VGVM: Efficient GPU Capabilities in Virtual Machines

Dimitrios Vasilas
Computing Systems Laboratory
National Technical University of Athens
dimvas@cslab.ece.ntua.gr

Stefanos Gerangelos
Computing Systems Laboratory
National Technical University of Athens
sgerag@cslab.ece.ntua.gr

Nectarios Koziris
Computing Systems Laboratory
National Technical University of Athens
nkoziris@cslab.ece.ntua.gr

Abstract

Graphics Processing Units (GPUs) have become a powerful platform, that can provide significant performance benefits to data parallel applications. Graphic processors are being increasingly introduced as accelerators in high performance computing (HPC) systems due to the development of GPGPU (General-Purpose Computation on GPUs). Furthermore, virtualization technologies are gaining interest in these domains, due to their benefits on server consolidation as well as the isolation and ease of management they offer. There is thus a growing need to combine the benefits of both fields by providing heterogeneous resources, particularly GPUs, in virtual environments.

In this paper we address the challenge of integrating GPGPU into virtualized environments. We propose VGVM, a mechanism that enables the execution of GPU accelerated applications within Virtual Machines (VMs). Our framework consists of two components: a user level library and a paravirtualized driver, which enables communication with the host's GPU driver. To validate our approach, we conduct experiments on a variety of GPU applications, focusing on the virtualization overhead and the scalability of our framework.

1. Introduction

Nowadays, Graphics Processing Units (GPUs), driven by the insatiable market demand for real-time, high-definition 3D graphics, have evolved into general-purpose, high performance, many-core processors capable of high computation throughput and memory bandwidth. In addition to being efficient at manipulating computer graphics and image processing, their highly parallel structure makes them well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity. As a result, GPUs are being introduced as accelerators in order to achieve speed-ups in applications traditionally handled by the central processing unit (CPU). This approach, known as general purpose computation on GPUs (GPGPU), is being increasingly adopted in HPC (High Performance Computing) applications. Research has demonstrated that computationally intensive applications from a wide range of scientific fields, such as finance [1], chemical physics [2], weather broadcast [3], fluid dynamics [4] etc. can leverage GPUs to obtain major gains in performance. In addition to the scientific domain, GPUs are used in software routers [5], encrypted networks [6] and database management systems [7] as well. One of the reasons general purpose computing on GPUs has been well established is the introduction of dedicated programming environments, compilers, and libraries such as CUDA from Nvidia.

On the other hand, virtualization technology has an increasing influence on how computational resources are used and managed. The growth in hardware performance and the increased demand for service consolidation from business markets, leads virtualized cloud environments to host an ever growing amount of computations. Virtual Machines (VMs) can improve resource utilization, as several different customers may share a single computing node with the illusion that they own the entire machine in an exclusive way, while providing process isolation and ease of management. Consequently, virtualization techniques are a promising effort to run high performance software on a grid, as obtaining virtualized cloud computational resources is an elastic, time and cost efficient alternative to the traditional way of obtaining resources. With the recent advances in both virtualization and GPU technology, there is an ever-growing need to provide heterogeneous resources, particularly GPUs, within the cloud environment in the same scalable and on-demand way as traditional virtualized hardware. Cloud providers are thus facing the challenge of integrating into their platforms. For example, Amazon Elastic Compute Cloud (EC2) [8] provides GPU instances as computing resources, but each client is assigned with an individual physical instance of GPUs. Unfortunately, in the cloud context I/O virtualization suffers from poor performance, due to the overhead incurred by indirect access to physical resources and the need to multiplex the applications access to I/O resources. Virtualization and sharing of GPU hardware face additional challenges due to the characteristics of graphic processing units that do not enable preemptive scheduling and time-sharing capabilities.

In this paper we propose, VGVM, an efficient approach to expose GPGPU capabilities in virtual machines. We present the design and implementation of a framework that enables applications executing in virtual environments, to accelerate their performance exploiting GPU resources. By using our framework, multiple VMs co-located in the same host computer can share physical GPU resources. As a proof of concept we target the virtualization of CUDA-enabled GPUs by enabling applications developed using the CUDA platform to execute within VMs. VGVM employs paravirtualization techniques and uses a split driver model. It consists of a user-level library, a frontend driver located at the guest OS and a backend driver implemented at the hypervisor. The framework's architecture is shown in Figure 1.

In summary, the main contributions of our work are:

- We propose an efficient GPU virtualization framework that enables GPGPU applications to execute within VMs, and implements GPU resource sharing among co-located VMs.
- We maintain CUDA Runtime binary compatibility, so that existing applications can use our framework without any source code modification.
- We categorize GPU accelerated applications based on their computation and memory access profile and discuss which types applications could benefit by using our framework.

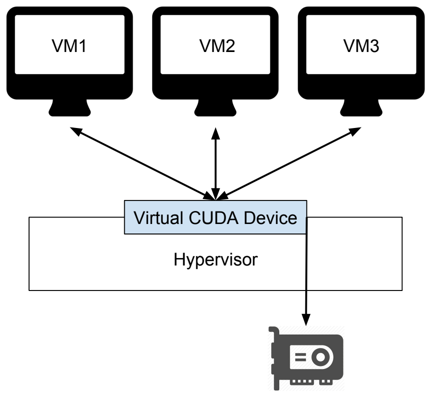

Figure 1. The VGVM architecture

Our performance evaluation shows that VGVM achieves low virtualization overhead, making GPU accelerated applications executing within VMs competitive to those executing in native environments with direct access to GPU resources.

