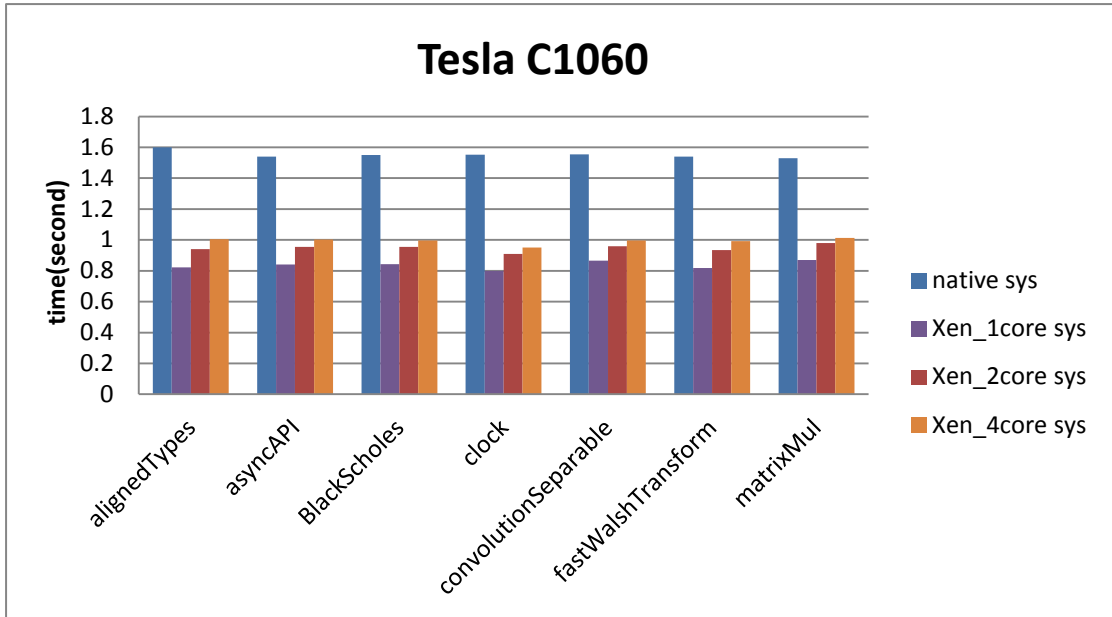


Figure 4-13. System Time with C1060

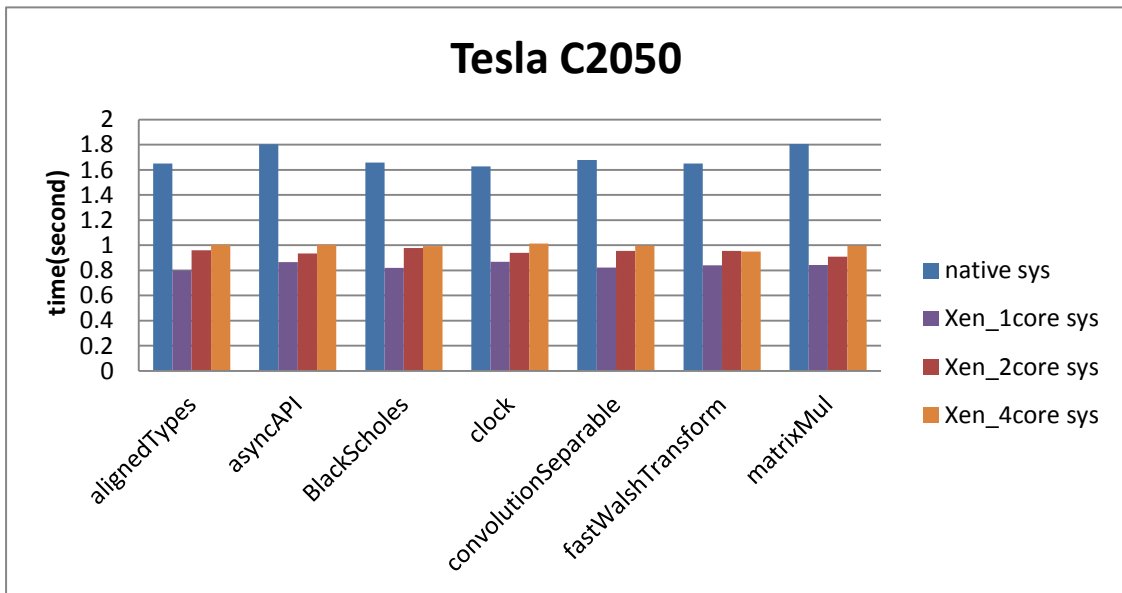


Figure 4-14. System Time with C2050

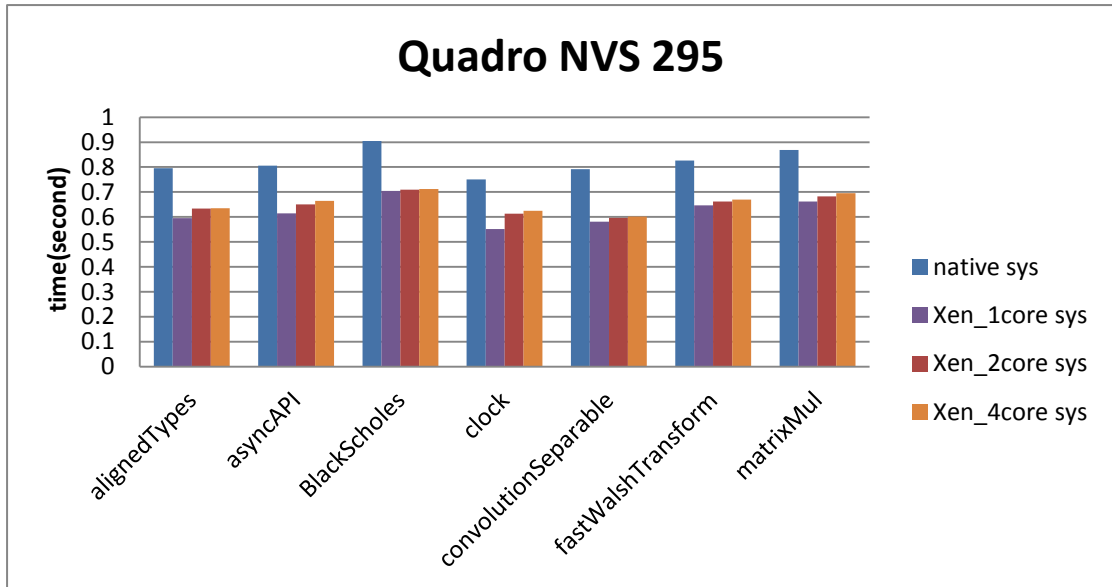


Figure 4-15. System Time with NVS295

In **Figure 4-13**, **Figure 4-14** and **Figure 4-15**, the system time of each SDK benchmark. Using the GPU accelerator to help compute and the system inner communication is also important. The native machine's system time is obviously much longer than virtual machines. And system time of the virtual machine with one CPU is shorter than the others which means if we run a program with heavily GPU computing, we can simply use one CPU in our virtual machine to save more resource on the host server for other users.

