Virtualizing GPUs for CUDA based HPC applications

Rangeen Basu, Siddharth Sharma

Abstract

Modern graphics processing units (GPUs) are being widely adopted as application accelerators in HPC owing to their massive floating point compute power that can be leveraged by data parallel algorithms. Consequently, need for virtualization of GPU resources has grown rapidly to provide on-demand efficient resource sharing. In this work, we present a detailed literature review of various GPU virtualization techniques. We then provide design details of a virtualization technique for CUDA applications, that we developed, based on API interception and redirection via Remote procedure calls. We demonstrate the viability of our virtualization framework by testing the performance on CUDA SDK examples and a set of micro-benchmarks.

Keywords

GPU Virtualization, CUDA, Call forwarding

1. INTRODUCTION

The rapid growth in virtualization technologies has resulted in a paradigm shift in computing trends towards on-demand computing, cloud computing and Software as a Service models. Smaller institutions lease compute resources from cloud vendors like Microsoft, Amazon and Google which in turn use virtualization to enable efficient resource sharing among different applications/users in order to maintain high overall system utilization.

The Graphics Processing Units (GPU) have played an essential role in providing rich visual experiences to users through graphics and physics acceleration. GPUs are needed to support graphics intensive applications like games, CAD tools and visualization software. Each of these are now being deployed onto the cloud as part of Software as a Service thus requiring GPU compute in cloud and making GPU virtualization an important technology.

More recently, GPUs have also been used to accelerate a wide range of applications beyond graphics such as bioinformatics, physics and chemistry, business forecasting and medical imaging applications [5], [8]. Programmers generally exploit GPUs by offloading compute intensive parallelizable parts of the application onto these devices. Newly developed programming models like CUDA and OpenCl provide libraries and programming framework (including device code compilers) to directly interact with the device making them easily accessible to the programmer. Widespread use of GPUs for general compute has resulted into further increase in demand for GPU compute capability in virtualized platforms offered by cloud vendors.

Due to multifold benefits in terms of performance and power savings GPUs are also being deployed in datacenters [11]. GPUs are being used to tackle big data analytics and advanced search for both consumer and commercial applications. Companies such as Shazam, Salesforce.com and Cortexica use GPUs to process massive data sets and complex algorithms for audio search, big data analytics and image recognition [10]. Since a configuration that dedicates one GPU per node on an HPC server cluster may not be able to fully utilize the device, GPU nodes are generally shared among multiple CPU nodes. Configurations with lesser number of GPUs require some sort of global scheduler and task manager to deploy work onto GPUs. This may be handled by decoupling execution hardware from the software via virtualization making GPU virtualization a very important building block of an efficient GPU compute server.

The focus of our work is on exploring GPU virtualization techniques for HPC clusters. As most GPU based HPC applications use CUDA framework [6] we further limit our discussion and implementation to virtualizing GPU workloads that use CUDA. In the next section, we discuss two middleware techniques that virtualize GPUs for CUDA applications , namely, rCUDA [2] and gVirtuS [3]. We also discuss PCI PassThrough technique which is commonly offered by Citrix Xen [12] and VMwares vSphere [9]. Section 3 describes the implementation of our GPU virtualization solution which is motivated by vCUDA [7]. Section 4 presents results and evaluations. We provide conclusions in section 5 and future work in section 6.

2. GPU VIRTUALIZATION

Despite high demand, GPU virtualization technology is still in its nascent stage. GPU architecture traditionally provided limited support for virtualization and GPU vendors remain highly secretive of their proprietary architecture leaving limited room for development of sophisticated virtualization techniques.

Currently available GPU virtualization techniques can be classified into two groups [1] - Front end and Back end techniques. Rest of the section describes these techniques with examples.

2.1 Back End Techniques

Backend virtualization techniques are built around device-dedication logic in which driver stack remains on the guest OS and the guest has direct exclusive control over the GPU device. A common example of this technique is PCI Passthrough provided by Xen and vSphere to allow GPU rendering capability for virtual desktops. PCI pass through techniques utilize IOMMU support provided by Intel VT-d [4] to dedicate GPU devices to guest machines. Since guest VM has complete control over the device, this techniques does not allow device sharing and multiplexing.