The rest of this paper is organized as follows: We first introduce some necessary background in Section 2. Section 3 describes the design and implementation of VGVM, while Section 4 presents a detailed performance evaluation. Then, in Section 5 we discuss how different types of GPGPU application can benefit from using our framework. In Section 6 we discuss related work and finally, in Section 7 we conclude and present directions for future work.

2. Background

2.1 GPGU Programming Interfaces

CUDA (Compute Unified Device Architecture) [9] and OpenCL (Open Computing Language) [10] are two widely used interfaces offering general-purpose computing capabilities on GPUs. Both present similar features but through different programming interfaces. OpenCL, developed by the Khronos Group, is an open standard for cross-platform, parallel programming on heterogeneous platforms consisting of central processing units (CPUs), GPUs, digital signal processors (DSPs) and other types of processors or hardware accelerators. CUDA is a parallel computing platform introduced by Nvidia. It offers a proprietary API and set of language extensions that can be used to perform computations on CUDA-enabled GPUs.

CUDA offers two programming interfaces: (1) the Runtime API and (2) the Driver API. The Runtime API is a high-level interface that provides a set of routines and language extensions and offers implicit initialization, context and module management. The Driver API is a low-level interface that offers an additional level of control by exposing lower level concepts such as CUDA contexts, the analogue of host processes for the device, and CUDA modules, the analogue of dynamically loaded libraries for the device. However, using the Driver API requires more code and effort to program and debug. Figure 2 shows CUDA software stack. Most applications do not use the Driver API, as they do not need the additional level of control, and use the Runtime API instead in order to produce more concise code.

CUDA exposes its features through a runtime library as well as a set of language extensions. These language extensions allow programmers to define device functions (kernels), configure and execute them on CUDA-enables GPUs. Extensions include function type qualifiers that specify whether a function executes on the host or the device and whether it can be called from the host or the device, variable type qualifiers, that specify the memory location of a variable on the device and build-in variables, that specify dimensions and indices for the GPU's multiple cores. Kernels are configured and launched using the execution configuration extension, denoted with the `<<<...>>>` syntax. A function declared as:

```
__global__ void Func(float* parameter);
```

is called using:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

where Dg specifies the dimension and size of the grid, Db the dimension and size of each thread block and Ns the number of bytes in shared memory that is allocated for this call.

Source files of CUDA applications contain both host and device code. More specifically, they contain language extensions and device functions that need to be compiled with the `NVCC` compiler. `NVCC` separates device code from host code and compiles the device code into an assembly form of CUDA instruction set architecture (`PTX` code) or binary form (`cubin` objects). Host code is modified by replacing the execution configuration extension (`<<<...>>>`) with the necessary runtime function calls to load and launch each compiled kernel from the `PTX` code or `cubin` object. This procedure is illustrated in Figure 3. Device code is loaded from `cubin` or `PTX` files either during initialization from the runtime or explicitly using the Driver API.

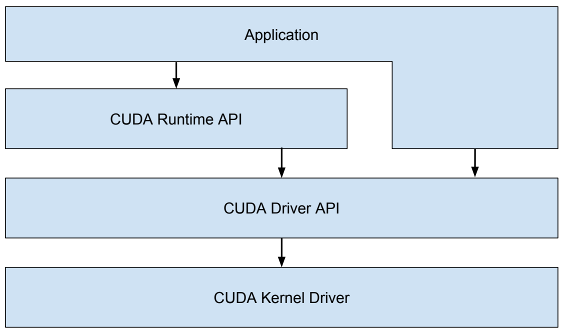

Figure 2. CUDA Software Stack

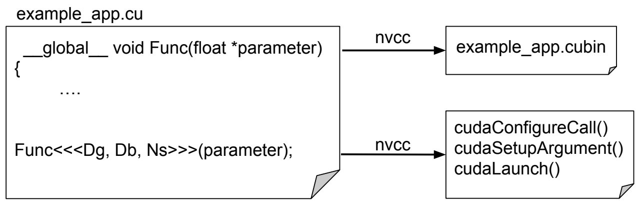

Figure 3. Compilation Output

2.2 I/O Virtualization

Paravirtualization is a common technology used to virtualize I/O devices. It enables low overhead I/O device virtualization providing efficient communication between host and guest. In this approach virtual hardware is optimized for the virtualization layer and exposes a software interface to the guest, which is similar but not identical to that of the underlying hardware. It is implemented by creating communication channels between hypervisor

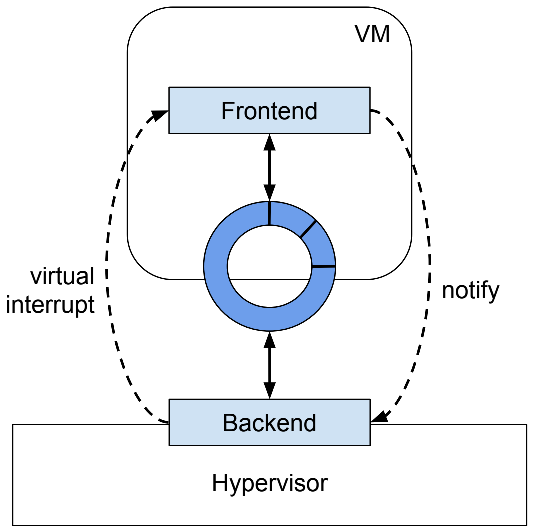

Figure 4. Data Transport Mechanism

and guest operating system. Paravirtualized frontend drivers post I/O requests to backend drivers directly, with minimal overhead. To address the issue of having a unified model for those paravirtualized drivers across different virtualization systems, virtio [11] has been proposed. Virtio provides a standardized interface for the development of virtualized devices, as well as a mechanism to support guest-to-hypervisor communication.

