GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor

Lan Vu
University of Colorado Denver,
1380 Lawrence St., Denver, CO 80204, USA
lan.vu@ucdenver.edu

Hari Sivaraman, Rishi Bidarkar
VMware,
3401 Hillview Ave, Palo Alto, CA 94304, USA
{hsivaraman, rishi}@vmware.com

Keywords: GPGPU, CUDA, virtual machine, high performance computing, virtualization.

Abstract

Graphics Processing Units (GPU) have become important components in high performance computing (HPC) systems for their massively parallel computing capability and energy efficiency. Virtualization technologies are increasingly applied to HPC to reduce administration costs and improve system utilization. However, virtualizing the GPU to support general purpose computing presents many challenges because of the complexity of this device. On VMware's ESX hypervisor, DirectPath I/O can provide virtual machines (VM) high performance access to physical GPUs. However, this technology does not allow multiplexing for sharing GPUs among VMs and is not compatible with vMotion, VMware's technology for transparently migrating VMs among hosts inside clusters. In this paper, we address these issues by implementing a solution that uses "remote API execution" and takes advantage of DirectPath I/O to enable general purpose GPU on ESX. This solution, named vmCUDA, allows CUDA applications running concurrently in multiple VMs on ESX to share GPU(s). Our solution requires neither recompilation nor even editing of the source code of CUDA applications. Our performance evaluation has shown that vmCUDA introduced an overhead of 0.6% - 3.5% for applications with moderate data size and 14% - 20% for those with large data (e.g. 12.5 GB - 237.5GB in our experiments).

1. INTRODUCTION

High performance computing has recently seen the emergence of GPUs as co-processors to provide the parallel processing capability that was traditionally dominated by CPUs [1]. An increasing number of supercomputers in the TOP500 have been equipped with GPUs as the main computing components. Modern GPUs like Nvidia's Tesla, Fermi, and Kepler class that have hundreds to thousands of processing elements can now support massively parallel data processing and have become ideal computing resources for large data centers [2]. The development of general purpose applications that leverage GPU computing is easier now with the introduction of programming models like CUDA[3] and OpenCL[4].

1.1. Motivation

The integration of virtualization, one of the key components that enable cloud computing, in HPC is welcomed because it helps to significantly save the cost of acquisition, administration, maintenance, and energy [5-6]. The trend of providing general purpose GPU (GPGPU) support is beginning to accelerate. Amazon Web Services is now providing access to GPGPUs in their clouds and Nvidia recently announced its offerings in this area. Hence, it is essential to enable the GPGPU capability in virtualization platforms like VMware [7] or Xen [8] to support the general purpose GPU computing.

Several research efforts [9-14] have investigated mechanisms to provide GPGPU in virtual machines with different degrees of integration into the hypervisors. However, they are mostly targeted for open-source virtualization platforms like Xen, Kernel-based Virtual Machine (KVM) or VirtualBox and their solutions generally have high overheads. For example, one benchmark in the Nvidia GPU Computing SDK [15], rCUDA [9] introduced 60.8% overhead on KVM and 200.6% on VirtualBox; the overhead of vCUDA[14] was 73.5% and the overhead of GViM [12] was 25% on Xen. None of these efforts provide a comprehensive solution to support GPGPU on VMware's ESX hypervisor which is one of the most popular virtualization platforms currently.

1.2. Contribution

In our study, we investigate a virtualized GPGPU solution that provides multiple VMs high speed access to shared physic GPU on VMware's ESX to offload the general purpose computing workload. The main contributions of our research work include:

(1) Proposing a framework named vmCUDA that gives VMs the ability to leverage CUDA, a GPGPU solution by Nvidia. By deploying the "remote API execution" model and utilizing the DirectPath I/O technology, CUDA applications on multiple VMs can have concurrent high performance access to virtualized GPUs without the requirement of recompiling or editing the CUDA code like the rCUDA approach [9, 16].

(2) Providing the experimental evaluation of our proposed work with the introduction and analysis of the techniques that we applied to utilize the benefits of ESX and minimize the performance overhead of our framework. Using the performance results, we show the efficiency of our approach.

The rest of this paper is organized as follows: Section 2 provides basic knowledge of how hypervisors like ESX virtualize GPUs to enable both graphical and non-graphical operations. A brief introduction of CUDA is also provided. In Section 3, we present our proposed solution to support GPGPU on ESX. We evaluate its efficiency in Section 4. Section 5 discusses related work. The final section presents our conclusions and future work.

2. BACKGROUND

Virtualization creates virtual devices and platforms that give software applications enhanced access to underlying hardware and provide better utilization of hardware resources. Although virtualization technology has been successfully applied to a variety of devices (i.e. CPU, network device, HDD, etc.) [17], virtualizing modern GPUs is facing challenges due to the

complexity and rapid change of this device [18]. We present, in this section, current techniques to offload graphics operations and other computing workloads to GPUs on VMware's ESX hypervisor as well as their limitations which lead to the introduction of our research work. We also introduce CUDA, the computing platform for GPGPU by Nvidia that we support in our proposed GPGPU virtualization solution.

