GPUvm: GPU Virtualization at the Hypervisor

Yusuke Suzuki, Shinpei Kato, *Member, IEEE,* Hiroshi Yamada, *Member, IEEE,* and Kenji Kono, *Member, IEEE*

Abstract—Graphic processing units (GPUs) provide a massively-parallel computational power and encourage the use of general-purpose computing on GPUs (GPGPU). The distinguished design of *discrete GPUs* helps them to provide the high throughput, scalability, and energy efficiency needed for GPGPU applications. Despite the previous study on GPU virtualization, the tradeoffs between the virtualization approaches remain unclear, because of a lack of designs for or quantitative evaluations of the hypervisor-level virtualization for discrete GPUs. Shedding light on these tradeoffs and the technical requirements for the hypervisor-level virtualization would facilitate the development of an appropriate GPU virtualization solution. GPUvm, which is an open architecture for hypervisor-level GPU virtualization with a particular emphasis on using the Xen hypervisor, is presented in this paper. GPUvm offers three virtualization modes: the full-, naive para-, and high-performance para-virtualization. GPUvm exposes low- and high-level interfaces such as memory-mapped I/O and DRM APIs to the guest virtual machines (VMs). Our experiments using a relevant commodity GPU showed that GPUvm incurs different overheads as the level of the exposed interfaces is changed. The results also showed that a coarse-grained fairness on the GPU among multiple VMs can be achieved using GPU scheduling.

Index Terms—GPU, Virtualization, GPGPU, Virtual Machines.

1 Introduction

Graphics processing units (GPUs) are recognized as distinguished accelerators because of their significant performance benefit and high energy efficiency. GPUs are composed of thousands of compute cores, which enable massively data-parallel computations. Beyond the graphics purpose, general-purpose computing on GPUs (GPGPU) is appealing in various application domains, such as in scientific simulations [1], [2], network systems [3], [4], file systems [5], [6], web servers [7], database management systems [8], [9], [10], [11], complex control systems [12], [13], and autonomous vehicles [14], [15].

Discrete GPUs are widely accepted for their intensive computational ability when compared with *integrated* GPUs in the GPGPU field. GPUs are classified into *discrete* (on-board and off-chip) and *integrated* (on-chip) GPUs. Discrete GPUs are connected on the PCI express bus (PCIe) and are composed of a huge number of cores tightly coupled with a specialized high bandwidth device memory, while integrated GPUs reside on the same chip as the CPUs and share system memory with CPUs. As discussed in [16], the discrete GPU design delivers a greater computational performance and a higher energy efficiency, whereas integrated GPUs are oriented to a lower latency and a lower thermal design power (TDP). The more recent research has leveraged discrete GPUs to create high-performance, scalable, and more energy efficient cloud applications [16], [17], [18]. This paper focuses on discrete GPUs.

Virtualizing GPUs is required in order to allow GPUs to be shared among the users in a cloud. Since GPUs and their relevant system software are not virtualized, GPGPU computations are prevented from becoming isolated among multiple clients. Thus, GPUs cannot be shared among multiple tenants without first virtualizing them. Supporting GPU virtualization would enable GPU computations to be isolated among the virtual machines (VMs), which are used as the logical unit of the computing resources in a cloud.

We have summarized the pros and cons of the current approaches to GPU virtualization. They are classified into I/O pass-through [19], API remoting [20], [21], [22], [23], [24], [25], [26], hybrid [27], or mediated pass-through [28]. These approaches are also referred to as *back-end*, *front-end*, *para*, and *full* virtualization, respectively [27].

I/O pass-through, which directly exposes the GPU hardware to guest device drivers, can provide close to a native performance, but a physical GPU is assigned to a single VM by hardware design.

API remoting is more suitable for multi-tasking and is relatively easy to implement. A high-level API such as CUDA in this approach is exported to the guest VMs by installing a wrapper library. The API calls from the guest VMs through a wrapper library are routed to the server owning the GPUs, and then, the server invokes the APIs through the original library. Although this approach is simple, it lacks flexibility in the choice of languages and libraries and can cause a version incompatibility between a wrapper library and an original library. 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 the need to accommodate for additional libraries and drivers in the server.

Para-virtualization provides an ideal device model through the hypervisor and allows multiple VMs to concurrently access the GPU. It can provide lower-level control to the guest drivers than in API remoting and minimizes the overhead of the virtualization, but the guest device drivers must be modified to support the device model.

Full-virtualization enables for multiplexing without needing any drivers or runtime modification. It allows guest VMs to use vanilla device drivers while providing resource

- *Y. Suzuki and K. Kono are with Keio University.*
- *S. Kato is with Nagoya University.*
- *H. Yamada is with Tokyo University of Agriculture and Technology.*

isolation on multiple VMs for GPGPU. These features are attractive to IaaS environments on which the users can use existing GPGPU software stacks without any guest modifications. However, to the best of our knowledge, no designs or evaluations of the full-virtualization for discrete GPUs have been reported. gVirt [28] enables for the full-virtualization of the Intel integrated GPUs, but they have different hardware designs than those for discrete GPUs.

Despite all the study on the GPU virtualization, the tradeoffs between the virtualization approaches remain unclear because of a lack of designs for and quantitative evaluations of the hypervisor-level virtualization for discrete GPUs. Exploring the hypervisor-level virtualization that includes both the full- and para-virtualizations clarifies the tradeoffs and technical difficulties in the virtualization approaches and allows hypervisor developers and hardware designers to design and discuss efficient virtualization solutions for GPUs.

Contribution: GPUvm, which is an open architecture for the GPU virtualization is presented in this paper. We developed three GPU virtualization modes: full-, naive para- and high-performance para-virtualization, in order to more closely study the hypervisor-level GPU virtualization. In the full- and naive para-virtualizations, GPUvm exposes a native GPU device model to provide a low-level interface through memory-mapped I/O (MMIO). In the naive para-virtualization, GPUvm provides a hypercall interface to mitigate the major source of overhead in the full-virtualization. In high-performance para-virtualization, which is called PVDRM, GPUvm exposes the high-level interface. PVDRM leverages the *Direct Rendering Manager* (DRM) APIs as an interface. DRM is a widely used GPU abstraction layer in Linux. It is used in open-source GPU drivers including i915 for the Intel Integrated GPUs, AMDGPU for the AMD GPUs, and Nouveau for the NVIDIA GPUs. We also developed several optimization techniques to reduce the overhead in each GPU virtualization. We describe the design and implementation of GPUvm based on the Xen hypervisor [29], and it is provided as a complete open-source software [1], [2].

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

2 Model

The system is composed of a multi-core CPU and a GPU connected on the bus. A compute-intensive function offloaded from the CPU to the GPU is called a *GPU kernel*, which can 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.

