# vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines

Lin Shi, *Student Member*, *IEEE*, Hao Chen, *Member*, *IEEE*, Jianhua Sun, and Kenli Li

**Abstract**—This paper describes vCUDA, a general-purpose graphics processing unit (GPGPU) computing solution for virtual machines (VMs). vCUDA allows applications executing within VMs to leverage hardware acceleration, which can be beneficial to the performance of a class of high-performance computing (HPC) applications. The key insights in our design include API call interception and redirection and a dedicated RPC system for VMs. With API interception and redirection, Compute Unified Device Architecture (CUDA) applications in VMs can access a graphics hardware device and achieve high computing performance in a transparent way. In the current study, vCUDA achieved a near-native performance with the dedicated RPC system. We carried out a detailed analysis of the performance of our framework. Using a number of unmodified official examples from CUDA SDK and third-party applications in the evaluation, we observed that CUDA applications running with vCUDA exhibited a very low performance penalty in comparison with the native environment, thereby demonstrating the viability of vCUDA architecture.

**Index Terms**—CUDA, virtual machine, GPGPU, RPC, virtualization.

## 1 Introduction

System-level virtualization has recently attracted much attention from both industry and academe. This interest stems from the continued growth in hardware performance and the increased demand for service consolidation from business markets. Virtualization technology allows multiple virtual machines (VMs) to coexist in a physical machine under the management of a virtual machine monitor (VMM). VMM has been applied to many areas including intrusion detection [10], high-performance computing (HPC) [16], device driver reuse [25], and so on.

Over the past few years, graphics processing units (GPUs) have quickly emerged as inexpensive parallel processors due to their high computation power and low price. The modern GPU is not only a powerful graphics engine, but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth [13], [31], [33]. The newest GPU architecture supports high-precision floating-point computation and error-correcting code (ECC) memory, which are both important requirements for HPC.

GPU programming has been extensively used in the last several years for resolving a broad range of computationally demanding and complex problems. The introduction of some vendor specific technologies, such as NVIDIA's Compute Unified Device Architecture (CUDA) [7], further accelerates the adoption of high-performance parallel computing to commodity computers.

Although virtualization provides a wide range of benefits, such as system security, ease of management, isolation, and live migration, it is not widely adopted in the high-performance computing area. This is mainly due to the overhead incurred by indirect access to physical resources such as CPU, I/O devices, and physical memory, which is one of the fundamental characteristics of virtual machines. In addition, the GPU hardware interface is proprietary, rather than standardized. GPU designers are highly secretive of the specifications of their hardware, which leaves no room to develop an elegant virtualization scheme for a GPU device in the hardware abstraction layer. As a result, the powerful data processing ability in GPUs cannot be directly used by applications running in virtualized environments.

In this paper, we present vCUDA, a framework for HPC applications that uses hardware acceleration provided by GPUs to address the performance issues associated with system-level virtualization technology. vCUDA works by intercepting and redirecting Compute Unified Device Architecture APIs in VMs to a privileged domain with an enabled real graphics device. vCUDA accomplishes the computations through the vendor-supplied CUDA library and GPU driver. A VMM-specific remote procedure call (RPC) tool, Virtual Machine Remote Procedure Call (VMRPC) [6], was developed to accelerate the data transfer. Using detailed performance evaluations, we demonstrate that hardware-accelerated high-performance computing jobs can run as efficiently in a virtualized environment as in a native host.

In summary, the main contributions of our work are as follows:

- A framework that allows high-performance computing applications in VMs to benefit from GPU acceleration. To demonstrate the framework, we developed a prototype system on top of KVM and CUDA.
- The development of a dedicated RPC infrastructure for virtualized environments to improve the performance of remote procedure calls between different VMs.

- *The authors are with the School of Information Science and Engineering, Hunan University, Changsha 410082, P.R. China.*
*E-mail: {shilin, haochen, jhsun}@aimlab.org, lkl510@263.net.*

*Manuscript received 17 Nov. 2010; revised 15 Apr. 2011; accepted 29 May 2011; published online 14 June 2011.*
*Recommended for acceptance by R. Gupta.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-11-0634.*
*Digital Object Identifier no. 10.1109/TC.2011.112.*

- Two functional extensions built within our framework (i.e., resource multiplexing and suspend/resume (S&R)) without modifying the applications.
- A detailed performance evaluation on the overhead of our framework. This evaluation shows that the vCUDA framework is practical and performs well in HPC applications comparable with those in native environments.

Some preliminary results of this work were presented in our IEEE IPDPS'09 conference paper [32]; however, additional technical details are described in the present paper. In addition, we have developed a new model (SHARE model) for vCUDA that significantly improves its efficiency. The proposed vCUDA is adjusted accordingly to reflect these changes. The results show that, for the new model, only vCUDA produces a slight overhead in a virtualized environment. Most importantly, vCUDA has been updated (from CUDA 1.1 and G80) to support CUDA 3.2 and the Fermi GPU. However, in attempting to port vCUDA to new platforms, we failed to install a working Fermi driver in the Xen-enabled linux kernel for the domain0 of almost all major Linux distributions. Finally, we chose to port vCUDA and VMRPC to KVM [21] (a full virtualization solution).

The rest of the paper is organized as follows: in Section 2, we provide some necessary background for the current work. We then present our vCUDA design in Section 3, and the implementation in Section 4. An example is shown in Section 5. We carry out a detailed performance analysis in Section 6. Finally, we introduce related work in Section 7, and conclude the paper in Section 8.

## 2 Background

### 2.1 VMM and GPU Virtualization

System-level virtualization simulates details of the underlying hardware in software, provides different hardware abstractions for each operating system instance, and concurrently runs multiple heterogeneous operating systems. System-level virtualization decouples the software from the hardware by forming a level of indirection, which is traditionally known as the virtual machine monitor. A variety of VMM platforms [21], [4], [36] provide a complete and consistent view of the underlying hardware to the VM running on top of it.

I/O virtualization is an important part of system-level virtualization. So far, there are four main I/O device virtualization methods: VMM pass-through [39], device emulation [21], self-virtualized hardware [38], and protocol redirection (para-virtualization) [4], [23]. Each method has corresponding advantages and disadvantages. Basically, the first three approaches own the characteristics of transparency to guestOS, while the protocol redirection approach needs to change the I/O control flow in guestOS.

Although virtualization has been successfully applied to a variety of I/O devices, virtualization of the GPU in VMM is difficult. One main reason is the lack of a standard interface at the hardware level and driver layer. The VMware [8] classifies the existing GPU virtualization approaches into two main categories: front-end virtualization (API remoting and device emulation) and back-end virtualization (VMM pass-through). API remoting [23] is a kind of protocol redirection, which redirects the GPU API from guestOS to hostOS by replacing system dynamic link libraries containing APIs. API remoting is device independent but significant issues remain with this approach when the methods for managing the state in the original APIs are not designed with this approach in mind. Furthermore, the protocol redirection method in API level can only deal with a part of the device function in a time, for GPU which involved many different kinds of APIs, e.g., OPENGL, Direct3D, CUDA, OpenCL, DirectCompute, etc., each of these interfaces needs a stand-alone virtualization solution. As the device emulation method, it is nearly intractable to emulate a full function GPU corresponding to a real high-end modern GPU because it is a highly complicated and underdocumented device. The VMM pass-through method means the VMM assigns part of the host's real I/O device to the guestOS exclusively, any other VMs, sometimes even the hostOS, do not have any access to the device anymore. Since the guestOS communicates directly with the hardware, it is able to use the native drivers and its full capabilities. This method does not require any guestOS changes and in theory providing near-native performance. But this method violates the resource consolidation and multiplex philosophy of virtualization.

### 2.2 Inter-VM Communication

VMM enforces the isolation between different VMs, while simultaneously severely compromises the effectiveness of inter-VM communication. The communication between two guestOS needs to frequently pass through the hostOS, leading to a significant inter-VM communication overhead. Some techniques in this research area, such as Xensocket [41], Xenloop [37], and Xway [22], have addressed this problem by building a fast inter-VM channel based on a shared memory. By redirecting or changing the traditional network transport path (socket layer or TCP/IP protocol stacks), these techniques obtain a better performance in both latency and throughput. The latest development in this area is Fido [2], which is a high-performance inter-VM communication mechanism tailored to software architectures of enterprise-class server applications.

However, in order to maintain transparency in these applications, these projects choose to utilize the shared memory mechanism in the VMM or OS layer, resulting in frequent OS and VMM layer context switches. Borrowing the idea from Bershad et al. [3], we developed a dedicated inter-VM communication channel in user space that removes the redundant context switches and significantly decreases the communication cost between VMs. The user space communication channel is integrated into a lightweight RPC system, VMRPC [6], which is used as the main middleware in the proposed vCUDA framework to redirect the API flow. Section 4.3.1 provides more details on this topic.

