GPUvm: Why Not Virtualizing GPUs at the Hypervisor?

Yusuke Suzuki, ***Keio University;*** **Shinpei Kato,** ***Nagoya University;*** **Hiroshi Yamada,**
Tokyo University of Agriculture and Technology; **Kenji Kono,** ***Keio University***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

GPUvm: Why Not Virtualizing GPUs at the Hypervisor?

Yusuke Suzuki
Keio University

Shinpei Kato
Nagoya University

Hiroshi Yamada
Tokyo University of Agriculture and Technology

Kenji Kono
Keio University

Abstract

Graphics processing units (GPUs) provide orders-of-magnitude speedup for compute-intensive data-parallel applications. However, enterprise and cloud computing domains, where resource isolation of multiple clients is required, have poor access to GPU technology. This is due to lack of operating system (OS) support for virtualizing GPUs *in a reliable manner*. To make GPUs more mature system citizens, we present an open architecture of GPU virtualization with a particular emphasis on the Xen hypervisor. We provide design and implementation of full- and para-virtualization, including optimization techniques to reduce overhead of GPU virtualization. Our detailed experiments using a relevant commodity GPU show that the optimized performance of GPU para-virtualization is yet two or three times slower than that of pass-through and native approaches, whereas full-virtualization exhibits a different scale of overhead due to increased memory-mapped I/O operations. We also demonstrate that coarse-grained fairness on GPU resources among multiple virtual machines can be achieved by GPU scheduling; finer-grained fairness needs further architectural support by the nature of non-preemptive GPU workload.

1 Introduction

Graphics processing units (GPUs) integrate thousands of compute cores on a chip. Following a significant leap in hardware performance, recent advances in programming languages and compilers have allowed compute applications, as well as graphics, to use GPUs. This paradigm shift is often referred to as general-purpose computing on GPUs, *a.k.a.*, GPGPU. Examples of GPGPU applications include scientific simulations [26,38], network systems [13,17], file systems [39,40], database management systems [14,18,22,36], complex control systems [19,33], and autonomous vehicles [15,27].

While significant performance benefits of GPUs have attracted a wide range of applications, main governors of practically deployed GPU-accelerated systems, however, are limited to non-commercial supercomputers. Most GPGPU applications are still being developed in the research phase. This is largely due to the fact that GPUs and their relevant system software are not tailored to support virtualization, preventing enterprise servers and cloud computing services from isolating resources on multiple clients. For example, Amazon Elastic Compute Cloud (EC2) [1] employs GPUs as computing resources, but each client is assigned with an individual physical instance of GPUs.

Current approaches to GPU virtualization are classified into I/O pass-through [1], API remoting [8,9,11,12,24,37,41], or hybrid [7]. These approaches are also referred to as *back-end*, *front-end*, and *para* virtualization, respectively [7]. I/O pass-through, which exposes GPU hardware to guest device drivers, can minimize overhead of virtualization, but the owner of GPU hardware is limited to a specific VM by hardware design.

API remoting is more oriented to multi-tasking and is relatively easy to implement, since it needs only to export API calls to outside of guest VMs. Albeit simple, this approach lacks flexibility in the choice of languages and libraries. The entire software stack must be rewritten to incorporate an API remoting mechanism. Implementing API remoting could also result in enlarging the trusted computing base (TCB) due to accomodation of additional libraries and drivers in the host.

The para-virtualization allows multiple VMs to access the GPU by providing an ideal device model through the hypervisor, but guest device drivers must be modified to support the device model. According to these three classes of approaches, it is difficult, if not impossible, to use vanilla device drivers for guest VMs while providing resource isolation on multiple VMs. This lack of reliable virtualization support prevents GPU technology from the enterprise market. In this work, we explore GPU virtualization that allows multiple VMs to share underlying GPUs without modification of existing device drivers.

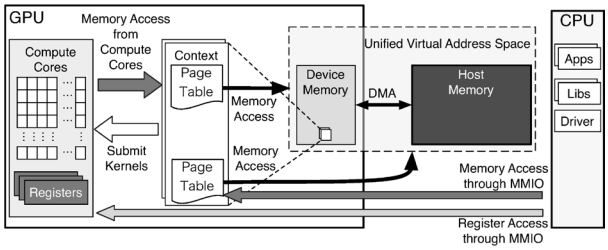


Figure 1: The GPU resource management model.

Contribution: This paper presents GPUvm, which is an open architecture of GPU virtualization. We provide the design and implementation of GPUvm based on the Xen hypervisor [3], introducing virtual memory-mapped I/O (MMIO), GPU shadow channels, GPU shadow page tables, and virtual GPU schedulers. These pieces of resource management are provided with both full- and para-virtualization approaches by exposing a native GPU device model to guest device drivers. We also develop several optimization techniques to reduce overhead of GPU virtualization. To the best of our knowledge, this is the first piece of work that addresses fundamental problems of GPU virtualization. GPUvm is provided as complete open-source software [1].

Organization: The rest of this paper is organized as follows. Section 2 describes a system model behind this paper. Section 3 provides a design concept of GPUvm, and Section 4 presents its prototype implementation. Section 5 shows experimental results. Section 6 discusses related work. This paper concludes in Section 7.

2 Model

The system is composed of a multi-core CPU and an onboard GPU connected on the bus. A compute-intensive function offloaded from the CPU to the GPU is called a *GPU kernel*, which could produce a large number of compute threads running on a massive set of compute cores integrated in the GPU. The given workload may also launch multiple kernels within a single process.

Product lines of GPU vendors are closely tied with programming languages and architectures. For example, NVIDIA invented the Compute Unified Device Architecture (CUDA) as a GPU programming framework. CUDA was first introduced in the Tesla architecture [30], followed by the Fermi and the Kepler architectures [30,31]. The prototype system of GPUvm presented in this paper assumes these NVIDIA technologies, yet the design concept of GPUvm is applicable for other architectures and programming languages.

Figure 1 illustrates the GPU resource management model, which is well aligned with, but is not limited to, the NVIDIA architectures. The detailed hardware mechanism is not identical among different vendors, though recent GPUs adopt the same high-level design presented in Figure 1.

MMIO: The current form of GPU is an independent compute device. Therefore the CPU communicates with the GPU via MMIO. MMIO is the only interface that the CPU can directly access the GPU, while hardware engines for direct memory access (DMA) are supported to transfer a large size of data.

GPU Context: Just like the CPU, we must create a context to run on the GPU. The context represents the state of GPU computing, a part of which is managed by the device driver, and owns a virtual address space in GPU.

GPU Channel: Any operation on the GPU is driven by commands issued from the CPU. This command stream is submitted to a hardware unit called a GPU *channel*, and isolated from the other streams. A GPU channel is associated with exactly one GPU context, while each GPU context can have one or more GPU channels. Each GPU context has GPU channel descriptors for the associated hardware channels, each of which is created as a memory object in the GPU memory. Each GPU channel descriptor stores the settings of the corresponding channel, which includes a *page table*. The commands submitted to a GPU channel is executed in the associated GPU context. For each GPU channel, a dedicated command buffer is allocated in the GPU memory visible to the CPU through MMIO.

GPU Page Table: Paging is supported by the GPU. The GPU context is assigned with the GPU page table, which isolates the virtual address space from the others. A GPU page table is set to a GPU channel descriptor. All the commands and programs submitted through the channel are executed in the corresponding GPU virtual address space.