2.1. Virtualizing GPU for Graphic Operations

Virtualizing the graphical capabilities of GPUs has been added to hypervisors in last few years and the major hypervisors now provide support for offloading graphics operations to GPUs. This virtualization is typically achieved by using a split driver model, where the frontend driver exists in the guest VM, and the backend driver is located in the control domain (Hyper-V, Xen) or the hypervisor itself (VMware's ESX). The frontend driver works inside the guest OS as a graphics driver. It accepts requests from applications and the guest OS, and routes them to the backend driver. The backend driver operates like a proxy by forwarding the requests to the vendor driver that actually interacts with the physical device. Additionally, the backend driver also undertakes the load balancing, scheduling and resource sharing to allow multiple VMs to securely and robustly share a GPU. For VMware products, the GPU virtualization solution for graphic operations, which is named vGPU, also work with its vMotion technologies [19] that allow VMs to migrate between hosts without interrupting graphics operations running on a physical GPU.

2.2. GPGPU Virtualization

While the compute capabilities of GPGPUs are not currently virtualized by ESX, applications running on a VM can still leverage CUDA [3] and the other compute APIs by using VMware's DirectPath I/O technology (also known as direct pass-through or direct pass-by) which is the most commonly used technique to expose a GPGPU to a VM [9, 12, 14, 20]. This technology allows a VM to directly access a PCIe device and users can install and use off-the-shelf CUDA and OpenCL drivers in their VM. DirectPath I/O can provide applications in VMs on ESX a performance very close to that of native execution. We have tested and found that most CUDA SDK examples [15] performed at 98% of their native performance using DirectPath I/O. However, using direct pass-through requires an explicit one-to-one binding between the GPU and the VM, necessitating a GPU device per active VM, resulting in the under-utilization that virtualization is meant to address. In addition, a VM using direct pass-through to access a device is not compatible with vMotion [19], an important feature of VMware's products.

2.3. CUDA

CUDA (Compute Unified Device Architecture) [3] is a complete GPGPU solution by Nvidia that includes GPU devices whose architecture was designed to efficiently support both graphics and non-graphics operations. Its software interface brings ease to building and executing non-graphics applications on these devices. CUDA is among the most popular GPU computing frameworks for developing high performance general purpose applications. The CUDA software stack is composed of a hardware driver, an application programming interface (API) and its runtime library. CUDA applications are written in "C" and use the CUDA APIs to access the GPUs at runtime to get device information, copy data and shift the computing workload to GPUs. The programming and execution model of CUDA applications usually involve following tasks: (1) copy input data from main memory to GPU memory. (2) Launch kernel to be executed on GPU. Each kernel is a computational unit doing a specific task. (3) GPU executes the kernel that has been launched. (4) Copy output data back from GPU memory to main memory.

3. A FRAMEWORK FOR VIRTUAL GPGPU ON ESX HYPERVISOR

In this section, we present our GPGPU solution for CUDA applications on ESX that overcomes the limitations of the current approach like one-to-one binding and no vMotion support as presented in Section 2.2. Generally, we use the "remote API execution" model in which an appliance VM is used to manage the physical GPUs on the host and to provide compute services to multiple client VMs (a client VM is the VM running the user's CUDA application). The overview of our developed framework, named vmCUDA, is presented in Figure 1.

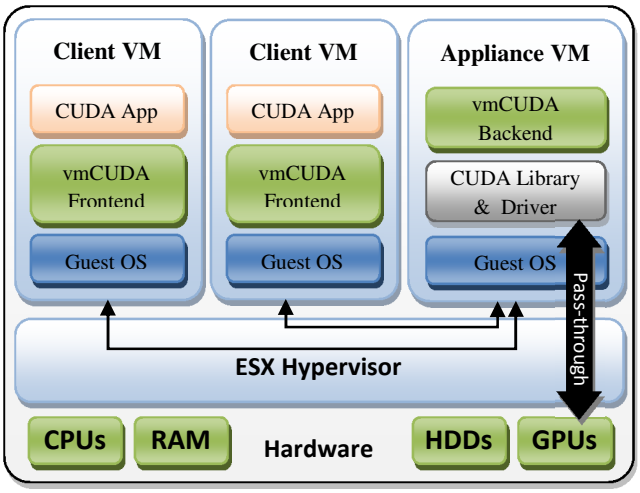

Figure 1: Overview of the proposed framework on ESX

This framework has the following features:

- It allows CUDA applications running on multiple VMs to access and share GPUs of the system to maximize the utilization of this computing resource.
- It is compatible with vMotion technology of VMware because the client VMs can either be co-located with the appliance VM on the same host or on entirely different hosts. Similarly, CUDA applications can access GPU resources on its ESX host or on a remote one.
- Our approach does not require recompilation or even access to the source code of CUDA applications. This is an advanced feature compared to the related solutions [9, 16].
- The DirectPath I/O technology is utilized to maximize the performance of CUDA applications.
- It works independently of the hypervisor. Hence, no revision of Vmware's ESX is required to integrate this framework to ESX. In addition, our framework can leverage the existing stock CUDA drivers which eliminates the cost of developing customized CUDA driver for ESX hypervisor.

3.1. The Architecture

We virtualize the GPGPU stack at the CUDA API level by using two modules: the vmCUDA backend module for the appliance VM and the vmCUDA frontend for the client VM (Figure 1).

These two modules interact with each other to transfer the CUDA API requests and data between CUDA applications in client VMs and the CUDA software stack in appliance VM. The critical components in this framework are shown in Figure 2.

