

GViM: GPU-accelerated Virtual Machines

Vishakha Gupta, Ada Gavrilovska,
Karsten Schwan, Harshvardhan Kharche

Georgia Institute of Technology
vishakha,ada,schwan,hkharache3@cc.gatech.edu

Niraj Tolia, Vanish Talwar,
Parthasarathy Ranganathan

HP Labs, Palo Alto
firstname.lastname@hp.com

Abstract

The use of virtualization to abstract underlying hardware can aid in sharing such resources and in efficiently managing their use by high performance applications. Unfortunately, virtualization also prevents efficient access to accelerators, such as Graphics Processing Units (GPUs), that have become critical components in the design and architecture of HPC systems. Supporting General Purpose computing on GPUs (GPGPU) with accelerators from different vendors presents significant challenges due to proprietary programming models, heterogeneity, and the need to share accelerator resources between different Virtual Machines (VMs).

To address this problem, this paper presents GViM, a system designed for virtualizing and managing the resources of a general purpose system accelerated by graphics processors. Using the NVIDIA GPU as an example, we discuss how such accelerators can be virtualized without additional hardware support and describe the basic extensions needed for resource management. Our evaluation with a Xen-based implementation of GViM demonstrate efficiency and flexibility in system usage coupled with only small performance penalties for the virtualized vs. non-virtualized solutions.

General Terms GPGPU, GViM

Keywords amorphous access, split driver model

1. Introduction

Current trends in the processor industry indicate that future platforms will not merely exhibit an increased number of cores, but in order to better meet market power/performance requirements, will also rely on specialized cores – accelerators, better suited for execution of various components of the common software stacks – such as for graphics [26; 3], crypto and security operations [13], commu-

nications or computationally intensive operations such as those needed in scientific HPC codes or financial applications [29; 11]. Such tightly coupled heterogeneous many-core systems have long been present in the high performance community [6; 2], but current processor and interconnect technologies create opportunities for performance levels traditionally reserved for the HPC domain only, to be supported on commodity platforms. This is particularly true for many-core systems with graphics accelerators, and clusters built with such platforms have already started to penetrate the ranks of the world’s fastest systems [28; 1].

Concurrently, virtualization technology is making significant impact on how modern computational resources are used and managed. In the HPC domain, significant evidence points out the feasibility of lightweight, efficient virtualization approaches suitable for high end applications [22], its utility for capacity systems and HPC Grid environments [23], and the benefits it can provide with respect to improved portability and reduced development and management costs [9]. Furthermore, for HPC platforms used in application domains like finance, transportation, gaming, etc. (see Top500 list for diversity of application areas [1]), virtualization is becoming a de facto standard technology. This is particularly true of the banking sector, for which some estimate that the top 10 US banks have three times the number of processors as the top 10 supercomputers.

Our work is exploring efficient virtualization mechanisms for tightly coupled heterogeneous manycore systems, such as those (to be) used in HPC environments. Specifically, we focus on platforms with specialized graphics accelerators, and on such platforms, we are seeking to efficiently execute virtual machines (VMs) that run applications with components targeted for execution on GPUs. These GPU components are referred to as kernels in the rest of this paper.

Our approach, GViM, builds on existing virtualization solutions by integrating novel mechanisms for improved support of GPU-accelerated Virtual Machines (GViM). In comparison to prior work that has begun to consider the virtualization of GPU resources [16], our novel contributions address these platforms’ performance and flexibility needs, such as those present in the high performance community:

- *Improved programmability and portability* – In comparison to prior work, GViM virtualizes the graphics accelerator at the level of abstraction familiar to programmers, leveraging the CUDA APIs [20] and their open source counterparts [14]. This not only makes it easier to run and port standard applications, but it also relieves HPC application programmers of concerns about the physical positioning of accelerators and about driver and accelerator hardware versions. Deploying GViM requires minimal modification to the guest VMs running on the virtualized HPC platform.
- *Efficient accelerator virtualization* – GViM-based GPU virtualization offers low overheads and is competitive with kernels running in guest OSs that have direct access to accelerator resources. Attaining such high performance involves the careful management of the memory and device resources used by GPUs.
- *Coordinated resource management* – Using the GViM environment makes it easier for applications to ignore the issues related to efficiently sharing GPU resources. This is achieved by integrating methods into GViM for managing an application’s joint use of general purpose and accelerator cores. This paper establishes the importance of coordinated resource management for general purpose and graphics cores.

For large-scale parallel machines, the long-term goal of our work is to present ‘amorphous’ images of machine resources to applications, where VMs with special needs (e.g., the need to run GPU kernels) are deployed onto resources suitable and available for running them, without undue programmer involvement, and where multiple parallel applications efficiently share underlying heterogeneous computing platforms. The methods used to attain this goal combine the virtualization of machine resources with the active management of their use. Applications targeted by our work include compute-intensive physical simulations sharing HPC resources, high performance codes such as weather forecasting and financial codes, and applications requiring both computational and enterprise services like next generation web applications. By freely and dynamically using and combining accelerators with general purpose processing cores via VMs configured with the appropriate software support for GPU access, GViM creates rich opportunities for runtime resource consolidation and management (e.g., for power savings [19] and increased reliability [15]).