### 2.3 CUDA

CUDA [7] is a general-purpose graphics processing unit (GPGPU) solution of NVIDIA that provides direct access to the GPU hardware interface through a C-like language (CUDA C), rather than the traditional approach that relies on the graphics programming interface. CUDA extends C by allowing the programmer to define C functions, called CUDA *kernels*, that, when called, are executed $n$ times in parallel by $n$ different CUDA threads, as opposed to only once like regular C functions.

CUDA provides two programming interfaces: driver API and runtime API. The runtime API is built on top of the CUDA driver API. The virtualization of the CUDA runtime API is the main focus of this paper.

## 3 Design

There are many challenges in designing a good CUDA virtualization scheme. In formulating our design, we had several goals in mind:

**Transparency.** A well-designed CUDA virtualization scheme should be intuitive and useful in a wide variety of CUDA applications. Furthermore, the scheme should be binary compatible to applications originally designed for nonvirtualized environment.

**Performance.** vCUDA should address the performance and management overhead associated with VM-based computing, and should make CUDA applications to run as fast as the native case.

**Functionality.** Whenever possible, vCUDA should be compatible with existing features in VMs, such as resource multiplexing, live migration, secure isolation, and so on. With vCUDA, applications running in VMs should work as usual, without any changes, when enabling VM-specific features.

The vCUDA framework is organized around two main architectural features. First, runtime APIs are encapsulated into RPC calls. Their parameters are properly queued and redirected to the hostOS, and internal objects are cached and kept persistent on both the server and the client sides. Through this kind of virtualization, the graphics hardware interface is decoupled from the software layer. Second, we developed a high-efficiency RPC tool working in a virtualization environment that exploits the shared memory mechanisms in VMM in order to enhance the performance of remote procedure call between different VMs. Lazy update is also adopted to reduce the frequency of remote calls.

CUDA is currently not a fully open standard, and some internal details have not been officially documented. We do not possess full knowledge regarding the states maintained by the underlying hardware driver or runtime library. In vCUDA, we achieved the virtualization functionality from the following three aspects:

- **Function parameters**. The intercepting library has no access to all the internals of an application to which it is linked. However, the fake library can obtain all the parameters of corresponding API calls, which can be used as inputs to the wrapper APIs defined in the intercepting library. These wrapper APIs with proper parameters can then be sent to a remote server for execution as normal calls.
- **Ordering semantics**. Ordering semantics are the set of rules that restrict the order in which API calls are issued. CUDA API is basically a strictly ordered interface, which means that some APIs must be launched in a specified order. This behavior is essential for maintaining internal consistency. However, in some cases, if possible, vCUDA uses less constrained ordering semantics when an increased performance is ensured.
- **CUDA states**. CUDA maintains numerous states in the hardware and software. The states contain attributes such as CUBIN handler, device pointers, server/client memory pointers, variable symbol, array pitch, texture, and so on. On a workstation with hardware acceleration, the graphics hardware and local CUDA library keep track of all of the CUDA states. However, in order to properly implement a remote execution environment for these APIs in virtual machines, keeping track of these states in the remote server is necessary for vCUDA.

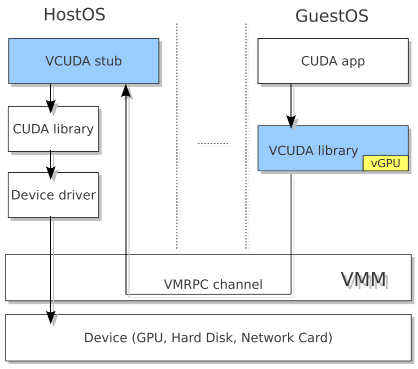


Fig. 1. vCUDA architecture.

## 4 Implementation

vCUDA uses a robust client-server model consisting of three user space modules: the vCUDA library (as a user space library in the guestOS), the virtual GPU (a database in the guestOS) , and the vCUDA stub in the server (as an application in the hostOS). Fig. 1 shows the vCUDA architecture. For the rest of this paper, the term "server memory" refers to the address space of vCUDA stub in the hostOS, the term "client memory" refers to the address space of CUDA application running in the guestOS, and "device memory" refers to the memory space in the graphics device of the hostOS.

### 4.1 System Component

**vCUDA Library** resides in the guestOS as a substitute for the standard CUDA runtime library (*libcudart.so* in Linux), and is responsible for intercepting and redirecting API calls from applications to the vCUDA stub. The virtualized CUDA library, hereafter named the vCUDA library, realizes the functions defined by the real CUDA runtime library. When an API is invoked by the application, the thread index, API index, and its arguments are packed and inserted into a global API queue. This queue contains a copy of the arguments, dereference information, synchronization flag, and so on. The queue contents are periodically pushed to the RPC system according to several predefined strategies (details in Section 4.3.2).

vCUDA now supports two different kinds of execution mode: the TRANSMISSION mode working with traditional RPCs such as XMLRPC [40], and the SHARE mode built on top of VMRPC, a dedicated RPC architecture for VMM platforms. The main difference between the two modes is the manner by which data from the client are delivered to the server, or vice versa. In TRANSMISSION mode, all data are transferred through the traditional TCP/IP protocol stack, whereas in SHARE mode, they are fetched from a shared zone in the address space of the application.

Two programming interfaces are provided by CUDA: runtime and driver APIs. We chose runtime API as the target of virtualization because it is the most widely used library in practice, as well as the officially recommended interface for programmers. However, we do not anticipate any main obstacle in the virtualization of the driver-level API. In addition to the runtime APIs described in the official programming guide, there are six other internal APIs in the dynamic linking library of CUDA runtime: *__cudaRegisterFatBinary()*, *__cudaUnregisterFatBinary()*, *__cudaRegisterVar()*, *__cudaRegisterTexture()*, *__cudaRegisterSurface()*, and *__cudaRegisterFunction()*. These six APIs are used to manage the CUDA *kernel* (device code) and device memory allocated to *CUDA variables*, *texture*, or *surface*. All the six APIs are compiled by NVCC into a final binary, thereby rendering it invisible to programmers. The virtualization methods of APIs are introduced in Section 4.2.

**vGPU** is created, identified, and used by the vCUDA library. In fact, vGPU is represented as a database in the memory maintained by the vCUDA library. vGPU provides three main functionalities. First, vGPU abstracts several features of the real GPU to give each application a complete view of the underlying hardware. Second, vGPU creates a virtual GPU context for each CUDA application, which contains the device attributes, such as GPU memory usage and texture memory properties. vCUDA sets up a synchronization mechanism between the vGPU and the stub to ensure data consistency. The third function of vGPU is to store the CUDA context for the need of suspend/resume as described in Section 4.4.2.

**vCUDA stub** receives remote requests and creates a corresponding execution context for API calls from the guestOS, then accomplishes the required CUDA task and returns the results to the guestOS. The stub manages the actual physical resources (device memory, and so on), dispatches the service threads and working threads, and keeps a consistent view of the states on both sides of the server and client through periodical synchronization with the vGPU. The vCUDA stub maintains the consistency of internal objects with the help of a Virtual Object List (VOL). The details of this data structure are illustrated in Section 4.2.

As the server, vCUDA stub spawns threads to manage the CUDA devices and answer CUDA requests. There are two different kinds of threads in vCUDA stub: working threads and service threads. One service thread corresponds to a remote client waiting for CUDA service. This thread receives the CUDA command via the RPC channel, translates the command into a server-side representation, and then forwards the command and related information to the working threads. Each of working threads corresponds to one physical GPU device. It obtains the CUDA function arguments from the service threads and manipulates the GPU device to complete the CUDA task. When the working threads are finished, the service threads collect the result and transfer it to the corresponding vCUDA library as the return of the RPC. The separation of the service and working threads is an important feature to realize the multiplex of GPU device in virtual machines. The details of multiplex are presented in Section 4.4.1.

## 4.2 Tracking CUDA State

**CUDA state**. For vCUDA, simply delivering the original arguments and directly calling the CUDA API on the server side are insufficient. As mentioned before, CUDA maintains a large amount of internal states, such as memory pointer, CUBIN handler, variable symbol, array pitch, texture, and so on. Remotely managing these states is a very complicated task because the original CUDA APIs were not designed with this approach (remote execution), not to mention the proprietary nature of the CUDA platform, in mind. In order to properly implement a remote execution environment for relevant APIs in VMs, keeping track of the CUDA state in software is necessary. For example, when a CUDA program begins, objects such as *texture* and *variable* are registered as symbols, and subsequent APIs manipulate these objects with these symbols. The vCUDA framework must know how to correctly manipulate these objects. Another example is the client memory pointer used by CUDA applications, which resides in and is only accessible by the guestOS. Directly delivering the raw pointers to the hostOS makes no sense. Therefore, the vCUDA stub must track these objects while the application is running.