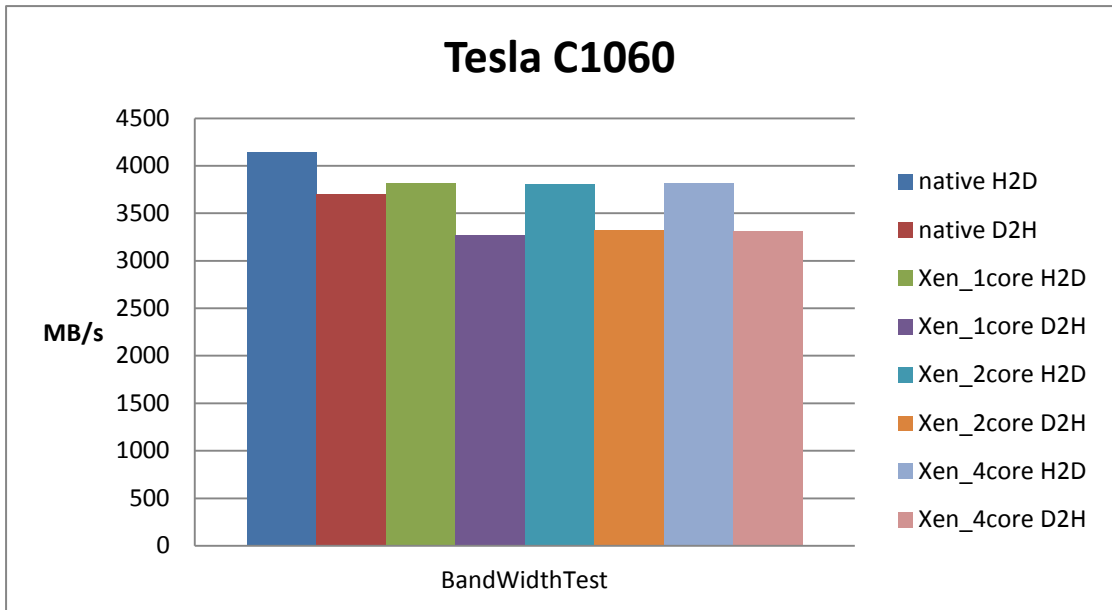


Figure 4-16. Bandwidth Test with C1060

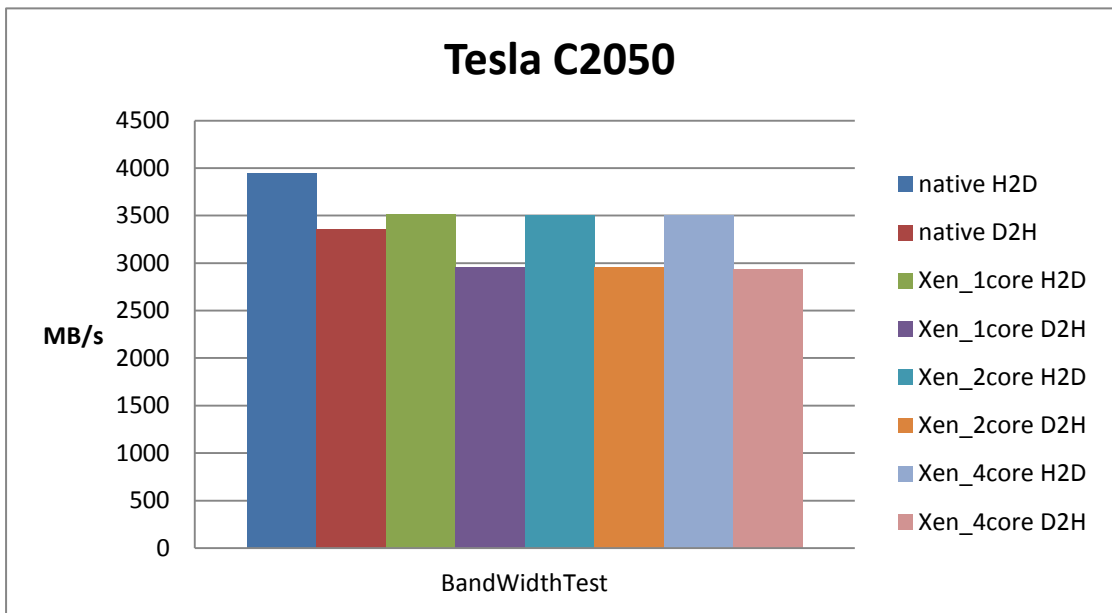


Figure 4-17. Bandwidth Test with C2050

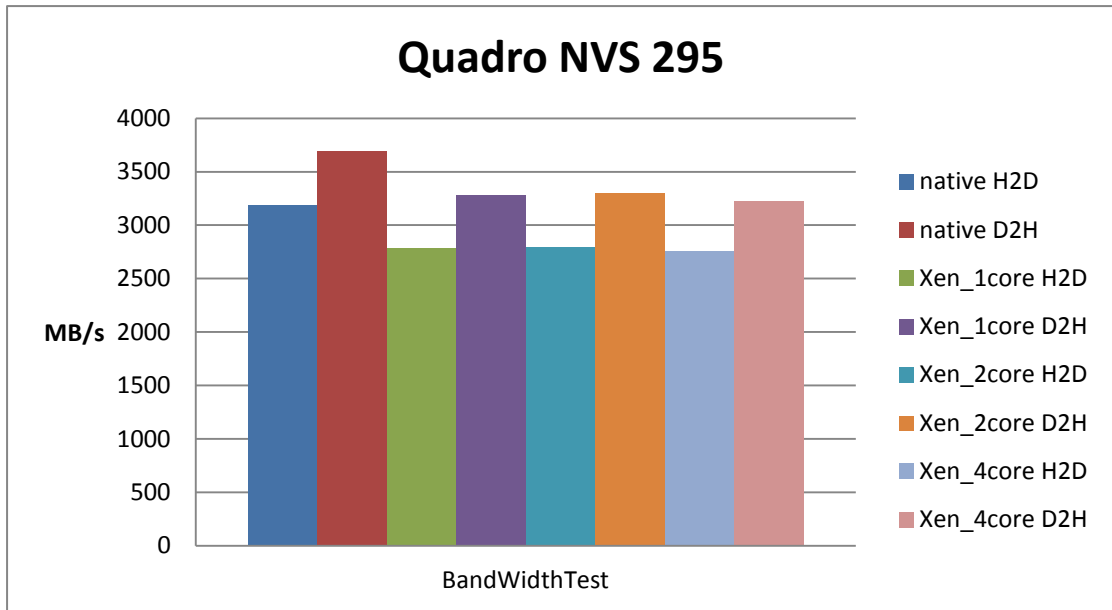


Figure 4-18. Bandwidth Test with NVS295

In **Figure 4-16**, **Figure 4-17** and **Figure 4-18**, the H2D means “Host to Device”, the D2H means “Device to Host”. There is another value called D2D, means “Device to Device” which the values are almost the same so we skip the discussion here. It is more obvious to see the bandwidth of native is higher than others. In this figure, the CPU numbers of virtual machines do not affect the bandwidth. PCI pass-through is the main reason that really affects some bandwidth between virtual machine and GPU accelerator- the virtual machine’s bandwidth is about 400 MB/s lower than native.