In Section 2, we describe the general architecture of virtualized GPGPU platforms. In Section 3, we discuss the virtualization of such platforms in detail. This is followed by a description of ways to coordinate the management of general purpose and accelerator cores in Section 4. GViM’s evaluation in Section 5 demonstrates its low overheads and motivates the need for coordinated core management. A discussion of related work appears in Section 6 followed by conclusions and future work.

2. System Architecture

Figure 1 shows the system architecture of a virtualized GPGPU system. The hardware platform consists of general purpose cores (e.g., x86 cores) and specialized graphics accelerators – multiple NVIDIA GPUs in our prototype platform. Any number of VMs executing applications which require access to the GPU accelerators may be concurrently deployed in the system. The application components targeted for execution on the platform’s GPU components are represented as kernels, and their deployment and invocation are supported by the CUDA API.

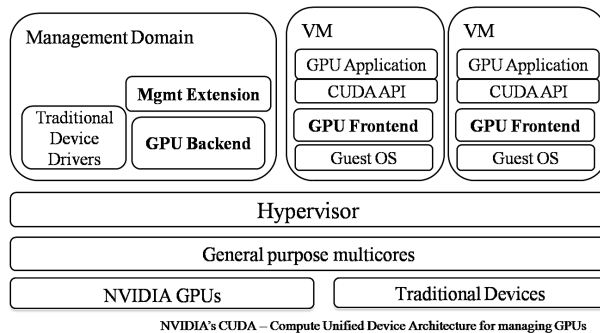


Figure 1. Virtualization of GPUs

The “split-driver model” depicted in Figure 1 delegates full control of the physical accelerators and devices to a management domain (i.e., dom0 in Xen). This implies that all accesses to the GPU will be routed via the frontend/backend drivers through the management domain and that data moved between the GPU and the guest VM application may require multiple copies or page remapping operations [24; 25]. While the approach is sufficiently general to handle a range of devices without additional virtualization-related capabilities, the overheads associated with it are prohibitive for HPC applications. This is particularly true for the GPU accelerators due to the potentially large size of the input and output data of the kernel that can span many pages of contiguous memory. The GViM approach described in this paper adopts the split-driver model described above, but makes substantial enhancements to make it more suitable for the performance requirements of HPC applications and the CUDA API on which many of them rely [30].

First, as CUDA is becoming an important API and programming model for high performance codes on GPU-accelerated manycore platforms, GViM virtualizes the GPU stack at the CUDA API level. While our choice of CUDA is a practical one that recognizes its substantial market penetration, it is also principled in that the level at which virtualization is done corresponds to that of other APIs used for accelerator interaction – IBM’s ALF [12], originally developed for the Cell processors, the recently announced OpenCL [14], and many ongoing industry and academic efforts towards uniform APIs and access methods with associated languages and runtimes [7; 27; 17]. Through this ap-

proach, GViM provides for improved productivity, allowing developers to deal with familiar higher-level APIs, and for increased portability, hiding low level driver or architecture details from guest VMs. Furthermore, since CUDA’s parallel programming model does not depend on the presence of graphics or other types of accelerators, CUDA kernels may also be deployed on the general purpose cores in the many-core platform, provided that appropriate binaries or translation tools exist. This gives GViM an additional level of flexibility to completely virtualize the heterogeneous platform resources. Our future work will focus on further enhancements to GViM to provide for and exploit such capabilities.

A key property of GViM is its ability to execute kernels with performance similar to that attained by VMs with direct access to accelerators. Toward this end, GViM implements efficient data movement between the guest VM’s application using the kernel and the GPU running it. The enhancements to the standard front end/back end model for this purpose are similar to the VMM-bypass mechanisms supported for high performance interconnects such as InfiniBand, or developed for specialized programmable network interfaces [21]. Common to these approaches is that either the hardware device itself is capable of enforcing isolation and coordination across device accesses originating from multiple VMs (e.g., in the case of InfiniBand HCAs), or the hardware and software stack are ‘open’, i.e., programmable, and the device runtime is programmed to provide such functionality. Neither one of these features is supported on our NVIDIA GPU accelerators or by their CUDA stack. NVIDIA’s proprietary access model and binary device drivers make it a ‘closed’ accelerator architecture. In response, GViM offers ‘VMM-bypass’-like functionality on the ‘data-movement’ path only, which means that all device accesses are still routed through the management domain, but then GViM uses lower-level memory management mechanisms to ensure that the kernels’ input and output data is directly moved between the guest VM and the GPU. The result is the elimination of costly copy and remapping operations.