GPU page tables can translate a GPU virtual address to not only a GPU device physical address but also a host physical address. This means that the GPU virtual address space is unified over the GPU memory and the host main memory. Leveraging GPU page tables, the commands executed in the GPU context can access to the host physical memory with the GPU virtual address.

PCIe BAR: The following information includes some details about the real system. The host computer is based on the x86 chipset and is connected to the GPU upon the PCI Express (PCIe). The base address registers (BARs) of PCIe, which work as windows of MMIO, are configured at boot time of the GPU. GPU control registers and GPU memory apertures are mapped on the BARs, allowing the device driver to configure the GPU and access the GPU memory.

To operate DMA onto the associated host memory,

[1] https://github.com/CS005/gxen

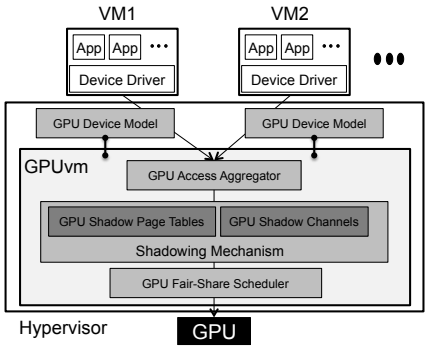


Figure 2: The design of GPUvm and system stack.

GPU has a mechanism similar to IOMMU, such as the graphics address remapping table (GART). However, since DMA issued from the GPU context goes through the GPU page table, the safety of DMA can be guaranteed without IOMMU.

Documentation: Currently GPU vendors hide the details of GPU architectures due to a marketing reason. Implementations of device drivers and runtime libraries are also protected by binary proprietary software, whereas the compiler source code has been recently open-released from NVIDIA to a limited extent. Some work uncovers the black-boxed interaction between the GPU and the driver [28]. Recently the Linux kernel community has developed Nouveau [25], which is an open-source device driver for NVIDIA GPUs. Throughout their development, the details of NVIDIA architectures are well documented in the Envytools project [23]. Interested readers are encouraged to visit their website.

Scope and Limitation: GPUvm is an open architecture of GPU virtualization with a solid design and implementation using Xen. Its implementation focuses on NVIDIA Fermi- and Kepler-based GPUs with CUDA. We also restrict our attention to Nouveau as a guest device driver. NVIDIA binary drivers should be available with GPUvm, but they cannot be successfully loaded with current versions of Xen, even in the pass-through mode, which has also happened in the prior work [11, 12].

3 Design

The challenge of GPUvm is to show that GPU can be virtualized at the hypervisor level. GPU is a unique and complicated device and its resources (such as memory, channels and GPU time) must be multiplexed like the host computing system. Although the architectural details of GPU are not known well, GPUvm virtualizes GPUs by combining the well-established techniques of CPU, memory, and I/O virtualization of the traditional hypervisors.

Figure 2 shows the high-level design of GPUvm and relevant system stack. GPUvm exposes a native GPU device model to VMs where guest device drivers are loaded. VM operations upon this device model are redirected to the hypervisor so that VMs can never access the GPU directly. The GPU Access Aggregator arbitrates all accesses to the GPU to partition GPU resources across VMs. GPUvm adopts a client-server model for communications between the device model and the aggregator. While arbitrating accesses to the GPU, the aggregator modifies them to manage the status of the GPU. This mechanism allows multiple VMs to access a single GPU in the isolated way.

3.1 Approaches

GPUvm enables full- and para-virtualization of GPUs at the hypervisor. To isolate multiple VMs on the GPU hardware resources, memory areas, PCIe BARs, and GPU channels must be multiplexed among those VMs. In addition to this spacial multiplexing, the GPU also needs to be scheduled in a fair-share manner. The main components of GPUvm to address this problem include GPU shadow page tables, GPU shadow channels, and GPU fair-share schedulers.

To aggregate accesses to a GPU device model from a guest device driver, GPUvm intercepts MMIO by setting these ranges as inaccessible.

In order to ensure that one VM can never access the memory area of another VM, GPUvm creates a GPU *shadow* page table for every GPU channel descriptor, which is protected from guest OSes. All GPU memory accesses are handled by GPU shadow page tables; a virtual address for GPU memory is translated by the shadow page table not by the one set by the guest device driver. Since GPUvm validates the contents of shadow page tables, GPU memory can be safely shared by multiple VMs. And by making use of GPU shadow page tables, GPUvm guarantees that DMA initiated by GPU never accesses memory areas outside of those allocated to the VM.

To create a GPU context, the device driver must establish the corresponding GPU channel. However, the number of GPU channels is limited in hardware. To multiplex VMs on GPU channels, GPUvm creates *shadow* channels. GPUvm configures shadow channels, assigns virtual channels to each VM and maintains the mapping between a virtual channel and a shadow channel. When guest device drivers access a virtual channel assigned by GPUvm, GPUvm intercepts and redirects the operations to a corresponding shadow channel.

3.2 Resource Partitioning

GPUvm partitions physical memory space and MMIO space over PCIe BARs into multiple sections of continuous address space, each of which is assigned to an in-

dividual VM. Guest device drivers consider that physical memory space origins at 0, but actual memory access is shifted by the corresponding size through shadow page tables created by GPUvm. Similarly, PCIe BARs and GPU channels are partitioned by multiple sections of the same size for individual VMs.

The static partitioning is not a critical limitation of GPUvm. Dynamic allocation is possible. When a shadow page table refers to a new page, GPUvm allocates a page, assigns it to a VM and maintains the mappings between guest physical GPU pages and host physical GPU pages. For ease of implementation, the current GPUvm employs static partitioning. We plan to implement the dynamic allocation in the future.

3.3 GPU Shadow Page Table

GPUvm creates GPU shadow page tables in the reserved area of GPU memory, which translates guest GPU virtual addresses to GPU device physical or host physical addresses. By design, the device driver needs to flush TLB caches every time a page table entry is updated. GPUvm can intercept TLB flush requests because those requests are issued from the host CPU through MMIO. After the interception, GPUvm updates the corresponding GPU shadow page table entry.

GPU shadow page tables play an important role in protecting GPUvm itself, shadow page tables, and GPU contexts from buggy or malicious VMs. GPUvm excludes the memory mappings to those sensitive memory pages from the shadow page tables. Since all the memory accesses by GPU go through the shadow page tables, any VMs cannot access those sensitive memory areas.

There is a subtle problem regarding a pointer to a shadow page table. In case of GPU, a pointer to a shadow page table is stored in GPU memory as part of the GPU channel descriptor. In the traditional shadow page tables, a pointer to a shadow page table is stored in a privileged register (e.g. CR3 in Intel x86); VMM intercepts the access to the privileged register to protect the shadow page tables. To protect GPU channel descriptors, including a pointer to a shadow page table, from being accessed by GPU, GPUvm excludes the memory areas for GPU channel descriptors from the shadow page tables. The accesses to the GPU channel descriptors from host CPUs are all through MMIO and thus, can be easily detected by GPUvm. As a result, GPUvm protect GPU channel descriptors, including pointers to shadow page tables, from buggy VMs.