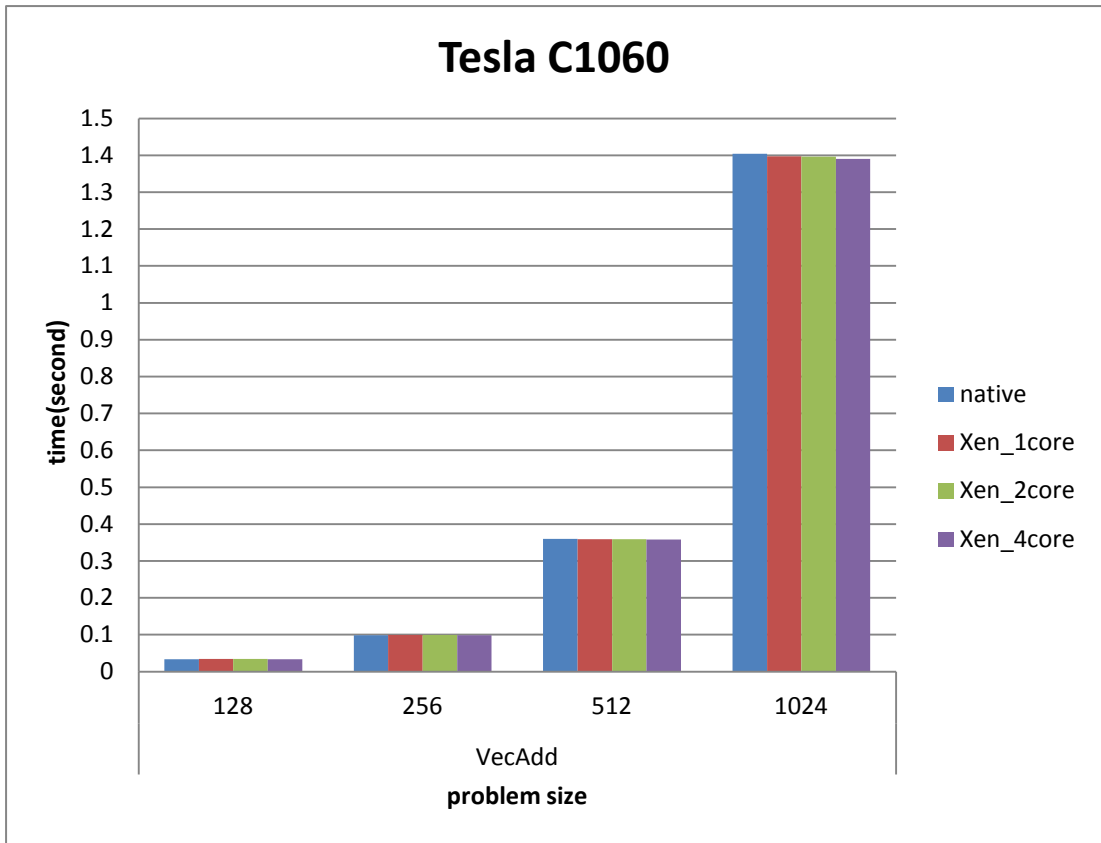


Figure 4-19. Execution time of VecAdd with C1060

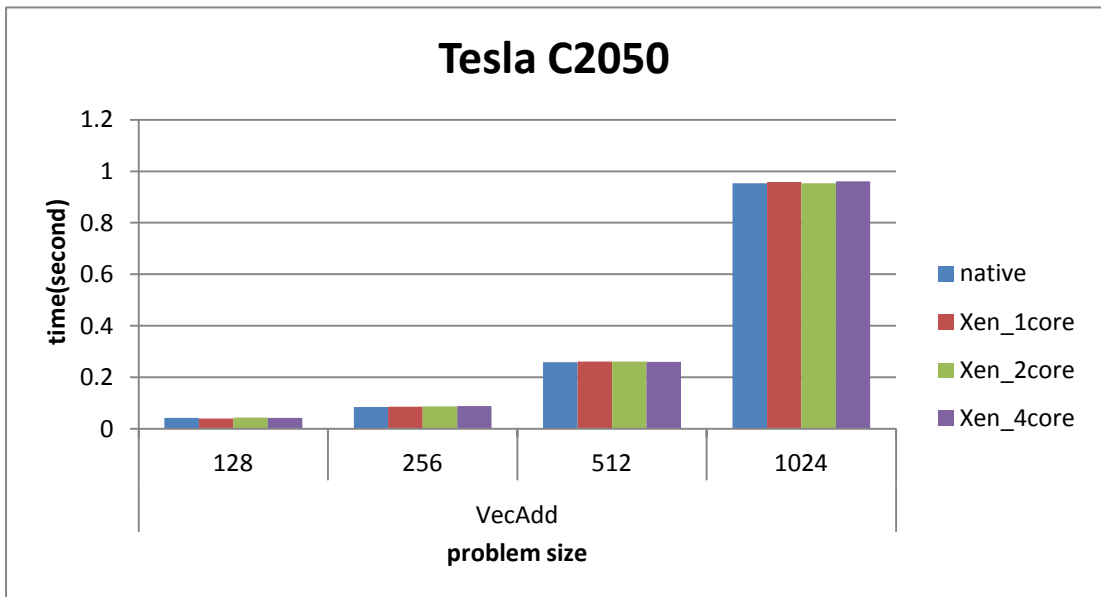


Figure 4-20. Execution time of VecAdd with C2050

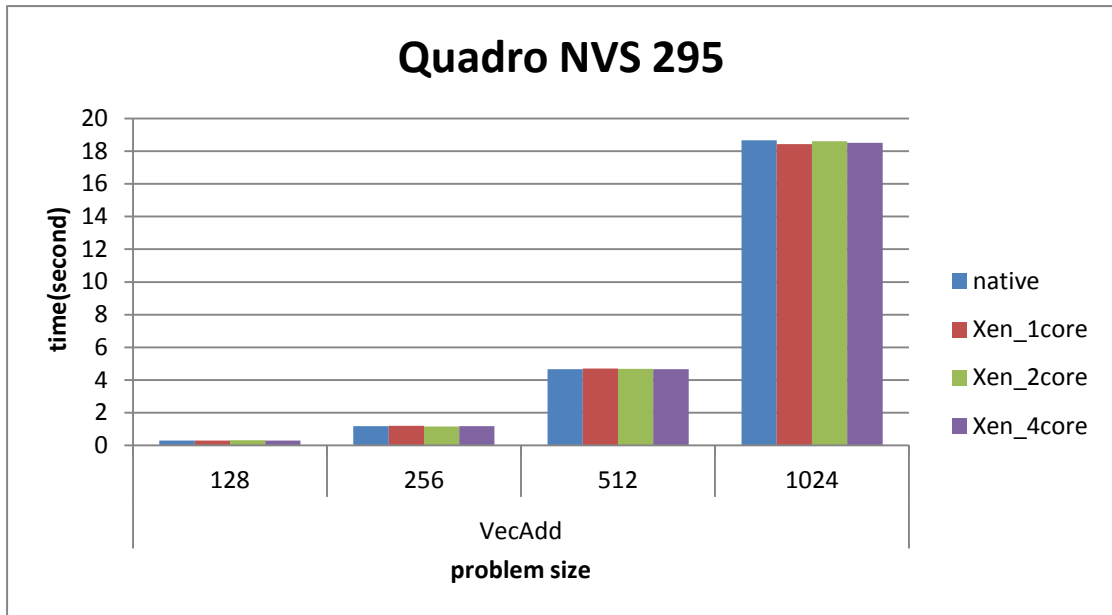


Figure 4-21. Execution time of VecAdd with NVS295

In **Figure 4-19**, **Figure 4-20** and **Figure 4-21**, the application is VecAdd. We use 128, 256, 512 and 1024 to be our problem size. The difference of these four environments is very slight. Even the execution time of virtual machine is shorter than native machine. We think it is caused by the difference of real machine and virtual machine. Besides, the GPU performance results between these four are all the same.