**Virtual object list**. In order to track CUDA objects, vCUDA keeps all the objects' information in a global list, called Virtual Object List. This list manages and synchronizes the content of objects shared between the server and client. When a client invokes a remote call to the server, the VOL locates the client-side objects corresponding to the API parameters and updates their contents. On return to the client, the VOL looks up the CUDA object associated with the parameters to propagate changes made by the server to the client. To perform this task, the VOL must track allocation and deallocation events to determine when to add or delete an object from the tracking list.

For example, vCUDA takes advantage of VOL to keep internal consistency of memory objects. There are two different kinds of memory objects: the device memory object in GPU and the host memory object in server/client RAM. As for the device memory object, the entry of an allocated device memory chunk is registered to the VOL when calling *cudaMalloc()*, and removed from the list when calling *cudaFree()*. Based on the VOL, vCUDA can determine the size and base address of all device memory chunks used by the CUDA runtime. The server/client memory object is also registered to the VOL, and vCUDA uses the VOL to control the data-transfer operation. If any CUDA API parameters are passed by the reference (typically a memory pointer), the vCUDA library delivers (in TRANSMISSION mode) or copies (in SHARE mode) the referent to the vCUDA stub.

Aside from the memory objects referenced by the device pointers or server/client pointers, the VOL must also manage other CUDA objects, such as the CUBIN handler and variable symbol. For symbols, vCUDA traces their usage (by the name and address), and then translates them to appropriate remote objects in the vCUDA stub. In the following sections, we explain in detail how symbols of variables are traced and translated. When *__cudaRegisterVar()* is called in the client, it binds the second argument, *hostVar*, a specific string, to its third argument, *deviceAddress*, which is an address in GPU. vCUDA then duplicates the original string in the stub and keeps its address in the VOL. When *cudaMemcpyToSymbol()* or *cudaMemcpyFromSymbol()* is called, the vCUDA stub searches the VOL according to the name, and launches the API with the obtained address. The virtualizations of *__cudaRegisterTexture()* and *__cudaRegisterSurface()* are similar to *__cudaRegisterVar()*. Aside from the symbols, the CUBIN handler in *__cudaRegisterFatBinary()* and f*unction stub* in *__cudaRegisterFunction()* are also translated between the server and client.

**Identify internal objects**. In order to keep track of the CUDA runtime states, vCUDA needs to identify all the arguments that represent the internal objects. For most of the CUDA APIs, this task is easy because the arguments are set up in a fixed location. But *cudaSetupArgument()* is very different from other APIs, it is used to push the arguments of a CUDA *kernel* (device function) to a device execution stack. The programmer is free to determine the number, type, and sequence of arguments for this API. Therefore, vCUDA has no prior knowledge for identifying items relevant to the internal objects. Fortunately, server/client memory objects are never constructed using these kinds of arguments in practice, which significantly reduces the complexity of virtualization. In our implementation, the vCUDA distinguishes normal arguments with state-related ones by comparing them with the value stored in the VOL.

**API virtualization without remote call**. Not all APIs need to be intercepted and redirected to a remote site. Those that do not modify the CUDA runtime states can be replaced by local ones. A typical example is *cudaGetDeviceProperties()*, which queries the GPU for device properties and compute capacity. vGPU relies on this API to return some customized information to create a fake GPU for applications. This also provides an opportunity for vCUDA to control the part of the features of physical GPUs exposed to the client.

In CUDA 3.2, the API *cudaHostAlloc()* is the same to *cudaMallocHost()* by default. However, this API supports additional features, such as "zero-copy." When the flag option *cudaHostAllocMapped* is set, *cudaHostAlloc()* maps the allocation into the CUDA address space. Then, the CUDA *kernels* are able to directly access CPU memory without any *cudaMemcpy()*. Using zero-copy, CUDA eliminates the need to copy data to/from the device memory. For the vCUDA, zero-copy confuses the local CPU address space and remote GPU address space, leaving the vCUDA with no means to distinguish them in a CUDA *kernel* launch. So far, zero-copy is not supported by vCUDA.

## 4.3 Optimizations

### 4.3.1 VMRPC: RPC Based on Shared Memory

In order for vCUDA to work correctly, the environment in which the CUDA routine is called from the stub in the hostOS must be identical to the environment in which it is called from the vCUDA library in the guestOS. In an earlier version of vCUDA [32], we developed a TRANSMISSION mode wherein all parts of this environment are transmitted from the vCUDA library to the stub, and vCUDA constructs a fake execution context for every CUDA call. The transfer task is fulfilled by a traditional RPC system, XMLRPC. Great effort was made to adapt the XMLRPC framework to the VMM environment; however, significant overhead was still introduced to vCUDA. The performance overhead comes from some aspects including the low bandwidth data channel between VMs, the expensive serialization/deserialization procedure, redundant data copies, and frequent VMM interventions.

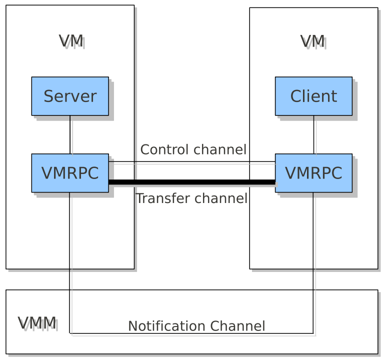


Fig. 2. VMRPC architecture.

Finally, we decided to develop a dedicated RPC framework for VMM, called Virtual Machine Remote Procedure Call [6]. VMRPC was designed based on the shared memory mechanism in mainstream VMMs such as KVM, Xen, and VMware. Fig. 2 shows the VMRPC architecture. VMRPC is made up of three components: transfer channel, control channel, and notification channel. Transfer channel is a preallocated large shared-data section dedicated to large-capacity and high-speed data transfer. The transfer channel directly opens a fast channel in the address space of two processes that reside in different VMs. The control channel is realized as tiny shared zones of server and client, which is responsible for constructing the RPC semantics as substituent of the interface definition language (IDL) in traditional RPC systems. The notification channel is an asynchronous notification mechanism, similar to hardware interrupt or software signal. The notification channel is implemented by leveraging the special inter-VM communication mechanism in certain VMM platforms. Its main task is to trigger the RPC actions, and to manage the synchronization of shared memory accesses. In VMRPC, the notification channel is the only part where the OS and VMM level activities must be involved.

**Memory mapping**. Unlike the other studies on inter-VM communication optimizations discussed in Section 2.2, VMRPC directly sets up the shared zone at the virtual address space of the process. Fig. 3 shows how VMRPC uses user space memory mapping to deploy control channel and transfer channel. In KVM, the IVSHMEM [27] tool supports an inter-VM shared memory device that maps a shared memory object as a PCI device in the guestOS. In Xen, the hostOS can map a foreign memory space from guestOS to its own address space, and then directly access the mapped memory. In the VMware workstation, virtual machine communication interface (VMCI) infrastructure provides memory-sharing capability between VMs on the same host.

**Transfer channel**. By using the memory mapping mechanism, VMRPC built a preallocated large memory region:

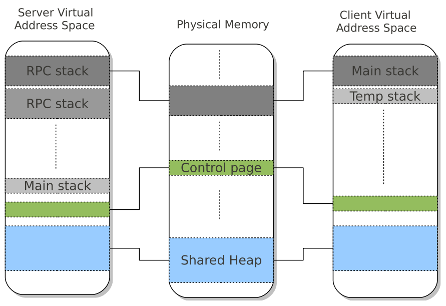


Fig. 3. Virtual address mapping in VMRPC.

*shared heap*, which is mapped into both server and client. The *shared heap* is different from the standard operating-system-provided main heap; however, the two can be interchangeably used by applications. For the *shared heap*, the VMRPC realizes an elegant heap management interface. When a piece of data needs to be shared, the programmer (the vCUDA designer in our case) should use *VMRPC_malloc()* instead of regular C function *malloc()* to allocate the virtual memory. This places the allocation in the shared heap instead of the regular private heap of system process, allowing the server to directly access the heap. When this space is no longer needed, the API *VMRPC_free()* should be invoked, which operates in the same way as the standard equivalent *free()*, but in the *shared heap* of VMRPC. VMRPC also provides APIs to help the programmer decide the size of *shared heap*, and when to set it up or tear it down. In VMRPC, OS layer data copying is omitted by removing the OS and VMM from the critical data-transfer path.

**Control channel**. A native procedure call does not need an interface definition language because both the caller and callee are statically compiled to pinpoint the arguments and their exact location. Since in VMM the caller and callee also reside in the same host (although located in different VMs), there is no need to perform complicated data manipulations, such as data serialization and deserialization.