GPUvm guarantees the safety of DMA. If a buggy driver sets an erroneous physical address when initiating DMA, the memory regions assigned to other VMs or the hypervisor can be destroyed. To avoid this situation, GPUvm makes use of shadow page tables and the unified memory model of GPU. As explained in Section 2, GPU page tables can map GPU virtual addresses to physical addresses in GPU memory and host memory. Unlike conventional devices, GPU uses GPU virtual addresses to initiate DMA. If the mapped memory happens to be in the host memory, DMA is initiated. Since shadow page tables are controlled by GPUvm, the memory access by DMA is confined in the memory region of the corresponding VM.

The current design of GPU poses an implementation problem of shadow page tables. In the traditional shadow page tables, page faults are extensively utilized to reduce the cost of constructing shadow page tables. However, GPUvm cannot handle page faults caused by GPU [10]. This makes it impossible to update the shadow page table upon page fault handling, and it is also impossible to trace changes to the page table entry. Therefore, GPUvm scans the entire page tables upon TLB flush.

3.4 GPU Shadow Channel

The number of GPU channels is limited in hardware and they are numerically indexed. The device driver assumes that these indexes start from zero. Since the same index cannot be assigned to multiple channels, channel isolation must be supported to multiplex VMs.

GPUvm provides GPU shadow channels to isolate GPU accesses from VMs. Physical indexes of GPU channels are hidden from VMs but virtual indexes are assigned to their virtual channels. Mapping between physical and virtual indexes is managed by GPUvm.

When a GPU channel is used, it must be activated. GPUvm manages currently activated channels by VMs. These activation requests are submitted through MMIO and they can be intercepted by GPUvm. When GPUvm receives the requests, GPUvm activates the corresponding channels by using physical indexes.

Each GPU channel has channel registers, through which the host CPU submits commands to GPU. Channel registers are placed in GPU virtual address space which is mapped to a memory aperture. GPUvm manages all physical channel registers and maintains the mapping between physical and virtual GPU channel registers. Since the virtual channel registers are mapped to the memory aperture, GPUvm can intercept the access to them and redirect it to the physical channel registers. Since the guest GPU driver can dynamically change the location of the channel registers, GPUvm monitors it and changes the mapping if necessary.

3.5 GPU Fair-Share Scheduler

So far we have argued virtualization of memory resources and GPU channels for multiple VMs. We herein provide virtualization of GPU time. Indeed this is a scheduling problem. The GPU scheduler of GPUvm is based on the bandwidth-aware non-preemptive device

(BAND) scheduling algorithm [21], which was developed for virtual GPU scheduling. The BAND scheduling algorithm is an extension of the CREDIT scheduling algorithm [3] in that (i) the prioritization policy uses reserved bandwidth and (ii) the scheduler intentionally inserts a certain amount of waiting time after completion of GPU kernels, which leads to fairer utilization of GPU time among VMs. Since the current GPUs are not preemptive, GPUvm waits for GPU kernel completion and assigns credits based on the GPU usage. More details can be found in [21].

The BAND scheduling algorithm assumes that the total utilization of virtual GPUs could reach 100%. This is a flaw because there must be some interval that the CPU executes the GPU scheduler during which the GPU remains idle, causing utilization of the GPU to be less than 100%. This means that even though the total bandwidth is set to 100%, VMs' credit would be left unused, if the GPU scheduler consumes some time in the corresponding period. The problem is that the amount of credit to be replenished and the period of replenishment are fixed. If the fixed amount of credit is always replenished, after a while all VMs could have a lot of credit left unused. As a result, the credit may not influence the decision of scheduling at all. To overcome this problem, GPUvm accounts for CPU time consumed by the GPU scheduler, and considers it as GPU time.

Note that there is a critical problem in guaranteeing the fairness of GPU time. If a malicious or buggy VM starts infinite computation on GPU, it can monopolize GPU time. One possible solution to this problem is to abort the GPU computation if the GPU time exceeds the pre-defined limit of the computation time. Another approach is to cut longer requests into smaller pieces, as shown in [4]. For the future directions, we are planning to incorporate the disengaged scheduling [29] at the VMM level. The disengaged scheduling provides fair, safe and efficient OS-level management of GPU resources. We believe that GPUvm can incorporate the disengaged scheduling without any technical issues except for engineering efforts.

3.6 Optimization Techniques

Lazy Shadowing: In principle, GPUvm has to reflect the contents of guest page tables to shadow page tables every time it detects TLB flushes. As explained in Section 3.3, GPUvm has to scan the entire page table to find the modified entries in the guest page table because GPUvm cannot use page faults to detect the modifications on the guest page tables. Since TLB flushes can happen frequently, the cost of scanning page tables introduces significant overhead. To reduce this overhead, GPUvm delays the reflection to the shadow page tables until an attempt is made to reference them. The shadow page tables are used when memory apertures are accessed or after the GPU kernels starts. Note that GPUvm can intercept memory apertures access and command submission. So, GPUvm scans the guest page table at this point of time and reflects it to the shadow page table. By delaying the reflection, GPUvm can reduce the number of the page table scans.

BAR Remap: GPUvm intercepts data accesses through BARs to virtualize GPU channel descriptors. By intercepting all data accesses, it keeps the consistency between shadow GPU channel descriptors and guest GPU channel descriptors. However, this design incurs non-trivial overheads because the hypervisor is invoked every time the BAR is accessed. The BAR remap optimization reduces this overhead by limiting the handling of BAR accesses. In the BAR remap optimization, GPUvm passes through the BAR accesses other than to GPU channel descriptors because GPUvm does not have to virtualize the values read from or written to the BAR areas except for GPU channel descriptors. Even if the BAR accesses are passed through, they must be isolated among multiple VMs. This isolation is achieved by making use of shadow page tables. The BAR accesses from the host CPU all go through GPU page tables; the offsets in the BAR areas are regarded as virtual addresses in GPU memory and translated to GPU physical addresses through the shadow page tables. By setting shadow page tables appropriately, all the accesses to the BAR areas are isolated among VMs.

Para-virtualization: Shadowing of GPU page tables is a major source of overhead in full-virtualization, because the entire page table needs to be scanned to detect changes to the guest GPU page tables. To reduce the cost of detecting the updates, we take a para-virtualization approach. In this approach, the guest GPU page tables are placed within the memory areas under the control of GPUvm and cannot be directly updated by guest GPU drivers. To update the guest GPU page tables, the guest GPU driver issues hypercalls to GPUvm. GPUvm validates the correctness of the page table updates when the hypercall is issued. This approach is inspired by the direct paging in Xen para-virtualization [3].

Multicall: Hypercalls are expensive because the context is switched to the hypervisor. To reduce the number of hypercalls, GPUvm provides a multicall interface that can batch several hypercalls into one. For example, instead of providing a hypercall that updates one page table entry at once, GPUvm provides a hypercall that allows multiple page table entries to be updated by one hypercall. The multicall is borrowed from Xen.

4 Implementation

Our prototype of GPUvm uses Xen 4.2.0, where both the domain 0 and the domain U adopt the Linux kernel

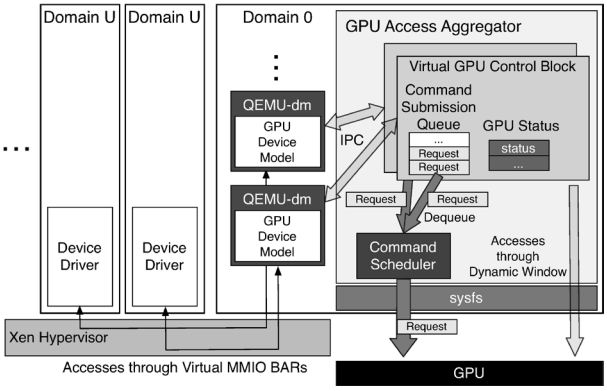


Figure 3: The prototype implementation of GPUvm.

v3.6.5. We target the device model of NVIDIA GPUs based on the Fermi/Kepler architectures [30, 31]. While full-virtualization does not require any modification to guest system software, we make a small modification to the GPU device driver called Nouveau, which is provided as part of the mainline Linux kernel, to implement our GPU para-virtualization approach.