GViM provides for effective sharing of available GPU resources among VMs by exploiting the fact that all accelerator kernel invocations are routed through the management domain. Policies dealing with a) GPU usage across multiple VMs and for multiple GPUs, and/or b) the coordination between the actions of the management domain with regards to scheduling the VMs’ kernels on accelerators, with VMM-level actions concerning the scheduling of VMs on general purpose cores, are thus enforced in this domain. Section 4 describes these mechanisms in greater detail, and experimental data in Section 5 demonstrates the importance of such coordination for meeting the performance requirements of HPC applications.

3. GPU Virtualization

This section describes how GViM virtualizes an NVIDIA-based GPGPU platform. The NVIDIA accelerator supports

the CUDA higher end parallel execution model with reported speedups ranging from 18x to 140x compared to general purpose CPUs. Virtualizing it, however, entails considerable complexity due to its proprietary access model and binary device drivers (i.e., its ‘closed’ architecture). We virtualize this accelerator on a hardware platform comprised of an x86-based multicore node with multiple accelerators attached via PCIe devices, using the Xen hypervisor and Linux as the target guest OS. The platform is designed to emulate future heterogeneous manycore chips comprised of both general and special purpose processors.

3.1 Design

With the Xen hypervisor, a GPU attached to the host system must run its drivers in a privileged domain that can directly access the hardware. An example of a privileged domain is Xen’s management domain (henceforth referred to as ‘Dom0’). This privileged domain, therefore, must also implement suitable memory management and communication methods for efficient sharing of the GPU by multiple guest VMs. In the rest of this paper, we assume that the drivers are run in Dom0 but note that they could also be run in a separate ‘driver domain’ [8].

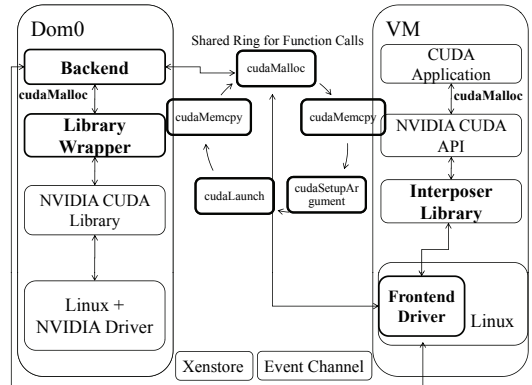


Figure 2. Virtualization components for GPU

Figure 2 shows the implementation components involved in virtualizing (and sharing) the GPU for access by multiple VMs. To attain high performance, GViM adapts Xen’s split driver model in multiple ways, the most important one being our mechanisms for data bypass, as described in further detail below.

The following adaptations exist on the guest side, as depicted on the right half of Figure 2, which shows a VM and the software layers being used for GPU access:

1. The GPU application uses the CUDA API – as explained in Section 2, GViM permits users to run any arbitrary CUDA-based application in a VM.
2. The Interposer Library provides CUDA access – In a non-virtualized environment, GPU applications written with the CUDA API rely on libraries available with the CUDA SDK. These libraries perform required checks and pass parameters to the lower level driver that triggers

execution on a GPU. Since the source code of the library and driver are ‘closed’, the interposer library running in the guest VM intercepts CUDA calls made by an application, collects arguments, packs them into a CUDA call packet, and sends the packet to the frontend driver described below. GViM thus maintains the abstraction level required for broad application use. The library currently implements all function calls synchronously and is capable of managing multiple application threads.

3. Frontend driver - The Frontend driver manages the connections between the guest VM and Dom0. It uses Xenbus and Xenstore [5] to establish event channels between both domains, receiving call packets from the interposer library, sending these requests to the backend for execution over a shared call buffer (or shared ring in Xen terminology), and relaying responses back to the interposer library. GViM’s implementation localizes all changes to the guest OS within the frontend driver, which can be loaded as a kernel module.

Function calls are carried out by several components in Dom0, which are described next and shown in the left half of Figure 2:

1. The Backend mediates all accesses to the GPU – located in Dom0, the backend is responsible for executing CUDA calls received from the frontend and for returning the execution results. It notifies the guest once the call has executed, and the result is passed via a shared ring. It is implemented as a user-level module for easier integration with the user-level CUDA libraries and to avoid additional runtime overhead due to accesses to userspace CUDA memory.
2. The Library Wrapper functions convert the call packets received from the frontend into function calls – the wrapper functions unpack these packets and execute the appropriate CUDA functions.
3. The NVIDIA CUDA library and driver are provided by NVIDIA – they are the components that interact with the actual device.

Jointly, these components enable a guest virtual machine to access any number of GPUs available in the system. The stack shown on the VM-side in Figure 2 is replicated in every guest in the system that wishes to access the GPU, while the single Dom0 stack is responsible for managing multiple guest domains as well as multiple GPUs, if available. In Section 4, we further discuss as ‘management extension’ the interaction between the backend and the scheduling of requests made by guests.