The product lines of the GPU vendors are closely tied to the programming languages and architectures. For example,

1. https://github.com/CPFL/gxen
2. https://github.com/CPFL/pvdrm

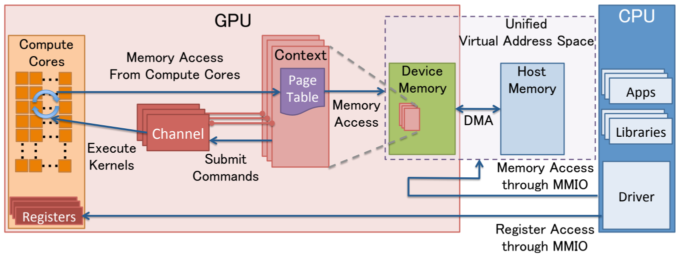

Fig. 1. GPU resource management model.

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

Fig. 1 illustrates the resource management model of our target GPU, which is well aligned with, but is not limited to, the NVIDIA architectures. The detailed hardware mechanism is not identical among the different vendors, although recent GPUs have adopted the same high-level design.

MMIO: The current GPU form is an independent computing device. Therefore, the CPU communicates with the GPU via MMIO. MMIO is the main interface that the CPU uses to directly access the GPU, while the hardware engines for the direct memory access (DMA) are supported for transferring large amounts of data. We must note that the I/O ports are used to indirectly access the above MMIO regions. The I/O port is rarely used since it is intended to be used in the real mode, which cannot map a high memory address. In fact, Nouveau, which is an open-source device driver, never accesses it.

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

GPU Channel: Any operation on the GPU is driven by commands (e.g., launching a kernel) issued from the CPU. This command stream is submitted to a hardware unit called a GPU *channel* and is 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 contains 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 are executed in the GPU *compute cores* and the execution is confined to within the associated GPU context. For each GPU channel, a dedicated command buffer is allocated in the GPU memory that is visible to the CPU through MMIO. The GPU commands can be simultaneously submitted from multiple GPU contexts through the GPU channels. The GPU context switching and command executions in the GPU compute cores are scheduled internally by the GPU hardware.

GPU Page Table: Paging is supported by the GPU. The GPU context is assigned using the GPU page table, which isolates the virtual address space from the others. The GPU

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2506582, IEEE Transactions on Computers

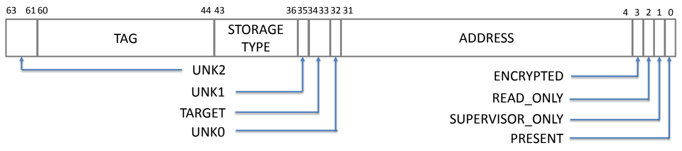

Fig. 2. Format of GPU page table entry.

page table is separated from the CPU page table. It resides in the GPU memory and its physical address is in a GPU channel descriptor. All the commands and programs submitted through the channel are executed in the corresponding GPU virtual address space.

The GPU page tables translate a GPU virtual address into not only a GPU device physical address but also a host physical address. Fig. 2 shows the format of the page table entries in the NVIDIA Fermi architecture [32]. `TARGET` indicates the memory type of the target page. We specify a memory type among the following three types; `VRAM`, `SYSRAM` and `SYSRAM_NO_SNOOP`. When the `TARGET` entry is `VRAM`, the GPU page table translates a given GPU virtual address to a GPU device physical address. When the `TARGET` entry is `SYSRAM` or `SYSRAM_NO_SNOOP`, the GPU page table translates a given GPU virtual address to a host physical address. This enables the GPU page table to unify the GPU memory and host main memory into the unified GPU virtual address space. The commands executed in the GPU context can access the host physical memory using the GPU virtual address by leveraging the GPU page tables.

The GPU context uses a GPU virtual address that indicates the host physical address in the GPU page table for initiating the DMA to the associated host memory.

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

Documentation: GPU vendors currently withhold the details of their GPU architectures due to marketing reasons. Implementations of the device drivers and runtime libraries are also protected by the binary proprietary software, whereas the compiler source code from NVIDIA has recently been open-released to a limited extent. Some previous works have uncovered the black-boxed interaction between the GPU and the driver [33]. The Linux kernel community has recently developed Nouveau [34], which is an open-source device driver for NVIDIA GPUs. Throughout their development, the details of the NVIDIA architectures have been well documented in the Envytools project [32]. Interested readers are encouraged to visit their website.

Scope and Limitation: GPUvm is an open architecture for a GPU virtualization that has a solid design and implementation using Xen. It does not support those GPUs prior to the NVIDIA Fermi architecture. We also restrict our attention to using Nouveau as the guest device driver. The NVIDIA binary drivers should be available with GPUvm, but they cannot be successfully loaded using the current versions of Xen, even in Xen domain 0, which also was the case in the previous work [22], [23].

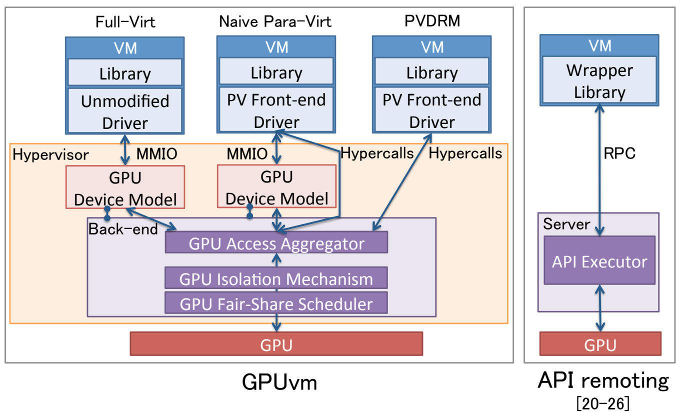

Fig. 3. Software stack of GPUvm.

3 Design

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

3.1 Approaches

Fig. 3 shows a high-level overview of the software stack of GPUvm. GPUvm exposes interfaces to each VM and aggregates the accesses to it. VM operations within the interfaces such as MMIO and hypercalls are redirected to the hypervisor so that the VMs can never directly access the GPU. The GPU Access Aggregator arbitrates the redirected operations to the multiplex GPU resources.

The GPU memory and channels must be multiplexed among multiple VMs to isolate them on the GPU hardware resources. In addition to this spacial multiplexing, the GPU also needs to be scheduled in a fair-share manner. GPUvm logically partitions GPU channels and assigns some of them to each VM. GPUvm also makes use of the GPU page table to isolate the GPU memory among the GPU contexts of different VMs. GPUvm introduces the GPU fair-share scheduler for the GPU command submissions in order to multiplex the GPU computation time.

GPUvm exposes the interfaces on several levels. For the full-virtualization, it exposes a native GPU device model to the VM where the guest device drivers are loaded. For the naive para-virtualization, in addition to the GPU device model, GPUvm provides the interface for updating the GPU page tables. While the full- and naive para-virtualizations use a low-level interface through MMIO, GPUvm exposes the high-level interface for PVDRM, which is the high-performance para-virtualization.

3.2 Full-Virtualization

GPUvm supports the hypervisor-level full-virtualization for GPUs., The memory areas, PCIe BARs, and GPU channels must be multiplexed in order to provide an isolated native GPU device model. The main components for the full-virtualization of GPUvm to address this problem include the GPU shadow page tables and GPU shadow channels.