Figure 3 shows the overview of implementation of the GPU Access Aggregator and its interactions with other components. The GPU device model is exposed in the domain U, which is created by the QEMU-dm and behaves as a virtual GPU. Guest device drivers in the domain U consider it as a normal GPU. It also exposes MMIO PCIe BARs, handling accesses to the BARs in Xen. All accesses to the GPU arbitrated by the GPU Access Aggregator are committed to the GPU through the `sysfs` interface.

The GPU device model communicates with the GPU Access Aggregator in the domain 0, using the POSIX inter-process communication (IPC). The GPU Access Aggregator is a user process in the domain 0, which receives requests from the GPU device model, and issues the aggregated requests to the physical GPU.

The GPU Access Aggregator has virtual GPU control blocks and the GPU command scheduler, which represent the state of virtual GPUs. The GPU device models update their own virtual GPU control blocks using IPC to manage the states of corresponding virtual GPUs when privileged events such as control register changes are issued from domain U.

Each virtual GPU control block maintains a queue to store command submission requests issued from the GPU device model. These command submission requests are scheduled to control GPU executions. This command scheduling mechanism is similar to TimeGraph [20]. However, the GPU command scheduler of GPUvm differs from TimeGraph in that it does not use GPU interrupts. It is very difficult, if not impossible, for the GPU Access Aggregator to insert the interrupt command to the original sequence of commands, because user contexts may also use some interrupt commands, and the GPU Access Aggregator cannot recognize them once they are fired. Therefore, our prototype implementation uses a thread-based scheduler polling on the request queue. Whenever command submission requests are stored in the queue, the scheduler dispatches them to the GPU. To calculate GPU time, our prototype polls a GPU control register value that is modified by the hardware just after GPU channels become active/inactive.

Another task of the GPU Access Aggregator is to manage GPU memory and maintain isolation of multiple VMs on partitioned memory resources. For this purpose, GPUvm creates shadow page tables and channel descriptors in the reserved area of GPU memory.

5 Experiments

To demonstrate the effectiveness of GPUvm, we conducted detailed experiments using a relevant commodity GPU. The objective of this section is to answer the following fundamental questions:

1. How much is the overhead of GPU virtualization incurred by GPUvm?
2. How does the number of GPU contexts affect performance?
3. Can multiple VMs meet fairness on GPU resources?

The experiments were conducted on a DELL PowerEdge T320 machine with eight Xeon E5-24700 2.3 GHz processors, 16 GB of memory, and two 500 GB SATA hard disks. We use NVIDIA Quadro 6000 for the target GPU, which is based on the NVIDIA Fermi architecture. We ran our modified Xen 4.2.0, assigning 4 GB and 1 GB of memory to the domain 0 and the domain U, respectively. In the domain U, Nouveau was running as the GPU device driver and Gdev [21] was running as the CUDA runtime. The following eight schemes were evaluated: **Native** (non-virtualized Linux 3.6.5), **PT** (pass-through provided by Xen's pass-through feature), **FV Naive** (full-virtualization w/o any optimization techniques), **FV BAR-Remap** (full-virtualization w/ BAR Remapping), **FV Lazy** (full-virtualization w/ Lazy Shadowing), **FV Optimized** (full-virtualization w/ BAR Remapping and Lazy Shadowing), **PV Naive** (para-virtualization w/o multicall), and **PV Multicall** (para-virtualization w/ multicall).

5.1 Overhead

To identify the overhead of GPU virtualization incurred by GPUvm, we run the well-known GPU benchmarks called Rodinia [5] as well as our microbenchmarks, as listed in Table 1. We measure their execution time on the eight platforms.

Table 1: List of the GPU benchmarks.

Benchmark	Description
NOP	No GPU operation
LOOP	Long-loop compute without data
MADD	1024x1024 matrix addition
MMUL	1024x1024 matrix multiplication
FMADD	1024x1024 matrix floating addition
FMMUL	1024x1024 matrix floating multiplication
CPY	64MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

5.1.1 Results

Figure 4 shows the execution time of the benchmarks on each platform. The x-axis lists the benchmark names while the y-axis exhibits the execution time normalized by one of Native. It is clearly observed that the overhead of GPU full-virtualization is mostly unacceptable, but our optimization techniques significantly contribute to reduction of this overhead. The execution times obtained in FV Naive are more than 100 times slower in nine benchmarks (nop, loop, madd, fmadd, fmmul, bp, hfs, hs, lid) than those obtained in Native. This overhead can be mitigated by using the BAR Remap and the Lazy Shadowing optimization techniques. Since these optimization techniques are complementary to each other, putting it together achieves more performance gain. The execution time is 6 times shorter in madd (the best case) while being 5 times shorter in mmul (the worst case). In some benchmarks, PT exhibits slightly faster performance than Native, especially in madd is 1.5 times shorter than PT. This is a GPU's mysterious behavior.

From these experimental results, we also find that GPU para-virtualization is much faster than full virtualization. The execution times obtained in PV Naive are 3 - 10 times slower than those obtained in Native except for pincpy. We discuss this reason in the next section. This overhead can also be reduced by our reduced hypercalls feature. The execution time increased in PV Multicall is at most 3 times.

5.1.2 Breakdown

The breakdown on the execution time of the GPU benchmarks is shown in Figure 5. We divide the total execution time into five phases; *init*, *htod*, *launch*, *dtoh*, and *close*. Init is time for setting up the GPU to execute a GPU kernel. Htod is time for host-to-device data transfers. Launch is time for the calculation on GPUs. Dtoh is device-to-host data transfer time. Close is time for destroying the GPU kernel. The figure indicates that the dominant factor of execution time in GPUvm is init and close phases. This tendency is significant for four GPUvm's full virtualization configurations. In FV Naive, init and close phases are more than 90 % in the execution times. By using optimization techniques, the phases' ratio becomes lowered.

Table 2 and 3 list BAR3 writes and shadow page table update counts in each benchmark. BAR3 is used for a memory aperture. BAR remapping achieves more performance gain in benchmarks that write more bytes in the BAR3 region. For example, the execution time of pincpy that writes 64 MB is 2 times shorter in FV BAR-Remap than FV Naive. Also, lazy shadowing works more effectively to the benchmarks that update shadow page tables more frequently. Specifically, the execution time of srad, where the shadow page table is updated in 52 times, is 2 times shorter in FV Lazy than FV Naive.