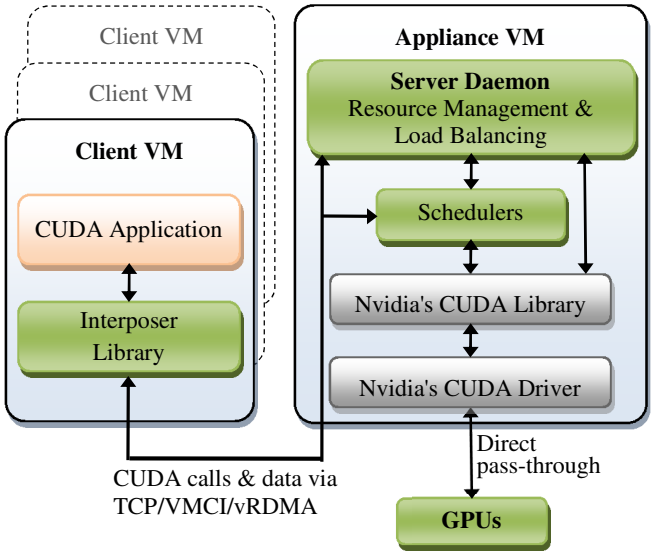

Figure 2: The architecture of the proposed framework

3.1.1. The frontend components

Our GPGPU virtualization uses an interposer-library, which intercepts the CUDA calls made by applications running in the client VM. The library collects all necessary information associated with the CUDA call, packetizes this information and communicates it to the appliance VM using VMware VMCI, VMware vRDMA or TCP/IP as appropriate. The connection between client VM and appliance VM is set up the first time a CUDA API is called by a CUDA application. This frontend module is responsible for executing data movement optimizations discussed in Section 3.2. The interposer-library is designed to be fully compatible with the standard CUDA APIs. Therefore, the existing CUDA applications can work with vmCUDA without source code revision or recompilation.

3.1.2. The backend components

On the appliance VM, our backend module comprises of a server daemon and schedulers. The server daemon is responsible for managing the GPUs and for initial communication with the client VMs. When a client VM starts executing a CUDA application, it connects to the server daemon, and sends it a copy of the binary for the CUDA application. The server daemon modifies the binary as described in Section 3.3 and then starts a new process to run this modified binary. Hereafter, the client VM communicates directly with the child process that is executing the modified binary.

The scheduler is responsible for assigning this process to a particular GPU. Currently this assignment is static and remains unchanged for the duration of the execution of the CUDA application. We plan to change this in the next implementation so that it is possible to move the CUDA application from one GPU to another mid-execution.

Using an appliance VM provides several key advantages over using the traditional split-driver implementation:

- First, it avoids the time, performance and stability problems associated with asking each and every GPU vendor to provide a driver port for ESX. Further, even if a vendor were to port an initial version, it is not at all clear that it will be updated and maintained as faithfully and frequently as a version that runs on Windows or Linux.
- Second, it avoids adding to the footprint of the hypervisor and keeps CUDA support more independent of the hypervisor.
- Finally, given that we have considerable freedom over the number of GPUs per appliance VM and the number of appliance VMs per host or cluster, we can choose to scale horizontally to maximize robustness and scalability, avoiding some of the problems associated with tying GPGPUs to a single hypervisor or control domain.

3.2. Data Movement

The data movement between the client VMs and the appliance VM has the potential to introduce significant overheads if handled inefficiently. Fortunately, ESX already supports a number of mechanisms for high-throughput, low latency communication between VMs. One such mechanism is TCP/IP using VMware's VMXNET3 driver. Another mechanism is vRDMA between co-located VMs. Our current implementation uses TCP/IP for communication but will use vRDMA as well in the future. Since both transport mechanisms expose a socket-based interface, it is easy for our solution to support all and dynamically choose the optimal mechanism, based on ESX version, and appliance location.

For the socket based data methods, i.e. TCP or vRDMA, we apply the data blocking model for CUDA data copying calls like cudaMemcpy, cudaMemcpyToSymbol, etc. This technique will affect the CUDA APIs whose data copying size are large. In our GPGPU model, each CUDA application will perform its data transfer independently using its own sockets and send/receive data buffers for performance efficiency.

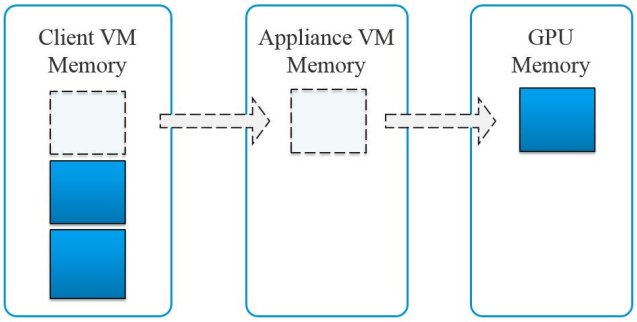

Figure 3. Data Copying with Blocking Model

Hence, large data buffers will result in a large amount of memory used in both client VMs and appliance VM which in turn reduce memory resource and increase memory overhead. This problem can become more critical when multiple VMs concurrently perform data transfer which increases the waiting time of CUDA requests in the queue. Therefore, it is necessary to divide the large data copy calls into smaller ones to share copying time fairly as well as reduce the memory usage. This data partition is executed by the interposer library of vmCUDA while the original CUDA application and its CUDA APIs are unchanged. The efficiency of this technique will be evaluated in Section 4.3