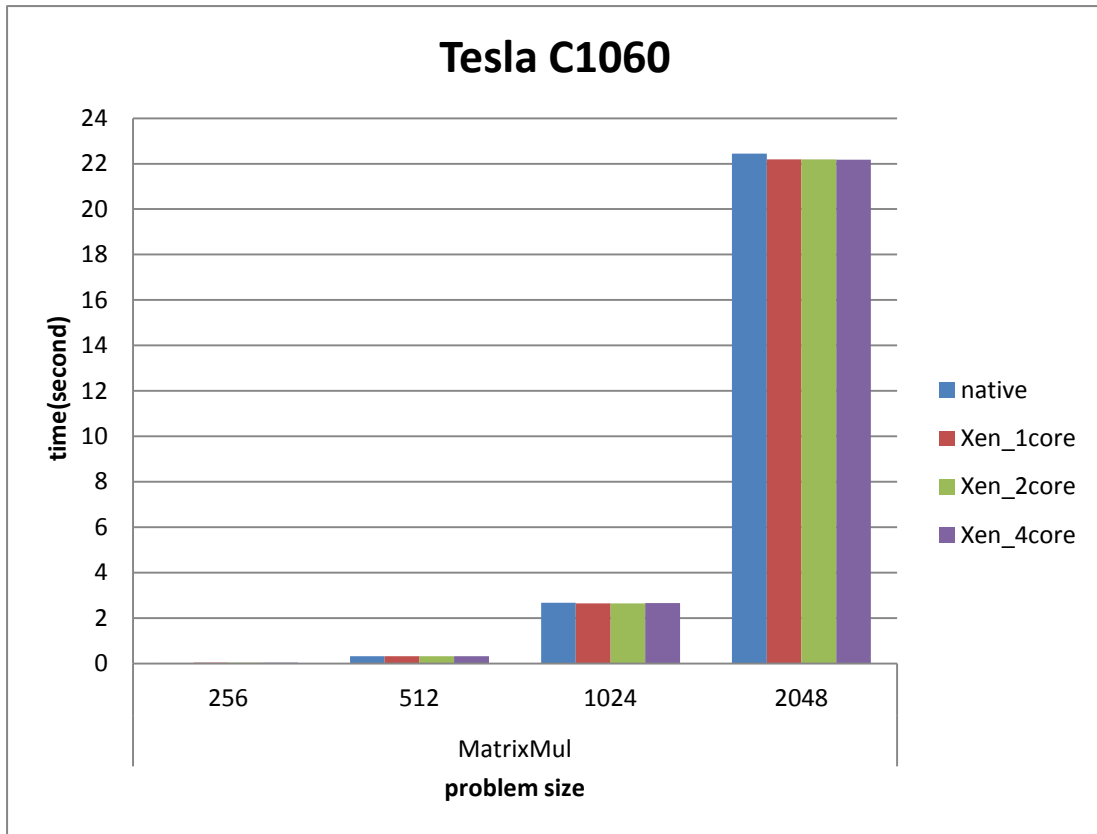


Figure 4-22. Execution time of MatrixMul with C1060

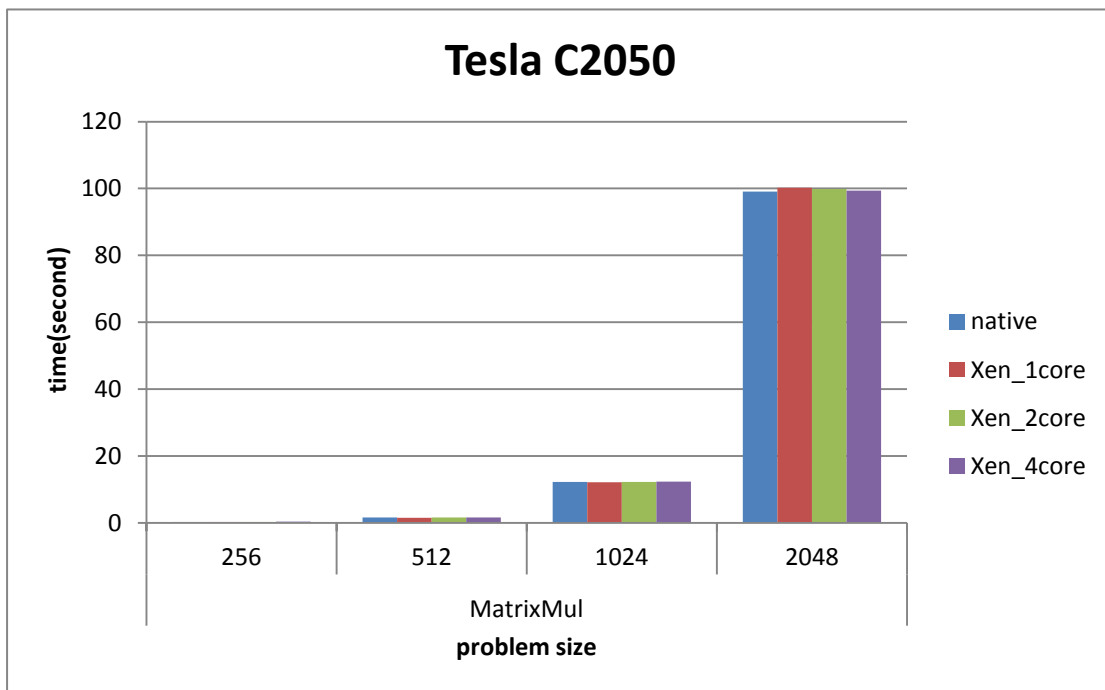


Figure 4-23. Execution time of MatrixMul with C2050

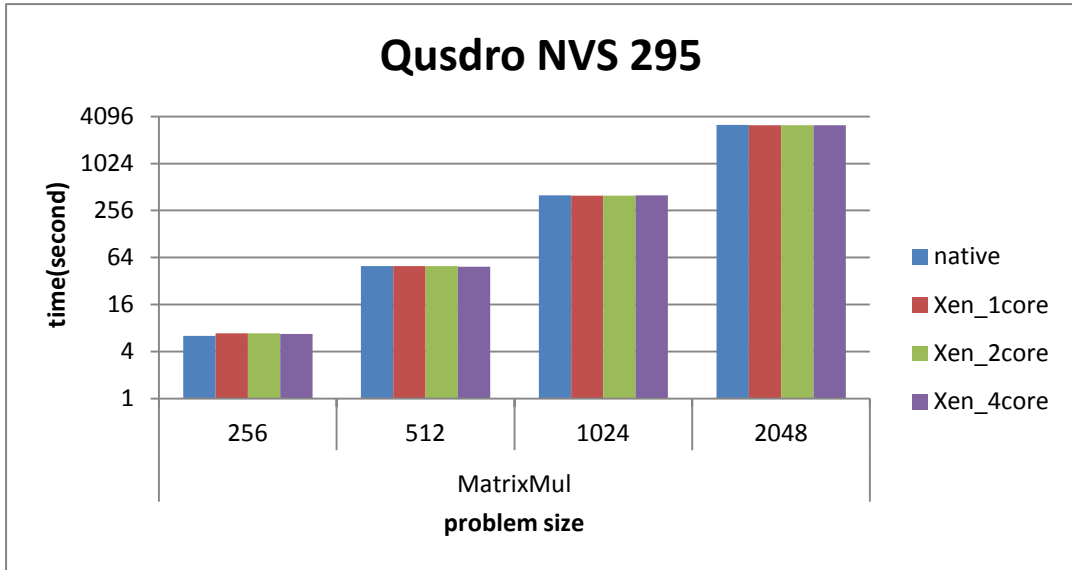


Figure 4-24. Execution time of MatrixMul with NVS295

In **Figure 4-22**, **Figure 4-23** and **Figure 4-24**, the execution time of MatrixMul. In this figure, we can see the similar result as the previous. The execution times of these four environments are very close. The execution time of virtual machine is also shorter than real machine. The execution time of problem size 256 is shorter than 0.1 second so it is difficult to see clear in this figure.