On the other hand, init and close phases in two GPUvm's para-virtualized platforms are much shorter than the full-virtualization configuration in all cases. Full-virtualization GPUvm performs many shadowing operations, including TLB flushes, since memory allocation are done frequently in the two phases. This cost can be significantly reduced in para-virtualization GPUvm in which memory allocations are requested by hypercall. Time spent in these phases is longer in pincpy since it issues more hypercalls than the other benchmarks. Table 4 lists the number of hypercall issues of each benchmark in PV Naive and PV Multicall. Compared to the other benchmarks, pincpy issues much more hypercalls. Also, our optimization dramatically reduces hypercall issues, resulting in the reduced overhead in PV Naive. As a result, the execution times in PV Multicall are close to those of PT and Native, compared with PV Naive's result. For example, the results in 7 benchmarks (mmul, cpu, pincpy, bp, hfs, srad, srad2) are similar in PV Multicall, PT, and Native.

5.2 Performance at Scale

To discern the overhead GPUvm incurs in multiple GPU contexts, we generate GPU workloads and measure their execution time in two scenarios; in the first scenario, one VM executes multiple GPU tasks, in the other scenario, multiple VM executes GPU tasks. We first launch 1, 2, 4, 8 GPU tasks in one VM with full-virtualized, para-virtualized, pass-throughed, GPU (*FV(1VM), PV(1VM), and PT*). These tasks are also run on native Linux (*Native*). Next, we prepare 1, 2, 4, 8 VMs and execute one GPU task on each VM with a full- or para-virtualized GPU (*FV and PV*) where all our optimizations are turned on. In each scenario, we run *madd* listed in Table 1. Specifically, we repeat the GPU kernel execution of madd 10000 times, and measure its execution time.

The results are shown in Fig. 6. The x-axis is the number of launched GPU contexts and the y-axis represents execution time. This figure reveals two points.

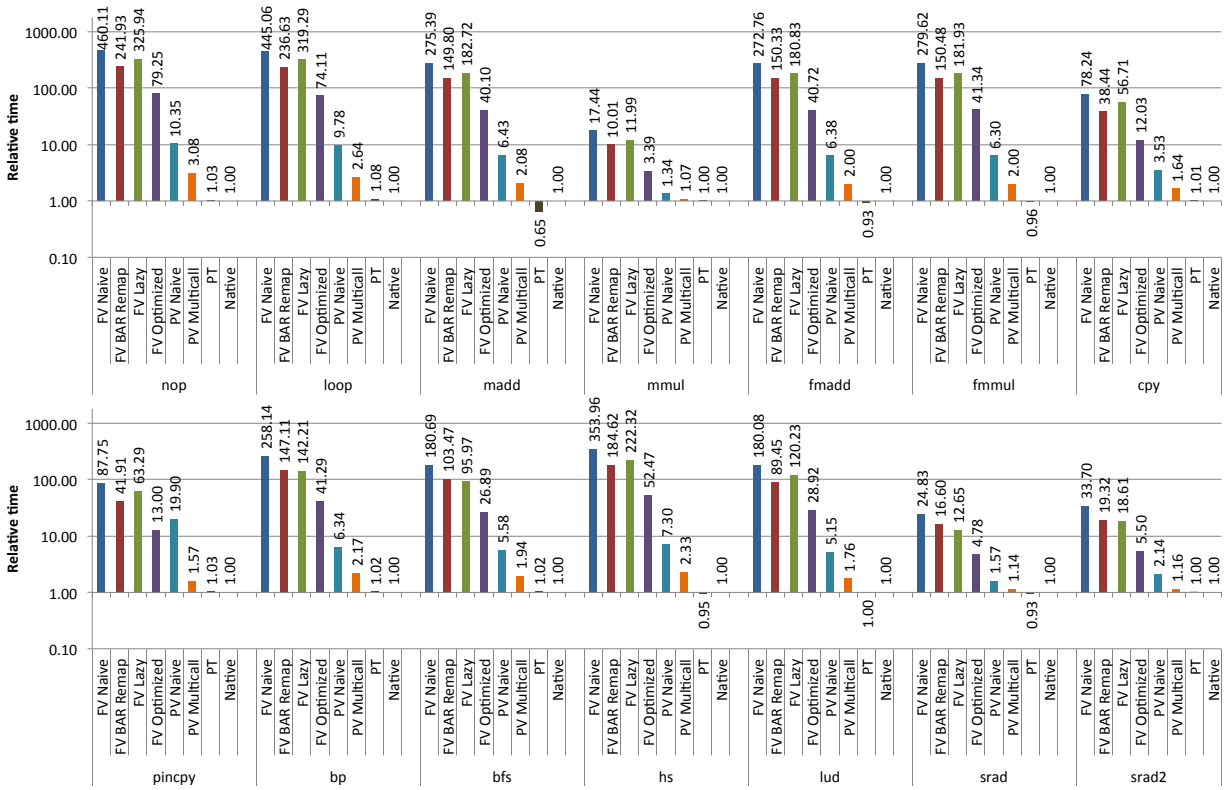


Figure 4: Execution time of the GPU benchmarks on the eight platforms.

Table 2: Total size of BAR access (bytes).

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
READ	0	0	0	0	0	0	0	0	0	0	0	0	0	0
WRITE	6688	6696	6648	6672	6680	6688	6168	268344	7280	6232	6736	7240	6352	6248

One is that full-virtualized GPUvm incurs larger overhead as GPU contexts are more. Time spent in init and close phases is longer in FV and FV (1VM) since GPUvm performs exclusive accesses to some GPU resources including the dynamic window. The other is that para-virtualized GPUvm achieves similar performance to pass-through GPU. The total execution time in PV and PV (1VM) is quite similar to those in PT even if GPU contexts are more. The kernel execution times in both FV (1VM) and FV are larger than the other GPU configurations, while those in the three GPU configurations are longer in 4 and 8 GPU contexts. This comes from GPUvm's overhead that it polls a GPU register to detect GPU kernel completion through MMIO.

5.3 Performance Isolation

To demonstrate how GPUvm achieves performance isolation among VMs, we launch a GPU workload on 2, 4, 8 VMs and measure each VM's GPU usage. For comparison, we use three schedulers; FIFO, CREDIT, and BAND scheduler. FIFO issues GPU requests in a first-in/first-out manner. CREDIT schedules GPU requests in a proportional fair share manner. Specifically, CREDIT reduces credits assigned to a VM in advance when its

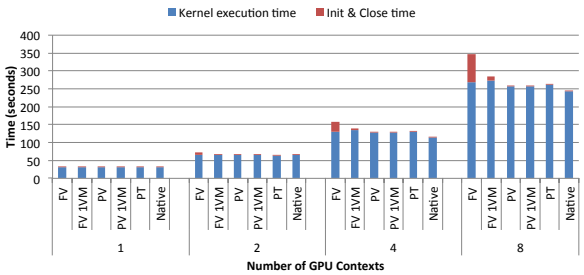


Figure 6: Performance across multiple VMs.

GPU requests are executed, and issues GPU requests of a VM whose credits are highest. BAND is our scheduler described in Sec.3.5. We prepare two GPU tasks; the one is madd, used in the previous experiment, and the other is an extended madd (*long-madd*), which performs 5 times more calculations than the regular madd. Each VM loops one of them. We run each task on a half of the VMs, respectively. For example, madd runs on 2 VMs while long-madd runs on 2 VMs in the 4-VM case.