3.3. Binary Modification

CUDA supports API calls that allow applications to execute arbitrary code fragments on the GPU. To support these API calls in vmCUDA we, first send a copy of the CUDA application to the server daemon when the client VM connects to the appliance VM. The server daemon modifies this binary, as described below, to allow the intercepting process to deploy successfully. This binary modification includes the following tasks:

- First, the server daemon replaces a CUDA API in the symbol table of the binary by an interception API in the vmCUDA backend module. This interception API handles all the CUDA API calls sent by the client VM.
- The daemon edits the dynamic linking information of the binary to allow the vmCUDA backend module to be loaded by the binary when it is launched.
- Third, the execution starting point of the binary is set to the interception API indicated in the first step.
- Finally, the server daemon writes the modified binary to disk, and forks a process to execute this modified binary. When this new process starts executing, it calls the interception API which waits for TCP/IP messages from the client VM. The TCP/IP messages contain CUDA calls which are executed on the GPU. The output from GPU is sent back to the client VM.

This technique is unique and hasn't been seen in previous research work on CUDA in a hypervisor environment. By applying it at the appliance VM side, we avoid the requirement of editing code or recompiling the original CUDA application to have it work with vmCUDA.

4. PERFORMANCE EVALUATION

In this section, we evaluate the performance of vmCUDA on VMware's ESXi 5.0 environment with VMs running a Linux based operating system and compare it with the native performance of CUDA on Linux running natively on the hardware (i.e. no ESXi is installed). We also analyze the key elements that impact the performance of vmCUDA and present our solutions to minimize its overhead.

4.1. Experiment Setup

4.1.1. Testbed

We conducted our experiments on a Dell Precision T7500 server with 6-core Xeon Intel CPU 2.8 Ghz, 24GB RAM and 500 GB hard drive. It is equipped with one Nvidia Quadro 4000 with 8 SMs, 256 CUDA cores and 2GB GDDR5. This machine is used for performance benchmarks with both virtualized and non-virtualized environments. For the virtualized environment, with VMware's ESXi 5.0, the client VMs are created with one VCPU, 2GM RAM and 48GB hard drive. The appliance VM has 3 VCPU, and 2GB RAM. All the VMs run Centos 5.6 (i.e. a 32 bit Linux-based distribution). The appliance VM has an Nvidia CUDA 5.0 driver & toolkit. The client VMs have vmCUDA installed. The communication among VMs is over TPC/IP using VMware's VMXNET3 driver. We used TPC/IP as the communication method between client VM and appliance VM to show the portability of vmCUDA for other virtualization platforms like Xen, VirtualBox, or KVM or even the non-virtualized environment like cluster to provide remote use of GPUs [9]. For the native environment, we use the same physical machine with 32-bit Centos 5.6 and CUDA 5.0 driver & toolkit.

4.1.2. Benchmarks

In other to analyze the performance of vmCUDA, we selected some CUDA benchmarks from the Nvidia Computing SDK [15] for our experiments. They vary in computational load, data size and domain and include Binomial Option (Finance), Black Scholes (Finance), Matrix Multiplication (Algebra), Stereo Disparity (Image Processing), Radix Sort (Sorting). Our initial tests using these original CUDA programs showed considerably small overheads of vmCUDA (e.g. 0% - 4% overheads) compared to native CUDA. Hence, we edited these programs so that they processed larger input and output data, as shown in Table 1, to provide better the understanding of vmCUDA performance in typical applications where the date set is large. Please note that this editing was for the purpose of increasing the data size only and did not affect the structure of the original CUDA programs because our vmCUDA framework works with CUDA applications without the requirement of CUDA code revision. In fact, vmCUDA does not even require that the application source code be available.

The execution times were measured from the start point to end point of CUDA programs at the client VM which reflects the real elapsed times. Programs were executed multiple times and their average execution time was computed.

Table 1: CUDA Benchmarks

Benchmark	Input Size	Output Size	Data Transfer Size
Binomial Option (BO)	80KB	16KB	96KB
Black Scholes (BS)	1144 MB	763MB	1907MB
Matrix Multiplication (MM)	128MB	64MB	196MB
Stereo Disparity (SD)	2496 MB	832MB	3328MB
Radix Sort (RS)	128MB	128MB	256MB

4.2. Overall Performance Evaluation

We studied the overhead of vmCUDA by comparing the execution time of CUDA programs in a single client VM using vmCUDA with the execution time whilst running natively. The measured data is shown in Table 2 and Figure 4.

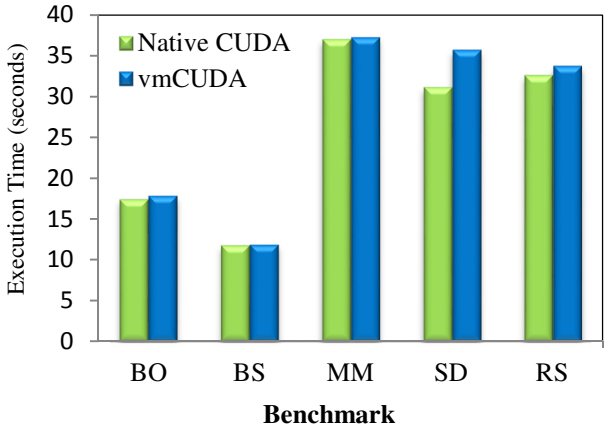

Figure 4. Performance of vmCUDA vs. native CUDA

In Table 2, we present the overhead of using vmCUDA compared to the native CUDA execution time. *Total Overhead* presents the percentage of time difference between native CUDA and vmCUDA; *vmCUDA Overhead* reflects the overhead created by vmCUDA itself. These results show that vmCUDA performance

is almost as good as the native one. Its overhead is in the range of 0.6% - 3.5% in most test cases. These results are much better than those from similar approaches. For example, with Black Scholes (BS), rCUDA[9] introduced an overhead of 60.8% on KVM and 200.6% on VirtualBox; the overhead of GViM [12] was 25% and the overhead of vCUDA [14] was 73.5% for the same benchmark on Xen platform.