3.2 Memory Allocation and Sharing

Many HPC applications using GPUs have large amounts of input and/or output data. Xen does not natively provide efficient (i.e., non copy-based) support for large data sharing between guest VMs and Dom0. Prior efforts to improve IO

performance [31; 24] for network and block devices do not address sharing large numbers of pages with contiguous virtual addresses between a guest domain and a Dom0 backend running as a user level process, as required for GViM applications and as shown in Figure 3, which depicts with dotted lines the memory-related interactions in GViM. The dark solid arrows indicate the path of a CUDA call (refer to Figure 2 for detail). The arrows between guest application, Malloc memory, and Frontend memory allocator (via Frontend) are explained below.

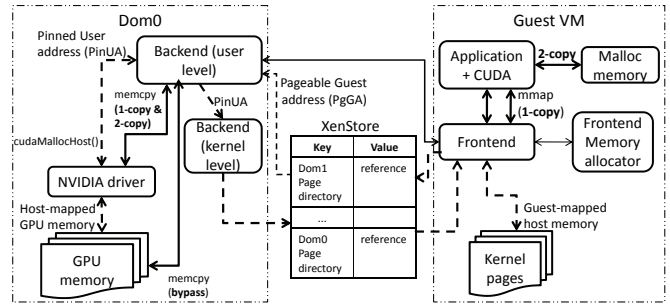


Figure 3. Memory management in GViM

In a non-virtualized environment, a memcpy operation invoked by a CUDA application has to go through one copy of data between host memory and GPU memory, managed by the NVIDIA driver. This can be avoided whenever possible/desirable by making a call to cudaMallocHost() which tries to return a pointer to host-mapped GPU memory, as shown in the left part of Figure 3. This is the pinned memory case and implies zero-copy of data. To understand the memory management in GViM in greater detail, it is important to understand the different memory kinds that can be allocated by the user for data movements:

1. 2-copy – A user CUDA application running in the guest VM can allocate memory for data buffers using malloc. The buffer is then resident in the user virtual address space; this makes it necessary to copy that data into a temporary kernel buffer before passing it on to Dom0 for some CUDA operation. However, it is possible to share the kernel buffer in advance with the backend and eliminate the copy from the guest kernel to the backend. The next step is a copy from host memory to GPU memory, as described above. Since the data has to go through two copies, this is the 2-copy solution.
2. 1-copy – From the previous solution, if we remove the guest ‘user to kernel’ copy before passing the call to the backend, we can reduce overheads, particularly for larger data movements. This is done by letting the user call mmap() into the frontend driver to get memory for its buffer instead of calling malloc. To avoid changing applications, the mmap call has been wrapped within cudaMallocHost() implementation of the interposer library. This memory is pre-allocated by the frontend, shared

with the Backend using Xenstore, and managed using the Frontend memory allocator. In order to allow a contiguous view of the memory at guest user level and at the Backend, the individual page numbers from the Frontend allocated buffer are loaded in a page directory structure that is shared with the Backend at frontend load time and is remapped by the Backend. We have thus, eliminated an extra level of copy potentially caused by virtualization.

3. Bypass – The ideal situation is a zero-copy data bypass whenever possible, as seen from Figure 6 in Section 5. Since the GPU memory is managed entirely by the ‘closed’ driver, we propose to let our Backend make a call to `cudaMallocHost()` when the system starts and map it through the kernel level module that can be loaded on its behalf into Dom0’s kernel address space. Portions of this region can be mapped into individual guests and further used to move data to and from the card, i.e., eliminating the ‘host to GPU’ data movement shown in Figure 3. In spite of its obvious performance benefit, the bypass approach limits the data sizes a VM can exchange with the GPU to the amount of available memory in the VM’s partition of the driver memory. Therefore the 1-copy mechanism described above remains useful.

4. Management Extension

The management extension to Dom0 is the software component that implements the resource management logic shown in Figure 4. The main task handled by this extension is to schedule domain requests for access to accelerator resources. The scheduler interacts with the backend described in Section 3 and instructs it to accept requests from a particular number of domains (depending on the number of GPUs available) for a certain period of time.

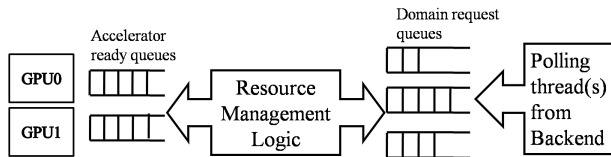


Figure 4. Interaction between scheduler and accelerator backend

Figure 4 shows a diagrammatic representation of the extension, the presence of which makes it possible to control access to the GPU ‘above’ the driver level. However, GViM does not control actions on the actual device, i.e., once the requests reach the driver. This is due to the closed nature of the NVIDIA graphics device. Instead, we monitor requests coming in from a guest over the shared call buffer using a thread that constantly polls the ring for requests. Since Dom0 has a shared call buffer per guest, the polling thread can only poll one guest’s buffer at a time. For the multiple GPUs shown in the figure, there can be a polling thread corresponding to every GPU.