Fig. 7 shows the results. The x-axis represents the elapsed time and the y-axis is VM's GPU usage over 500 msec. The figure reveals that BAND is the only scheduler that achieves performance isolation in all cases. In

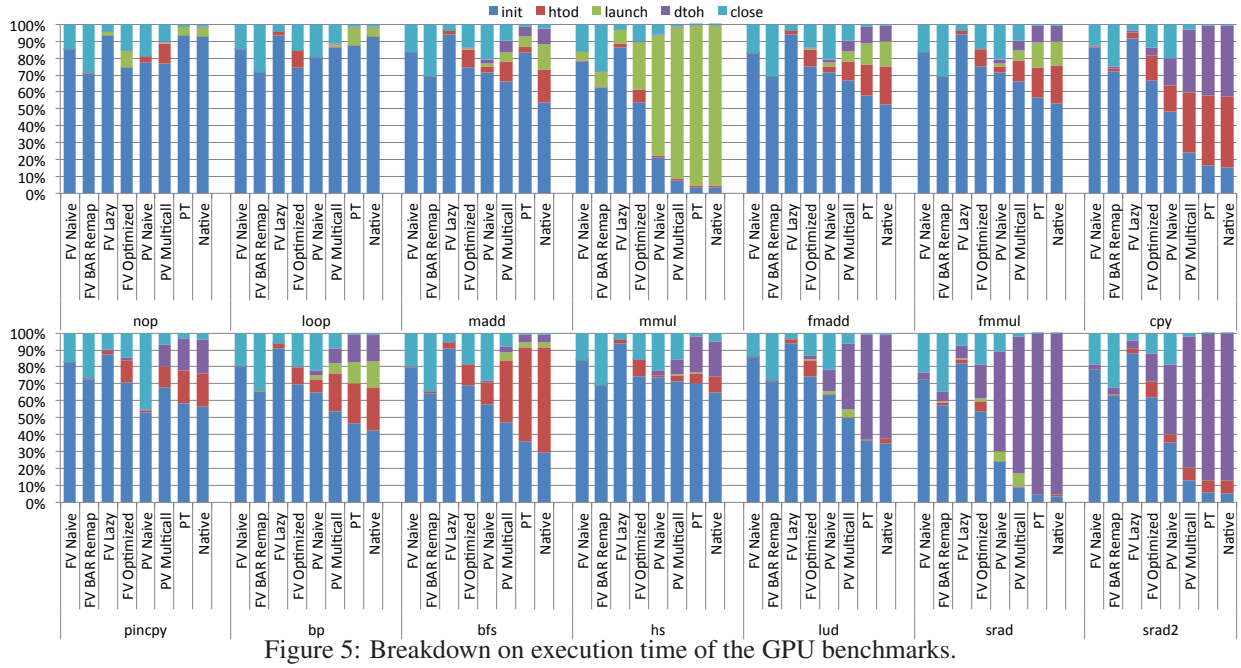


Figure 5: Breakdown on execution time of the GPU benchmarks.

Table 3: Update count for GPU shadow page tables.

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
FV Naive	30	30	34	34	34	34	26	28	40	42	34	30	52	40
FV Optimized	7	7	7	7	7	7	6	6	7	7	7	7	7	7

FIFO, GPU usages in long-madd are higher in all cases since FIFO dispatches more GPU commands from long-madd than madd. CREDIT fails to achieve the fairness among VMs in 2-VM case. Since the command submission request queue contains the requests only from long-madd just after the madd's commands completed, CREDIT dispatches the long-madd's requests. As a result, the VM running madd has to wait for the completion of the long-madd's commands. BAND waits for request arrivals for a short period just after a GPU kernel completes. BAND can handle the requests issued from the VMs whose GPU usage is less.

CREDIT achieves fair-share GPU scheduling in 4- and 8- VM. In these cases, CREDIT has more opportunities to dispatch less-credit VMs' commands for the following two reasons. First, GPUvm's queue has GPU command submission requests from two or more VMs just after a GPU kernel completes, differently from the 2-VM case. Second, GPUvm submits GPU operations from three or more VMs that complete shortly; GPUvm can have more scheduling points.

Note that BAND cannot achieve fairness among VMs in a fine-grained manner on the current GPUs. Fig. 8 shows VMs' GPU usages over 100 msec. Even with BAND, the GPU usages are fluctuated over time. This is because the GPU is a non-preemptive device. To achieve finer-grained GPU fair-share scheduling, we need a novel mechanism inside the GPU which effectively switches GPU kernels.

6 Related Work

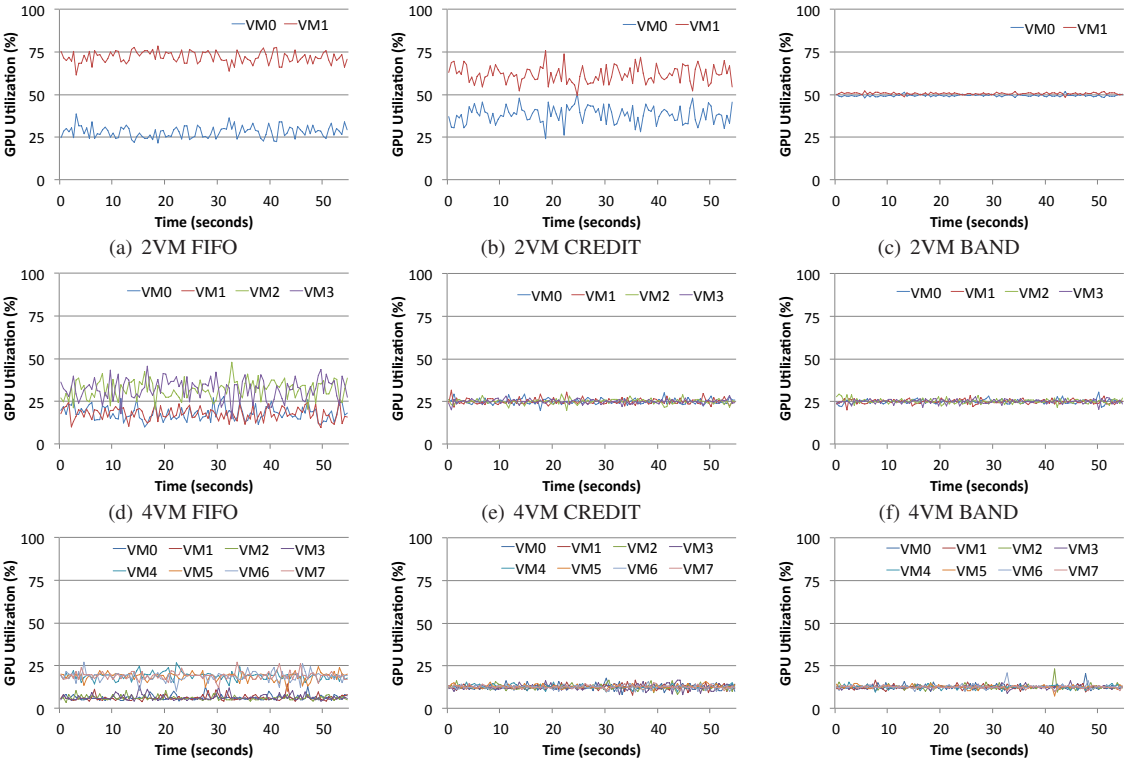
Table 5 briefly summarizes the characteristics of several GPU virtualization schemes, and compares them to GPUvm. Some vendors invent techniques of GPU virtualization. NVIDIA has announced NVIDIA VGX [32], which exploits virtualization supports of Kepler generation GPUs. These are proprietary so their details are closed. To the best of our knowledge, GPUvm is the first open architecture of GPU virtualization offered by the hypervisor. GPUvm carefully selects resources to virtualize, GPU page tables and channels, to keep its applicability to various GPU architectures.