Stereo Disparity (SD) is the only case whose overhead (14.7%) is quite large. In a virtualized environment, the overhead of an application running on VM is generally the combination of CPU, memory, network and disk I/O overheads. Stereo Disparity was different from other benchmarks because it processed larger amounts of data which resulted in larger network overhead due to the data communication between the client VM and the appliance VM. In addition, the input data of Stereo Disparity was read from files which introduced a considerable disk I/O overhead while the other benchmarks did not suffer from this type of overhead because their input data were initialized at runtime. Because this disk I/O overhead is not a part of the vmCUDA, the overhead of Stereo Disparity caused by vmCUDA itself in fact is 9.5% (Table 2).

Table 2. The execution time (seconds) and overhead of vmCUDA compared to the execution time of native CUDA

Bench mark	Native CUDA	vmCUDA	vmCUDA Overhead	Total Overhead
BO	17.3s	17.8s	2.1%	2.5%
BS	11.7s	11.8s	0.5%	0.6%
MM	36.9s	37.1s	0.6%	0.7%
SD	31.0s	35.6s	9.5%	14.7%
RS	32.5s	33.6s	2.9%	3.5%

4.3. Performance for Very Large Data

In order to analyze the performance of vmCUDA in case of very large datasets, we tested the Stereo Disparity benchmark with 100 iterations and many sets of image input files whose size varied from 64MB to 1248MB. In this experiment, the total amount of data transferred between the frontend and backend modules of vmCUDA ranges from 12.5GB to 237.5GB which can show the performance of vmCUDA in the worst cases. We present the results of this experiment in Figure 5. We found that the execution times of Stereo Disparity using vmCUDA and its native execution times increase linearly when data transfer size increases.

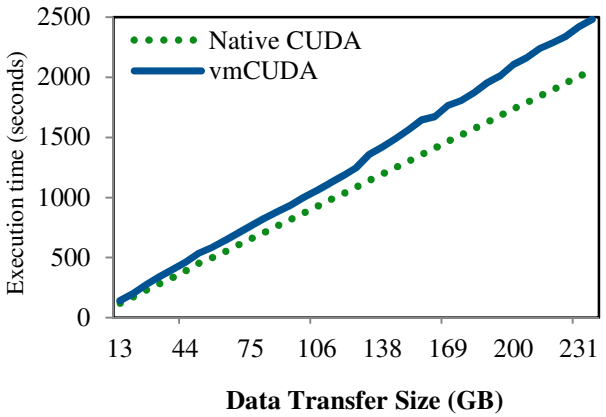

Figure 5. Execution time of Stereo Disparity with varying data size

The overhead of Stereo Disparity compared to its native execution time ranges from 14% to 20% in which the overhead caused by vmCUDA itself ranges from 9% - 14%. This results show the potential of our GPGPU solution for CUDA applications with large data sets. Most related research rCUDA[9], GViM[12], vCUDA[14] only has published data from experiments with much smaller data test sizes which made it difficult to predict the performance behavior of those implementations in case of large data sets.

4.4. Analyzing Optimization Techniques

The majority of vmCUDA overhead comes from the data communication between the two modules of vmCUDA running on client VM and appliance VM. Therefore, increasing this data transfer speed plays an important role in minimizing vmCUDA overhead. In this section, we analyze the techniques that we applied to achieve the performance presented in Section 4.2 and Section 4.3. These techniques helped to bring the data bandwidth of CUDA memory copy APIs using vmCUDA up to 6GB/s on ESX platform compared to the 126MB/s data bandwidth of rCUDA [9] or 3.1GB/s of GViM [12].

4.4.1. Using the virtual network on ESX

We found that the virtual network of client VMs and appliance VM inside the ESX server can have a huge impact on data communication performance. There are two types of virtual network supported by ESX that we can utilize for vmCUDA. The first one is a public network using a physical network adapter. The benefit of this network type is allowing CUDA applications to access GPU resources on remote ESX servers using vmCUDA. The second type is a private network which does not use the physical network device. Applying this private network usually provides better performance; it also avoids the overhead of sharing physical network devices with other network applications. In our experiments, we chose a private virtual network for our benchmarks because client VMs are co-located with appliance VM on the same host. Additionally, the use of a newer virtual network driver VMXNET3 in VMware Toolkit helped reduce network latency among the VMs on ESX.

