vCUDA: GPU Accelerated High Performance Computing in Virtual Machines

Lin Shi, Hao Chen and Jianhua Sun
Advanced Internet and Media Lab
School of Computer and Communications
Hunan University, Chang Sha, 410082, China
{linshi,haochen,jhsun}@aimlab.org

Abstract

This paper describes vCUDA, a GPGPU (General Purpose Graphics Processing Unit) computing solution for virtual machines. vCUDA allows applications executing within virtual machines (VMs) to leverage hardware acceleration, which can be beneficial to the performance of a class of high performance computing (HPC) applications. The key idea in our design is: API call interception and redirection. With API interception and redirection, applications in VMs can access graphics hardware device and achieve high performance computing in a transparent way. We carry out detailed analysis on the performance and overhead of our framework. Our evaluation shows that GPU acceleration for HPC applications in VMs is feasible and competitive with those running in a native, non-virtualized environment. Furthermore, our evaluation also identifies the main cause of overhead in our current framework, and we give some suggestions for future improvement.

1. Introduction

Recently, system level virtualization technology revivals again, which stems from the continued growth in hardware performance and increased demand for service consolidation from business markets. Virtual machine (VM) technologies allow different guest VMs coexisting in a physical machine under the management of a virtual machine monitor (VMM). VMM technology has been applied to many areas including intrusion detection [8], high performance computing [12] and device driver reuse [21] et al.

Over the past few years, there has been a marked increase in the performance and capabilities of graphics processing unit (GPU). The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth [10, 28, 29]. Research community has successfully mapped a broad range of computationally demanding and complex problems to GPU. The introduction of some vendor specific technologies (such as NVIDIA's CUDA [7]) are further accelerating the adoption of high performance parallel computing to commodity computers.

Although virtualization technologies provide a wide range of benefits such as system security, ease of management, isolation and live migration, VM technologies have not been widely adopted in high performance computing area. This is mainly due to the overhead incurred by indirect access to physical resources such as CPU, IO devices and physical memory, which is one of the fundamental characteristics of virtual machines.

In this paper, we propose a framework vCUDA for HPC which uses hardware acceleration provided by GPUs to address the performance issues associated with VMs. Due to the closure and diversity, the powerful graphic processing ability can not be directly used by application running in virtualization platforms. To achieve hardware acceleration non-invasively for general purpose computing applications in VMs, we propose a solution by intercepting CUDA API calls, which intercepts and redirects CUDA commands and data in VMs to a CUDA enabled graphics device, and does the real computations by the vendor-supplied GPU driver and CUDA library in VMM. With detailed performance evaluations, we demonstrate that hardware accelerated high performance computing jobs can run as efficiently in virtualized environment as in a native host. Although we focus on CUDA and Xen, we believe that our framework can be readily extended for other vendor specific GPGPU solutions and other VMMs. To the best of our knowledge, this is the first study to adopt GPU accelerated HPC in virtual machines.

In summary, the main contributions of our work are:

- We propose a framework which allows high performance computing applications to benefit from hardware acceleration in virtual machines. To demonstrate the framework, we have developed a prototype system using Xen virtual machine and CUDA.

- we present a set of extensions built with this framework such as multiplexing, suspend and resume without any modifications to applications.
- We carry out detailed performance evaluation on the overhead of our framework. This evaluation shows that the vCUDA framework is practical and can deliver high performance for HPC applications as those in native environments.

The rest of the paper is organized as follows: In Section 2, we provide some necessary background about this work. Then, we present our framework for hardware accelerated high performance computing in VMs in Section 3 and carry out detailed performance analysis in Section 4. In Section 5, we discuss several issues in our current implementation and how they can be addressed in future. we discuss the related work in Section 6 and conclude the paper in Section 7.

2. Background

2.1. VMM and GPU Virtualization

System level virtualization technologies simulate details of the underlying hardware in software, provide different hardware abstraction for each operating system instance, and run multiple heterogeneous operating systems concurrently. It decouples the software from the hardware by forming a level of indirection which traditionally known as the VMM. There are some different forms of VMM, but they all provide a complete and consistent view of underlying hardware to the VM running on it.

The VMM provides total mediation of all interactions between the virtual machine and underlying hardware, thus allowing strong isolation between virtual machines and supporting the multiplexing of many virtual machines on a single hardware platform. VMM layer can also map and remap virtual machines to available hardware resources at will and even migrate virtual machines across machines. Encapsulation also means that administrators can suspend virtual machines and resume them at arbitrary times or checkpoint them and roll them back to a previous execution state. With this general-purpose undo capability, systems can easily recover from crashes or configuration errors.

In the field of commercial software, Vmware [31] has become the de facto industry standard, and in the field of open-source software, Xen [2, 3] leads the emergence of paravirtualize technology. Both Xen and Vmware developed the hosted architecture. In this architecture, the virtualization layer uses the device drivers of a host operating system such as Windows or Linux to access devices.