To aggregate the accesses to a GPU device model from a guest device driver, GPUvm intercepts the MMIO by setting these ranges as inaccessible. The accesses to the I/O ports are trapped in the hypervisor, and they are emulated by changing these accesses into ones for the appropriate MMIO region.

In order to ensure that one VM can never access the memory areas of the other VMs, GPUvm creates a GPU *shadow* page table for every GPU channel descriptor. The entire GPU memory address translation is done using the GPU shadow page tables; a virtual address for the GPU memory is translated using the shadow page table not using the one set by the guest device driver. The GPU memory can be safely shared by multiple VMs because GPUvm validates the contents of the shadow page tables. The use of the GPU shadow page tables also guarantees that the DMA initiated from the GPU never accesses memory areas outside those allocated to the VM.

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

3.2.1 Resource Partitioning

GPUvm partitions the physical memory space into multiple sections of continuous address space, each of which is assigned to an individual VM. The guest device drivers consider that the physical memory space originates at 0, but the actual memory access is shifted by the corresponding size through the shadow page tables created by GPUvm. Similarly, the GPU channels are partitioned into multiple same-sized sections for individual VMs.

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

3.2.2 GPU Shadow Page Table

GPUvm creates GPU shadow page tables in the reserved area of the GPU memory, which translates the guest GPU virtual addresses into GPU device physical or host physical addresses. By design, the device driver needs to flush the TLB caches every time a page table entry is updated. GPUvm can intercept the TLB flush requests because they are issued from the guest device driver 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, the shadow page tables, the shadow channel descriptors, and the GPU contexts from buggy or malicious VMs. GPUvm excludes any memory mappings to the sensitive memory pages from the shadow page tables. Since all the memory accesses by the GPU go through the shadow page tables, no VMs can access these sensitive memory areas.

The current GPU design poses the technical challenge of maintaining consistency between the guest and shadow page tables. In the traditional shadow page tables, page faults are extensively used to detect the updates to the guest page table entries. However, the current NVIDIA GPUs abort the execution of GPU kernels after page faults occur [35], [36]. It is impossible to employ the typical shadow page technique that restarts the guest code after setting an appropriate page table entry during page fault handling. Therefore, GPUvm scans all the page tables during a TLB flush.

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

3.2.3 GPU Shadow Channel

GPUvm takes the approach of assigning dedicated GPU channels to each VM. As described in Section 2, the GPU has multiple GPU channels. They have dedicated command buffers and the driver can simultaneously push commands to the buffers. GPUvm partitions the GPU channels and assigns some of them to each VM. This design enables GPUvm to simultaneously accept GPU commands from the VMs. In addition, compared to multiplexing one GPU channel among the VMs, it does not incur any overhead when switching the GPU context belonging to the GPU channel. Since the GPU commands executed through the assigned GPU channels are confined by the GPU shadow page tables, the isolation among the VMs is maintained.

GPUvm provides GPU shadow channels to isolate the GPU accesses from the VMs. The physical indexes of the GPU channels are hidden from the VMs, but the virtual indexes are assigned to their virtual channels. Mapping between the physical and virtual indexes is managed by GPUvm. GPUvm intercepts the MMIO operations for the virtual GPU channels and then translates the virtual GPU channel indexes into shadow ones and performs the operations to the corresponding channel.

GPUvm provides virtual channel registers to each VM and maintains the mapping between the physical and virtual channel registers. Since the virtual channel registers are mapped to the memory aperture, GPUvm can intercept any access to them and redirect it to the physical channel registers. Furthermore, intercepting the command submission requests enables GPUvm to schedule the GPU command executions.

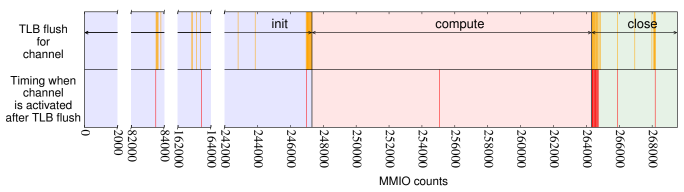

Fig. 4. Timings of TLB flush and channel activation in GPGPU application (*srad*). The orange lines in this figure represent the TLB flush timing and the red ones are the timings of the channel activations, where the corresponding shadow page table starts to be used.

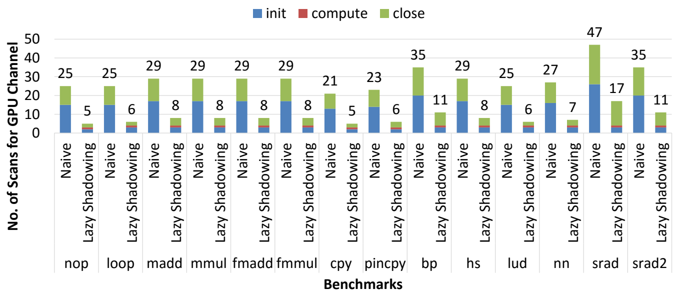

Fig. 5. No. of scanning shadow page tables in several benchmarks with/without Lazy Shadowing. The measurement is done in each execution phase: *init*, *compute*, and *close*.

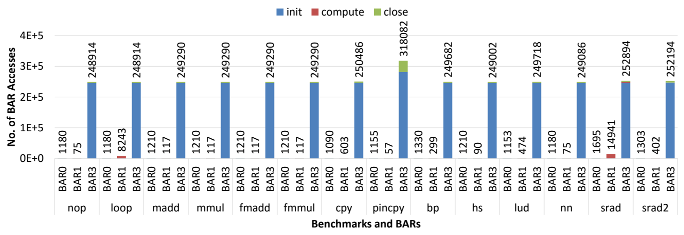

Fig. 6. No. of BAR0, 1, 3 accesses in benchmarks. The measurement is done in each execution phase: *init*, *compute*, and *close*

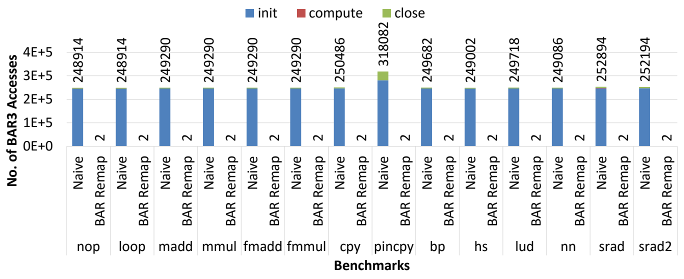

Fig. 7. No. of BAR3 traps in benchmarks with/without BAR Remap. The measurement is done in each execution phase: *init*, *compute* and *close*

GPUvm also creates a GPU *shadow* channel descriptor for each GPU shadow channel to achieve the isolation between virtual GPU channels. The GPU shadow channel descriptors are set for each GPU shadow channel and have a reference to the GPU shadow page table used by the corresponding channel. The GPU shadow channel descriptors reside in the reserved GPU memory and are protected from the VMs in a similar way as for GPU shadow page tables. GPUvm intercepts the GPU memory accesses through MMIO, detects the accesses to the guest channel descriptors, and maintains a consistency between the guest channel descriptors and the shadow ones.