4.4.2. Using Jumbo Frame for Network Optimization

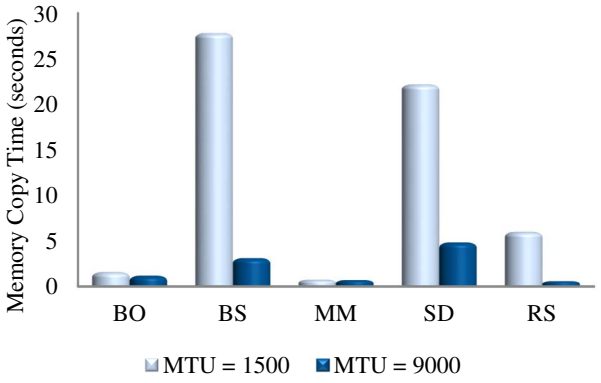

Figure 6. Memory copy time of vmCUDA with different MTUs

Using a large maximum transmission units (MTU) size (also known as Jumbo Frame) is a popular technique of network optimization [21]. In high performance computing environment, increasing the MTU size can provide significant performance

gain. The challenge is the fact that many network devices do not support large MTU size. However, on ESX hypervisor, jumbo frame can be set easily using the virtual private network. We used Jumbo frames in our experiments. Figure 6 presents the memory copy time (e.g. cudaMemcpy, cudaMemcpyToSymbol, etc.) of the benchmarks in Table 1 measured at the client VM with MTU= 1500 and MTU=9000. These results show that data transferring time between CUDA appliancations in client VM and GPU in appliance VM reduced sharply. The performance of data communication improved up to 939% (Radix sort) or 797% (Black Scholes).

4.4.3. Data Blocking Movement

We applied the data blocking model presented in Section 3.2 for CUDA memory copy APIs like cudaMemcpy, cudaMemcpyToSymbol, etc. Using this model, the original memory copy tasks are divided into smaller units where each handles the copying of one data block. This work is performed by the vmCUDA interposer library and no modification of the original CUDA application is required. In other to identify good block sizes, we studied the performance of these memory copy APIs with different block sizes from 128KB to 128MB. For each block size, we ran a same benchmark including many memory copy calls whose data size ranges from 32MB to 624MB and total data transfer size is ~50GB. The total memory copy times of this experiment are presented in Figure 7. It can be seen from this chart that a reduction in the data communication cost in the range 30% - 110% can be obtained by choosing a suitable block size.

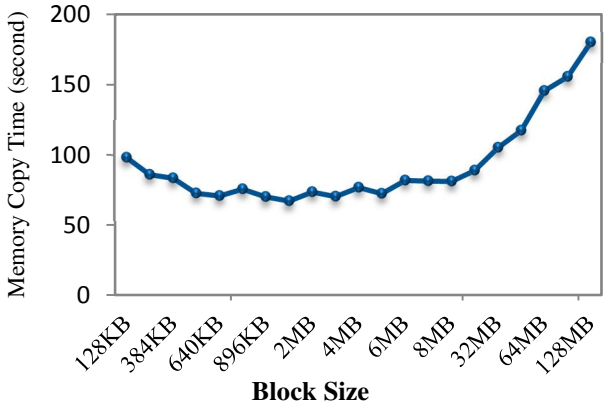

Figure 7. Memory copy time of vmCUDA with different block size

For small block sizes, the cost of conducting too many smaller memory copy calls adds to the overhead of total communication time. In contrast, large data blocks require more memory for data buffers which is a critical issue when many CUDA applications are launched at the same time. In our experiments, we chose a block size of 4MB for the data block communication. This size also provides better cache optimization because the data buffer can fit in the cache of most modern processors.

4.5. Multiple Virtual Machines Sharing a GPU

In other to demonstrate the ability to share GPU resource among VMs, an important feature of vmCUDA, we ran each CUDA program in Table 1 concurrently on multiple VMs. The average execution time of these programs with up to six concurrent VMs is shown in Figure 8.

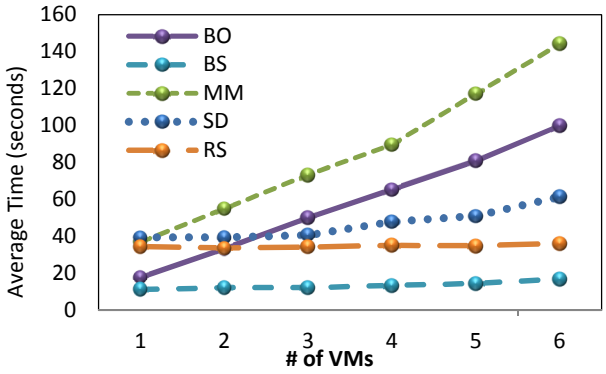

Figure 8. Performance of CUDA benchmarks running concurrently on multiple VMs on ESXi 5.0

The increase in execution time of CUDA benchmarks when the number of VMs increases is because VMs share a single Nvidia Quadro 4000 GPU. The benchmarks, Binomial Options (BO) and Matrix Multiplication (MM) show the sharpest increase in total execution time with number of VMs. The total execution times of Black Scholes (BS), Stereo Disparity (SD) and Radix Sort (RS) on multiple concurrent VMs are just slightly higher than that on a single VM. For better understanding of this difference, we measured the CPU time, GPU time, memory copy time, and CPU usage of each CUDA program. We present the time distribution in Figure 9 and the average CPU usage in Figure 10.