While virtualization technology has been successfully applied to a variety of devices, it is difficult to virtualize the GPU in VMM, one main reason is lacking standard interface in hardware level. One possibility is to redirect graphics processing requests from guestOS to hostOS using software emulation. However, this is not practical for modem graphics hardware because trapping requests at this level is generally too inefficient. Another choice is to intercept the graphics protocol stream at a higher (device-independent) point in the stream and redirect that to hostOS. The approach might be to replace system dynamic link libraries containing APIs with protocol stubs to redirect to the guestOS, though there are still significant issues with this approach when the methods for managing state in the original APIs have not been designed with this approach in mind.

In the rest of this paper, the term hostOS refers to the administrative OS (or domain0 according to Xen's semantics). The term guestOS refers to a VM (or domainU in Xen).

2.2. CUDA

CUDA (Compute Unified Device Architecture) [7] is a complete GPGPU solution that provides direct access to the hardware interface, rather than the traditional approach that must rely on the graphical interface API. The CUDA framework uses common C language as its programming language and provides a large number of high performance computing instructions and development capabilities, so that developers can establish more efficient data-intensive computing solutions on the basis of powerful GPU acceleration ability.

The CUDA software stack is composed of three layers: a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage. The runtime library is split into three parts: A host component that runs on the host and provides functions to control and access one or more compute devices from the host; A device component that runs on the device and provides device-specific functions; A common component that provides built-in vector types and a subset of the C standard library that are supported in both host and device code. For host runtime component, it is composed of two APIs: A low-level API called the CUDA driver API, and A higher-level API called the CUDA runtime API, which is implemented on top of the CUDA driver API. A important fact is they are mutually exclusive which means one application can only use one of them. NVCC is a compiler for CUDA, it simplifies the process of compiling CUDA code. Its basic work-flow consists in separating device code from host code and compiling the device code into a binary form or *cubin* object.

For the programmer the CUDA execution model as shown in Figure 1 is a collection of threads running in parallel. The programmer decides the number of threads to

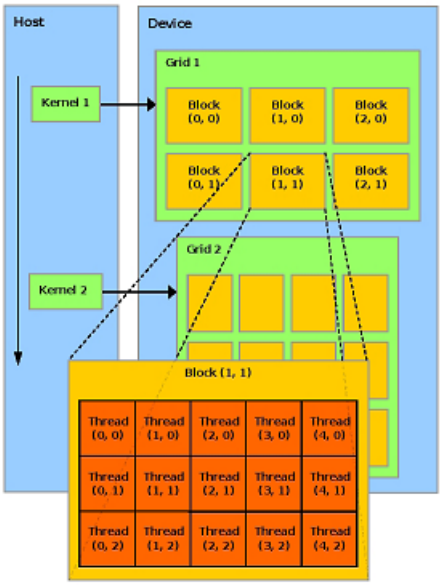

Figure 1. The CUDA execution model: A number of threads are batched together in blocks, which are again batched together in a grid.

be executed. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A single execution on a device generates a number of blocks. A collection of all blocks in a single execution is called a grid. All threads of all blocks executing on a single multi-processor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the kernel. The kernel is the core code to be executed on each thread. Using the thread and block IDs each thread can perform the kernel task on different set of data. Since the device memory is available to all the threads, it can access any memory location.

3. vCUDA

vCUDA framework is organized around two main architectural features:

- **Virtualization of CUDA API**. 31 APIs in total 56 runtime APIs were encapsulated into RPC calls. Their parameters were properly queued and redirected to hostOS, and variables was cached and kept persistence in both server and client side. Through this kind of virtualization, the graphics hardware interface was decoupled from software layer.

- **Lazy RPC Transmission**. vCUDA used XML-RPC [32] as the means of high-level communication between guestOS and hostOS, taking into account its compatibility and portability. XML-RPC is well-supported by a number of third-party libraries that provide language-agnostic ways of invoking RPC calls. In addition, we adopted a lazy RPC mode to improve efficiency of the original XML-RPC.

In addition, CUDA currently is not a fully open API, some internal details have not been documented in official software development kit SDK). we do not have full knowledge about the states maintained only by the underlying hardware driver or shared between applications and hardware. We achieve the virtualization functionality from the following three aspects:

- **Function parameters**. The intercepting library has no access to all the internals of an application linked with it. But it can get all the parameters of corresponding API calls, which can be used as inputs to the faked API calls defined in the intercepting library. These faked API calls with proper parameters can then be sent to remote server for execution as normal calls.

- **Ordering semantics**. Ordering semantics are the set of rules that constrain the order in which API calls may be executed. CUDA is basically a strictly ordered interface, which means some APIs must be launched in the order in which they are specified. This behavior is essential for maintaining internal persistency. But in some cases, if possible, vCUDA would use less constrained ordering semantics when it meant increased performance.

- **Device state**. CUDA maintains a large amount of states in hardware. The states contain attributes such as device point, symbol, pitch, texture and so on. On a workstation with hardware acceleration, the graphics hardware keeps track of most or all of the current state. However, in order to properly implement a remote execution for these APIs in virtual machine, it is necessary for the client to keep track of some of the state in software.