Using the interface defined in virtio, guest drivers communicate with the hypervisor by pushing data buffers to a shared queue. The guest posts request buffers, which are processed by the backend to produce corresponding responses. Virtio defines a virtual queue interface that can be used for guest-to-hypervisor communication. Specifically, it implements a shared ring buffer mechanism that enables guests to post buffers which host can consume. Each shared ring has a callback function associated with it, which is called when the hypervisor consumes the buffers. The communication scheme is shown in Figure 4. Using the virtio data transport interface, frontend drivers can enqueue buffers to the shared ring and notify the hypervisor. They can then either poll or wait to be notified when results become available through a virtual interrupt. Frontend virtio drivers, including drivers for network and block devices, have been added to the mainline Linux kernel. Additionally, backend virtio drivers have been implemented for the QEMU software. Those implementations use a data transport channel and a control mechanism which we also use in our approach.

3. Design and Implementation

In this section we describe VGVM's architecture and analyze how virtualization and sharing of the GPU device is accomplished. We design VGVM using a paravirtualization approach and employ API redirection in order to enable CUDA application to execute within VMs. VGVM consists of three software modules: a user level library, a frontend driver located at the guest OS and a backend that represents a virtual CUDA device. We accomplish virtualization by intercepting library calls and redirecting their arguments to the frontend driver. They are afterwards transfered to the backend through a communication channel and executed on the host. Results are eventually returned to the guest. GPU resource sharing is implemented by multiplexing execution requests at the backend side. Future work includes implementing a GPU resource management system that enables scheduling of execution requests posted by multiple guests. Data and control paths are depicted in Figure 5. Solid lines represent control path, while dashed lines represent data path.

VGVM Library: CUDA applications access GPU resources through routine calls as well as extensions to the C programming language. Routine calls are implemented in libraries provided by the CUDA SDK. Moreover, language extensions are replaced at compile time by internal function calls, not exposed to the programmer. Using our framework, CUDA applications developed with the Runtime API remain binary compatible since we expose the same interface in our library. In order to implement Runtime API routines in our library, we transfer routine arguments to the backend, where the execution occurs, and receive the execution results.

When a CUDA library call is made by an application (5a), we intercept the arguments, pack them to an execution request among with other required data, such as CUDA contexts, stored by the library, and forward the request to the frontend driver through an `ioctl()` system call (5b). When the system call returns, we unpack the results and return them to the calling process. Furthermore, we accomplish virtualization of the kernel launch syntax (`<<<...>>>`) by implementing the internal routines which replace the extension in our library. We intercept CUDA SDK routine calls and override them with VGVM library routines using the LD_PRELOAD environment variable.

Frontend Driver: We design VGVM's frontend driver as the intermediate component which forwards execution requests from guest to backend. We implement the frontend driver as a kernel module loaded to the guest Linux kernel. When a library routine makes an `ioctl()` call (5b), the frontend driver handles it by performing appropriate memory allocations and copying the intercepted arguments from user space to kernel space (5i). It

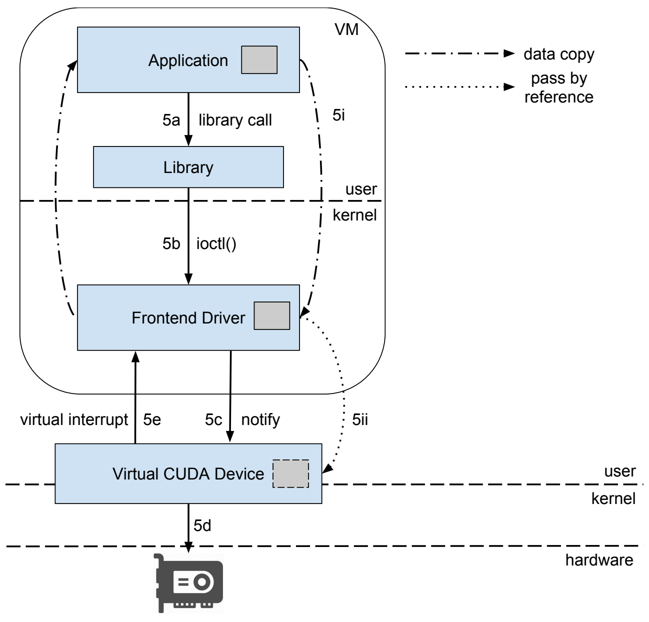

Figure 5. Data and Control Path

then uses the data transport mechanism described in Section 2 to make a request to the VGVM virtual CUDA device (5c). When processing is complete, the frontend driver copies results back to user space.