2.2 Front-end techniques

This class of virtualization technique is also known as middle-ware technique since it has a middle-ware sitting between guest application and host/VMM managing communication related to the GPU device. GPU device driver is located on the VMM or host OS and VMs access the GPU by remote call forwarding or device emulation. This class of virtualization supports multiplexing and sharing of device among multiple VMs.

2.2.0.1 rCUDA.

rCUDA [2] virtualizes and manages load on multi-gpus installed on a cluster and provides remote use of CUDA runtime APIs. The framework is based on a client server architecture consisting of client and server side middlewares. Client middle-ware constitutes a wrapper library that encapsulates the original CUDA runtime. The wrapper functions implement remote call forwarding functionality for each runtime function in the CUDA library. The functions forwarded by client rCuda wrapper are handled by server middle-ware. The server middle-ware consists of a set of worker threads each controlling one GPU device and a variable number of client service threads which listen to TCP ports for client side requests and services them. GPU sharing is accomplished by spawning different server process for each remote execution over a new GPU context. GPU virtualization is thus provided via API remoting.

2.2.0.2 gVirtuS.

The GPU virtualization service (gVirtuS) [3] is also based on API call forwarding with client middle-ware acting as call interceptor and server middle-ware servicing client side GPU runtime library calls. gVirtuS, however, is designed to accommodate a pluggable communication component which is independent of the server or client middleware allowing it to use efficient communicators if available to reduce remote call forwarding overheads.

In the next section we describe our implementation of a front-end virtualization technique via call forwarding which is essentially a variant of vCUDA. We start by providing an over-view of vCUDA virtualization technique and describe our implementation in later subsections.

3. IMPLEMENTATION DETAILS

3.1 General overview

The VCUDA system has a client server architecture that virtualizes the CUDA run time API. The client, which resides in the guest VM, intercepts the CUDA API and redirects them to the server. The VCUDA server, that is part of the VMM (host), manages the physical GPU device and also ensures resource sharing between multiple clients. For successful remote execution of CUDA programs, certain ordering semantics need to be strictly followed. CUDA has a strictly ordered execution model and does not guarantee data integrity if ordering is violated. Also, the VCUDA system needs to keep track of certain states and proper mapping of these states from client to server is necessary. These are tracked in special map tables by both the client and the server.

3.2 Client library

The client side consists of a custom shared library and a XMLRPC client stack. The library replaces the regular CUDA runtime library (libcudart.so) on the guest system. This library implements the CUDA runtime API and redirects the calls and data to the GPU server using appropriate remote function calls. The XMLRPC client stack enables remote function calls and provides XML encoding and XML-RPC protocol. On program start, the client sends a connect request to the master server and if the connection is successful, it receives the port number for a specific server instance (described below). Rest of the program is executed using this server instance. On program exit, the client sends a disconnect request which also ends the corresponding server instance.

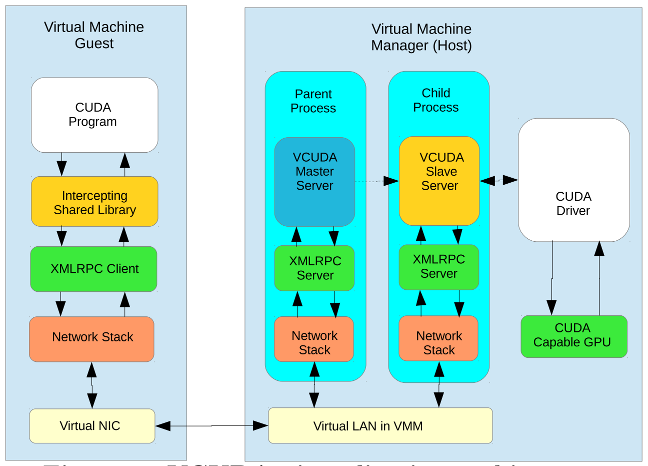

Figure 1: VCUDA virtualization architecture

3.3 Vcuda Server

The server has two main components, the XMLRPC server stack and the VCUDA server API implementation. The XMLRPC stack includes the web server and also the RPC handlers. These are responsible for communicating with the client using the XMLRPC protocol, XML decoding and passing on the arguments to the appropriate function. The RPCs, implemented in the VCUDA server, translate to appropriate CUDA runtime API calls. The CUDA run time library interfaces with the CUDA driver which manages execution on the GPU.