3.1. System Architecture

vCUDA uses a robust client-server model, which consists of three user space modules: the vCUDA library, virtual GPU in client and the vCUDA stub in server. Figure 2 shows the vCUDA architecture. In the rest of this paper, the term server memory refers to the hostOS memory space, the term client memory refers to the VM memory space,

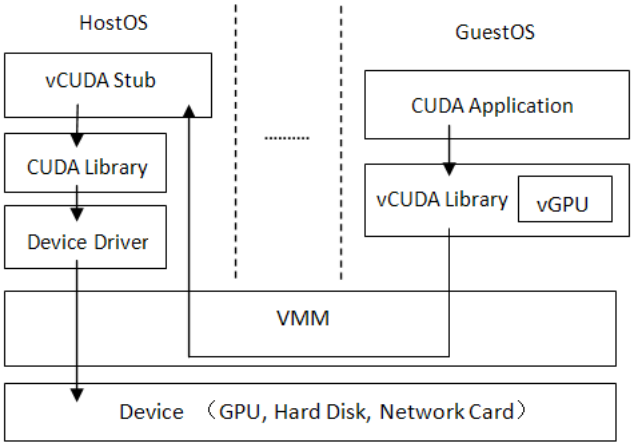

Figure 2. The vCUDA architecture.

and device memory refers to the memory space in graphics hardware reside in hostOS.

3.1.1. vCUDA Library

vCUDA library resides in guestOS as a substitute of standard CUDA runtime library, and it is responsible for intercepting and redirecting API calls from client to stub. There are two programming interface provided by CUDA, runtime API and driver API. We chose the runtime API as the target of virtualization because it is the most widely used library in practice and also the officially recommended interface for programmers. However we don't anticipate any main obstacle to virtualize the driver level API.

There are total 56 runtime APIs described in the official programming guide. But we also found other 6 internal APIs in the dynamic linking library of CUDA runtime, and they are not visible to programmers. It seems that they are used to manage device execution code and memory allocated to variables. These six internal APIs are compiled by NVCC into the final executable file, and never interact with other APIs. They were called before the choice of device therefore have nothing to do with specific GPU. For the six APIs, we just wrapped the corresponding parameter and sent it to the stub. We do not assume any internal logic to these functions because they might be changed in the future.

We use NVCC generated intermediate code, combined with control flow analysis to customize the virtual logic for each API. For each API we intercept in the faked library, all API calls are packed into a global API queue. This queue contains a copy of the arguments to the corresponding function as well as an opcode, which is encoded into a single byte. The contents in this queue were pushed to the stub periodically according to some pre-defined strategies (Section 3.2). In the current stage, we do not support the virtualization of 3D graphics APIs, which requires a large amount of engineering effort. Thus the discussion about these APIs is beyond the scope of this paper (Section 5).

3.1.2. vGPU

vGPU is created, identified and used by vCUDA library, and in fact it is represented as a large data structure in memory maintained by vCUDA library. vGPU provides three main functionalities. First, vGPU abstracts some features of real GPU to give each application a complete view of the underlying hardware. vCUDA library creates a virtual GPU context for each application, which contains device attributes such as GPU memory usage, texture memory properties. Another important role of vGPU is local device memory management. When a CUDA application allocates a device memory, vGPU will return a local virtual address to the application and notify remote stub to allocate the real device memory. vGPU is also responsible for maintaining the mappings of local and remote addresses to avoid unnecessary memory copies and leaks. The third function of vGPU is to store the CUDA API flow, most of the APIs' opcodes and parameters are stored in a global queue in memory or a file in file system to support suspend/resume as described in Section 3.4.

3.1.3. vCUDA Stub

vCUDA stub receives and interprets remote requests and creates a corresponding execution context for the API calls from guestOS, then returns the results to guestOS. The vCUDA stub manages the actual physical resources such as the allocation of hardware resources and threads, the matching of parameters of API calls, and also keeps a consistent view of states in both the stub and client sides by periodical synchronization with vGPU.

The vCUDA stub spawns one thread for each client. The main purpose of these threads is to receive CUDA commands via the RPC channel, and to execute those commands on behalf of the client application. Each stub thread receives the vCUDA API stream, decodes it, and translates it into a server-side representation. Then, for each client that vCUDA library interprets, the stub thread sets up an appropriate execution environment, and finally calls the native APIs. The process of this translation is crucial to the coherent and success of our system.

3.2. Lazy RPC

Thousands of CUDA APIs could be involved in a CUDA application. If vCUDA intercepts and redirects every API call in client applications, then the same number of RPCs

will be invoked and the overhead of world switch (execution context switch between different guest OSes) will be inevitably introduced into the system. In virtual machines, world switch is an extremely expensive operation and should be avoided whenever possible [23].

We classify the CUDA APIs into two categories. One is called instant APIs, whose executions have immediate effect on the state of CUDA runtime system. The other category is lazy APIs, which do not have any side effects on the runtime state until the invocation of a following instant API. This kind of classification allows vCUDA to reduce the frequency of world switch by updating states in the stub lazily, and decrease unnecessary RPCs by redirecting lazy APIs to stub side in a batched manner, thus boosting the system's performance. A potential problem with lazy mode needs mentioning is the delay of error report and time counting, thus it is not suitable for debugging and measurement purpose.