4.1 Scheduling in Management Domain

The concepts used for scheduling include the following:

1. Credits – credits are considered as the currency given to a guest, which grants it x msec of execution time at the cost of y credits. This term is adopted from the Xen credit-based scheduler [5].
2. Xen credits – these refer to the credits that are assigned to the guest by the user or by default when it boots. These values can be changed as the guest is running.

To improve compliance of domains with the weights assigned to them, GViM implements scheduling of requests destined for the GPU. However, since we do not have direct access to the GPU’s hardware threads or to the driver that controls request scheduling on the GPU, GViM’s scheduling layer is implemented ‘above’ the driver level intercepting requests before they reach the driver. Various scheduling policies can be implemented in this layer, and we describe two schemes below. These simple schemes provide basic QoS guarantees to guests, focusing on fairness.

Round robin scheduling (RR). In this scheme, the polling thread monitors a guest’s call buffer for a given period of time, and some guest is chosen every period. Any request coming from the guest during this period is executed. Of interest about this method is that we poll a domain only after receiving its first request which establishes some context on the GPU. Toward this end, the domain queue is checked for the corresponding initialization request, but the domain is not polled for the entire assigned time period if no such request exists. This is to prevent other domains from suffering due to some domain’s inactivity related to the GPU. Further, the queue is always re-checked for additional domain requests before putting the device into standby mode.

XenoCredit(XC)-based scheduling. Here, we exploit the fact that guest VMs are assigned Xen credits in order to provide them with basic QoS or fairness guarantees. The XenoCredit-based scheduler uses these credits to calculate the proportion of time a guest call buffer should be monitored. The higher the credits, the more time is given for the execution of guest requests on the accelerator. For example, with Dom1 - 1024, Dom2 - 512, Dom3 - 256, Dom4 - 512, the number of ticks will be 4, 2, 1, 2, respectively. The duration of ticks is adjustable and depends on the common execution time of most GPU applications. The credits assigned to the guests are acquired from the Xen credit scheduler [5] and are monitored during the lifetime of the domain to track any changes in their value.

We are currently developing and evaluating additional and alternative scheduling methods for GPU accelerators. Our intent is to better coordinate guest domain and accelerator scheduling, both with respect to how the GViM scheduler interacts with Xen’s scheduling methods and to explore scheduling methods appropriate for different classes of end

user applications, ranging from HPC codes to enterprise and web applications.

5. Experimental Evaluation

Testbed. The experimental evaluation of GViM is conducted on a GPGPU system based on a Xeon quad-core running at 2.5GHz and 2GB memory. For evaluation of virtualization overhead we have used an NVIDIA 8800 GTX PCIe card and for resource management experiment we have used an NVIDIA 9800 GTX PCIe card (with 2 GPUs on-board). The GPU driver version is 169.09. Virtualization is provided by Xen 3.2.1 running the 2.6.18 Linux kernel.

Benchmarks. All benchmarks used in the evaluation are part of the CUDA SDK 1.1. The specific applications are selected so as to represent a mix of data types and dataset sizes, data transfer times, number of iterations executed for certain computations and their complexity. These are:

Matrix multiplication – it is easy to change the size of inputs and corresponding output data for this benchmark. Matrix multiplication is also an important step in financial, image/signal processing, and other applications. Our evaluation uses 2048x2048 single precision floating point matrices, which amounts to an exchange of 16MB data per matrix. We refer to it as MM[2K].

Black Scholes – modern option pricing techniques are often considered among the most mathematically complex of all applied areas of finance. Most of the models and techniques employed by today’s analysts are rooted in a model developed by Fischer Black and Myron Scholes [4]. The example used has a tunable number of iterations to improve accuracy and option count. This makes it possible to vary the runtime of this benchmark, and it also allows us to measure the scheduling performance of the system over some given period of time. Unless stated otherwise, it by default generates values for 1 million call and put option prices over 512 iterations, labeled as BS[1m,512].

Fast Walsh transform – Walsh transforms belong to a class of generalized Fourier transformations, used in various fields of electrical engineering and in numeric theory. The sample application used is an efficient implementation of naturally-ordered Walsh transform (also known as Walsh-Hadamard or Hadamard transform) with CUDA, with a particular application to dyadic convolution computation [18]. This example is fairly data intensive, requiring 64MB in input and output buffers (FWT[64]).

5.1 Measurement Results

Experimental results attained with the benchmarks above and some micro-benchmarks can be classified based on the properties being evaluated.