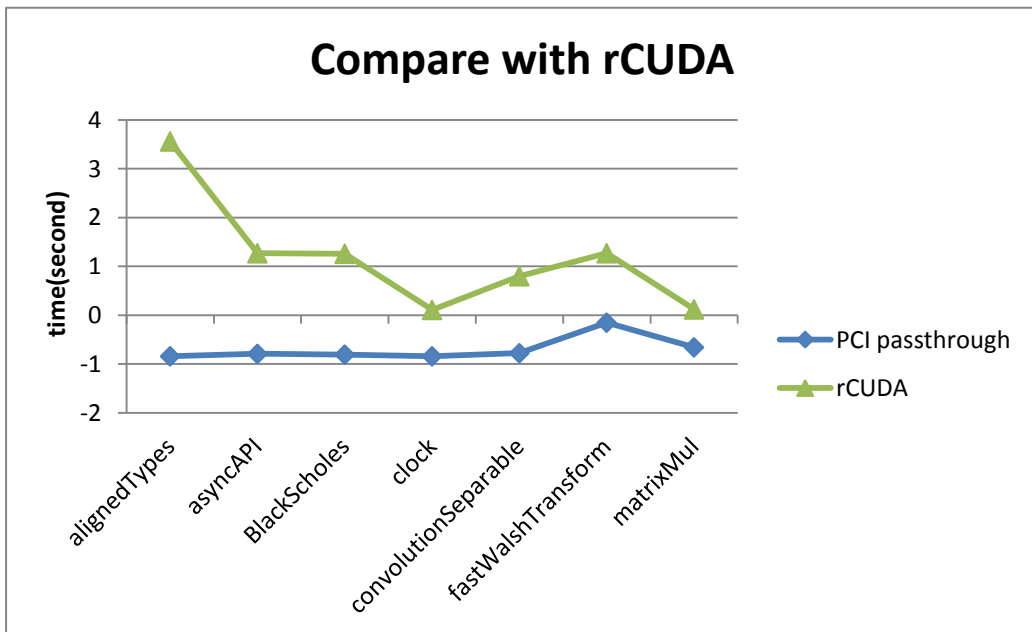


Figure 4-25. Compare with rCUDA

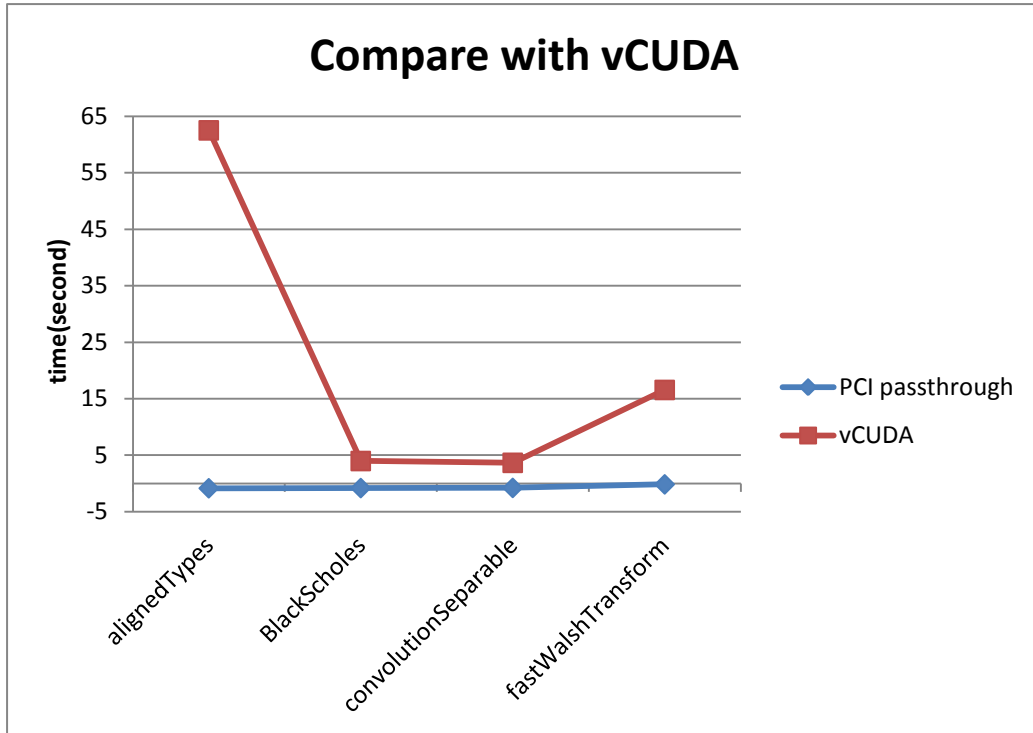


Figure 4-26. Compare with vCUDA

Also we compare with rCUDA and vCUDA. **Figure 4-25** and **Figure 4-26** show the comparison. The time in the figure is the time that we minus from the execution time before GPU virtualization and after. We use the time after GPU virtualization minus before GPU virtualization. The execution time is taken from [27] [30]. From these two figures, we can see that using PCI pass-through dose not add too much time. Compare with these two technologies, PCI pass-through is more efficient.