3.3. Multiplexing

One fundamental feature of VM technologies is device multiplexing. For example, in Xen, the physical network card can be multiplexed among multiple concurrently executing guest OSes. To enable this multiplexing, the privileged driver domain (domain0) and the unprivileged guest domains (domainU) communicate by means of a split network-driver architecture. The driver domain hosts the *backend* of the split network driver, and the domainU hosts the *frontend*.

In vCUDA, we implement the GPU multiplexing in application level through the cooperation of the vCUDA library in domainU and the stub in domain0, thus allowing multiple CUDA applications to execute concurrently in the same VM or different VMs. As described in Section 3.1.3, the vCUDA stub spawns one thread for each client, and there is an indicator (hash value of IP address, domain ID and process ID) for each thread to distinguish different clients from different VMs. Under the coordination of vCUDA stub, these threads allocate and manage hardware resources cooperatively to guarantee the correct execution semantics of client applications.

There are three situations when we handle the mappings of threads to GPU hardware. The first is one thread mapped to a single hardware device. The second is multiple threads running on a single device, and the last is that one thread controls multiple GPU devices. According to the Nvidia's official guide, the third case is not supported by the current GPU hardware. Although being not a officially recommended operation, the second case was also implemented in our framework and some performance evaluation will be given in Section 4.2.

3.4. Suspend and Resume

vCUDA provides support for suspend and resume, enabling client sessions to be interrupted or moved between computers [18]. Upon resume, vCUDA presents the same device state that the application observed before suspending while retaining hardware acceleration capabilities.

The basics to implement application suspend and resume is to store the CUDA API calls which affect device states and the coresponding states bound to these calls when necessary. While the guest is running, vCUDA stub and vGPU both snoop on the CUDA commands they forward to keep track of the state of device. Upon resume, vCUDA spawns a new thread in the stub, which is initialized by synchronizing it with the application vCUDA state stored by vCUDA. The time spent on resume depends on RPC efficiency, the GPU computing time of specific application and world switch overhead.

4. Experiments

While the previous sections have presented detailed technical descriptions of the vCUDA system, this section evaluates the efficiency of vCUDA using programs selected from official SDK examples: a set of general-purpose algorithms from various domains. The benchmark range from simple data management to more complex WalshTransform computation and MonteCarlo simulation. Table 1 shows the statistical characteristics of these benchmarks, such as the quantity of API calls, the device memory size they consume and the data volume transferred from or to GPU device.

These applications are evaluated concerning the following criteria:

- **Performance** How close does vCUDA come to providing the performance observed in unvirtualized environment with GPU acceleration?
- **Lazy RPC and Concurrency** How greatly can vCUDA reduce the frequency of network transmission by Lazy RPC mechanism? How well does vCUDA scale to support multiple CUDA applications running concurrently?
- **Suspend and Resume** What is the latency for resuming a suspended CUDA application? What is the size of an application's recorded CUDA state?
- **Compatibility** How compatible is vCUDA with a wide range of applications besides the examples distributed with CUDA SDK?

The following testbed has been used for all benchmarks: A personal computer equipped with one Intel Core

Table 1. Statistics of Benchmark Applications.

	Number of APIs	GPU RAM	Data Volume
AlignedTypes (AT)	1990	94.00MB	611.00MB
BinomialOptions (BO)	31	0.01MB	0.01MB
BlackScholes (BS)	5143	61.03MB	76.29MB
ConvolutionSeparable (CS)	48	108.00MB	72.00MB
FastWalshTransform (FWT)	144	128.00MB	128.00MB
MersenneTwister (MT)	24	91.56MB	91.56MB
MonteCarlo (MC)	53	187.13MB	0.00MB
ScanLargeArray (SLA)	6890	7.64MB	11.44MB

2 Duo E6550 processors running at 2.33 GHz with two single-threaded cores and provided with 2 GBytes of memory. Furthermore, the graphics hardware was NVIDIA's GeForce8600 GT. As for software, the test machine ran the RHEL 5.0 Linux distribution with the 2.6.16.29 kernel, with the official NVIDIA driver for linux version 169.09. We choose the XEN3.0.3 as our virtual platform, all paravirtualized virtual machines were setup with 512 MB RAM, 5G disk, and bridge mode network configuration.

4.1. Performance

Performance evaluation refers to the execution time of benchmarks in virtual machine compared to the native version. Our first test measures basic performance of vCUDA, all benchmarks were evaluated in two different configurations:

- **Native**: Every application has direct and exclusive access to hardware and native CUDA drivers. vCUDA was not used. This represents the upper bound on achievable performance for our experimental setup.
- **vCUDA**: a virtualized guest using vCUDA to provide hardware acceleration. All CUDA instructions were intercepted and redirected to hostOS.