We implement two approaches for the mechanism which awaits results from the virtual device, a polling based and an interrupt based. In the former approach the driver repeatedly checks if a buffer has been pushed to the shared ring buffer by the backend, while in the latter one the process' state is changed to interruptible sleep, until a virtual interrupt is received, indicating that a buffer has been pushed to the shared ring. The interrupt handler then pops the buffer from the ring and wakes up the process. We perform evaluation of the aforementioned implementations regarding their performance and CPU utilization.

We enable communication between frontend and backend utilizing the previously described data transport mechanism. More specifically, we issue execution requests to the virtual CUDA device by pushing buffers to the shared ring buffer and notifying the backend. This communication mechanism introduces a limitation. Each data buffer has to be allocated in a physically contiguous way, which is not always feasible, especially in large data transfers. This is a limiting issue in the case of data copies between host and device memory. We therefore develop a mechanism which falls back to a scatter-gather technique with smaller physically contiguous buffers, in case the required memory cannot be allocated contiguously. The backend can then access each piece of memory and reconstruct the original buffer.

Moreover, we implement GPU resource sharing among processes executing concurrently in the same virtual machine. We accomplish it by enabling co-executing processes to access the shared ring buffer concurrently using a synchronization scheme. Each process can independently push requests to the ring buffer and wait for them to be processed.

Virtual CUDA Device: We design the backend part of VGVM as a dispatcher which handles execution requests from multiple co-located VMs. We implement the virtual CUDA device as a QEMU PCI device. VGVM's backend component operates as a request handler, receiving requests for routine execution as well as the required arguments and executing them in the host environment. The backend can directly access buffers provided through the data transport mechanism, without copying them. This is possible since the guest's physical address space is accessible from the QEMU process' virtual address space through a translation mechanism. When an execution request is received, we decode it, retrieve required arguments and trigger execution on the GPU (5d).

We eventually handle execution requests at the backend by executing appropriate CUDA Driver API routines. We choose the approach of implementing the Runtime API using Driver API routines, since the Driver API allows explicit context and module management. Explicitly managing module loading and CUDA context switching is required in order to implement GPU resource sharing. We ensure isolation and protection among CUDA applications by switching the CUDA context associated with the calling process to the current one before issuing routine execution, since each CUDA context has its own unique address space. When execution is completed, we push buffers representing execution results to the shared ring buffer and notify the guest by triggering a virtual interrupt (5e).

In order to accomplish sharing of the physical device among processes executing concurrently in co-located VMs, we implement a separate shared ring buffer between the virtual CUDA device and each VM. We treat each request interdependently and multiplex execution requests from co- located virtual machines. By multiplexing execution requests from CUDA application executing concurrently on multiple VMs we enable them to share their access to GPU resources.

3.1 Runtime API Implementation Details

We implement virtualization of Runtime API routines by intercepting library call arguments and forwarding them to the backend for execution. Standard library routines' implementation is fairly straightforward since there are respective Driver API routines offering the same functionality. However, implementing routine calls that replace the kernel launch extension (`<<<...>>>`) requires more effort, since those routines are not exposed to the programmer and CUDA is proprietary software not providing source code. We therefore employ reverse engineering techniques in order to accomplish virtualization of the kernel launch extension. We perform library call tracing in order to discover declaration of internal routines that implement kernel launching. We implement those routines in our library exposing the same interface so that we can intercept launch arguments and forward them. Multiple routines are used to configure and launch the kernel. In our implementation we gather the appropriate arguments, store them in data structures, and forward them lazily on the last call.

Moreover, before launching a kernel execution, we need to load device code to the GPU from the appropriate CUDA object file. However, the Driver API routine that implements module loading requires the programmer to provide the object file name. Additionally, Runtime API implicitly manages module loading, not exposing the respective file names. We therefore develop a mechanism which, at the beginning of CUDA application execution, searches for `.cubin` files and uses symbol extraction to determine which kernels are defined in each object file. We store the mapping between object files and kernels in a data structure in our library and search for the corresponding file to load every time a kernel is launched.

3.2 Isolation and Security

Our implementation ensures protection between applications executing within a VM as well as between separate VMs. We achieve isolation using the mechanism of CUDA contexts. CUDA contexts are the equivalent of CPU processes. Each context has each own unique address space and, as a result, GPU pointer values from different contexts reference different physical memory locations. We associate a context with each application in order to ensure isolation and protection from other applications executing in the same as well as different VMs. Then, we set the current context each time at the backend side based on the calling process.

VGVM has been designed in such a way, that it can be enhanced with more advanced features as future extensions. For instance, a scheduling mechanism could be added in order to ensure fairness among separate VMs. Currently, in our prototype implementation requests are multiplexed on the host in a FIFO order. This can be extended in a future work by introducing different scheduling algorithms which can apply fairness and protect VMs from other poorly configured or malicious VMs that try to hog GPU resources.