Chapter 5

Conclusions and Future Work

5.1. Concluding Remark

In our work, we can see the GPU performance is the same in native and virtual machine. No matter how many CPUs in virtual machine, the GPU provide the same performance by PCI pass-through. Even if we use virtual machine, the system time is less than real machine; the system time of the virtual machine with one CPU is less than four CPUs. The inner communication in virtual machine is not through the real hardware but simply relies on the memory of the real machine.

Data transfer time is shorter than rCUDA because rCUDA is network related and the speed of network is the key of rCUDA. Code needs to be rewritten if using rCUDA, but not PCI pass-through. Though rCUDA can let the virtual machine run not only in local GPU, but also in remote GPU by network.

Taking PCI pass-through is more direct if we want to make comparison of vCUDA. vCUDA uses the middleware as the connect point but it takes more time than PCI pass-through. Using the PCI pass-through to implement that computing with GPU accelerator in virtual machines can save more resource but has the same high performance in real machine overall.

5.2. Future Work

In the future, we may test more GPU board for PCI pass-through and implement GPU hot-plug to virtual machine. GPU hot-plug is very useful for virtual machine and the whole system. There is an opensourced monitor system called OpenNebula that the interface of virtual machine can be controled through webpage. Therefore, we may use OpenNebula to control our virtual machine with GPU PCI pass-through.

Bibliography

- [1] TOP 500, <http://www.top500.org/>
- [2] nVidia, <http://www.nvidia.com>
- [3] Cloud computing, http://en.wikipedia.org/wiki/Cloud_computing
- [4] GPGPU, <http://en.wikipedia.org/wiki/GPGPU>
- [5] PCI-pass-through, <http://www.ibm.com/developerworks/linux/library/l-pci-passthrough>
- [6] CUDA, http://www.nvidia.com.tw/object/cuda_home_new_tw.html
- [7] National Institute of Standards and Technology, <http://www.nist.gov/index.html>
- [8] Virtualization, <http://en.wikipedia.org/wiki/Virtualization>
- [9] Full Virtualization, http://en.wikipedia.org/wiki/Full_virtualization
- [10] Para Virtualization, <http://en.wikipedia.org/wiki/Paravirtualization>
- [11] Xen, <http://www.xen.org/>
- [12] KVM, http://www.linux-kvm.org/page/Main_Page
- [13] NVIDIA CUDA SDK, <http://developer.nvidia.com/cuda-cc-sdk-code-samples>
- [14] Download CUDA, <http://developer.nvidia.com/object/cuda.htm>
- [15] NVIDIA CUDA Programming Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
- [16] CUDA-wiki, <http://en.wikipedia.org/wiki/CUDA>
- [17] Fred V. Lionetti, Andrew D. McCulloch and Scott B. Baden, “Source-to-Source Optimization of CUDA C for GPU Accelerated Cardiac Cell Modeling,” *Lecture Notes in Computer Science*, 2010, Volume 6271, Euro-Par 2010 - Parallel Processing, Pages 38-49.
- [18] Sungbo Jung, “Parallelized pairwise sequence alignment using CUDA on multiple GPUs,” *BMC Bioinformatics*, 2009, Volume 10, Supplement 7, A3.
- [19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron, “A Performance Study of General-Purpose Applications on

Graphics Processors Using CUDA,” *Journal of Parallel and Distributed Computing*, Volume 68, Issue 10, October 2008, Pages 1370-1380.

- [20] OpenCL, <http://www.khronos.org/opencv/>
- [21] OpenCL-wiki, <http://en.wikipedia.org/wiki/OpenCL>
- [22] M.J. Harvey, G. De Fabritiis, “Swan: A tool for porting CUDA programs to OpenCL,” *Computer Physics Communications*, Volume 182, Issue 4, April 2011, Pages 1093-1099.
- [23] QEMU, http://wiki.qemu.org/Main_Page
- [24] VirtualBox, <https://www.virtualbox.org/>
- [25] Chia-Tien Dan Lo, Kai Qian, "Green Computing Methodology for Next Generation Computing Scientists," *Proceedings of IEEE 34th Annual Computer Software and Applications Conference*, pp.250-251, 2010.
- [26] Benjamin Zhong, Ming Feng, Chung-Horng Lung, “A Green Computing Based Architecture Comparison and Analysis”, *GREENCOM-CPSCOM '10 Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pp.386-391, 2010.
- [27] J. Duato, A. J. Peñna, F. Silla, R. Mayo, and E. S. Quintana-Ortí, “rCUDA: Reducing the number of GPUbased accelerators in high performance clusters,” *Proceedings of the 2010 International Conference on High Performance Computing & Simulation (HPCS 2010)*, Jun. 2010, pp. 224–231.
- [28] J. Duato, A.J. Pena, F. Silla, J.C. Fernandez, R. Mayo, E.S. Quintana-Orti, “Enabling CUDA acceleration within virtual machines using rCUDA,” *Proceedings of High Performance Computing (HiPC), 2011 18th International Conference*, 2010, Pages 1-10.
- [29] J. Duato, A. J. Peñna, F. Silla, R. Mayo, and E. S. Quintana-Orti, “Performance of CUDA virtualized remote GPUs in high performance clusters,” *Proceedings of International Conference on Parallel Processing (ICPP)*, Sep. 2011, Pages 365-374.
- [30] L. Shi, H. Chen, and J. Sun, “vCUDA: GPU accelerated high performance computing in virtual machines,” *Proceedings of IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, 2009, Page 1-11.