Figure 3 shows the results from running the benchmarks under two configurations described above. The first two bars shows the execution time with and without vCUDA, the third bar represents the overhead caused by XML-RPC encode/decode procedures.

The experiments result shows that the time consumption with vCUDA is one to five times bigger than the native version. But further observation will discover the main cause that affects the efficiency is the encode/decode time of XML-RPC implementation. Benchmark programm AlignedType involves data transfer about 600MBytes, whose encode/decode time took more than 60% of the total execution time. This indicates that the XML-RPC is not

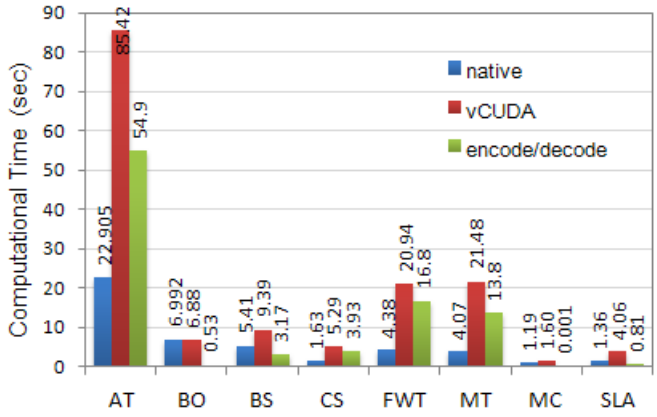

Figure 3. vCUDA performance.

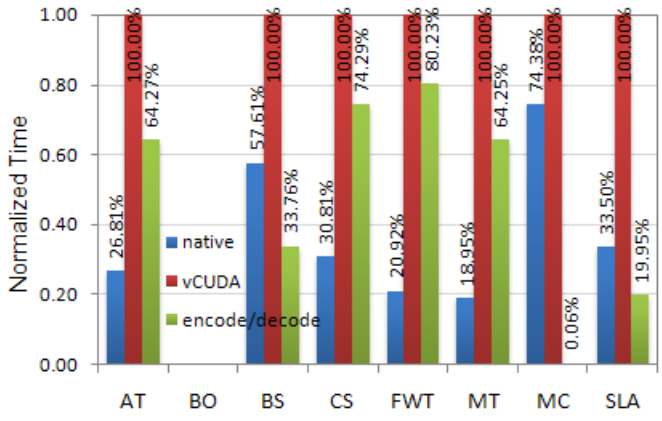

Figure 4. vCUDA performance - normalized view.

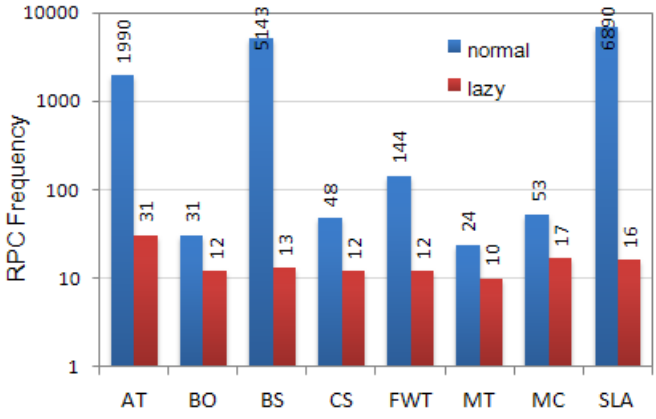

Figure 5. Effect of lazy RPC mode.

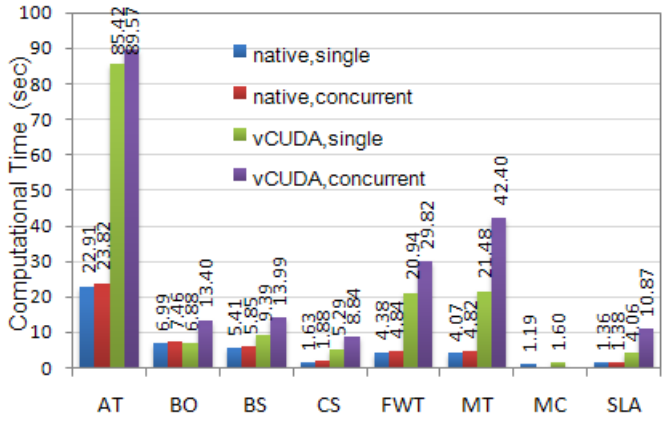

Figure 6. Evaluation of scalability by running two CUDA applications concurrently.

a efficient enough protocol for high-capacity data transmission. We are investigating a customized inter-domain communication mechanism in virtual machine to improve the efficiency. Other benchmarks that involve large data transfer exhibit similar characteristics. On the other hand, the less the data volume is transferred, the closer performance is to the native version, such as benchmarks BO and MC.

Figure 4 normalizes the native and encode/decode results against the results obtained in VM (i.e. native and encode/decode time divided by vCUDA time). The purpose of normalization is to compare the results more intuitively. For example, although the execution time in vCUDA is not exactly the sum of native and encode/decode time, we can infer a coarse performance penalty of vCUDA. In the case of benchmark AT, vCUDA itself incurred 8.92% (1-26.81%-64.27%) overhead except for encode/decode time.