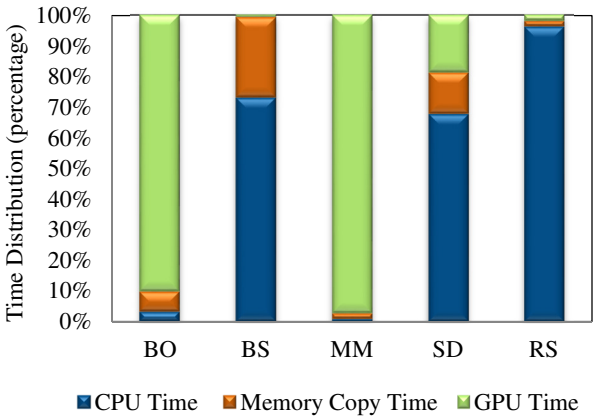

Figure 9. Time distribution of CUDA benchmark

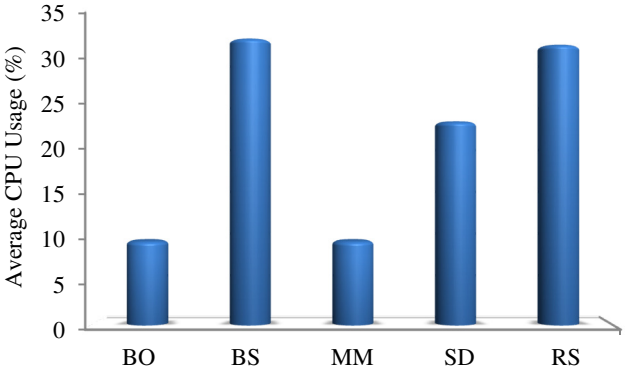

Figure 10. Average CPU usage of CUDA benchmark

It shows that most of the execution time in the benchmarks, BO and MM is spent in the GPU. Since the GPU is shared it causes the execution time of these two benchmarks to increase almost linearly with the number of VMs. In contrast, BS, SD and RS have CPU bound execution time. Since the CPU computation of each client VM is handled by its own VCPU, the execution time is not changed much when more VMs are added.

5. RELATED WORK

There are several research projects in progress to support CUDA in a virtualized or remote environment. The majorities of these efforts [9-14] use a split-driver model, but vary in the degree of integration into the hypervisor. We believe that our "remote API execution" approach is superior to the split-driver approach for three reasons; first, it eliminates the requirement of asking for each and every GPU vendor to provide customized drivers ported for the hypervisor and avoids its related problems like the time, performance and stability. Second, it keeps CUDA support more independent of the hypervisor by avoiding changes to the hypervisor. Finally, this solution also avoids some of the problems associated with tying GPGPUs to a single hypervisor or control domain and helps easily scaling HPC systems.

The approach taken by Gupta et al [11-13], Shi et al. [14] use the split-driver approach with the backend driver embedded in the hypervisor/host OS. In some cases, standard TCP sockets are utilized to connect the frontend and backend drivers, whereas, in others, there is deep integration with the hypervisor, exposing pinned pages to guest VMs in an effort to reduce data transfer overheads [9-14]. Further, there has been significant focus on the batching of operations wherever possible to reduce the overheads associated with communicating requests to remote GPUs. Our approach is distinct from these implementations as we leverage VMware's DirectPath I/O technology to directly expose one or more GPUs to an appliance VM, creating a more flexible solution.

The model (called rCUDA) used by Duato et al [9, 10] is closest to that proposed by us but it appears to be implemented in a hosted environment. Second, to the best of our knowledge, rCUDA [9] modifies the source either manually or using a custom compiler component. Our approach does not require any acces to the source for the CUDA application. vCUDA by Shi et al. [14] implements a subset of the CUDA Runtime version 1.1 and maintains a significant amount of state information in the client VM. As evidenced by our approach, this represents a significant and unnecessary overhead with no tangible benefit. Further, it will complicate efforts to keep pace with changes in the CUDA specifications in different versions. The GViM [12] project uses a split-driver with the back-end located in "Dom0" in Xenserver. Significant effort has been made in GViM to reduce the number of data copies and to achieve fair scheduling for multi-client-VM environments, leading to modest virtualization overheads.

Performance data available from [14] indicate that the expected overhead for using a split-driver approach to remote CUDA, as opposed to running the application natively can be as high as 50%. This large overhead can be readily attributed to the fact that the implementations are not fully optimized and use expensive, but generic, communication protocols with a desire for portability across platforms.

Finally, Nvidia recently introduced Nvidia Grid that provides both a software hypervisor and hardware enhancements to the GPU to support virtualization [22]. The VGX Hypervisor allows VMs to interact directly with a GPU and manages the GPU resources such that multiple users can share common hardware.

This direct interaction can be extremely compelling for graphics operations, where VMs are offloading tens of latency sensitive operations per second. However, for the compute operations, where the typical kernel duration is often measured in seconds, the overheads associated with transmitting compute requests to the control domain or hypervisor can more be effectively amortized over the duration of the compute kernel. That said, as more details of these hardware enhancements emerge, we plan to investigate their use in our appliance.

6. CONCLUSION AND FUTURE WORK

In this paper we present vmCUDA, an efficient solution to support high performance GPU computing in the ESX virtualization environment. vmCUDA uses a "remote API execution" model. This model has several advantages over a split-driver approach; first, it avoids the need to build a GPU driver in the hypervisor. Second, it avoids adding to the hypervisor footprint and attack surface. Third, scheduling, scaling and robustness issues can be handled from within a Linux/Windows appliance VM allowing us to leverage off-the shelf solutions implemented in these domains. vmCUDA utilizes VMware's DirectPath I/O technology to gain direct, high performance access to the physical GPUs. It applies different techniques to minimize the overhead of executing CUDA applications in a virtualized environment. The performance evaluation shows that vmCUDA introduced overhead in the range of 0.6% - 3.5% for CUDA applications with average data input size and 14% - 20% for those with very large data sets (e.g. 12.5GB - 237.5GB in our experiments). These results are significantly better than related work reported for other virtualization platforms like Xen or KVM.