VMware SVGA2 [7] para-virtualizes GPUs to mitigate the overhead of virtualizing GPU graphics features. The SVGA2 handles graphics-related requests by using an architecture-independent communication to efficiently perform 3D rendering and hide GPU hardware. While this approach is specific to graphics acceleration, GPUvm coordinates interactions between GPUs and guest device drivers.

Gottschalk et al. proposes low-overhead GPU virtualization, named LoGV, for GPGPU applications [10]. Their approach is categorized into para-virtualization where device drivers in VMs send requests for resource allocation and mapping memory into system RAM to the hypervisor. Similar to our work, this work exhibits para-virtualization mechanisms to minimize GPGPU virtualization overhead. Our work reveals which virtualization technique for GPUs is efficient in a quantitative way.

Table 4: The number of hypercall issues.

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
PV Naive	1230	1230	1420	1420	1420	1420	2010	34784	1628	1993	1169	1429	1985	2681
PV Multicall	93	93	97	97	97	97	81	218	117	117	97	89	149	107



(a) 2VM FIFO (b) 2VM CREDIT (c) 2VM BAND

(d) 4VM FIFO (e) 4VM CREDIT (f) 4VM BAND

(g) 8VM FIFO (h) 8VM CREDIT (i) 8VM BAND

Figure 7: VMs' GPU usages (over 500 ms) on the three GPU schedulers.

API remoting, in which API calls are forwarded from the guest to the host which has the GPU, have been studied widely. GViM [11], vCUDA [37], and rCUDA [8] forward CUDA APIs. VMGL [24] achieves API remoting of OpenGL. gVirtuS [9] supports API remoting of CUDA, OpenCL, a part of OpenGL. In these approaches, applications are inherently limited to APIs the wrapper-libraries offer. Keeping the wrapper-libraries compatible to the original ones is not trivial task since new functionalities are frequently integrated into GPU libraries including CUDA and OpenCL. Moreover API remoting requires that the whole GPU software stacks including device drivers and runtimes become part of the TCB.

Amazon EC2 [1] provides GPU instances. It makes use of pass-through technology to expose a GPU to an instance. Since a pass-throughed GPU is directly managed by the guest OS, we cannot multiplex the GPU on a physical host.

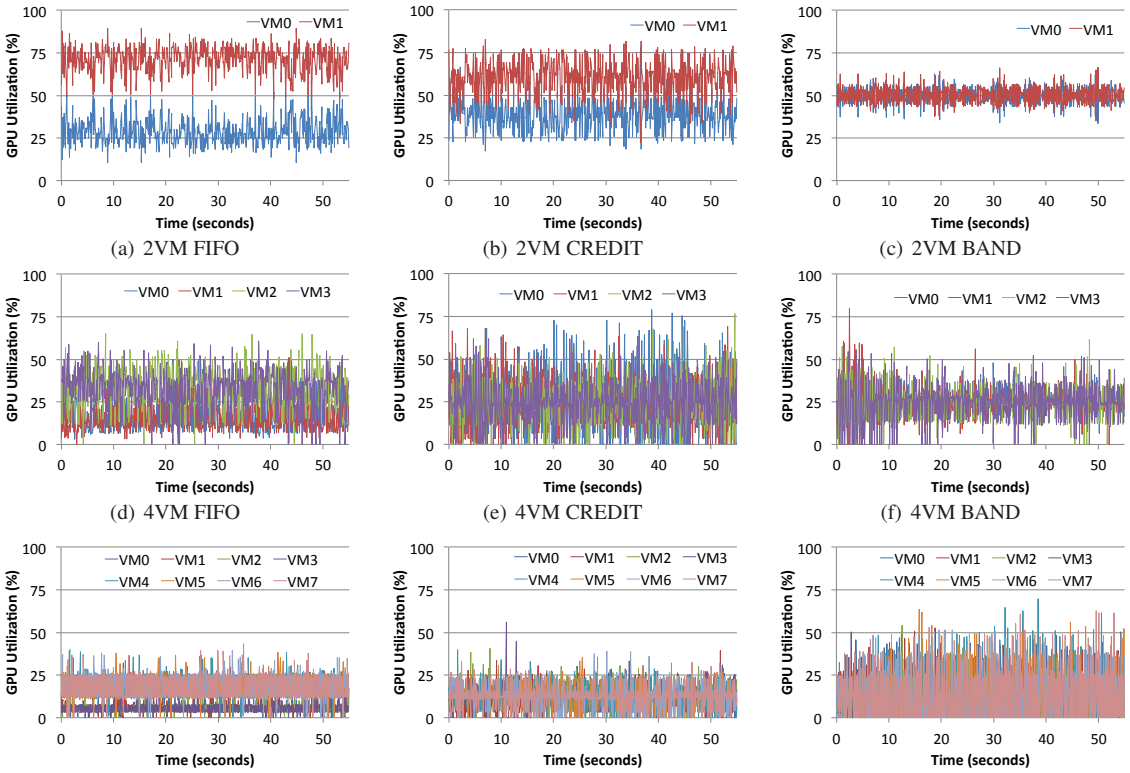
GPUvm is complementary to GPU command scheduling methods. VGRIS [41] enables us to schedule GPU commands in SLA-aware, proportional-share or the hybrid scheduling. Pegasus [12] coordinates GPU command queuing and CPU dispatching so that multi-VMs can effectively share CPU and GPU resources. Disengaged Scheduling [29] applies fair queuing scheduling with a probabilistic extension to GPU, and provides protection and fairness without compromising efficiency.

XenGT [16] is GPU virtualization for Intel on-chip GPUs with the similar design to GPUvm. Our work provides detailed analysis of the performance bottlenecks of the current GPU virtualization. It is useful to design the architecutre of the GPU virtualization, and also useful for GPU vendors to design the future GPU architecture which supports virtualization.

Some work aims at the efficient management of GPUs at the operating system layer such as GPU command scheduler [20], kernel-level GPU runtime [21], OS abstraction of GPUs [34,35] and file system for GPUs [39]. These mechanisms can be incorporated into GPUvm.

7 Conclusion

In this work, we try to answer the question; why not virtualizing GPU at the hypervisor? This paper have presented GPUvm, an open architecture of GPU virtualization. GPUvm supports full-virtualization and para-virtualization with optimization techniques. Experimental results using our prototype showed that full-



(a) 2VM FIFO (b) 2VM CREDIT (c) 2VM BAND

(d) 4VM FIFO (e) 4VM CREDIT (f) 4VM BAND

(g) 8VM FIFO (h) 8VM CREDIT (i) 8VM BAND

Figure 8: VMs' GPU usages (over 100 ms) on the three GPU schedulers.

Table 5: A comparison of GPUvm to other GPU virtualization schemes.

	Category	W/o Library mod.	W/o kernel mod.	Multiplexing	Scheduling
vCUDA [37], rCUDA [8]	API Remoting	×	√	√	×
VMGL [24]		×	√	√	×
VGRIS(gVirtuS) [9,41]		×	√	√	SLA-aware, Proportional-share
Pegasus(GViM) [11,12]		×	√	√	SLA-aware, Throughput-based, Proportional fair-share
SVGA2 [7]	Para-Virt.	√	×	√	×
LoGV [10]		√	×	√	×
GPU Instances [1]	Pass-through	√	×	×	×
GPUvm	Full-Virt.& Para-Virt.	√	√	√	Credit-based fair-share

virtualization exhibits non-trivial overhead largely due to MMIO handling, and para-virtualization provides yet two or three times slower performance than pass-through and native approaches. Also the results reveal that para-virtualization is preferred in performance, though highly compute-intensive GPU applications may also benefit from full-virtualization if their execution times are much larger than the overhead of full-virtualization.