4.2. Lazy RPC and Concurrency

Figure 5 compares the RPC frequencies in two cases leaving lazy mode open or close. As shown in Figure 5, the lazy mode transmission significantly reduce the frequency of RPC between two domains at the level of 40% to 70%.

To examine vCUDA's ability to support concurrent applications, we compared the performance of two applications executing concurrently in an unvirtualized configuration, to the performance of the same two applications executing concurrently in vCUDA. We launched each benchmark concurrently with a reference application to test the performance of concurrency in vCUDA. The benchmark BO was chosen as a reference application, because it consumed smaller device memory and can run concurrently with most of other benchmarks. The only exception was MC, which consumed too much memory in device RAM to execute concurrently with BO.

Figure 6 presents the results for the concurrent execution of two applications, compared to the results for a single instance (taken from Figure 3) in two circumstances. The test results in unvirtualized configuration all present good scalability, and the overheads for all applications are all below 16% (3.8% for AT, 6.3% for BO, 7.5% for BS, 13.3% for CS, 9.5% for FWT, 15.6% for MT and 1.4% for SLA). In the contrary, the counterparts in vCUDA show obvious performance degradation, the overhead ranging from 4% to 170% (4.9% for AT, 94.8% for BO, 49% for BS, 67.1% for CS, 42.4% for FWT, 99.9% for MT and 167.7% for SLA). We attribute this to the current unoptimized implementation of our system, such as the management of concurrent accesses to GPU device of different stub threads and inefficient inter-domain data transfer that also incurs significant overhead of world switch. Despite the performance issue, the concurrency evaluation validates the GPU multiplexing functionality described in Section 3.3.

4.3. Suspend and Resume

To measure the performance of vCUDA's suspend and resume, we suspended the benchmarks at the end of each application's API call flow in time. We then resumed the guest and verified successful resumption of the CUDA application. We measured the size of the CUDA state necessary to synchronize the vCUDA stub to the current application state, and the time it took to perform the entire resume operation. The results of these experiments are shown in Figure 8 and Figure 9.

Note that since *__unregistfatbinary* is always the last API called in CUDA applications, we put the suspending point before the first occurrence of *__unregistfatbinary*. That is the worst situation because the maximum states need to be

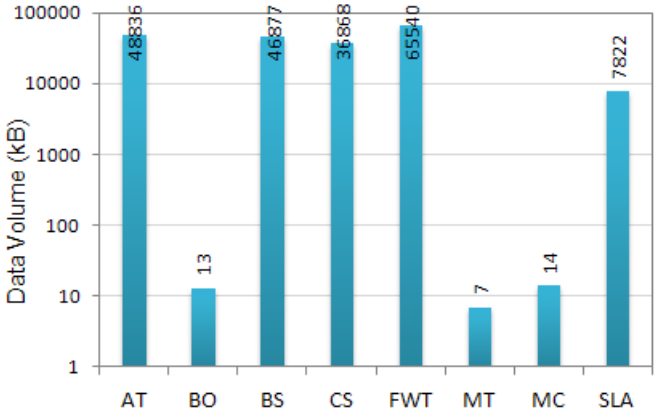

Figure 7. Size of suspended CUDA state.

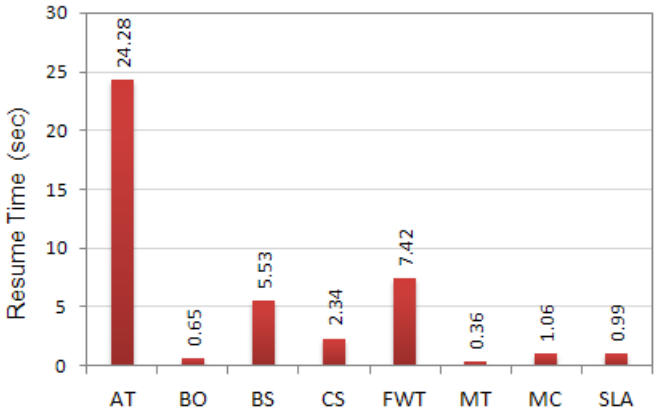

Figure 8. Resume time.

synchronized, thus the experimental results represent an upper bound of performance.

Figure 7 illustrates the data volume need to be restored for resumption. The resume time (Figure 8) is strongly dependent on the size of the suspended CUDA state (Figure 7), which can be as large as 65MB for the FWT benchmark. The benchmarks AT, BS, CS and FWT took more time to perform the resume operation than others due to their larger data volumes. Furthermore, Figure 9 compares the time consumption in three different circumstances (native, vCUDA and resume). The resume operation took much less time than the execution in vCUDA in all benchmarks, but took a little more time than the native execution in benchmarks AT, BS, CS and FWT, which all involved more data transfer. However, the resume time is almost negligible in benchmarks BO (0.65s) and MT (0.36s), where the state size is much smaller (13KB for BO and 7KB for MT).