3.3 Current Limitations

Although our framework provides transparent access to CUDA devices, it introduces two functionality restrictions. Due to undocumented internal library calls, currently only CUDA Toolkit 5.0

Input Size (KB)		8	16	32	64	128	256	512	1024	2048
Busy Wait	Execution Time (ms)	5.0	8.5	8.7	9.0	14.1	34.9	64.1	240.8	463.0
	CPU Usage (%)	100	100	100	100	100	100	100	100	100
Sleep	Execution Time (ms)	13.8	14.2	14.8	15.0	15.3	35.2	64.1	241.1	462.7
	CPU Usage (%)	10	12	10	10	9	9	10	6	7

Table 1. Comparison of Sleep and Busy Wait Implementations

is supported in the guest. CUDA toolkit 7.5 can be used at the backend, where the actual execution occurs. Additionally, CUDA object files, which are used to load device code, need to be available to the host at runtime. They can be either copied before the execution or accessed through a shared file system.

4. Experimental Evaluation

All performance evaluations are conducted on a test system consisting of two Intel Xeon X5650 CPUs (@2.66 GHz) with 48 GB of main memory. It is equipped with one Nvidia Tesla M2050 GPU. The host system is running Ubuntu Linux 14.04 distribution with kernel 3.19.0 and Nvidia driver version 352.39. The virtualization software used is QEMU-KVM 2.3. All virtual machines are configured to use one VCPU and 1 GB RAM. The guest OS is Debian 3.16.7 with kernel version 3.16.0.

In order to evaluate the performance of our prototype, we use benchmarks from the official CUDA SDK 7.5 [12] as well as the Rodinia benchmark suite [13]. We select benchmarks to represent a wide range of GPGPU applications, and use varying computational loads, data sizes, and different CUDA features.

We first use synthetic microbenchmarks to compare the two implementations that the frontend uses to wait for results by the backend. Furthermore, we perform breakdown analysis and examine the overhead introduced by the VGVM software stack. Subsequently, we use an application to evaluate the corresponding performance using our framework compared to native execution. Finally, we evaluate the scalability of VGVM as the number of concurrently executing applications in co-located VMs increases.

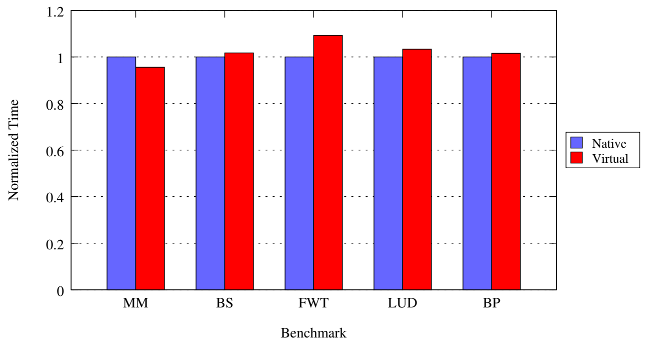

Figure 6. Microbenchmark Performance

4.1 Sleep and Busy Wait Implementations

We first perform an evaluation of the two aforementioned mechanism used by the frontend driver to wait for execution results. We use a microbenchmark from CUDA samples, that performs matrix multiplication which allows us to adjust the size of input data. Matrix multiplication is an important operation used in a variety of applications, such as financial and signal processing applications. We compare total execution time as well as CPU utilization for each method. We evaluate the two metrics for a range of input data sizes. Results of this experiment are depicted in Table 1.

Results show that for small input sizes the busy wait method performs much better than the sleep method regarding to total execution time. This is expected, since the overhead of triggering a virtual interrupt and executing the interrupt handler is higher compared to polling in a busy wait loop. However, as input size increases the difference in performance becomes negligible. Additionally, when the busy wait method is used, applications fully utilize the CPU throughout their execution, creating unnecessary load to the system. In the case of the sleep method, applications have lower CPU utilization, since they release the CPU during waiting time.

In order to benefit from advantages of both methods, we implement a hybrid approach. For small input sizes we use the busy wait method in order to achieve better performance. In this case, CPU is fully utilized for a short period of time, since backend execution does not usually last long for small input sizes. Conversely, for larger input sizes we use the sleep method in order to achieve low CPU utilization as well as high performance.

4.2 Microbenchmark Performance

We conduct experiments with several benchmarks running in a native environment compared to executing in virtual machines with VGVM. We measure the execution time of all CUDA operations and do not include any computation performed on the CPU as our framework introduces overhead only on CUDA operations. The normalized execution times on the native and virtualized environment are presented in Figure 6. Experiment results show that performance degradation of BlackScholes (BS), LU Decomposition (LUD) and Back Propagation (BB) (where we executed the GPU kernel 1000 times) benchmarks executing in a VM is negligible compared to the native execution. Their execution time in a VM is 1.74%, 3.32% and 1.56% higher than native respectively. The largest overhead in execution time is 9.23% for the fastWalshTransform (FWT) benchmark. This benchmark has a short kernel launch time (only a third of the total GPU execution time) and thus overhead from memory copy between host and device has a higher impact on its performance, due to the aforementioned copy between user and kernel space. This overhead can be alleviated by applying zero copy techniques such as memory pinning, which can be implemented as future work. Moreover, we observe a 4.40% improvement in CUDA execution time of the matrix multiplication (MM) benchmark. This improvement can be attributed to the conversion of Runtime API to Driver API at the backend.