3.2.4 Optimization Techniques

Several optimization techniques are introduced to reduce the overhead in GPUvm.

Lazy Shadowing: In principle, GPUvm reflects the updates of the guest page tables to the shadow page tables every TLB flush. As explained in Section 3.2.2, GPUvm scans the entire page table to find the updated entries in the guest page table. Since TLB flushes frequently occur and the page table size is large, the cost of scanning the page tables is significant. Fig. 4 shows the timings of the TLB flushes and channel activations during the execution of a GPGPU application. Although the TLB is frequently flushed, the shadow page table is often unused immediately after it.

GPUvm lazily scans the guest page tables to reduce the frequency of the scans. It scans the guest page tables each GPU channel activation, which is the timing for using the GPU page table. GPUvm detects the activation by checking the intercepted MMIO operations. Fig. 5 shows the number of guest page table scans in the benchmarks summarized in Table 2. Lazy Shadowing reduces the page table scans in all the benchmarks. Since some of the scans in the *init* phase are delayed until the channel activation point, the scan happens in the *compute* phase.

BAR Remap: GPUvm intercepts the data accesses through the BARs to virtualize the GPU channel descriptors. By intercepting all the data accesses, it maintains the consistency between the shadow GPU channel descriptors and guest GPU channel descriptors. However, this design incurs non-trivial overhead because the hypervisor is invoked every time the BAR is accessed. Fig. 6 shows the number of BAR accesses in the benchmarks. Even simple benchmarks such as madd significantly access BAR3, causing overhead from the MMIO trappings.

In the BAR Remap optimization, GPUvm passes through the BAR3 accesses other than those for the GPU channel descriptors. Specifically, GPUvm logically partitions the BAR3 area, which is used for a memory aperture. It exposes part of them as virtual BARs for each VM, and then, GPUvm creates a shadow page table for the physical BAR3. All the accesses to the BAR areas are isolated among the VMs by setting up shadow page tables in the same way as the shadow channels. The number of trapped BAR3 accesses under this optimization is shown in Fig. 7. This optimization significantly reduces the BAR3 traps in all the benchmarks.

3.3 Naive Para-Virtualization

Shadowing the GPU page tables is a major source of overhead in the full-virtualization, because the entire page table needs to be scanned to detect any changes to the guest GPU page tables. We take the naive para-virtualization approach to reduce the cost of detecting the updates. In this approach, we introduce a new hypercall interface for controlling the GPU page tables and integrate it into the full-virtualization mechanisms. The guest GPU page tables are placed within the memory areas under the control of GPUvm and cannot be directly updated by the guest GPU drivers. The guest GPU driver issues a hypercall to the hypervisor to update the guest GPU page tables. The hypervisor validates the correctness of the given page table updates. This approach is inspired by the direct paging in the Xen para-virtualization [29].

We take into account the hypercall invocation cost, which is expensive since the context is switched from the VM to the hypervisor. GPUvm uses the multicall interface that

TABLE 1
No. of hypercall issues.

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	hs	lud	nn	srad	srad2
without multicall	11383	11383	11573	11573	11573	11573	12163	44937	11777	11429	11783	11469	12379	13031
with multicall	118	118	122	122	122	122	105	122	137	122	114	118	185	133

batches multiple hypercalls to reduce any hypercall issues. For example, instead of calling a hypercall to update one page table entry, GPUvm calls one multicall to update multiple page table entries to be updated. Table 1 lists the number of hypercalls for each benchmark in the naive para-virtualization with and without the multicall optimization. In all the benchmarks, the multicall dramatically reduces the hypercall issues. Compared to the other benchmarks, pincpy issues many more hypercalls in the naive para-virtualization without the multicall. Pincpy suffers from a larger overhead but the multicall improves its performance, which is described in Section 5.1.2.

3.4 PVDRM

While the naive para-virtualization avoids scanning the GPU page tables using a hypercall, it still incurs overhead caused by the low-level interceptions through MMIO and frequent hypercall issues. We also developed PVDRM, the high-performance para-virtualization approach that uses a set of high-level interfaces to address this issue.

PVDRM uses the Direct Rendering Manager (`DRM`) as a boundary of the para-virtualization instead of MMIO. The DRM is a widely used GPU abstraction layer in Linux, and it is used in multiple existing open-source GPU drivers such as in the i915 for the Intel Integrated GPUs, Radeon for the AMD GPUs, and Nouveau for the NVIDIA GPUs. AMDGPU, which is an official driver for the AMD GPUs under development, uses DRM [37]. The use of the DRM provides high-level para-virtualization interfaces and enables for existing software stacks depending on the DRM to work on PVDRM without needing any modification. In addition, since the DRM is used through `ioctls` and each ioctl command semantic rarely changes [38], PVDRM easily maintains the compatibility over driver version changes. In fact, we can interchangeably use Linux kernel v3.6.5 and v3.17.2 on our PVDRM prototype.

PVDRM uses the split driver model [39]. The front-end driver resides in the guest and provides the DRM interfaces to the guest software stack. The DRM operations on the front-end driver are routed to the back-end driver that conceptually runs in the hypervisor. The back-end driver adjusts the routed operations and performs them in the DRM stack.

PVDRM inherits the existing isolation mechanism of DRM to achieve the isolation among the VMs. The DRM has already been integrated with a mechanism that uses the GPU page table to isolate multiple GPU contexts. PVDRM simply uses this isolation mechanism to protect the GPU contexts of different VMs. Shadowing the GPU page tables is unnecessary because the front- and back-end drivers are aware of the GPU virtualization and depend on the DRM isolation mechanism.

3.5 GPU Fair-Share Scheduler

So far we have discussed the virtualization of the memory resources and GPU channels for multiple VMs. We herein provide information on the virtualization of the GPU time. This is indeed a scheduling problem. The GPU scheduler of GPUvm is based on the bandwidth-aware non-preemptive device (BAND) scheduling algorithm [40], which was developed for virtual GPU scheduling. The BAND scheduling algorithm is an extension of the CREDIT scheduling algorithm [29] in that (i) the prioritization policy uses a reserved bandwidth and (ii) the scheduler intentionally inserts a certain amount of delay after completion of the GPU kernels, which leads to a fairer utilization of the GPU time among the VMs. Since the current GPUs are not preemptive, GPUvm waits for GPU kernel completion and assigns credits based on the GPU usage. More details about this can be found in [40].

The BAND scheduling algorithm assumes that the total utilization of the virtual GPUs could reach 100%. This is a flaw because there must be some interval in which the CPU executes the GPU scheduler during which the GPU remains idle, causing the utilization of the GPU to be less than 100%. This means that even though the total bandwidth is set to 100%, the credit for the VMs would remain unused, if the GPU scheduler consumes a given amount of 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 given period of time all the VMs could have a lot of credit remaining. As a result, the credit may not influence the scheduling decision at all. GPUvm accounts for the CPU time consumed by the GPU scheduler and considers it as the GPU time to overcome this problem. Specifically, GPUvm charges CPU time equally to each VM to avoid an unbalanced charge to a VM that issues short requests frequently.