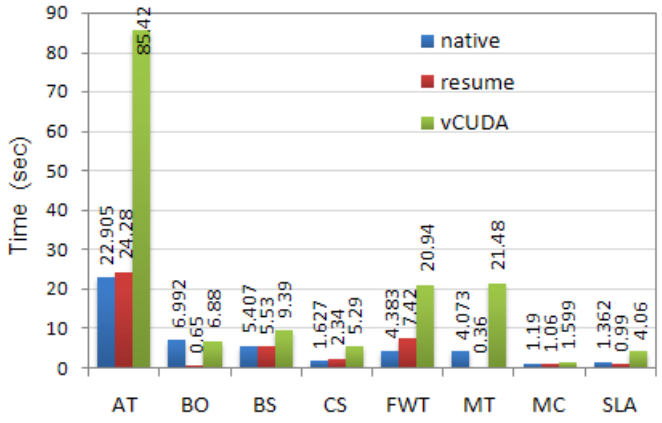

Figure 9. Comparison of time consumption among resume operation, native and virtualized executions.

4.4. Compatibility

A well designed API interface virtualization scheme should be not only transparent but also compatible to a wide range of applications except for the examples in official SDK. In order to verify the compatibility of vCUDA, we chose five applications from CUDA zone [7] that can run correctly in our testbed. These applications were mp3 lame encoder in CUDA contest [25], Molecular Dynamics Simulation with GPU [22], matrix-vector multiplication algorithm in CUDA [9], storeGPU [1] and MRRR implementation in CUDA [20].

All the five third party applications passed the test and returned the same results as in native executions. The details of these tests are depicted in Table 2, which shows that when running in vCUDA framework, these applications exhibit similar performance characteristics as those discussed in Section 4.1. For example, the performance degradation of application MV is mainly due to the higher data volume transfer compared with other applications.

5. Discussion

In this section we discuss several issues with our current prototype and how they can be addressed in future.

vCUDA has not yet achieve virtualization of all APIs of CUDA version 1.1. As visual computing is becoming very popular and widespread and the virtualization of 3D graphics interface such as OpenGL could be beneficial in practice, we are planing to integrate existing 3D virtualization technology such as VMGL [19] into our framework.

Another aspect needs to improve is the efficiency of network transmission. So far our work has focused on porta-

Table 2. Statistics of the Third Party Applications.

	Number of APIs	GPU RAM	Data Volume	Native Time(s)	vCUDA Time(s)
GPUmg	89	294KB	2448KB	0.242s	0.387s
storeGPU	32	983KB	860KB	0.301s	0.413s
MRRR	370	1893KB	2053KB	0.591s	0.686s
MV	31	15761KB	61472KB	0.776s	3.814s
MP3encode	94	1224KB	391KB	0.252s	0.515s

bility across VMMs and OSs, therefore avoided all performance optimizations that might compromise portability. The underlying data channels have not been fully utilized, leading to relatively low efficiency. One of our goals in future is to develop a specific communication strategy between different domains in Xen by adopting technologies such as XWAY [17].

CUDA currently is only Nvidia's private GPGPU interface standard, which means vCUDA only supports Nvidia's graphics cards. Recently the industry has announced other competitive frameworks such as [26] for the same purpose, and we expect that methodologies discussed in our framework can also be applied to other interfaces.

6. Related Work

Research community have adopted various methods to expand and reuse API, a typical method is to replace the graphic API library with an "intercept" library that looks exactly like the original graphic library.

According to the specific features and practical requirements, many existing systems intercept calls to the graphics library for various purposes. VirtualGL [30] virtrualizes GLX to grant remote rendering ability. WireGL [14] and its successor Chromium [13] intercept OpenGL [27] to generate different output like distributed displays. Chromium provides a mechanism for implementing plug-in modules that alter the stream of GL commands, allowing the distribute parallel rendering. HijackGL [24] uses the Chromium library to exploring new rendering styles. In VMM platform like Xen this methodology is used to achieve 3D hardware acceleration in a virtual machine. [19] deploys a fake stub in guestOS and redirect the OpenGL flow to hostOS. Blink project [11] intercepts OpenGL to multiplex 3D display in several ClientOS. Another main category is the tools to help performance analysis and debugging. IBM's ZAPdb OpenGL debugger [15] uses this interception technique to aid in debugging OpenGL programs. Intel's Graphics Performance Toolkit [16] uses a similar method to instrument graphics application performance.

High level middleware- and language-based virtual machines have been studied and used for high performance computing, such as HPVM [6] and Java. In [12], the authors proposed a framework for HPC applications in VMs, which addresses the performance and management overhead associated with VM-based computing. They explained how to achieve high communication performance for VMs by exploiting the VMM-bypass feature of modern high speed interconnects such as InfiniBand, and reduce the overhead of distributing and managing VMs in large scale clusters with scalable VM image management schemes.