Comparison of individual function call timings to study virtualization overhead – Figure 5 shows the difference in execution time at a function call level between a virtualized guest and our base case of Dom0. These functions represent

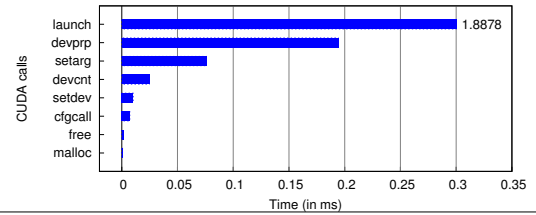


Figure 5. CUDA Calls - Execution Time Difference

the most commonly used CUDA calls in GPU applications. The number of bytes transferred per CUDA call each way without data buffers is about 80Bytes which we refer to as the standard packet size (SPS). With the exception of devprp, setarg, launch, and memcpy (dealt with later), most calls exchange standard packet data both ways. Most commonly used CUDA calls do not see more than a 0.07msec increase in execution time except devprp (retrieving device properties) which is called once at the beginning and launch (cudaLaunch() for a GPU kernel). A typical sequence of operations for compute kernel execution on a GPU is a) configuring a call with appropriate thread and block sizes on the GPU, b) setting up arguments and c) launching the kernel. While these calls execute individually in a non-virtualized environment, we combine them together with launch in the backend due to requirements imposed by the driver level API. This leads to a higher launch time but keeps the overhead for configuring a call smaller.

Impact of input data sizes – The cost of memcpy (memory copy) varies with the amount of data transferred. These results become very important when applications transfer much larger data sizes (possibly multiple times during the application execution), as discussed in Section 3.

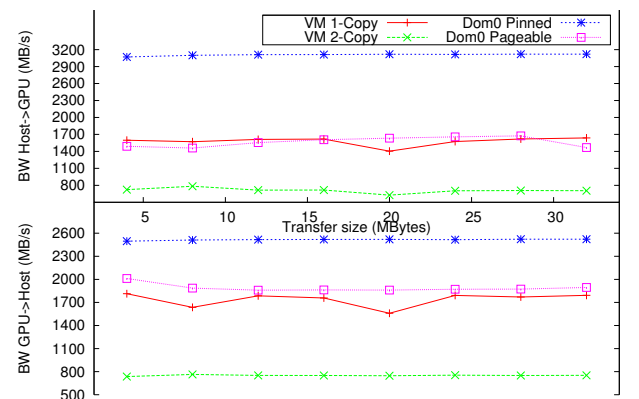


Figure 6. GPU bandwidth for memory copy operations

Figure 6 shows the bandwidth in MB/sec into (upper half of figure) and out (lower half) of the GPU for the guest VM, as well as Dom0 for the cases discussed in Section 3. The results (the Y-axis in figure does not start from 0) are obtained by running the GPU bandwidth test from the CUDA SDK. Dom0 pageable and pinned in the figure refer to the 1-copy and bypass options respectively for applications running in Dom0. As seen from the graphs, our 1-copy solu-

tion achieves almost as much bandwidth as Dom0 Pageable. Since most applications are written to use pageable memory, this is a very good option for even large benchmarks being run in guest VM (as long as it has sufficient memory). The Dom0 pinned case shows much higher bandwidth, and we are working towards this bypass solution for our virtualized guests, as well.

Runtimes compared to Linux, Dom0 without GPU virtualization and Guest(s) – Figure 7 shows the total execution time and time for CUDA calls of the aforementioned GPU applications in (i) a non-virtualized Linux environment, (ii) virtualized but with direct access to the GPU in the Dom0 environment, and (iii) with our GPU virtualization stack in a guest VM. As we can see from the graph, the introduction of a hypervisor and the execution of the application in Dom0 without going through any virtualization stack does not introduce much overhead. Therefore, the Dom0 times serve as our base case and the performance target for the GPU virtualization. For all these experiments we have pinned guest VMs to a physical CPU.

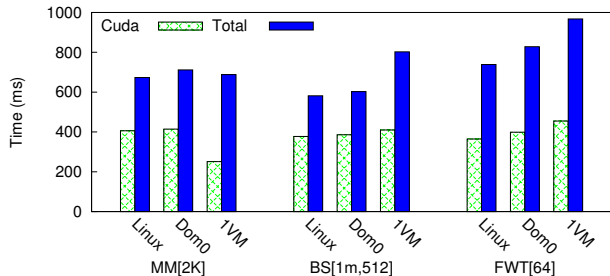


Figure 7. Virtualization overhead - Benchmark evaluation

As seen from the figure, the worst case overhead in total execution time observed is 25% for BS[1m,512] but the CUDA contribution in this case is about 6%. On the other hand, the overhead for FWT is 14% for the total runtime with a 12% contribution from CUDA execution. The best case time observed is actually an improvement of 3% in the CUDA execution time for matrix multiplication with total time comparable to native Linux. This shows that increase in virtualized execution time for benchmarks is a function of memory allocation (FWT has multiple host allocation and free calls in a tight loop) and transfer as well as the number of kernel invocations which is in accordance with Figures 5 and 6. Therefore, as VM-VM communication latencies continue to further improve in next generation hardware, they will significantly reduce the GViM overheads observed for the kernel-inocations for more iterative applications. The improvement seen in MM is attributable primarily to the conversion of host level CUDA API to driver level API in the backend. This conversion has been done to allow the backend to load the compute kernel for an application explicitly using its name which otherwise requires locating it from the application binary. This is almost impossible due to obfuscation of names by the NVIDIA compiler.