Note that there is a critical problem in guaranteeing the GPU time fairness. If a malicious or buggy VM starts an infinite computation on the GPU, it can monopolize the GPU time. One possible solution to this problem is to abort the GPU computation if the GPU time exceeds the pre-defined limit of computation time. Another approach is to cut longer requests into smaller pieces, as shown in [41].

For future directions, we are planning to incorporate disengaged scheduling [42] on the hypervisor level. The disengaged scheduling provides a fair, safe, and efficient OS-level management of the GPU resources. We believe that GPUvm can incorporate this disengaged scheduling without incurring any technical issues except for the engineering efforts.

4 Implementation

Our GPUvm prototype uses Xen 4.2.0, where both domain 0 and domain U adopt the Linux kernel v3.6.5. We target

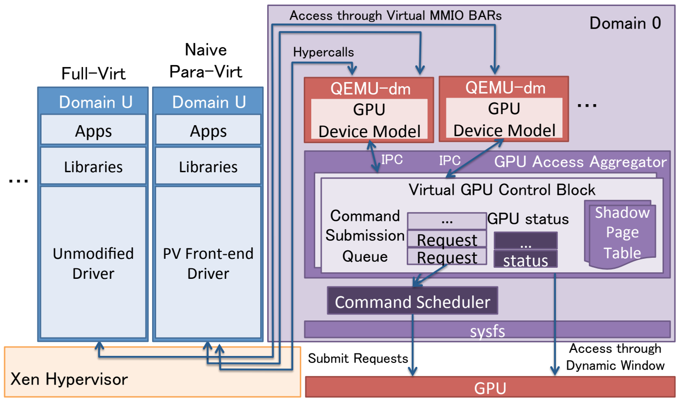

Fig. 8. Prototype overview of GPUvm full- and naive para-virtualization.

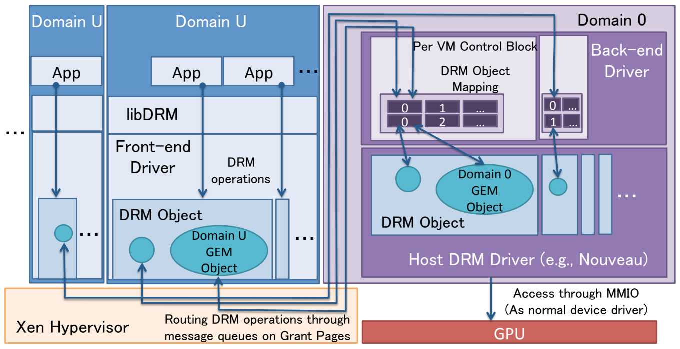

Fig. 9. Overview of GPUvm PVDRM prototype.

the device model for the NVIDIA GPUs that is based on the Fermi architectures [30]. While the full-virtualization does not require any modification to the 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.

4.1 Full- and Naive Para-Virtualizations

Fig. 8 shows an overview of our implementation of the GPU device model, the GPUvm back-end, and their interactions with the other components. QEMU-dm is used to create GPU device models and behave as virtual GPUs. The guest device drivers in domain U consider it a normal GPU. It exposes virtual MMIO PCIe BARs to domain Us, trapping the accesses to them. All the accesses to the GPU aggregated by the GPU Access Aggregator are committed to the physical GPU through the `sysfs` interface.

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

The GPU Access Aggregator has virtual GPU control blocks that represent the states of the virtual GPUs. The GPU device models update their own virtual GPU control blocks using the IPC to manage the states of the 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 the command submission requests issued from the GPU device model. These command submission requests are scheduled to control the GPU executions in the GPU command scheduler. This command scheduling mechanism is similar to TimeGraph [43]. However, the GPU command scheduler of GPUvm differs from that of 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 into the original sequence of commands, because the 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. Whenever command submission requests are stored in the queue, the scheduler dispatches them to the GPU. Our prototype polls a GPU control register that is modified by the hardware just after the GPU channels become active/inactive to calculate the GPU time.

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

4.2 PVDRM

Fig. 9 overviews our PVDRM prototype. PVDRM front- and back-end drivers are running in the domain U and domain 0, respectively. To access the GPU in the DRM environment, an application creates DRM objects, each of which has its own GPU address space. The DRM object has a memory instance named Graphics Execution Manager (GEM). The application runs its GPU code by sending a request to the GEM object through a special device file. When an application requests to create a DRM object in domain U, the front-end creates a stub DRM object and the back-end prepares the corresponding DRM object. The back-end driver manages the mapping between the stub and corresponding objects. The front-end driver forwards received operations to the back-end, and the default DRM manager running in domain 0 handles the operations. Since each DRM object is isolated, PVDRM guarantees the isolation of each VM DRM object.

PVDRM creates a message queue between the front- and back-end drivers. The queue is used for the front- and back-end drivers to forward the requests and return the operation results, respectively. Both drivers poll a message to efficiently handle the requests and responses.

One technical challenge is to handle a `mmap` operation, which maps a GEM object to the address space of an application. PVDRM needs to map the GEM objects in domain 0 to the address space of the application running in domain U to successfully handle the mmap operation. This means that page sharing across domains is needed. PVDRM shares the pages of the target back-end GEM object with domain U and maps the shared pages to the address space of the application to accomplish this. However, sharing the pages involves several hypercalls, which causes non-trivial overhead. PVDRM pools the allocated shared GEM objects in domain U and domain 0 to mitigate the performance penalty. When the mmap is requested to a cached GEM object, the front-end driver can directly map the object pages to the address space of the application since the cached object pages are already backed to those of the back-end object.

4.3 Discussion

GPUvm focuses on the hypervisor-level GPU virtualization for GPGPU, and thus, virtualizes only the GPU resources required for it. This means that GPUvm does not virtualize all of the GPU resources. For example, our prototype does not virtualize a frame buffer used for graphics that is typically used by the X server. Such a resource virtualization is out of the scope of this paper.

In shadowing channel descriptors, GPUvm traps the MMIO accesses to the guest channel descriptors to propagate their updates to the shadow ones. The current design of GPUvm implicitly assumes that the updates of the channel descriptors are only done by the CPUs. In other words, GPUvm cannot handle the channel descriptor updates from the GPUs. However, we believe that this assumption is reasonable since the channel descriptors are typically updated by the device drivers. From our experience, we have never seen GPU applications that update the channel descriptors from the GPU contexts.

We must also note the current status of the prototype. Our prototype does not perfectly handle initialization operations. For the ease in implementation, the prototype initializes the physical GPUs by using the domain 0 GPU device driver, and ignores the initialization operations from the guest GPU device drivers just enough to successfully execute the GPU applications in domain U. We need to pay the engineering cost to carefully analyze the initialization operations and sophisticate the corresponding part of the prototype to improve the stability of GPUvm.