4.3 Breakdown Analysis

We analyze the overhead introduced by our virtualization framework and perform breakdown analysis of individual CUDA Runtime API routines, using the bandwidthTest benchmark provided by the CUDA samples. Results are shown in Figure 7. We choose to perform measurements of `cudaMemcpy` routine, as it introduces the largest virtualization overhead because of memory copy operations. We divide the execution of a routine into

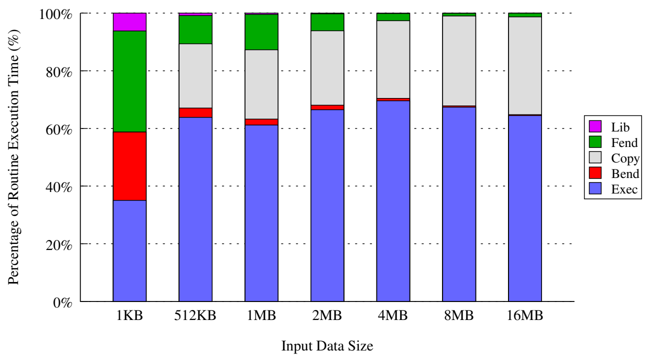

Figure 7. Breakdown Analysis

five phases (`lib`, `copy`, `fend`, `exec`, `bend`) representing the different components of VGVM's software stack as well as operations causing significant overhead. The first phase (`lib`) includes operations performed by the VGVM library except the `ioctl()` system call that transfers control to the frontend driver. `Copy` represents the overhead introduced by copying memory from user space to kernel space, while `fend` is the time consumed at the rest of frontend driver's operations. Finally, `exec` is the time of Driver API routine execution at the backend and `bend` is the time spent at the rest of the operations performed by the backend, such as memory allocations, argument unpacking etc.

The figure indicates that the dominant factor of execution time is Driver API routine execution at the backend. This phase takes up to 69% of the total execution time for large memory copies. Since this phase represents the actual execution on the GPU, the rest of execution phases can be characterized as the virtualization overhead caused by our framework. Results show that this overhead remains consistent for memory copies larger than 1 MB at 27% of the total execution time on average. This favors applications in which execution time is dominated by computation on the GPU rather than memory copying. The phase of memory copy between user and kernel space consumes 30% of the total execution time on average and constitutes the major factor of virtualization overhead. Applying zero copy mechanisms, as mentioned earlier, could lower this overhead. Such techniques can be implemented as future work.

Further measurements depict that the rest of library routines introduce a constant overhead. `CudaMalloc` operation introduces an overhead of 24 μs, `cudaFree` overhead is 26 μs and kernel launch overhead is 45 μs. Relative applications' performance overhead is reduced as input data size increases, since this constant overhead becomes a smaller portion of the total execution time as time of execution on the GPU increases.

4.4 Application Performance

We evaluate the robustness and efficiency of VGVM when used by a higher level application. We use GPU MrBayes [14], a bio-informatics application which uses DNA data to infer phylogeny. The application implements the Bayesian method to infer phylogeny, using Metropolis coupled Markov chain Monte Carlo $(MC)^3$ on CUDA and uses real-world DNA data as input (1.1 MB input size). We measure the total time of execution as well as the execution time of CUDA operations. Table 2 depicts experiment results for execution on native and virtual environment using VGVM. As in previous experiments, `Cuda` represents only the CUDA related functions calls, while `Total` represents the total time of execution including both CPU and GPU processing. Results show that our framework adds 12.4% overhead on the total execution time and 10% ovehread on CUDA operations time.

	Native	Virtual
Total (s)	60.93	67.02
Cuda (s)	50.37	56.64

Table 2. MrBayes Application Performance

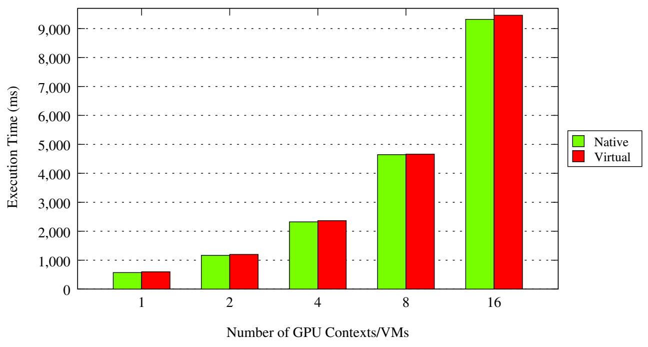

Figure 8. Scaling Measurements

4.5 Performance at Scale

We evaluate the overhead VGVM introduces at multiple concurrently executing GPU contexts by conducting experiments in two setups: (1) we issue multiple processes on the native system executing the same application (`native`) and (2) we launch multiple VMs and execute one application per VM (`virtual`). We measure application's CUDA execution time for these settings and evaluate the overhead introduced by VGVM, as the number of GPU contexts and VMs increases respectively. We use the BlackScholes benchmark provided by the CUDA samples. Results are depicted in Figure 8.

We make two observations based on the results. One is that native execution time increases linearly as the number of GPU contexts increases. This is expected, since legacy GPUs enforce serialization of GPU tasks from different contexts. We later discuss the effect of this GPU characteristic on different types of applications. However, recent Nvidia GPUs (e.g. Kepler [15]) implement actual sharing of GPU resources between concurrently executing CUDA jobs. Another observation is that performance degradation of concurrently executing GPU applications in multiple VMs is negligible compared to native. Operations performed by VGVM software modules, such as copies between user and kernel space, are executed in multiple VMs in parallel and do not introduce additional overhead as the number of VMs increases.