7. Conclusions

In this paper we proposed a framework vCUDA, a GPGPU high performance computing solution for virtual machines. vCUDA allows applications executing within virtual machines (VMs) to leverage hardware acceleration, which can be beneficial to the performance of a class of high performance computing (HPC) applications. We explained how to access graphics hardware in VMs transparently by API call interception and redirection. Our evaluation showed that GPU acceleration for HPC applications in VMs is feasible and competitive with those running in a native, non-virtualized environment. In future, we will add 3D graphics virtualization to our framework and port it to newer versions of CUDA. We plan to investigate high performance inter-domain communication schemes to improve the efficiency of data transfer in our system.

8. Acknowledgments

The authors would like to thank the anonymous reviewers for their useful suggestions and comments on this paper. This research was supported in part by the National Basic Research Program of China under grant 2007CB310900, and the National Science Foundation of China under grants 60803130 and 60703096.

References

[1] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems.

In Proc. ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC), Boston, MA, Jun. 2008.

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

[3] M. Ben-Yehuda, J. Mason, O. Krieger, and J. Xenidis. Xen/IOMMU, Breaking IO in New and Interesting Ways. http://www.xensource.com/files/xs0106 xeniommu.pdf.

[4] I. Buck, G. Humphreys, and P. Hanrahan. Tracking Graphics State for Networked Rendering. In Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, pages 87-95, Interlaken, Switzerland, Aug. 2000.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, F. Kayvon, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In Proc. ACM SIGGRAPH 2004, pages 777-786, New York, NY, 2004.

[6] A. Chien et al. Design and Evaluation of an HPVM-Based Windows NT Supercomputer. The International Journal of High Performance Computing Applications, 13(3):201-219, Fall 1999.

[7] CUDA: Compute Unified Device Architecture. http://www.nvidia.com/object/cuda_home.html. (accessed September, 2008).

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M.Chen. Revirt: Enabling Intrusion Analysis Through Virtual Machine Logging and Replay. In Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI), Boston,MA, Dec. 2002.

[9] N. Fujimoto. Faster Matrix-Vector Multiplication on GeForce 8800GTX. In Proc. 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), LSPP-402, Apr. 2008.

[10] GPGPU: General Purpose Programming on GPUs. http://www.gpgpu.org/w/index.php/FAQ#What programming APIs exist for GPGPU.3F.

[11] J. G. Hansen. Blink: 3d Display Multiplexing for Virtualized Applications. Technical Report, DIKU - University of Copenhagen, Jan. 2006. http://www.diku.dk/ jacobg/pubs/blink-techreport.pdf.

[12] W. Huang, J. Liu, B. Abali, D. K. Panda. A Case for High Performance Computing with Virtual Machines. In Proc. 20th Annual International Conference on Supercomputing, June 28-July 01, 2006, Cairns, Queensland, Australia.

[13] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern,P. D. Kirchner, and J. T. Klosowski. Chromium: a Streamprocessing Framework for Interactive Rendering on Clusters. In Proc. 29th Annual Conference on Computer Graphics and Interactive Techniques, pages 693-702, New York, NY, USA,2002.

[14] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In Proc. SIGGRAPH, pages 129-140, August 2001.

[15] IBM's ZAPdb OpenGL debugger, 1998. Zapdb. Computer Software.

[16] Intel Graphics Performance Toolkit. Computer Software.

[17] K. Kim, C. Kim, S. I. Jung, H. S. Shin, J. S. Kim. Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In Proc. VEE 2008. ACM Press, Mar. 2008.

[18] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In Proc. Fourth IEEE Workshop on Mobile Computing Systems and Applications, Callicoon, New York, June 2002.

[19] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de La-ra. VMM-independent Graphics Acceleration. In Proc. VEE 2007. ACM Press, June 2007.

[20] C. Lessig, An Implementation of the MRRR Algorithm on a Data-Parallel Coprocessor. Technical Report, University of Toronto, 2008.

[21] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In Proc.6th Symposium on Operating Systems Design and Implementation, San Francisco,CA, Dec. 2004.

[22] MDGPU. http://www.amolf.nl/˜vanmeel/mdgpu/about.html.

[23] A. Menon, J. R. Santos and Y. Turner et al. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments, pages 13-23, Chicago, IL, June 2005.

[24] A. Mohr and M. Gleicher. HijackGL: Reconstructing from Streams for Stylized Rendering. In Proc. the Second International Symposium on Non-photorealistic Animation and Rendering, 2002.

[25] MP3 LAME Encoder (Nvidia's CUDA Contest). http://cudacontest.nvidia.com/index.cfm?action=contestdownload&contestid=2.

[26] OpenCL: Parallel Computing on the GPU and CPU. In Beyond Programmable Shading Course of SIGGRAPH 2008, August 14, 2008.

[27] OpenGL - The Industry Standard for High Performance Graphics. http://www.opengl.org.

[28] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. Journal of Computer Graphics Forum,26:21-51,2007.

[29] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), New York, NY, USA, 2006.

[30] VirtualGL. http://virtualgl.sourceforge.net/.

[31] VMware Workstation. http://www.vmware.com/products/ws/.

[32] XML-RPC. http://www.xmlrpc.com/.