We apply the BAR remap optimization to only BAR3 in the prototype, which is dominantly accessed in several benchmarks. In principle, the same optimization can be applied to BAR1 that is used as a memory aperture. This optimization is expected to slightly reduce the overhead in the full- and naive para-virtualizations.

5 Experiments

We conduct detailed experiments using a relevant commodity GPU to show the effectiveness of GPUvm. The objective of this section is to answer the following fundamental questions: 1) How much is the overhead of the GPU virtualization incurred by GPUvm?, 2) How does the number of GPU contexts affect the performance?, 3) Can multiple VMs meet the necessary fairness for the GPU resources?, and 4) How much is the overhead of the schedulers we used?. The experiments are conducted on a DELL PowerEdge T320 machine with eight Xeon E5–24700 2.3-GHz processors, 16 GB of memory, and one 2-TB SATA hard disk. We use a NVIDIA Quadro 6000 as the target GPU, which is based on the NVIDIA Fermi architecture. We run our modified Xen 4.2.0, assigning 4 and 1 GB of memory to domain 0 and domain U. Nouveau is running as the GPU device driver and the latest user-mode Gdev [40] is running as the CUDA runtime in domain U. In our previous paper [44], the kernel-mode Gdev was used instead of the user-mode Gdev. The following ten configurations are evaluated: **Native** (non-virtualized Linux 3.6.5), **PT** (pass-through provided by pass-through feature of Xen), **FV Naive** (full-virtualization w/o any optimization techniques), **FV BAR-Remap** (full-virtualization w/ BAR Remap), **FV Lazy** (full-virtualization w/ Lazy Shadowing), **FV Optimized** (full-virtualization w/ BAR Remap and Lazy Shadowing), **PV Naive** (naive para-virtualization w/o multicall), **PV Multicall** (naive para-virtualization w/ multicall), **PVDRM** (GPUvm w/o pooling allocated GEMs), and **PVDRM Pooling** (GPUvm w/ pooling allocated GEMs).

TABLE 2
List of 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)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
NN	K-nearest Neighbors (data mining)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

5.1 Overhead

We pick several benchmarks from the well-known GPU benchmarks called Rodinia [45] as well as our microbenchmarks to identify the overhead of the GPU virtualization incurred by GPUvm, as listed in Table 2. We measure their execution time on the ten models of virtualization. For each benchmark, we run it eleven times, once for warming up and ten times for the results. We use the later ten iterations.

5.1.1 Results

Fig. 10 shows the average execution times of the benchmarks on each model. The x-axis is the benchmark names while the y-axis exhibits the execution time normalized by one of Native. It is clearly observed that the overhead of the GPU full-virtualization is mostly unacceptable, but our optimization techniques significantly contribute to the reduction of this overhead. The execution times obtained in FV Naive are more than 100 times slower in nine of the benchmarks (nop, loop, madd, fmadd, fmmul, bp, hs, lud, and nn) 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 improves the performance. The execution time is 8.2 times shorter in pincpy (best case) while being 3.4 times shorter in srad (worst case).

We also find from these experimental results that the naive para-virtualization is much faster than the full-virtualization. In most cases, the execution times obtained in PV Naive are 2–20 times slower than those obtained in Native. PV Naive exceeds FV Optimized overall in the execution times. However, in pincpy, it is defeated by FV Optimized. We discuss the reason for this in the next section. The overhead can also be reduced by using our multicall

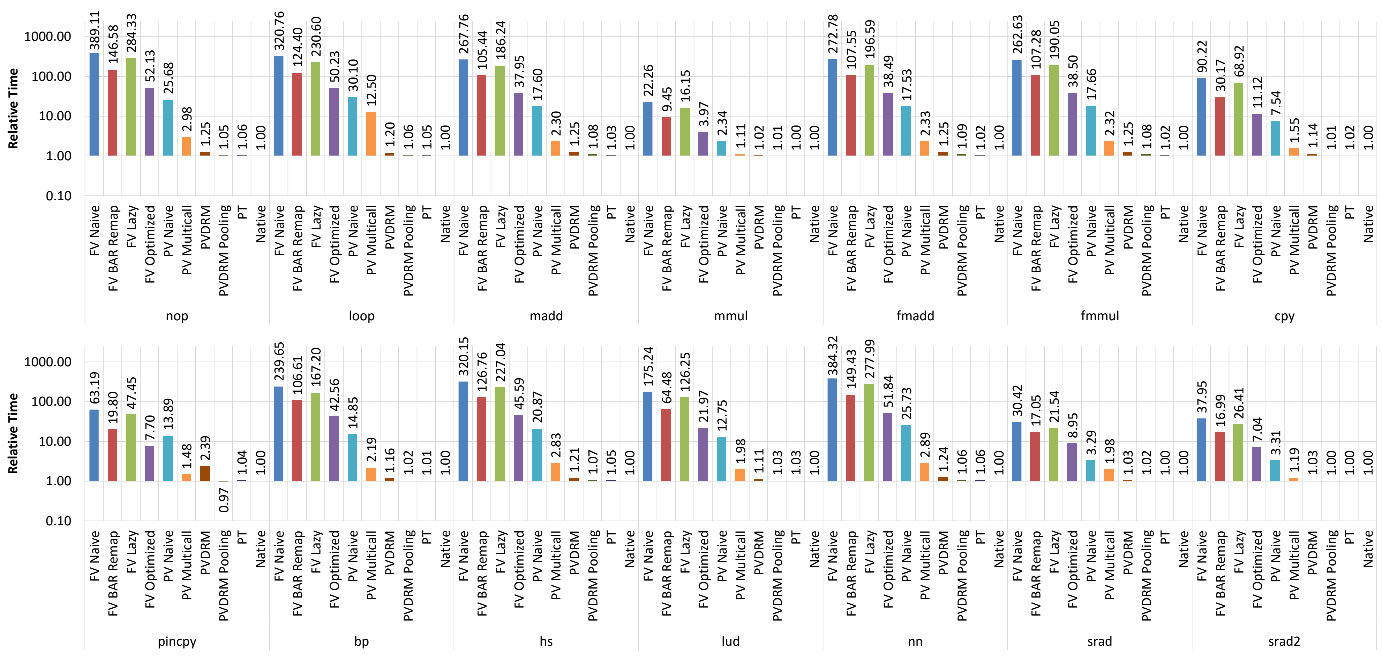

Fig. 10. Execution time of GPU benchmarks on ten configurations.

optimization. PV Multicall is at most 3 times slower than Native except in the case of loop.

PVDRM outperforms PV Naive and PV Multicall because the high-level interface reduces the frequency of the MMIO interceptions and hypercalls. In PVDRM, the overhead becomes 2–25% except for when using pincpy. PVDRM Pooling incurs only a 4% overhead on average. Interestingly, PVDRM Pooling exceeds Native in the performance of pincpy. We discuss this in detail in the next section.