5. Related Work

Various approaches to implement virtualization of graphic processing hardware and address GPU resource sharing have been proposed by the research community. In the meantime, cloud providers, such as Amazon [8], Microsoft Azure [16] etc. [17] are starting to offer GPU computing resources as a service. A class of proposed solutions makes use of pass-through technology in order to grant VMs direct access to host devices. This approach can minimize the overhead of virtualization, as performance measurements on the Xen platform indicate [18], but since a pass-throughed GPU is exclusively managed by the guest OS, it does not allow multiple VMs to share the same device. Nvidia GRID [19] technology enables assignment of the physical GPU to multiple VMs at the same time. Gdev [20] is able to virtualize a physical GPU into multiple logical GPUs, which can then be pass-throughed to VMs, thus enabling GPU resource sharing.

A full GPU virtualization solution that allows the native graphics driver to execute in the guest system has been presented by Tian et al. [21]. This approach is implemented on Intel Processor Graphics GPU and evaluation is oriented on 2D and 3D graphic applications. Suzuki et al. [22] propose an architecture based on the Xen hypervisor that implements both full virtualization and paravirtualization. This approach differs from VGVM, as it virtualizes the GPU at a lower level and uses Gdev [23] as the CUDA Runtime.

vCUDA [24] and rCuda [25, 26] both employ API call interception and redirection, in order to allow applications executed within VMs to transparently access Nvidia hardware. They both use network protocols in order to implement communication mechanisms. Although network communication mechanisms make these solutions VMM-independent, they cause significant overhead. rCUDA also features an optimized implementation of the communication mechanism for InfiniBand interconnects, in order to take advantage of the high speed fabric. Giunta et al. [27] propose gVirtuS, a GPU virtualization service which focuses on hypervisor independence relying on a communication component independent from the communication channel. Comparison with measurements in [27] reveals that VGVM introduces lower overhead than this approach. This can be attributed to the execution of Runtime API routines at the backend. GViM [28], uses a paravirtualization approach in order to virtualize and manage GPU resources. It employs Xen- specific mechanisms, including shared memory buffers, in order to implement the communication mechanism. This solution presents a higher overhead compared to our framework, as the comparison between measurements in [28] and ours depict. Gottschlag et al. [29] propose LoGV, a solution that implements GPGPU virtualization at a lower level, by leveraging protection mechanisms present in modern GPUs. It virtualizes the API of the `pscnv` GPU driver and uses open-source software in order to support the CUDA API. Moreover, this approach does not ensure protection between individual applications within VMs but only between VMs. In DS-CUDA [30] the authors present a middleware with the goal to address major difficulties in programming multi-node heterogeneous computers. The system implements virtualization of a cluster of computers equipped with GPUs so that they appear as if they were attached to a single node, in order to simplify the programming of multi-GPU applications. In the context of sharing GPU resources among co-located virtual machines Nanos et al. propose V4VSockets [31], a framework for efficient, low-overhead intra-node communication in the Xen hypervisor. They show that rCUDA can be deployed over V4Vsockets to efficiently enable GPU resource sharing among co-located VMs.

6. Discussion

Sharing physical hardware among multiple concurrently executing OSes is a fundamental aspect of hardware virtualization. VGVM enables sharing of GPU resources by multiplexing requests for routine execution at the hypervisor. However, multiplexing applications' accesses on the GPU introduces additional overhead and negatively impacts its performance. Moreover, applications with different execution patterns can be affected differently by sharing their access to the GPU.

GPU accelerated applications can be divided according to their characteristics into different classes. One such class contains applications that can be be characterized as batch jobs. These applications copy large amounts of data from host to device memory and then issue intensive computations without further user interaction. Results are calculated and copied back to host memory before execution is completed. Examples include HPC scientific applications from fields such as bioinformatics [32] and material science [33]. Multiplexing GPU accesses of concurrently executing applications of this class cause performance degradation, due to the inability of legacy GPUs to offer actual resource sharing. The critical performance metric is total execution time. However, the main characteristic of such applications is that their execution time is dominated by GPU resources utilization. Therefore multiplexing GPU accesses of multiple concurrently executing applications decreases performance of all applications, since execution requests from different CUDA contexts are serialized on the GPU. This is an inherent characteristic of legacy GPUs' architecture. Thus, performance degradation of each individual application is negligible in case applications are submitted to run sequentially, for instance on a resource scheduling system (e.g. Torque). However, even this class of applications is expected to behave better on modern GPUs.

On the other hand, a different class involves long-running interactive applications which typically begin by copying required data to the device, outside of the critical execution path, and then repeatedly receive smaller amounts of data as input which trigger computations. Examples of applications following this execution pattern include Big Data applications that perform queries on large data sets [34]. Applications of this class sometimes have low latency characteristics and even real-time requirements. The critical performance factor of this class is the response time when input is received. Their execution includes alternations between idle periods user input is awaited, and computation periods when requested results are being calculated. This execution pattern is well fitted for device sharing among concurrently executing applications. Multiplexing their accesses to the GPU is feasible as idle periods of some applications can overlap with computation periods of others.