As Fig. 3 depicts, the control channel consists of the *control page* and the *RPC stack*. The *control page* serves as the control information exchange tunnel for the RPC caller and callee. When VMRPC initializes, it stores the information about the size, start address of *RPC stack* and *shared heap*, and so on. When the client launches an RPC, it stores the RPC index and stack frame information. After the server finishes the call, the acknowledgment and return value also stay in the *control page*. The *RPC stack* is shared between the server and client, and stores all the arguments of CUDA API calls. When the client issues a remote call, the private execution stack of the client is forwarded (shared) to the server as an *RPC stack*. In VMRPC, the server directly uses the contents in this stack. All the function arguments and any in-stack structures that are pointed to are made available. This procedure does not involve large-capacity data replication, analysis, or coding. No matter how complicated the arguments and return value of the calls are, the overall overhead in VMRPC is only proportional to the cost incurred by the notification channel described below. Therefore, VMRPC simultaneously achieves a low latency and low CPU load.

**Notification channel**. In VMRPC, the VMM-specific asynchronous mechanism is used as a notification channel, such as the VMCHANNEL [35] in KVM, the event channel in Xen, and the VMCI datagram in VMware workstation. The use of VMM-specific notification mechanism is helpful in reducing the latency in the RPC system, which is critical for highly interactive applications. Our evaluation shows that latency of VMRPC in KVM and Xen is only 10 percent of that of the native socket.

In summary, VMRPC completely bypasses the OS and VMM, eliminates the data serialization/deserialization operations, and takes full advantages of VMM-specific mechanisms to improve the overall system performance.

#### *4.3.2 Lazy RPC*

Thousands of CUDA APIs could be called in a CUDA application. If vCUDA sends every API call to remote site at the moment the API is intercepted, the same number of RPCs will be invoked and the overhead of excessive world switch (execution context switch between different VMs) will be inevitably introduced into the system. In virtual machines, the world switch is an extremely expensive operation and should be avoided whenever possible [28]. We adopted an optimization mechanism called Lazy RPC to improve the system performance by intelligently batching specific API calls.

We classify the CUDA APIs into two categories: instant APIs, whose executions have immediate effects on the state of CUDA runtime system, and lazy APIs, which are side-effect-free on the runtime state. This kind of classification allows vCUDA to reduce the frequency of world switch by lazily updating states in the stub. vCUDA can defer decisions about the specific state to transmit until the application calls an instant API because vCUDA records parts of the CUDA states, rather than immediately launching remote call for all APIs. vCUDA decreases unnecessary RPCs by redirecting lazy APIs to the stub side in batches, thereby boosting the performance of the system. A potential problem with lazy mode is the delay of error reporting and time counting. Therefore, the lazy mode may not be suitable for debugging and measurement purposes.

An example of instant API is *cudaMalloc()*, which returns a device address and changes the CUDA state in the client. Each *cudaMalloc()* must be immediately launched as a remote call. *cudaMemcpy(DeviceToHost)* directly changes the client state, and is also an instant API. Two typical lazy APIs, *cudaConfigureCall()* and *cudaSetupArgument()*, prepare for subsequent *cudaLaunch()*, which can also be delayed until an API returns the device state (*cudaMemcpy(DeviceToHost)* in most cases).

Classifying an API as instant or lazy is sometimes difficult. For example, in some CUDA applications, several consecutive *cudaMemcpy(HostToDevice)* could be grouped into one remote call, until a *cudaMemcpy(DeviceToHost)* changes the client state. But vCUDA cannot determine whether the application will reuse the host memory between the calls of *cudaMemcpy(HostToDevice)*. Considering the overhead in memory copy operation, vCUDA chooses to remotely invoke every *cudaMemcpy(HostToDevice)* (i.e., as an instant API).

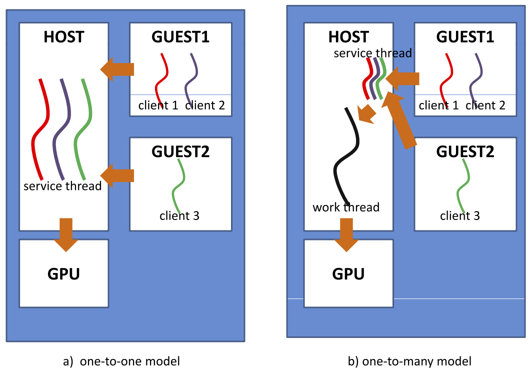


Fig. 4. Two models for CUDA multiplexing.

## 4.4 Extra Functionalities

In a precious study [8], the authors used four primary criteria in comparing the virtualization method of GPU: performance, fidelity, multiplexing, and interposition. The former two indicate how close the virtualization method is in providing a complete GPU illusion to the operating system and user. The latter two emphasize the special functionalities of virtualization, such as resource multiplexing, live migration, secure isolation, and so on. The performance and fidelity are discussed in Section 6. In this section, we describe two advanced functionalities of virtualization supported by vCUDA.

### 4.4.1 Multiplexing

One fundamental feature provided by VMs is device multiplexing. The VMM/hostOS manipulate the native hardware while preserving the illusion that each guestOS occupies a complete stand-alone device. For example, in Xen, the physical network card can be multiplexed among multiple concurrently executing guestOS. To enable this kind of multiplexing, the privileged driver domain (domain0) and the unprivileged guest domain (domainU) communicate by means of the frontend-backend driver architecture. The driver domain hosts the *back end*, and the guest domain hosts the *front end*. Nevertheless, the GPU does not support this kind of multiplexing. Unlike network controllers, GPU specification is privately held by hardware vendors, and often dramatically changes across revisions. Thus, multiplexing a physical GPU device in the way commonly used by popular virtual machines to virtualize hardware devices is nearly intractable.

CUDA supports three different compute modes: default (two and more host threads can share one GPU), exclusive compute (only one host thread can use the device at any given time), and prohibited compute (no host thread can use the device). Naturally, default mode is the best choice to multiplex the GPU in a virtualized environment because it is already able to share the GPU. Fig. 4 shows a one-to-one model executing the remote CUDA task. A dedicated service thread is allocated for each CUDA client (regardless of whether they come from the same VM or different VMs). The CUDA runtime has the responsibility to manage the service thread and serially execute different CUDA *kernels* of threads in a FIFO order. The access to GPU is blocked when another service thread has already occupied the device.

Nevertheless, this model has a shortcoming in that it relies too heavily on the CUDA framework to directly multiplex the GPU devices. First, CUDA is a private and fast-changing framework for GPGPU computing. The multiplex feature of CUDA and its interface may change in the future. If the device multiplex function can be separated from the CUDA runtime, it will provide the maximum flexibility in the changing face of technological constraints and opportunities. Second, not all GPU hardware and software frameworks support the default mode of sharing GPUs between multiple threads, especially for GPUs and APIs dedicated to graphics. Third, some applications need the GPU in the host to run in the exclusive mode. For example, a node equipped with three graphic cards may set each of its GPUs to exclusive mode in order to accelerate one of its multi-GPU CUDA applications.

In short, vCUDA should expose the device multiplex function using itself as basis, independent of specific API framework such as CUDA, OpenCL, or DirectCompute. Thus, vCUDA develops a one-to-many model to multiplex GPU device in the VM. Fig. 4 shows the one-to-many model. In this model, only one working thread is allocated for each GPU device in hostOS, and all CUDA requests are completed in its context. As described in Section 4.1, the vCUDA stub spawns one service thread for each vCUDA client, which obtains the remote CUDA request from the vCUDA client then forwards the request to the working thread. Under the coordination of the vCUDA stub, two different service threads can cooperatively manipulate one hardware resource by connecting to a single working thread. The working thread is a standard CUDA application that serializes all the requests of the vCUDA clients and executes the CUDA *kernels* one by one.

Regardless of whether multiple vCUDA clients reside in the same VM or come from different VMs, vCUDA allows them to be concurrently executed. Our tests show that there was no marked difference between the two scenarios. In all cases, the concurrent vCUDA clients compete for resources in two levels: hardware and software. Hardware resource, such as global memory, is occupied by client_A when *cudaMalloc()* is called, and is released by calling *cudaFree()*. Before *cudaFree()*, this memory region cannot be reassigned to any other vCUDA clients, even if, for the time being, client_A does not launch any CUDA *kernels*. Hardware resources on the chip, such as registers and shared memory, are shared for all running CUDA *kernels*. Software resources include the virtual address space and handler. Because the Fermi GPU already supports the 40-bit unified address space, these would not be a significant problem, even for hundreds of concurrent threads.

### 4.4.2 Suspend and Resume

Most VMMs support suspend/resume functionality, which enables the user to pause a guestOS and resume it at a later time, as if it was never stopped. The key idea of S&R is to save and restore the inner state of the entire operating system. The typical states include CPU registers, memory content, storage content, network connections, and so on. However, these are not sufficient to suspend and resume a guestOS that runs CUDA applications, because portions of their states reside in GPU (DRAM, on-chip memory, and registers), not in the CPU or system memory. These states are complex because the GPU has multiple memory levels and thousands of registers. Furthermore, no general method exists to directly read/write most of these states. In our early implementation [32], a full API flow record and replay strategy was used to restore the whole GPU state, which consumes a significant amount of resources.