5.1.2 Breakdown

A breakdown of the execution times of the GPU benchmarks is shown in Fig. 11. We divide the total execution time into five phases: *init*, *htod*, *launch*, *dtoh*, and *close*. Init is the time necessary for setting up the GPU to execute a GPU kernel. Htod is the time for the host-to-device data transfers, launch is the time for the calculation on the GPUs, dtoh is the device-to-host data transfer time, and close is time for destroying the GPU kernel and context. The figure indicates that the dominant factors in the execution time for GPUvm are the init and close phases. This tendency is significant for four of the GPUvm full-virtualization configurations. In FV Naive, the init and close phases comprise more than 90% of the execution time. The ratios of these phases can be lowered by using optimization techniques and naive para-virtualization. In particular, PV Multicall significantly lowers the ratios of the two phases during computation heavy workloads (mmul, lud, srad, and srad2).

In the case of loop, PV Naive and PV Multicall are more than 15 times slower than Native. Fig. 11 shows that loop for PV Naive and PV Multicall spends a large amount of time in the htod and dtoh phases. This is because the current GPUvm prototype is not integrated with the BAR Remap for BAR1. Loop transfers a small amount of memory (4KB) back and forth between the host and device. In such cases, the Gdev CUDA runtime uses BAR1 for transferring memory instead of the DMA, and thus, this causes overhead for intercepting the BAR1 accesses.

As explained in Section 3.2.4, BAR Remap and Lazy Shadowing effectively work to reduce the overhead. In all the benchmarks, FV BAR-Remap and FV Lazy incur less overhead than FV Naive. Since both techniques are orthogonal, the use of both optimizations (FV Optimized) results in a bigger gain in performance. On average, the execution times are 2.5 times in FV BAR-Remap, 1.4 times in FV Lazy, and 4.9 times in FV Optimized shorter than in FV Naive.

On the other hand, the init and close phases in two of the GPUvm naive para-virtualized configurations are much shorter than those of the full-virtualization in almost all cases. Full-virtualization performs many operations related to the shadow page tables since memory allocations and deallocations that touch the GPU page tables frequently occur in the two phases. This cost can be significantly reduced in the naive para-virtualizations (PV Naive and PV Multicall) in which those operations are requested by hypercalls. The only exception is pincpy. Pincpy reports larger overhead in PV Naive than one in FV Optimized. This is because pincpy issues many more hypercalls to map a large amount of the host memory in a GPU page table. As shown in Table 1, pincpy issues about 4 times more hypercalls than the other benchmarks. Using the multicall effectively reduces this overhead, and pincpy in PV Multicall is 9 times faster than in PV Naive. The multicall optimization reduces the hypercall issues, and as a result, the execution times in PV Multicall are closer to those of PT and Native than the ones of PV Naive. The relative times in six of the benchmarks (mmul, cpy, pincpy, lud, srad, and srad2) in PV Multicall are within twice of ones in Native. In the case of mmul, PV Multicall incurs only 11% overhead.

PVDRM performs comparably to the Native except for pincpy. Fig. 11 depicts that the init and close times of PVDRM take on a larger ratio than in PV Multicall for pincpy. This is because in the case of pincpy PVDRM issues many hypercalls for granting access to a large amount of the domain 0 memory to the domain U. Pincpy requires

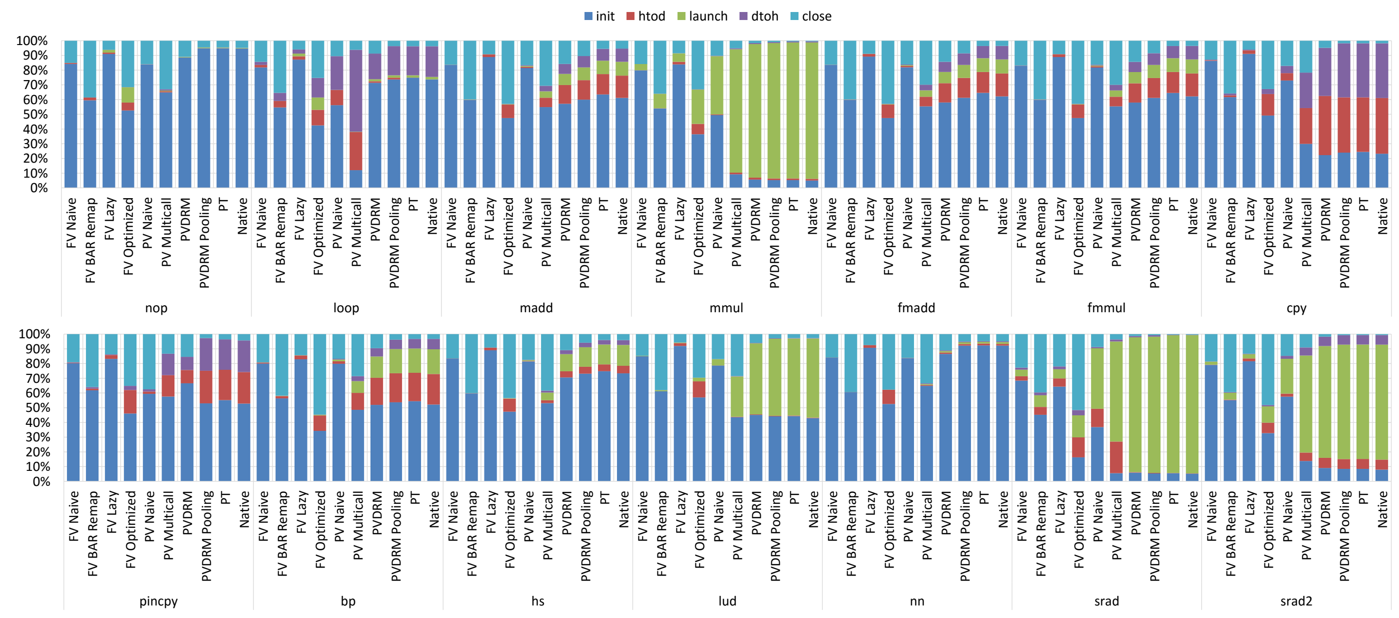

Fig. 11. Breakdown on execution time of GPU benchmarks.

that a wide region of the host memory is mmap-ped and it is accessed. Since pincpy involves a mmap operation for a large GEM object, the front- and back-end drivers interact with the hypervisor to map the pages of the back-end GEM object to the guest. PVDRM Pooling uses already-mapped GEM objects, and thus, improves the performance due to less interaction with the hypervisor.

5.2 Performance at Scale

We generate GPU workloads and measure their execution times using two scenarios to discern the overhead GPUvm incurs in multiple GPU contexts. In the first scenario, one VM executes multiple GPU tasks, and multiple VMs execute GPU tasks in the other scenario. We first launch 1, 2, 4, and 8 GPU tasks in one VM with the full-virtualized, naive para-virtualized, PVDRM, domain 0, and pass-throughed GPU (*FV(1VM), PV(1VM), PVDRM(1VM), Dom0, and PT*). These tasks are also run on the native Linux (*Native*). Next, we prepare 1, 2, 4, and 8 VMs and execute one GPU task on each VM with the full-, naive para-virtualized, and PVDRM(*FV, PV, and PVDRM*), where all our optimizations are turned on. In each scenario, we run the *madd* listed in Table 2. Specifically, we repeat the GPU kernel execution of madd 10000 times, and measure its execution time.