It is thus evident that the effect of sharing GPU resources can be different according to the application's execution pattern. A GPU resource management system could distinguish between the previously described application classes and schedule their accesses to GPU resources accordingly. Future work in VGVM involves applying profiling techniques in order to categorize applications and using different scheduling algorithms to multiplex accesses to the GPU. These techniques will also be evaluated on modern GPUs, which provide more advanced sharing features among concurrent tasks.

7. Conclusion

In this work we propose, VGVM, a framework for low overhead GPU resource virtualization and sharing among co-located VMs. Our implementation employs API redirection through a split driver approach, in order to allow GPGPU applications to access the physical hardware. VGVM is open-source software available online at `https://github.com/dimvass/VGVM`. Evaluation of our prototype shows that VGVM achieves near native performance for medium and large data sizes. Moreover, multiple applications executing concurrently in co-located VMs can efficiently share the host GPU.

We design VGVM in a way that enables scheduling mechanisms to be easily added to the current version. We plan to implement execution request scheduling in order to achieve quality of service between VMs as well individual applications, in a future work. An extension to VGVM's backend can implement GPU resource management and ensure fairness by identifying applications' GPU execution profile and appropriately schedule their access to the GPU. To this end, the mechanism could detect and slow down VMs with high demands on GPU resources. Finally, future endeavors also include evaluating our framework on recent Nvidia GPU with enhanced features regarding resource sharing.

References

[1] Abhijeet Gaikwad and Ioane Muni Toke. Gpu based sparse grid technique for solving multidimensional options pricing pdes. In *Proceedings of the 2Nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 6:1–6:9, New York, NY, USA, 2009. ACM.

[2] D.P. Playne and K.A. Hawick. Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA09)*, pages 104–110, Las vegas, USA, 13-16 July 2009. WorldComp.

[3] J. Michalakes and M. Vachharajani. Gpu acceleration of numerical weather prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, April 2008.

[4] Everett H. Phillips, Yao Zhang, Roger L. Davis, and John D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, number AIAA 2009-565, jan 2009.

[5] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[6] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and Kyoungsoo Park. Sslshader: cheap ssl acceleration with commodity processors. In *In Proceedings of the 8th USENIX conference on Networked systems and implementation, NSDI11*. USENIX Association, 2011.

[7] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[8] Amazon GPU Instances. http://aws.amazon.com/ec2/instance-types/.

[9] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[10] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[11] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[12] NVIDIA. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/5_0/rel-update-1/installers/cuda_5.0.35_linux_64_ubuntu11.10-1.run.

[13] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[14] Jie Bao, Hongju Xia, Jianfu Zhou, Xiaoguang Liu, and Gang Wang. Efficient implementation of mrbayes on multi-gpu. *Molecular Biology and Evolution*, 30(6):1471–1479, 2013.

[15] Nvidia. Nvidias Next Generation CUDA Compute Architecture Kepler GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[16] Microsoft Azure. https://azure.microsoft.com/en-us/.

[17] NVIDIA. GPU Cloud Computing. http://www.nvidia.com/object/gpu-cloud-computing-services.html.

[18] A.J. Younge, J.P. Walters, S. Crago, and G.C. Fox. Evaluating gpu passthrough in xen for high performance cloud computing. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 852–859, May 2014.

[19] Nvidia. NVIDIA GRID Virtual GPU Technology. http://www.nvidia.com/object/grid-technology.html.

[20] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[21] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[22] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 109–120, Philadelphia, PA, June 2014. USENIX Association.

[23] S. Kato. Gdev CUDA Runtime. https://github.com/shinpei0208/gdev.

[24] Lin Shi, Hao Chen, and Jianhua Sun. vcuda: Gpu accelerated high performance computing in virtual machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11, May 2009.

[25] Antonio J. Pea, Carlos Reao, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ort, and Jos Duato. A complete and efficient cuda-sharing solution for {HPC} clusters. *Parallel Computing*, 40(10):574 – 588, 2014.

[26] J. Duato, A.J. Pena, F. Silla, J.C. Fernandez, R. Mayo, and E.S. Quintana-Orti. Enabling cuda acceleration within virtual machines using rcuda. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, Dec 2011.

[27] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 379–391, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[29] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. Logv: Low-overhead gpgpu virtualization. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1721–1726, 2013.

[30] A. Kawai, K. Yasuoka, K. Yoshikawa, and T. Narumi. Distributed-shared cuda: Virtualization of large-scale gpu systems for programmability and reliability. In *FUTURE COMPUTING 2012, The Fourth International Conference on Future Computational Technologies and Applications*, page 712, 2012.

[31] Anastassios Nanos, Stefanos Gerangelos, Ioanna Alifieraki, and Nectarios Koziris. V4vsockets: Low-overhead intra-node communication in xen. In *Proceedings of the 5th International Workshop on Cloud Data and Platforms*, CloudDP '15, pages 1:1–1:6, New York, NY, USA, 2015. ACM.

[32] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. Barracuda - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):1–7, 2012.

[33] Stefan Maintz, Bernhard Eck, and Richard Dronskowski. Speeding up plane-wave electronic-structure calculations using graphics-processing units. *Computer Physics Communications*, 182(7):1421 – 1427, 2011.

[34] GPUdb. GPUdb The first GPU accelerated In-Memory Distributed database. http://www.gpudb.com/.