However, the GPU state is complex only when one CUDA *kernel* is running. CUDA *kernels* represent the core computing part of the CUDA application and are executed by thousands of parallel threads inside the GPU. When a CUDA *kernel* is finished, all its output is stored in the DRAM of the GPU device, and there is no need to rebuild the inner state of each *kernel*. This observation led us to develop an out-of-the-kernel S&R scheme:

In the suspend stage, when a suspend signal is received, vCUDA checks whether a CUDA *kernel* is running. If so, the vCUDA waits until the kernel is finished. When there is no active *kernel*, vCUDA library notifies the vCUDA stub to start the suspend process. The device memory object is transferred from the GPU DRAM to the host DRAM using *cudaMemcpy(DeviceToHost)*. All objects in VOL are transferred to and duplicated in vGPU. The CUDA application will then be frozen along with the guestOS. In the resume stage, vCUDA initiates a new working thread on the host, and synchronizes the working thread with the CUDA state stored by vGPU. The synchronization is accomplished by reallocating the GPU DRAM, transferring the data from the guestOS to hostOS, and finally, rebuilding the device object through *cudaMemcpy(HostToDevice)*.

A potential difficulty exists with vCUDA resume. Although vCUDA can rebuild the virtual object in the device memory, it cannot decide the address of the object in virtual space. In normal CUDA applications, the device memory address is obtained by calling *cudaMalloc()*, which means that the address is managed by CUDA runtime in an opaque way. When vCUDA reallocates the GPU DRAM to rebuild the virtual object in the resume stage, *cudaMalloc()* may return an address different from the preceding address in the suspend stage. This leads to an inconsistent state of CUDA applications between the suspend/resume points. Our solution to this problem is to maintain a GPU memory pool, which is a two-phase memory management scheme on top of the CUDA runtime. At the suspend stage, the vCUDA stub preallocates a bulk of GPU memory using *cudaMalloc()*. Subsequent client memory allocation requests (through *cudaMalloc()*) are replaced by *vcudaMalloc()*, which allocates the GPU memory from the previously established memory pool. As a result, vCUDA can track and control the device memory allocation operations in this manner. At the resume stage, the vCUDA stub rebuilds the GPU memory pool. The device memory objects of the suspended CUDA application were restored by launching *vcudaMalloc()* with the special address argument.

With the GPU memory pool, vCUDA provides support for suspend and resume, enables client sessions to be interrupted or moved between nodes. Upon resume, vCUDA presents the same device state that the application observed before being suspended, while retaining hardware acceleration capabilities.

```
__dText_c20[879]= {...}; __dText_sm20_c20[229]= {...};
__dText_c10[898]= {...}; __dText_sm10_c10[233]= {...};

__cudaFatPtxEntry __ptxEntries[] = {{"compute_20",__dText_c20},
                                    {"compute_10",__dText_c10},{0,0}};
__cudaFatCubinEntry __cubinEntries[] = {{0,0}};
__cudaFatDebugEntry __debugEntries0 = {0, 0, 0, 0} ;
__cudaFatElfEntry __elfEntries0 = {0, 0, 0, 0} ;
__cudaFatElfEntry __elfEntries1 = {"sm_10@compute_10",__dText_sm10_c10,
                                   &__elfEntries0, sizeof(__dText_sm10_c10)};
__cudaFatElfEntry __elfEntries2 = {"sm_20@compute_20",__dText_sm20_c20,
                                   &__elfEntries1, sizeof(__dText_sm20_c20)};

__fatDeviceText = {0x1ee55a01,0x00000004,0x2e00b786,"310b66c2f6ce6ed2","matrixMul.cu",
" ",__ptxEntries,__cubinEntries,&__debugEntries0,0,0,0,0,0,0xa359ea02,&__elfEntries2};

__cudaFatCubinHandle=__cudaRegisterFatBinary(&__fatDeviceText);
__cudaRegisterFunction(__cudaFatCubinHandle,matrixMul,"_Z9matrixMulPfS_S_ii",
                       "_Z9matrixMulPfS_S_ii",-1,0,0,0,0,0);

cudaGetDeviceCount(&device_count);
cudaGetDeviceProperties(&deviceProp,0);
cudaSetDevice(0);

int devID; cudaDeviceProp props;
cudaGetDevice(&devID); cudaGetDeviceProperties(&props,devID);

float* h_A=malloc(51200);float* h_B=malloc(25600);float* h_C=malloc(51200);
randomInit(h_A,12800);randomInit(h_B,6400);

float* d_A; float* d_B; float* d_C;
cudaMalloc(&d_A,51200); cudaMalloc(&d_B,25600);
cudaMemcpy(d_A,h_A,51200,cudaMemcpyHostToDevice);
cudaMemcpy(d_B,h_B,25600,cudaMemcpyHostToDevice);
cudaMalloc(&d_C,51200);

dim3threads(16,16);dim3grid(5,10);
cudaConfigureCall(grid,threads)?((void)0):matrixMul(d_C,d_A,d_B,80,80);

cudaThreadSynchronize();
for (intj=0; j<30; j++)
  cudaConfigureCall(grid,threads)?((void)0):matrixMul(d_C,d_A,d_B,80,80);

struct T20{float* par0;float* par1;float* par2;int par3;int par4;int dummy_field;};

void matrixMul(float* par0,float* par1,float* par2,int par3,int par4){
  struct T20 *T28 = 0;
  cudaSetupArgument((void*)(char*)&par0, sizeof(par0), (size_t)&T28->par0);
                                  ...
  cudaSetupArgument((void*)(char*)&par4, sizeof(par4), (size_t)&T28->par4);
  char *__f = ((char*)((void(*)(float *, float *, float *, int, int))matrixMul));
  cudaLaunch(((char*)((void(*)(float *, float *, float *, int, int))matrixMul)));
}

cudaThreadSynchronize();
cudaMemcpy(h_C,d_C,51200,cudaMemcpyDeviceToHost);
cudaFree(d_A);cudaFree(d_B);cudaFree(d_C);cudaThreadExit();
__cudaUnregisterFatBinary(__cudaFatCubinHandle);
```

Fig. 5. Code snippet of matrixMul.cu.c.

## 5 Illustrating Example

This section presents an illustrating example of vCUDA. In Fig. 5, we illustrate the internals of vCUDA by walking the reader through a simple example from CUDA 3.2 SDK. Matrix multiplication is an application in the official SDK, which implements matrix multiplication, namely, $C = A \times B$. The matrix dimensions are $80 \times 160$, $80 \times 80$, and $80 \times 160$ in A, B, and C, respectively, as the default configuration. According to the workflow of CUDA compilation, the source file (matrixMul.cu) coded in the extended CUDA language is compiled by NVCC to a normal ANSI C file (matrixMul.cu.c). Its main part is presented in Fig. 5 (where the *randomInit()* is an auxiliary function defined to initialize the memory with random values):

**Virtualizing CUDA objects**. Fifteen CUDA APIs are involved in this example (for simplicity, the *cudaGetErrorString()* and *cudaGetLastError()* are not counted), each with different kinds of arguments. Some of the arguments can be directly wrapped, including immediate values (*51200, 0, sizeof(par*)*), and predefined constants (*cudaMemcpyDeviceToHost, cudaMemcpyHostToDevice*).

Fig. 5 shows how the CUDA *kernel* is organized. First, the entire device code in matrixMul.cu (INCLUDE matrixMul_kernel.cu) is compiled to plain text *__dText_** by NVCC. In our case, two different kinds of device text are present: the PTX file that contains the ASCII instruction sets and the CUBIN file organized as an ELF image. The options "-gencode=arch=compute_10" and "-gencode=arch=compute_20" are the default settings for NVCC. Therefore, two device texts are generated for each format.

All the device texts are combined to *__fatDeviceText*, and are registered to a handler *__cudaFatCubinHandle*. A device stub *matrixMul* is then bound to this handle through *__cudaRegisterFunction()*. Finally, the stub is launched by *cudaLaunch()* as the reference to the actual CUDA *kernel*. These operations (register, bind, and launch) correspond to three explicit CUDA APIs that can be exploited by vCUDA to realize virtualization. vCUDA stores their relationship as two tuples (text to handle and handle to stub) in the VOL, and dereferences them in the vCUDA stub when *cudaLaunch()* is invoked. In this example, *__cudaRegisterTexture()*, *__cudaRegisterSurface()*, and *__cudaRegisterVar()* are not present.