Resource Management – In typical HPC scenarios, the applications tend to expect a chunk of resources to be al-

located to them for their entire duration or a significantly long time without too much sharing. GViM allows sharing of GPU resources but also has the option to define the proportions. The XenoCredit scheme described earlier proves to be pretty useful in letting GViM allow for such resource allocation. A simple experiment with executing BlackScholes (for 4096 iterations with expected per iteration time) shown in Table 1 shows how resource management policies can be adapted keeping in mind different application requirements. The numbers in the table have been acquired from a host with 2GPUs running 4 VMs that could have overlapping GPU execution. The expected iterations/msec shown is from explicit profiling of the application without sharing of resources. As seen from the columns for RR vs. XC, XC actually comes closer to the expectations of the VM in accordance with its credits unlike RR which tries to balance resources equally among all.

VMs	Credits	Expected iter/msec	RR	XC
VM1	512	1.8	2.632	1.867
VM2	256	2.4	2.567	3.12
VM3	256	2.4	2.804	3.213
VM4	256	2.4	2.901	3.3

Table 1. BlackScholes: XenoCredit vs. Round Robin

5.2 Discussion

As evident from experimental measurements, the GViM virtualization stack itself introduces only small additional overheads. Bandwidth results show that such overheads can be reduced further by achieving complete hypervisor bypass. More interestingly, the XenoCredit scheduling scheme improves system fairness and provides better QoS guarantees to guest domains, thereby motivating further work on new methods for resource management in accelerator-based systems. Specifically, it is evident that request scheduling on GPUs and more generally, on accelerators must be coordinated with the manner in which guest domains are scheduled. The coordination methods used should consider guest level QoS needs (e.g., required degree of parallelism for HPC codes or fairness for enterprise codes) and resource availability on accelerators. With closed accelerators like NVIDIA GPUs, since coordination does not have access to details about accelerator resources and their states, we are currently experimenting with using code profiling for accelerator codes to better use accelerator resources and with more adaptations to the scheduling scheme based on different application requirements. We are also experimenting with coordination mechanisms and methods suitable for open accelerators, using IBM’s Cell processor.

6. Related Work

Given the difficulty of virtualizing closed and proprietary GPUs, there has been very little work in the efficient use of GPGPUs with hypervisors such as Xen and VMware.

Most systems allow VMs to directly access the hardware but, unlike GViM, this approach prevents any sharing of these hardware resources. The most closely related work to our technique of virtualizing the GPU is VMGL [16], a method that virtualizes the OpenGL API to allow hardware acceleration for applications running in VMs. In contrast, GViM is tailored towards programmatic APIs for GPGPU access and uses significantly optimized interfaces, unlike VMGL's dependance on TCP/IP as the transfer protocol.

GViM shares its goals with our own prior work on virtualizing the Cellule accelerator [10]. The latter treats the Cell processor as a stand-alone system instead of the tight integration favored by GViM. Finally, there is a large body of work on the efficient virtualization of networking hardware in Xen [24], including efforts to provide direct multi-VM access to networking hardware [25]. However, most of this work has been greatly simplified by the presence of open and narrow hardware interfaces.

7. Conclusion and Future Work

The GViM virtualization infrastructure for a GPGPU platform enables the sharing and consolidation of graphics processors. Resource management methods associated with GViM provide reasonable guarantees to the guest domains using GPU(s). Experimental measurements of a Xen-based GViM implementation on a multicore platform with multiple attached NVIDIA graphics accelerators demonstrate small performance penalties for virtualized vs. non-virtualized settings, coupled with substantial improvements concerning fairness in accelerator use by multiple VMs. With solutions like GViM, therefore, it becomes possible to simultaneously run multiple HPC codes on the same platform, without unpredictably perturbing individual applications.

Our future work will extend the GViM virtualization architecture with respect to its resource management methods, to include methods that use kernel profile information and accommodate other accelerators like the IBM Cell processor. We will also evaluate the scalability and stability of our scheduling schemes and introduce power-awareness into GViM's scheduling policies.

References

[1] AAD J. VAN DER STEEN AND JACK J. DONGARRA. Overview of recent supercomputers. <http://www.top500.org/resources/orsc>.