3.4 Multiplexing Guests

One important aspect of virtualization is resource sharing between multiple guest VMs. VCUDA enables sharing of a single physical GPU among multiple VMs. It achieves this by using different execution contexts for different clients in the CUDA driver. CUDA driver is capable of managing multiple contexts and efficiently switching between them, although preemption is not possible with current GPU technology. Multiple CUDA programs, that run as independent processes, can share the GPU resources. In our current implementation Figure 1, VCUDA server exploits this by spawning multiple server instances for each client. Each instance is a separate process and hence has a separate CUDA context. When the Master VCUDA instance receives a connect request from a new client, it spawns a child process that listens on a new TCP port, and sends back the port number to the client. Thereafter, the client only communicates with the new VCUDA instance at the new TCP port. A server process ends when the client sends a disconnect request. The CUDA driver efficiently multiplexes work from all contexts, blocking them when GPU resources are busy. This is not the best implementation and a multi-threaded implementation of the server Figure 2 will have better overlapping of tasks from multiple clients as opposed to the multi-process implementation, although not without added complexity. In case of the multi-threaded server, a dispatcher thread pushes work into the input queues and the scheduler thread is responsible for maintaining ordering and atomicity (certain API sequences). Since the entire server is a single process,

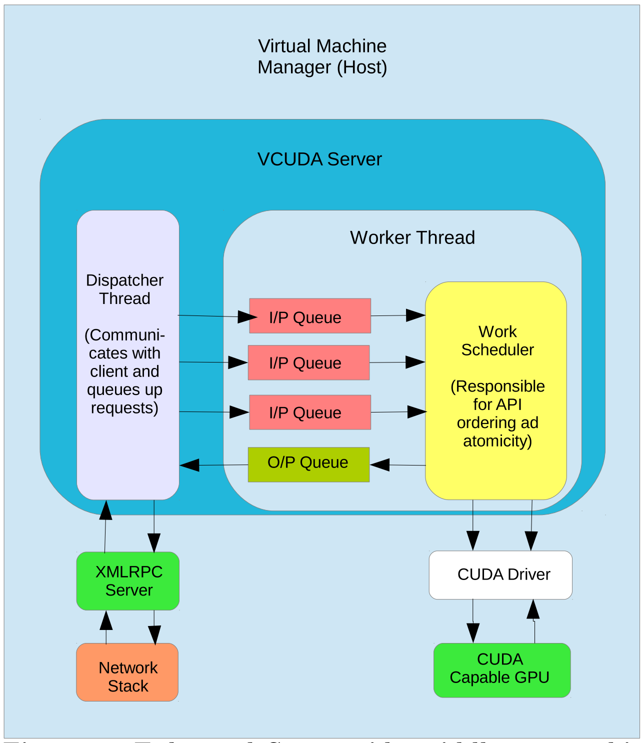

Figure 2: Enhanced Server side middleware architecture

and there is only one worker thread, everything runs in a single CUDA context and API calls from different clients are not blocked except in cases when an atomic API sequence is being run. For example, calls like cudaMalloc() from different clients can be overlapped whereas, a kernel launch sequence (cudaConfigureCall, cudaSetupArgument, cudaLaunch) must be atomic.

4. EXPERIMENTS AND DISCUSSIONS

Our vCuda implementation that forwards GPU related calls to a remote server may incur three types of overheads : RPC data-packing/unpacking overhead, network overhead, VM switching overhead for multiclient case. We perform xml-rpc call forwarding via base64 encoding byte stream data. This data exchange between server and client involves numerous procedure calls that perform data packing/unpacking, server-client handshakes and process of placing and reading data from TCP buffer. We call it *RPC-overhead*. *VM-switching* overhead arises when multiple VMs are accessing the GPU via vCUDA server. This is a standard virtualization overhead due to necessary context switches between VMs. The third class of overhead comes from network through which we exchange RPC data. The effect of this overhead as well as the RPC overhead highly depends on the amount of data being exchanged between client and server which can be approximated by the amount of data exchanged between GPU and the host for a given application.