The device memory pointers *d_A*, *d_B*, and *d_C* are virtualized with the help of the VOL, as described in Section 4.2. When *cudaMalloc()* is invoked, the size and base address of the device memory object are stored in the VOL, and is removed from the list when calling *cudaFree()*. If any device memory pointers are used as the arguments of subsequent CUDA API (typically *cudaMemcpy()*), the pointers are dereferenced in the vCUDA stub according to the information saved in the VOL.

The server/client memory referenced by pointers must be reproduced in another address space either by memory transfer or sharing. All these pointers can be divided into five main categories: C fashion strings (*__dText_c20*, *__dText_c10*), array (*__dText_sm20_c20*, *__dText_sm10_c10*), structure pointers *&__fatDeviceText*, *& deviceProp*, *grid*, *threads*, heap pointers *h_A*, *h_B*, *h_C*, and stack pointers *& par0*, *& par1*, *& par2*, *& par3*, *& par4*. We can obtain the length of memory objects that these pointers indicate at either compiling time or runtime. First, the size of the structures is obtained from C header files at the compiling time. Second, the length of string is easily determined at runtime because the string always ends with a NULL. Third, the structure *__cudaFatElfEntry* shows the size of *__dText_sm{*}_c{*}*. Fourth, *cudaMemcpy()* reveals the size of heap content through its third argument. Finally, *cudaSetupArgument()* introduces its first argument, a stack pointer, by its second argument.

**Identify CUDA objects**. In this example, the programmer defines five arguments for *cudaSetupArgument()*, of which two are immediate values and three are device memory pointers. As previously mentioned, vCUDA registers these pointers in the VOL beforehand when they are assigned by *cudaMalloc()*. When *cudaSetupArgument()* is called, each argument is searched in the VOL. When one of list entry matches the value, the appropriate argument is treated as an internal object pointer and is expanded.

**API virtualization without remote call**. Device query APIs (*cudaGetDeviceCount()*, *cudaGetDeviceProperties()*, *cudaSetDevice()*), and *cudaGetDevice()*) are replaced by local ones, which directly respond to the application request from the vGPU component without issuing a remote call. When initializing vCUDA, the properties of virtual GPUs are customized according to the physical GPU configuration.

**VMRPC optimization**. In this example, the data of matrix need to be transferred from the client to the server, which rely on the matrix dimensions. When the matrix becomes bigger, more time will be spent on the data transfer. For data-intensive applications, the loading time for large memory object can be quite long and may seriously hinder the performance of RPC. This is the main reason vCUDA exhibits poor performance in TRANSMISSION mode. To highlight the essential features of VMRPC, we provide a detailed description about the virtualization of *cudaMemcpy()* in SHARE mode. There are three *cudaMemcpy()* in this example, two for the input matrices A and B, and one for output matrix C. For all *cudaMemcpy()*, VMRPC allocates memory space for these calls in transfer channel (*shared heap*) according to the third argument of these calls. VMRPC then copies the data of matrix to or from the transfer channel. Because the transfer channel is already shared between the server and the client, the expensive data-transfer overhead is replaced by the time consumption of VMRPC initialization and local memory copies, which is far less than the cost in TRANSMISSION mode in most cases.

**Lazy RPC**. The *__cudaRegisterFatBinary()* is an instant API because its return value, CUBIN handler, is later referenced by subsequent APIs. The subsequent APIs can be delayed until a *cudaMalloc()* is called because its return value (device memory address) changes the CUDA runtime state in the client. In this example, the two *cudaMemcpy(HostToDevice)* could be integrated into one remote call in logic because the host memory spaces they occupy (the content of the matrices A and B) are separated from each other. However, as previously mentioned, vCUDA treats them as two instant APIs. *cudaConfigureCall()*, *cudaSetupArgument()*, and *cudaLaunch()* are batched and launched until the next instant API *cudaMemcpy(DeviceToHost)* is called. The three *cudaFree()*are lazy APIs. The last *__cudaUnregisterFatBinary()* announces the end of the CUDA *kernel* execution in our example, whereas, in other CUDA applications, this API can be launched multiple times corresponding to multiple *kernels*, and can be delayed until the last one is called.

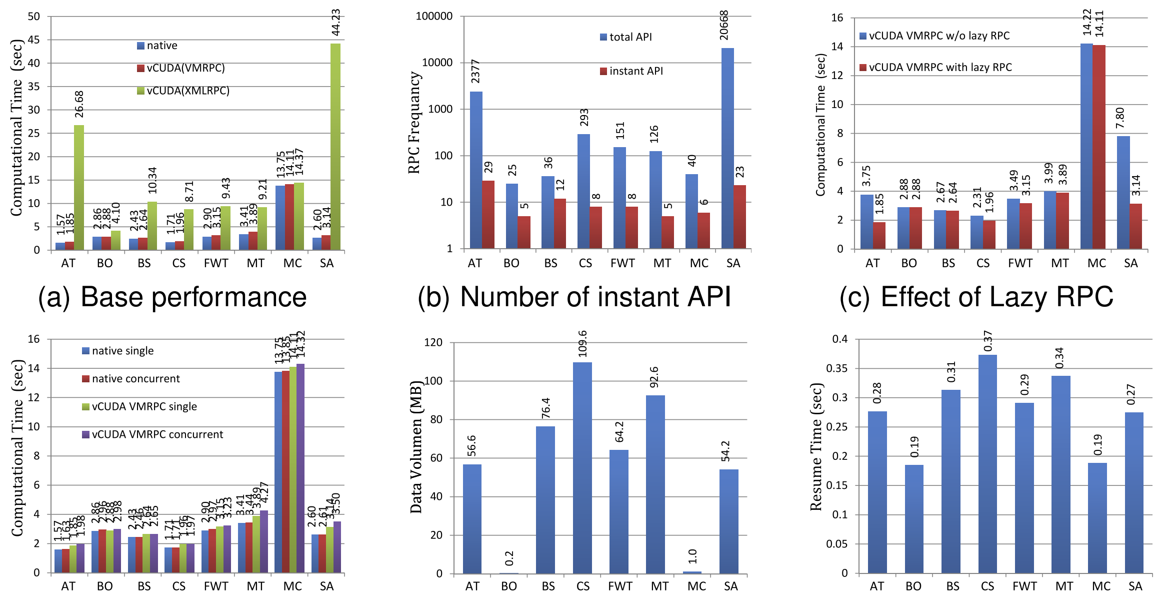
TABLE 1
Statistics of Benchmark Applications

| | Number of API calls | Number of kernel launch | GPU RAM (MB) | Data Volume (MB) |
|---|---|---|---|---|
| AlignedTypes (AT) | 2377 | 384 | 55.63 | 361.60 |
| BinomialOptions (BO) | 25 | 1 | 0 | 0.01 |
| BlackScholes (BS) | 36 | 1 | 76.29 | 76.29 |
| ConvolutionSeparable (CS) | 293 | 34 | 108.00 | 72.00 |
| FastWalshTransform (FWT) | 151 | 22 | 64.00 | 64.00 |
| MersenneTwister (MT) | 126 | 22 | 91.56 | 91.56 |
| MonteCarlo (MC) | 40 | 2 | 1.00 | 0.01 |
| Scan (SA) | 20668 | 3300 | 52.25 | 468.00 |

## 6 Evaluation

Whereas the previous sections present detailed technical descriptions of the vCUDA system, this section evaluates the efficiency of vCUDA using benchmarks selected from the official SDK of NVIDIA. The benchmarks range from simple data management to more complex Walsh Transform computation and MonteCarlo simulation. All benchmarks are from the standard CUDA SDK distribution (3.2) without any modification, except for the BlackScholes test. The iteration number of BlackScholes was set to 1 instead of 512 to meet the device capacity restriction of our testbed.

Table 1 shows the statistical characteristics of these benchmarks, such as the quantity of API calls, the amount of device memory they consume, and the data volume transferred from/to a GPU device.



(a) Base performance (b) Number of instant API (c) Effect of Lazy RPC

(d) Performance of concurrency (e) Data volume for S&R (f) Time of resume

Fig. 6. Performance evaluation of vCUDA.

These applications are evaluated according to the following criteria:

- **Performance**. How close is the vCUDA in providing the performance similar with that observed in an unvirtualized environment with GPU acceleration?
- **Lazy RPC**. How greatly can vCUDA boost overall performance by Lazy RPC mechanism?
- **Concurrency**. How well does vCUDA scale in supporting multiple CUDA applications concurrently running?
- **Suspend and resume**. How much of the CUDA state needs to be saved for a vCUDA suspend operation? What is the latency for resuming a suspended CUDA application?
- **Compatibility**. How compatible is vCUDA with a wide range of third-party applications aside from the examples distributed with CUDA SDK?