For future directions, it should be investigated that the optimization techniques proposed in vIOMMU [2] can be applied to GPUvm. We hope our experience in GPUvm gives insight into designing the support of device virtualization such as SR-IOV [6].

8 Acknowledgements

This work was supported in part by the Grant-in-Aid for Scientific Research B (25280043) and C (23500050).

References

[1] AMAZON.COM. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`, 2014.

[2] AMIT, N., BEN-YEHUDA, M., TSAFRIR, D., AND SCHUSTER, A. vIOMMU: Efficient IOMMU Emulation. In *USENIX ATC* (2011), pp. 73–86.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *ACM SOSP* (2003), pp. 164–177.

[4] BASARAN, C., AND KANG, K.-D. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *Proc. of Euromicro Conf. on Real-Time Systems* (2012), IEEE, pp. 287–296.

[5] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of IEEE Int'l Symp. on Workload Characterization* (2009), pp. 44–54.

[6] DONG, Y., YU, Z., AND ROSE, G. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Proc. of Workshop on I/O Virtualization* (2008).

[7] DOWTY, M., AND SUGERMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[8] DUATO, J., PENA, A. J., SILLA, F., MAYO, R., AND QUINTANA-ORTI, E. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. of IEEE Int'l Conf. on High Performance Computing Simulation* (2010), pp. 224–231.

[9] GIUNTA, G., MONTELLA, R., AGRILLO, G., AND COVIELLO, G. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proc. of Int'l Euro-Par Conf. on Parallel Processing* (2010), Springer, pp. 379–391.

[10] GOTTSCHLAG, M., HILLENBRAND, M., KEHNE, J., STOESS, J., AND BELLOSA, F. LoGV: Low-overhead GPGPU Virtualization. In *Proc. of Int'l Workshop on Frontiers of Heterogeneous Computing* (2013), IEEE, pp. 1721–1726.

[11] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. GViM: GPU-Accelerated Virtual Machines. In *Proc. of ACM Workshop on System-level Virtualization for High Performance Computing* (2009), pp. 17–24.

[12] GUPTA, V., SCHWAN, K., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *USENIX ATC* (2011), pp. 31–44.

[13] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 195–206.

[14] HE, B., YANG, K., FANG, R., LU, M., GOVINDARAJU, N., LUO, Q., AND SANDER, P. Relational Joins on Graphics Processors. In *ACM SIGMOD* (2008), pp. 511–524.

[15] HIRABAYASHI, M., KATO, S., EDAHIRO, M., TAKEDA, K., KAWANO, T., , AND MITA, S. Gpu implementations of object detection using hog features and deformable models. In *Proc. of IEEE Int'l Conf. on Cyber-Physical Systems, Networks, and Applications* (2013), pp. 106–111.

[16] INTEL. Xen - Graphics Virtualization (XenGT). `https://01.org/xen/blogs/srclarkx/2013/graphics-virtualization-xengt`, 2013.

[17] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *USENIX NSDI* (2011), pp. 1–14.

[18] KALDEWEY, T., LOHMAN, G. M., MÜLLER, R., AND VOLK, P. B. GPU Join Processing Revisited. In *Proc. of Int'l Workshop on Data Management on New Hardware* (2012), pp. 55–62.

[19] KATO, S., AUMILLER, J., AND BRANDT, S. Zero-copy I/O processing for low-latency GPU computing. In *Proc. of ACM/IEEE Int'l Conf. on Cyber-Physical Systems* (2013), pp. 170–178.

[20] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., AND ISHIKAWA, Y. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX ATC* (2011), pp. 17–30.

[21] KATO, S., MCTHROW, M., MALTZAHN, C., AND BRANDT., S. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX ATC* (2012), pp. 401–412.

[22] KIM, C., CHHUGANI, J., SATISH, N., SEDLAR, E., NGUYEN, A. D., KALDEWEY, T., LEE, V. W., BRANDT, S. A., AND DUBEY, P. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *ACM SIGMOD* (2010), pp. 339–350.

[23] KOSCIELNICKI, M. Envytools. `https://0x04.net/envytools.git`, 2014.

[24] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. VMM-Independent Graphics Acceleration. In *ACM VEE* (2007), pp. 33–43.

[25] LINUX OPEN-SOURCE COMMUNITY. Nouveau Open-Source GPU Device Driver. `http://nouveau.freedesktop.org/`, 2014.

[26] MARUYAMA, N., NOMURA, T., SATO, K., AND MATSUOKA, S. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–12.

[27] MCNAUGHTON, M., URMSON, C., DOLAN, J. M., AND LEE, J.-W. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of IEEE Int'l Conf. on Robotics and Automation* (2011), pp. 4889–4895.

[28] MENYCHTAS, K., SHEN, K., AND SCOTT, M. L. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. In *USENIX ATC* (2013), pp. 291–296.

[29] MENYCHTAS, K., SHEN, K., AND SCOTT, M. L. Disengaged scheduling for fair, protected access to fast computational accelerators. In *ACM ASPLOS* (2014), pp. 301–316.

[30] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi. `http://www.nvidia.com/`, 2009.

[31] NVIDIA. NVIDIA's next generation CUDA computer architecture: Kepler GK110. `http://www.nvidia.com/`, 2012.

[32] NVIDIA. NVIDIA GRID VGX SOFTWARE. `http://www.nvidia.com/object/grid-vgx-software.html`, 2014.

[33] RATH, N., BIALEK, J., BYRNE, P., DEBONO, B., LEVESQUE, J., LI, B., MAUEL, M., MAURER, D., NAVRATIL, G., AND SHIRAKI, D. High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak. *Fusion Engineering and Design* (2012), 1895–1899.

[34] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *ACM SOSP* (2011), pp. 223–248.

[35] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., AND FETTERLY, D. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *ACM SOSP* (2013), pp. 49–68.

[36] SATISH, N., KIM, C., CHHUGANI, J., NGUYEN, A. D., LEE, V. W., KIM, D., AND DUBEY, P. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *ACM SIGMOD* (2010), pp. 351–362.

[37] SHI, L., CHEN, H., SUN, J., AND LI, K. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers 61*, 6 (2012), 804–816.

[38] SHIMOKAWABE, T., AOKI, T., TAKAKI, T., ENDO, T., YAMANAKA, A., MARUYAMA, N., NUKADA, A., AND MATSUOKA, S. Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–11.

[39] SILBERSTEIN, M., FORD, B., KEIDAR, I., AND WITCHEL, E. GPUfs: Integrating a File System with GPUs. In *ACM ASPLOS* (2013), pp. 485–498.

[40] SUN, W., RICCI, R., AND CURRY, M. L. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In *Proc. of Int'l Systems and Storage Conf.* (2012), pp. 9:1–9:12.

[41] YU, M., ZHANG, C., QI, Z., YAO, J., WANG, Y., AND GUAN, H. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *ACM HPDC* (2013), pp. 203–214.