- [31] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, “GViM: GPU-accelerated virtual machines,” in *3rd Workshop on System-level Virtualization for High Performance Computing*. NY, USA: ACM, 2009, pp. 17–24.
- [32] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, “A GPGPU transparent virtualization component for high performance computing clouds,” in *Euro-Par 2010 - Parallel Processing, ser. LNCS, P. D Ambra, M. Guarracino*. D. Talia, Eds. Springer Berlin / Heidelberg, 2010, vol. 6271, pp. 379–391.
- [33] Front and back ends, http://en.wikipedia.org/wiki/Front_and_back_ends
- [34] VMGL, <http://sysweb.cs.toronto.edu/vmgl>
- [35] Nadav Amit, Muli Ben-Yehuda and Ben-Ami Yassour, “IOMMU: Strategies for Mitigating the IOTLB Bottleneck,” *Lecture Notes in Computer Science, 2012*, Volume 6161, Computer Architecture, Pages 256-274.
- [36] NVIDIA Tesla C1060 Computing Processor, http://www.nvidia.com/object/product_tesla_c1060_us.html
- [37] NVIDIA Quadro NVS 295, http://www.nvidia.com.tw/object/product_quadro_nvs_295_tw.html
- [38] NVIDIA Tesla C2050 Computing Processor, http://www.nvidia.com.tw/object/product_tesla_C2050_C2070_tw.html
- [39] CentOS, <http://www.centos.org/>
- [40] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, “VMM-independent graphics acceleration,” in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2007, pp. 33–43.
- [41] C.T. Yang, C.L. Huang and C.F. Lin, “Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters,” *Computer Physics Communications*, Vol. 182, Issue 1, pp. 266-269, June 25, 2010.
- [42] C.T. Yang, C.L. Huang, C.F. Lin and T.C. Chang, “Hybrid Parallel Programming on GPU Clusters,” *Proceedings of International Symposium on Parallel and Distributed Processing with Applications (ISPA) 2010*, pp. 142-147, Sept. 2010.
- [43] C.T. Yang; T.C. Chang; H.Y. Wang; Chu, W.C.C.; C.H Chang, “Performance Comparison with OpenMP Parallelization for Multi-core Systems,” *Proceedings*

of Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium, 2011, pp.232-237.

Appendix

A. Setup Xen on CentOS

- I. Make sure that SELinux is disabled or permissive

```
# vi /etc/sysconfig/selinux
```

```
# reboot
```

- II. Creating A Network Bridge

```
# yum install bridge-utils
```

```
# vi /etc/sysconfig/network-scripts/ifcfg-br0
```

- III. Modify /etc/sysconfig/network-scripts/ifcfg-eth0

```
# vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

```
# /etc/init.d/network restart
```

- IV. Installing Xen

```
# yum install http://au1.mirror.crc.id.au/repo/kernel-xen-release-6-3.noarch.rpm
```

```
# yum install kernel-xen xen
```

Edit /boot/grub/menu.lst

```
# vi /boot/grub/menu.lst
```

Replace the first word 'kernel' and 'initrd' with module

Then add the line `kernel /xen.gz dom0_mem=1024M cpufreq=xen
dom0_max_vcpus=1 dom0_vcpus_pin` after the root line

- V. Install the libvirt and make a patch

```
# yum install libvirt python-virtinst
```

```
# yum groupinstall 'Development Tools'
```

```
# yum install python-devel xen-devel libxml2-devel xhtml1-dtds readline-devel
```

```

ncurses-devel      libtasn1-devel    gnutls-devel     augeas           libudev-devel
libpciaccess-devel yajl-devel        sanlock-devel    libpcap-devel    libnl-devel      avahi-devel
libselenium-devel  cyrus-sasl-devel  parted-devel     device-mapper-devel
numactl-devel      libcap-ng-devel   netcf-devel      libcurl-devel    audit-libs-devel
systemtap-sdt-devel

# mkdir /root/src

# cd /root/src

# wget

http://vault.centos.org/6.2/os/Source/SPackages/libvirt-0.9.4-23.el6.src.rpm

# rpm -i libvirt-0.9.4-23.el6.src.rpm

# wget http://pasik.reaktio.net/xen/patches/libvirt-spec-rhel6-enable-xen.patch

# cd /root/rpmbuild/SPECS

# cp -a libvirt.spec libvirt.spec.orig

# patch -p0 < ~/src/libvirt-spec-rhel6-enable-xen.patch

# rpmbuild -bb libvirt.spec

# cd /root/rpmbuild/RPMS/x86_64/

#          rpm          -Uvh          --force          libvirt-0.9.4-23.el6.x86_64.rpm

libvirt-client-0.9.4-23.el6.x86_64.rpm libvirt-python-0.9.4-23.el6.x86_64.rpm

# reboot

```

B. Setup PCI passthrough

I. Enable iommu

```
# vi /boot/grub/menu.lst
```

Add 'iommu=1' at the end of kernel

II. Binding Devices to pci-stub

```
# lspci -n
```

Get the Device ID like '01:00.0 0200: 8086:10b9 (rev 06)'

Then

```
# echo "8086 10b9" > /sys/bus/pci/drivers/pci-stub/new_id
```

```
# echo "0000:01:00.0" > /sys/bus/pci/devices/0000:01:00.0/driver/unbind
```

```
# echo "0000:01:00.0" > /sys/bus/pci/drivers/pci-stub/bind
```

Viewing Devices

```
# xm pci-list-assignable-devices
```

III. Add Devices to VM

Through a software in linux 'VM Manager' or command like

```
# virt-install --host-device=HOSTDEVS
```

C. CUDA Installation

I. Download gpu driver from nVidia website

Install it

```
# sh gpudriver.sh
```

II. Download cudatoolkit from nVidia website

Install it

```
# sh cudatoolkit.sh
```

Setup PATH

III. Download CUDA SDK from nVidia website

Install it

```
$ sh SDK.sh
```

```
$ cd SDK/C
```

```
$ make
```