The following testbed was used for our evaluation: a machine was equipped with an Intel Xeon E5504 2 GHz processor and 4 GB memory. The graphics hardware was GTX470 of NVIDIA. The test machine ran Fedora 13 Linux distribution with the kernel version 2.6.33.3 as both hostOS and guestOS, and the official NVIDIA driver for Linux version 260.19.26. We used the CUDA toolkits and SDK 3.2, and chose KVM 0.14.0 as our VMM. All virtual machines were set up with 1 GB RAM, 10 G disk, and bridge mode network configuration. All the evaluation results are averaged across 20 runs.

## 6.1 Performance

We conducted the performance evaluation by comparing the execution time in virtual machine with that of native runs. All test programs were evaluated in three different configurations:

- **Native**. Every application has direct and exclusive access to the hardware and native CUDA drivers. vCUDA was not used in this case, which represents an upper bound of achievable performance for our experimental setup.
- **vCUDA with VMRPC**. Applications running in the guestOS leveraged the vCUDA to access real hardware device. All CUDA API calls were intercepted and redirected to the hostOS through VMRPC.
- **vCUDA with XMLRPC**. This is similar with the previous case, except that VMRPC is replaced with XMLRPC.

Fig. 6a depicts the results from running the benchmarks under the three configurations described above. The results show that the performance of vCUDA (VMRPC) is very close with native runs, whereas vCUDA (XMLRPC) incurs a significant overhead, especially for cases involving large data transfer, such as programs AT and SA. We can also observe that when the amount of data transferred is less, the closer the performance of vCUDA (XMLRPC) is to the native and vCUDA (VMRPC) runs, such as in the cases of BO and MC. This indicates that normal RPC system, such as XMRPC, is not efficient enough for high-capacity data transfer.

Fig. 6a also shows that all eight programs with vCUDA (VMRPC) caused performance degradation from 1 to 21 percent (an average of 11 percent), as compared with the overhead with vCUDA (XMLRPC) ranging from 43 to 1,600 percent, except for MC. The MC benchmark shows nearly the same performance in all the three cases because it spent much more time on the CPU than on the GPU. Based on this observation, the key bottleneck of vCUDA lies in the data representation and transfer, which is the main motivation for us to design and implement the VMRPC.

The overhead of vCUDA (VMRPC) can be attributed to several aspects: vCUDA library overhead (managing the local vGPU and maintaining the lazy API queue), vCUDA stub overhead (dispatching the remote task between the service thread and working thread, access validation, and managing the VOL), RPC overhead (initialization of

TABLE 2
Statistics of the Third-Party Applications

| | Number of API calls | GPU RAM | Data Volume | Native Time | vCUDA XMLRPC Time | vCUDA VMRPC Time |
|---|---|---|---|---|---|---|
| GPUmg | 89 | 294KB | 2448KB | 0.358s | 0.522s | 0.360s |
| estoreGPU | 32 | 983KB | 860KB | 0.418s | 0.564s | 0.435s |
| MRRR | 370 | 1893KB | 2053KB | 0.470s | 0.593s | 0.498s |
| MV | 31 | 15761KB | 61472KB | 0.654s | 3.648s | 0.710s |
| MP3encode | 94 | 1224KB | 391KB | 0.243s | 0.623s | 0.264s |

shared memory and memory copies), VMM overhead (world switching and VMM scheduling), and the absence of some CUDA features in vCUDA, such as the page-locked memory optimization. We leave further quantitative analysis regarding the overhead of vCUDA for future work.

### 6.2 Lazy RPC and Concurrency

Fig. 6b compares the frequencies of RPC call in two cases with lazy mode enabled and disabled, respectively. As it shows, the lazy RPC transmission mode significantly reduces the frequency of remote calls between two domains. Fig. 6c presents the performance of vCUDA (VMRPC) with and without lazy RPC. It proved that the lazy RPC mode improves the overall performance of vCUDA, especially for applications containing a large number of API calls, such as AT and SA.

To examine the ability of vCUDA in supporting concurrent execution of multiple applications, we compared the performance of two applications running concurrently in the native default mode of NVIDIA GPU with the performance of the same two applications running concurrently in vCUDA (VMRPC). In this experiment, each test program runs concurrently with the same reference application. The benchmark program BO was chosen as the reference application because it consumes less device resource and has successful in running concurrently with all other programs.

Fig. 6d presents the results of these experiments. The benchmarks with the unvirtualized configuration in all cases present good scalability, and the overheads for all applications are all below 4 percent (3.6 percent for AT, 3.5 percent for BO, 1.3 percent for BS, 0.1 percent for CS, 2.5 percent for FWT, 0.9 percent for MT, 0.7 percent for MC, and 0.2 percent for SA). In contrast, the counterparts in vCUDA (VMRPC) show obvious performance degradation. The overhead ranges from 0.5 to 11.5 percent. This phenomenon reflects the substantial difference between the default sharing mode of CUDA and the one-to-many mode of vCUDA. Although they both need to manage multiple CUDA threads and reschedule CUDA *kernels*, the CUDA runtime undoubtedly has more knowledge regarding the internals of CUDA framework, thereby showing better performance than vCUDA.

### 6.3 Suspend and Resume

When testing the S&R functionality of vCUDA, we first suspended the test program at arbitrary points in time, and then resumed the program to check if the resume operation was successful. We measured the size of CUDA state that must be saved to synchronize the vCUDA stub with the current state of the application and the duration needed to perform the entire resume operation. The results of these experiments are shown in Figs. 6e and 6f, respectively.

Note that all objects in the VOL have their own life cycles. When the suspend action occurs if the object is alive, it belongs to the state that needs to be saved. In most cases, the biggest data volume of the CUDA object is the device memory object, which is reflected in Fig. 6e.

The resume time in Fig. 6f is strongly dependent on the size of the suspended CUDA states, which can be as large as 109 MB for CS. The test program CS takes more time to perform the resume operation than others due to its larger data volume of CUDA states. In contrast, there is a nontrivial resume overhead for BO (0.19 s) and MC (0.19 s), even if their databases of CUDA states are very small. This can be explained by the constant overhead for vCUDA resume operation, such as the creation of GPU memory pool.

### 6.4 Compatibility

In order to verify the compatibility of vCUDA, we chose five applications from the CUDA zone [7]: mp3 lame encoder [30], molecular dynamics simulation with GPU [26], matrix-vector multiplication algorithm [11], storeGPU [1], and MRRR implementation in CUDA [24]. These applications were selected because none of them use 3D graphic API or drive API, which is not supported by vCUDA. Although CUDA supports the interoperability between CUDA and traditional graphics API, virtualization graphic interfaces such as OpenGL and Direct3D are beyond the scope of this paper.

All five third-party applications passed the test and returned the same results as in native executions. The details of these tests are presented in Table 2, which shows that when running in vCUDA, these applications exhibit similar performance characteristics as those discussed in Section 6.1. For example, the performance degradation of application MV in vCUDA (XMLRPC) is mainly due to the higher data volume transfer compared with other applications.

## 7 Related Work

According to the specific features and practical requirements, many existing systems intercept calls to the graphics library for various purposes. VirtualGL [34] virtualizes GLX to grant remote-rendering ability. WireGL [18] and its successor Chromium [17] intercept OpenGL to generate different outputs, such as distributed displays. Chromium provides a mechanism for implementing plug-in modules that alter the flow of GL commands, allowing the distribution of parallel rendering. HijackGL [29] uses the Chromium library in exploring new rendering styles. In VMM platform, this methodology is used to achieve 3D hardware acceleration in a VM. VMGL [23] deploys a fake stub in guestOS and redirects the OpenGL flow to hostOS. The Blink Project [15] intercepts OpenGL to multiplex 3D display in several client OS. Another main category is the tool in helping performance analysis and debugging. The ZAPdb OpenGL debugger of IBM [19] uses interception technique to aid in debugging OpenGL programs. The Graphics Performance Toolkit of Intel [20] uses a similar method to instrument graphics application performance.

As the GPGPU library, rCUDA [9], GViM [14], and gVirtuS [12] are three recent research projects on CUDA virtualization in GPU clusters and virtual machines. They all use an approach similar to vCUDA. The rCUDA framework creates virtual CUDA-compatible devices on those machines without a local GPU to enable a remote GPU-based acceleration, and communicate using the sockets API between the front end and back end. rCUDA can be used in the virtualization environment, but it requires the programmer to rewrite the CUDA applications to avoid the use of the CUDA C extensions, and requires to change the standard compile options to separate the host and device code into different files. GViM is a Xen-based system, allows virtual machines to access a GPU through XenStore between a front end executed on the VM and a back end on the Xen Domain0. GViM requires modification to the guest VMs running on the virtualized platform, a custom kernel module must be inserted to the guestOS. gVirtuS is a transparent and VMM independent framework to allow an instanced virtual machine to access GPUs. It implements various communicator components (TCP/IP, VMCI for VMware, VMSocket for KVM) to connect the front end in guestOS and back end in hostOS. The design of communicator seems similar to the transfer channel in VMRPC, but we believe RPC-based optimization that exploits the inherent semantics of the application, such as data presentation, would perform better than those merely optimizing data-transfer channels, as proven in our previous work [6]. None of the three projects support suspend/resume functionality of CUDA applications.