[2] ALAM, S. R., AGARWAL, P. K., SMITH, M. C., VETTER, J. S., AND CALIGA, D. Using fpga devices to accelerate biomolecular simulations. *Computer* 40, 3 (2007), 66–73.

[3] AMD CORPORATION. Amd white paper: The industry-changing impact of accelerated computing. http://www.amd.com/us/Documents/AMD_fusion.Whitepaper.pdf, 2009.

[4] BLACK, F., AND SCHOLES, M. The pricing of options and corporate liabilities. In *Journal of Political Economy* (1973), pp. 81:637–659.

[5] CHISNALL, D. *The Definitive Guide to the Xen Hypervisor*, 1st ed. Prentice Hall Open Source Software Development Series, 2008.

[6] CRAY INC. Cray xd1, 2.2 ghz. <http://www.top500.org/system/7657>.

[7] DIAMOS, G., AND YALAMANCHILI, S. Harmony: An execution model and runtime for heterogeneous many core systems. In *HPDC*

Hot Topics.

[8] FRASER, K., H, S., NEUGEBAUER, R., PRATT, I., ET AL. Safe hardware access with the xen virtual machine monitor. In *I Workshop on Operating System and Architectural Support for on demand IT InfraStructure (OASIS)* (Oct. 2004).

[9] GAVRILOVSKA, A., KUMAR, S., RAJ, H., SCHWAN, K., ET AL. High-performance hypervisor architectures: Virtualization in hpc systems. In *HPCVirt* (2007), pp. 1–8.

[10] GUPTA, V., XENIDIS, J., AND SCHWAN, K. Cellule: Virtualizing cell/b.e. for lightweight execution. Georgia Tech STI Cell/B.E. Workshop, June 2007.

[11] HOFSTEE, H. P. Power efficient processor architecture and the cell processor. In *HPCA* (2005), pp. 258–262.

[12] IBM. Accelerated library framework for cell broadband engine programmers guide and api reference. <http://tinyurl.com/c2z4ze>, October 2007.

[13] INTEL CORPORATION. Enabling consistent platform-level services for tightly coupled accelerators. <http://tinyurl.com/clcr3n>.

[14] KHRONOS OPENCL WORKING GROUP. The opencl specification. <http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf>, December 2008.

[15] KUMAR, S., TALWAR, V., RANGANATHAN, P., AND RIPAL NATHUJI, K. S. M-channels and m-brokers: Coordinated management in virtualized systems. In *Workshop on Managed Multi-Core Systems* (June 2008).

[16] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. Vmm-independent graphics acceleration. In *VEE* (2007), pp. 33–43.

[17] LINDERMAN, M. D., COLLINS, J. D., WANG, H., AND MENG, T. H. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS* (2008), pp. 287–296.

[18] MANZ, J. A sequency-ordered fast walsh transform. In *IEEE Transactions on Audio and Electroacoustics* (August 1972).

[19] NATHUJI, R., AND SCHWAN, K. Virtualpower: coordinated power management in virtualized enterprise systems. In *SOSP* (2007).

[20] NVIDIA. Nvidia cuda compute unified device architecture - programming guide. <http://tinyurl.com/cx3t13>, June 2007.

[21] RAJ, H., AND SCHWAN, K. High performance and scalable i/o virtualization via self-virtualized devices. In *HPDC* (2007).

[22] RANADIVE, A., KESAVAN, M., GAVRILOVSKA, A., AND SCHWAN, K. Performance implications of virtualizing multicore cluster machines. In *HPCVirt* (2008), pp. 1–8.

[23] RODRIGUEZ, M., TAPIADOR, D., ET AL. Dynamic provisioning of virtual clusters for grid computing. In *3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing Euro-Par* (2008).

[24] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND PRATT, I. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX Annual Technical Conference* (June 2008).

[25] SANTOS, J. R., TURNER, Y., AND MUDIGONDA, J. Taming heterogeneous nic capabilities for i/o virtualization. In *Workshop on I/O Virtualization* (Dec. 2008).

[26] SEILER, L., CARMEAN, D., SPRANGLE, E., ET AL. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics* 27, 3 (2008), 1–15.

[27] STRATTON, J., STONE, S., AND MEI HWU, W. Mcuda: An efficient implementation of cuda kernels on multi-cores. Tech. Rep. IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[28] SUN MICROSYSTEMS. The tsubame grid redefining supercomputing. <http://www.top500.org/lists/2006/11>.

[29] TURNER, J. A. The los alamos roadrunner petascale hybrid supercomputer: Overview of applications, results, and programming, March 2008.

[30] VOLKOV, V., AND DEMMEL, J. Lu, qr and cholesky factorizations using vector capabilities of gpus. Tech. Rep. UCB/EECS-2008-49, May 2008.

[31] YU, W., AND VETTER, J. S. Xen-based hpc: A parallel i/o perspective. In *CCGRID* (2008), pp. 154–161.