4.1 Hardware Configuration

The platform used for hosting GPU machine and the server middle-ware consists of a system with one quad-core 2.27 GHz intel Xeon processor hosting a Fermi C2075 GPU. The

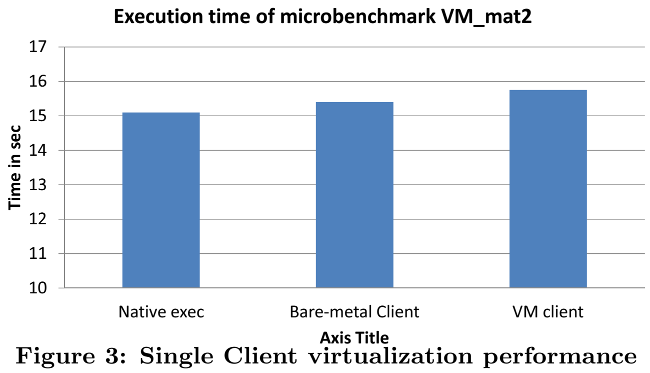

Figure 3: Single Client virtualization performance

system is running RHEL 5.5 with Nvidia driver 304.54 and CUDA toolkit 4.1. Remote procedure calls are implemented using xmlrpc-c library. [13] [14]. For client side we create multiple VMs running Ubuntu ontop of the host system with single core CPU capability.

4.2 Microbenchmarks

Vitalizing GPU not only provides GPU capability to remote guest systems but also allows efficient resource usage via context multiplexing. GPU can be most effectively and beneficially shared between multiple clients when they are running Heterogeneous HPC applications. This is because heterogeneous applications are characterized by the presence of sequential sections of GPU and CPU compute that often form a loop. Hence CPU compute of one client can effectively overlap on to GPU compute of some other allowing high overall compute throughput. Thus in order to test effective multiplexing capabilities of our virtualization technique we design a micro-benchmark application(VM_MAT2) that imitates heterogeneous application behavior with multiple sections of alternating GPU and CPU compute. We also create benchmark kernel with minimal GPU compute on large data-transfer that is transferred to and from the GPU device memory (VM_MAT1). This allows us to effectively characterize RPC-overhead and network latency overheads.

4.3 Single client remote execution

Fig 3 illustrates runtimes of native GPU execution (no virtualization) against three client side configurations. They are as follows:

- Bare-Metal Client **(BMC)** - In this configuration client program is running on the same host system and effectively only utilizes the client wrapper libraries instead of the actual CUDA libraries to access the GPU via server middleware. This configuration suffers from no VM-switch overhead and minimal network latency overhead allowing us to clearly identify RPC call overheads.
- Virtual machine client **(VMC)** - For this configuration, we create virtual machines on the same host system thus only adding VM switching overhead.
- Remote client **RC** - This configuration allows completely remote access to GPU with client sitting somewhere far away in the network allowing us to take network latencies linked with data transfer into account. Due to limited time and scarce real-world application

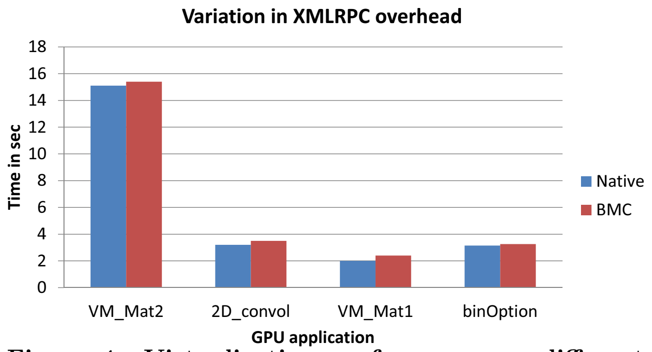

Figure 4: Virtualization performance on different apps showing RPC overhead

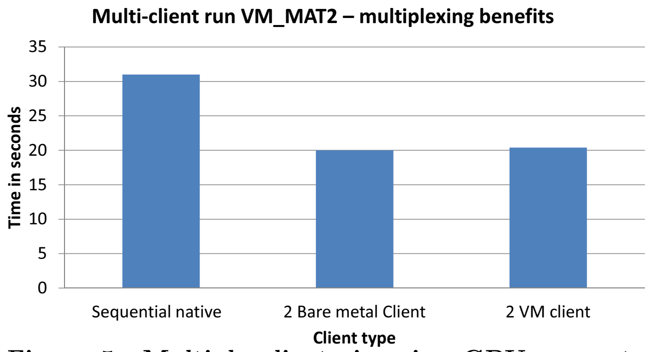

Figure 5: Multiple clients issueing GPU compute calls simultaneously

of this case (since HPC applications would most likely be running on VMs sitting close to host system and not on remotely placed thin client) we do not collect data for it.