High-level middleware- and language-based VMs have been studied and used for high-performance computing, such as HPVM [5] and Java. In [16], the authors proposed a framework for HPC applications in VMs, addressing 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 in distributing and managing VMs in large-scale clusters with scalable VM image management schemes.

## 8 Conclusions

In this paper, we have proposed vCUDA, a GPGPU high-performance computing solution for virtual machines, which allows applications executing within virtual machines to leverage hardware acceleration. This can be beneficial to the performance of a class of high-performance computing applications. We have explained how to transparently access graphics hardware in VMs 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, nonvirtualized environment.

## Acknowledgments

The authors would like to thank the anonymous reviewers for numerous valuable comments. 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, 60703096, and 61070057, and the Fundamental Research Funds for the Central Universities. The corresponding author is Hao Chen.

## References

[1] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems," *Proc. Int'l Symp. High Performance Distributed Computing (HPDC '08)*, June 2008.

[2] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L.N. Bairavasundaram, K. Voruganti, and G.R. Goodson, "Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances," *Proc. Conf. USENIX Ann. Technical Conf. (USENIX '09)*, June 2009.

[3] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "User-Level Interprocess Communication for Shared Memory Multiprocessors," *ACM Trans. Computer Systems*, vol. 9, no. 2, pp. 175-198, May 1991.

[4] 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," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03)*, pp. 164-177, Oct. 2003.

[5] A. Chien et al. "Design and Evaluation of an HPVM-Based Windows NT Supercomputer," *The Int'l J. High Performance Computing Applications*, vol. 13, no. 3, pp. 201-219, 1999.

[6] H. Chen, L. Shi, and J. Sun, "VMRPC: A High Efficiency and Light Weight RPC System for Virtual Machines," *Proc. 18th IEEE Int'l Workshop Quality of Service (IWQoS '10)*, 2010.

[7] CUDA: Compute Unified Device Architecture. http://www.nvidia.com/object/cuda_home_new.html, 2010.

[8] M. Dowty and J. Sugerman, "GPU Virtualization on VMware's Hosted I/O Architecture," *SIGOPS Operating Systems Rev.*, vol. 43, pp. 73-82, July 2009.

[9] J. Duato, A. Pena, F. Silla, R. Mayo, and E.S. Quintana, "rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters," *Proc. Int'l Conf. High Performance omputing and Simulation (HPCS '10)*, pp. 224-231, July 2010.

[10] 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," *Proc. Fifth Symp. Operating Systems design and Implementation (OSDI '02)*, Dec. 2002.

[11] N. Fujimoto, "Faster Matrix-Vector Multiplication on GeForce 8800GTX," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS '08)*, Apr. 2008.

[12] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," *Proc. Int'l Euro-Par Conf. Parallel Processing*, pp. 379-391, 2010.

[13] "General Purpose Programming on GPUs: What programming APIs exist for GPGPU," *GPGPU* http://www.gpgpu.org/w/index.php/FAQ# 2011.

[14] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "Gvim: Gpu-Accelerated Virtual Machines," *Proc. ACM Workshop System-Level Virtualization for High Performance Computing (HPCVirt '09)*, pp. 17-24, 2009.

[15] J.G. Hansen, "Blink: 3d Display Multiplexing for Virtualized Applications," technical report, DIKU - Univ. of Copenhagen, http://www.diku.dk/jacobg/pubs/blinktechreport.pdf, Jan. 2006.

[16] W. Huang, J. Liu, B. Abali, and D.K. Panda, "A Case for High Performance Computing with Virtual Machines," *Proc. 20th Ann. Int'l Conf. Supercomputing*, June 2006.

[17] 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," *Proc. 29th Ann. Conf. Computer Graphics and Interactive Techniques,* pp. 693-702, 2002.

[18] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan, "WireGL: A Scalable Graphics System for Clusters," *Proc. ACM SIGGRAPH,* pp. 129-140, Aug. 2001.

[19] IBM's ZAPdb OpenGL Debugger, Computer Software, 1998.

[20] Intel Graphics Performance Toolkit. Computer Software.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor," *Proc. Linux Symp.,* pp. 225-230, 2007.

[22] K. Kim, C. Kim, S.I. Jung, H.S. Shin, and J.S. Kim, "Inter-Domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen," *Proc. Int'l Conf. Virtual Execution Environments (VEE '08),* pp. 11-20, Mar. 2008.

[23] H.A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de La-ra, "VMM-Independent Graphics Acceleration," *Proc. Int'l Conf. Virtual Execution Environments (VEE '07),* June 2007.

[24] C. Lessig, "An Implementation of the MRRR Algorithm on a Data-Parallel Coprocessor," technical report, Univ. of Toronto, 2008.

[25] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," *Proc. Sixth Symp. Operating Systems Design and Implementation (OSDI '04),* Dec. 2004.

[26] MDGPU, http://www.amolf.nl/~vanmeel/mdgpu/about.html, 2011.

[27] IVSHMEM, http://www.linux-kvm.org/wiki/images/e/e8/0.11.Nahanni-CamMacdonell.pdf, 2011.

[28] A. Menon et al. "Diagnosing Performance Overheads in the Xen Virtual Machine Environment," *Proc. First ACM/USENIX Int'l Conf. Virtual Execution Environments (VEE '05),* pp. 13-23, June 2005.

[29] A. Mohr and M. Gleicher, "HijackGL: Reconstructing from Streams for Stylized Rendering," *Proc. Second Int'l Symp. Non-Photorealistic Animation and Rendering,* 2002.

[30] MP3 LAME Encoder (Nvidia's CUDA Contest), http://cudacontest.nvidia.com, 2010.

[31] 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," *J. Computer Graphics Forum,* vol. 26, pp. 21-51, 2007.

[32] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU Accelerated High Performance Computing in Virtual Machines," *Proc. Int'l Symp. Parallel and Distributed Processing (IPDPS '09),* pp. 1-11, May 2009.

[33] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," *Proc. 12th Int'l Conf. Architectural Support for Programming guages and Operating Systems (ASPLOS),* 2006.

[34] VirtualGL, http://virtualgl.sourceforge.net/, 2011.

[35] VMCHANNEL, http://www.linux-kvm.org/page/VMchannel_Requirements, 2011.

[36] VMware Workstation, http://www.vmware.com/products/ws/, 2011.

[37] J. Wang, K. Wright, and K. Gopalan, "XenLoop: A Transparent High Performance Inter-VM Network Loopback," *Proc. 17th Int'l Symp. High Performance Distributed Computing (HPDC '08),* pp. 109-118, June 2008.

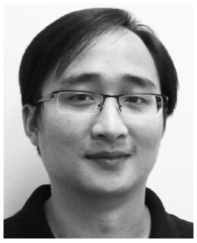
[38] P. Willmann, J. Shafer, D. Carr, A. Menon, and S. Rixner, "Concurrent Direct Network Access for Virtual Machine Monitors," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture (HPCA '07),* pp. 306-317, 2007.

[39] Xen VGA Passthrough, http://wiki.xensource.com/xenwiki/XenVGAPassthrough, 2011.

[40] XMLRPC, http://www.xmlrpc.com/, 2011.

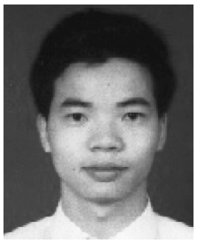
[41] X. Zhang, S. McIntosh, P. Rohatgi, and J.L. Griffin, "Xensocket: A High-Throughput Interdomain Transport for Virtual Machines," *Proc. Eighth ACM/IFIP/USENIX Int'l Conf. Middleware,* pp. 184-203, Nov. 2007.

**Lin Shi** is working toward the PhD degree at the School of Computer and Communication, Hunan University, China. His research interests include virtual machines and GPGPU computing. He is a student member of the IEEE.



**Hao Chen** received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2005. He is an associate professor at the School of Computer and Communication, Hunan University, China. His research interests include virtual machines, operating systems, distributed and parallel computing, and security. He is a member of the IEEE.

**Jianhua Sun** received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2005. She is an associate professor at the School of Computer and Communication, Hunan University, China. Her research interests are in security and operating systems.



**Kenli Li** received the BS degree in mathematics from Central South University, China, in 2000, and the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2003. He has been a visiting scholar at the University of Illinois at Champaign and Urbana from 2004 to 2005. He is now a professor of computer science and technology at Hunan University. He is a senior member of the CCF. His major research contains parallel computing, grid and cloud computing, and DNA computer.