In future work, we plan to add advanced scheduling capabilities to vmCUDA and investigate mechanisms to move an application from one GPU to another mid-execution to achieve dynamic load balancing. We would also like to integrate vmCUDA with VMware's existing support for shared graphics in ESX.

7. ACKNOWLEDGEMENT

The authors would like to thank Lawrence Spracklen for many useful and productive discussions during the early stages of this project. We acknowledge his extraordinary early efforts, enthusiasm, and leadership that were primarily responsible for initiating and launching this project. We also would like to thank Vikram Makhija, Razvan Cheversan, Tariq Magdon-Ismail for their discussions, suggestions and invaluable help to this project.

References

[1] Morgan, T., "Top 500 supers – The Dawning of the GPUs," http://www.theregister.co.uk, 31st May 2010.

[2] Hou, R., Jiang, T., Zhang, L., Qi, P., Dong, J., Wang, H., Gu, X., Zhang, S.. "Cost effective data center servers," In *the Proc. of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2013, pp. 179-187.

[3] Nvidia CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/index.html

[4] Munshi, A., "OpenCL 1.0 Specification," *Khronos OpenCL Working Group*, 2008.

[5] Mergen, M. F., Uhlig, V., Krieger, O., Xenidis, J., "Virtualization for high-performance computing," in *ACM SIGOPS Operating Systems Review Newsletter*, Volume 40 Issue 2, April 2006, New York, NY, pp. 8 - 11, .

[6] Younge , A.J., Henschel, R., Brown, J.T., Laszewski, G., Qiu, J., Fox, G.C., "Analysis of Virtualization Technologies for High Performance Computing Environments", in *the Proceeding 2011 IEEE International Conference on Cloud Computing (CLOUD)*, 4-9 July 2011, Washington, DC, pp. 9-16.

[7] Rosenblum, M., "VMware's Virtual Platform: A virtual machine monitor for commodity PCs," in *Proceeding of Hot Chips 11: Stanford University*, August 15–17 , 1999, Stanford, CA.

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A, "Xen and the Art of Virtualization," In *Proc. 19th ACM Symposium on OperatingSystems Principles (SOSP)*, Oct. 2003, Bolton Landing, NY, pp. 164-177.

[9] Dato, J., Peña, A. J., Silla, F., Mayo, R. & Quintana-Ort, E. S., "Enabling CUDA acceleration within virtual machines using rCUDA", in *the Proceedings of HiPC 2011*.

[10] Duato, J., Peña, A. J., Silla, F., Mayo, R. & Quintana-Orti, E. S, "Performance of CUDA Virtualized Remote GPUs in High Performance Clusters", in *the Proceedings of 2011 International Conference on Parallel Processing (ICPP)*, pp. 365-374.

[11] Gupta, V., Schwan, K., Tolia,N., Talwar, V., and Ranganathan, P., "Pegasus: Coordinated Scheduling for Virtualized Accelerator-based systems" , in *the Proceedings of USENIX ATC* 2011.

[12] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P., "GViM: GPU-accelerated virtual machines", in *Proceedings of the 3rd Workshop on System-level Virtualization for High Performance Computing*, NY, USA: ACM, 2009, pp. 17–24.

[13] Merritt, A., Gupta, V., Verma, A., Gavrilovska, A., and Schwan, K., "Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies", in *the Proceedings of VTDC* 2011.

[14] Shi, L., Chen, H., Sun, J., "vCUDA: GPU accelerated high performance computing in virtual machines," in *Proceedings of IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, 2009.

[15] Nvidia GPU Computing SDK, https://developer.nvidia.com/gpu-computing-sdk

[16] Reano, C., Pea, A.J., Silla, F., Duato, J.; Mayo, R., Quintana-Orti, E.S., "CU2rCU - towards the Complete rCUDA Remote GPU Virtualization and Sharing Solution," in the *Proc. of the 2012 19th International Conference on High Performance Computing (HiPC)*, Dec. 2012, pp. 1 - 10.

[17] Adams, K., Agesen, O., "A comparison of software and hardware techniques for x86 virtualization," in *Operating Systems Review*, 40(5):2–13, Dec. 2006.

[18] Huang, W., Liu, J., Abali, B., D. K. Panda, D.K., Muraoka, Y. "A case for high performance computing with virtual machines", in the *Proceedings of 20th Annual International Conference on Supercomputing*, G. K. Egan, Ed., Cairns, Queensland, Australia, Jun. 2006, pp. 125–134.

[19] VMware vSphere vMotion Architecture, Performance and Best Practices in VMware vSphere 5, http://www.vmware.com/files/pdf/vmotion-perf-vsphere5.pdf

[20] Dowty M., Sugerman, J., "GPU virtualization on VMware's hosted I/O architecture," in *Newsletter of ACM SIGOPS Operating Systems Review archive,* Volume 43 Issue 3, July 2009, New York, NY, pp. 73-82.

[21] Ciliendo, E., Kunimasa, T., "Linux Performance and Tuning Guidelines," in *IBM Redbooks*, 05 July 2007.

[22] Nvidia Grid, http://www.nvidia.com/object/cloud-gaming.html