The X-axis of fig 3 denotes different client configurations running our microbenchmark and Y-axis denotes execution time. We see about 2.6% RPC-overhead and about 3% VM switching overhead. While VM-switch overhead remains almost consistent RPC overhead vary with different applications. This can be seen in Fig 4 which compares native and BMC run-times of 3 more benchmarks including micro-benchmark VM_mat and two CUDA SDK benchmarks namely, 2d-Convolution and binomial Option sorting. The experiments showed xml overheads to vary from less than 1% to about 15%. It is worth mentioning that as we do not port all cuda runtime library functions, some functionalities of CUDA SDK applications are not supported. Binomial Option sorting originally uses constant memories which we change to standard device memory variables to allow this app to use our virtualization framework. Writing wrappers for all functions of CUDA runtime library is part of future work.

4.4 Multi client remote execution

We tested configurations where multiple clients simultaneously issue GPU compute library calls. As we spawn multiple server processes to allow effective driver level multiplexing, multiple GPU contexts are formed each queuing their GPU calls onto a single driver managed execution queue. The decision to spawn separate processes for each client allows us to effectively utilize well established context switching capabilities on GPU drivers. The Fig 5 compares time taken to complete simultaneous client runs of 2 BMC and VMC clients running VM_MAT2 benchmark against the time taken by two sequential runs of VM_MAT2 on native machine. The large time savings illustrates benefits obtained from effective multiplexing of GPU compute on one client with CPU compute on the other.

5. CONCLUSIONS

We proposed and implemented a framework for virtualizing GPU in a virtual machine environment. This is extremely useful for GPGPU applications hosted on clouds. This enables effective use of hardware resources and cost advantage for the data center operation. We presented a few examples of similar implementations and explained our implementation in detail. Our current implementation transparently enables remote execution of CUDA programs without changing the programmer's view or tool flow. Results obtained from experiments using micro- benchmarks show very little overhead in a virtualized environment. This infrastructure is also capable of handling remote execution on GPU clusters (Network attached GPUs) and with slight modification, will be capable of providing GPGPU as a service on the cloud.

6. FUTURE WORK

Future work on this can involve supporting other high bandwidth and lower latency transport mechanisms such as RDMA. Implementing the entire CUDA runtime API is also very important to support a broad spectrum of CUDA applications. Another very important enhancement could be implementing multi-GPU support and introduction of fault tolerance. New GPU architectures (eg. Kepler) supports simultaneous execution of multiple kernels and VCUDA can take advantage of such technology with some modifications.

7. REFERENCES

[1] M. Dowty and J. Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[2] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. Quintana-Orti. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231. IEEE, 2010.

[3] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing*, pages 379–391. Springer, 2010.

[4] R. Hiremane. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine*, 4(10), 2007.

[5] J. Nickolls and W. J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[6] C. Nvidia. Nvidia cuda programming guide, 2011.

[7] L. Shi, H. Chen, J. Sun, and K. Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on*, 61(6):804–816, 2012.

[8] G. M. Striemer and A. Akoglu. Sequence alignment with gpu: Performance and design challenges. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[9] Web. Gpu computing in a vm, http://blogs.vmware.com/performance/2011/10/gpgpu-computing-in-a-vm.html, 2011.

[10] Web. http://www.datacenterknowledge.com/archives/2013/03/22/nvidia -conference-gpu-can-power-big-data-analytics, 2011.

[11] Web. Top 500 super computer sites webpage, http://www.top500.org, 2011.

[12] Web. Xen pci-passthrough, http://wiki.xen.org/wiki/xen_pci_passthrough, 2011.

[13] Web. Xml-rpc-c documentation, http://xmlrpc-c.sourceforge.net/doc/, 2011.

[14] Web. Xml-rpc source code, http://sourceforge.net/projects/xmlrpc-c/, 2011.