The results are shown in Fig. 12. The x-axis is the number of launched GPU contexts and the y-axis represents the execution time. This figure shows that the full-virtualization takes longer in the init and close phases as the number of GPU contexts increased. FV also takes longer during these phases as more VMs are running. This is because GPUvm forces the VMs to exclusively access unique GPU resources including the dynamic window.

The graphs also show that the naive para-virtualization and PVDRM of GPUvm have a similar performance to pass-through GPU. The total execution times in PV, PV (1VM), PVDRM, and PVDRM (1VM) are quite similar to those in PT even if there are more GPU contexts.

We can see from this figure that PV (multiple VMs) has a shorter kernel execution time than PV (1VM). This overhead

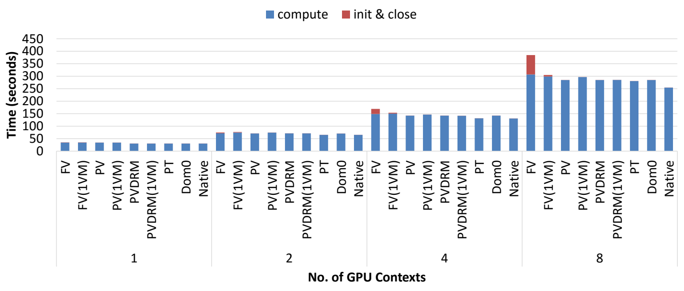

Fig. 12. Performance across multiple VMs.

seems to occur for the following two reasons. One is that the device driver uses coarser-grained locks for the GPU resource accesses, and thus, the GPU contexts are executed more sequentially than for one GPU context over multiple VMs; our prototype locks GPU resources in a finer-grained manner. The other is that the MMIO operations of multiple GPU contexts are serialized in the 1VM case since our prototype generates one thread for each VM to emulate the MMIO operations. On the other hand, the ones for a GPU context over several VMs are handled by several threads and are emulated in parallel on physical cores.

The kernel execution time in Native is shorter than Dom0 since Native does not suffer from virtualization overhead caused by Xen.

5.3 Performance Isolation

We launch a GPU workload on 2, 4, and 8 VMs and measure the GPU usage on each VM to show how GPUvm achieves the performance isolation among the VMs. For comparison, we use three schedulers: FIFO, CREDIT, and BAND. FIFO issues GPU requests in a first-in/first-out manner. CREDIT schedules GPU requests in a proportional fair share manner. Specifically, CREDIT reduces the credits assigned to a VM after its GPU requests are executed, chooses a VM whose credit number is positive, and issues its GPU requests. CREDIT reassigns the credits to the VMs for a given interval. BAND is our scheduler that was described in Section 3.5. We prepare two GPU tasks, madd, which is used in the

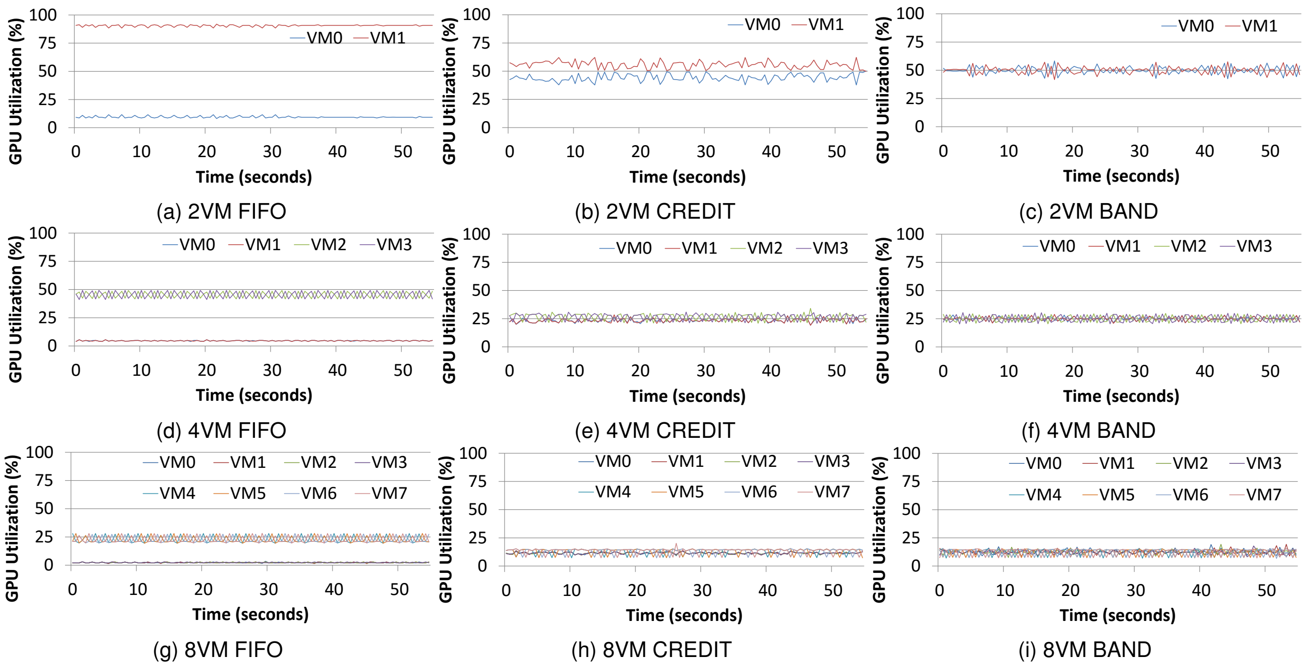

Fig. 13. GPU usage of VMs (over 500 ms) for three GPU schedulers.

previous experiment and an extended madd (*long-madd*), which performs 15 times more calculations than the regular one. Each VM loops one of them. We run each task on half of the VMs. For example, madd runs on 2 VMs while long-madd runs on 2 VMs in the 4VM case.

Fig. 13 shows the results. The x-axis represents the elapsed time and the y-axis is the GPU usage of the VMs over 500 msec. The figure reveals that BAND is the only scheduler that achieves performance isolation in all cases. In FIFO, the GPU usages of the VMs running long-madd are higher in all cases since it dispatches almost the same number of GPU commands from each VM at a given time.

CREDIT fails to achieve the fairness among the VMs in the 2VM case. When the command submission request queue contains requests from only long-madd just after the madd commands have completed, CREDIT dispatches long-madd requests even if it does not have a credit. On the other hand, BAND achieves the fairness because it waits for the request arrivals from madd for a short time period, and thus, handles the requests issued from the VMs whose GPU usage is less.

CREDIT achieves fair-share GPU scheduling in the 4VM and 8VM cases. In these cases, CREDIT has more opportunities to dispatch less-executed VM commands for the following two reasons. First, the GPU operations whose execution times are short are issued more frequently in the 4 and 8VM cases so that there are more points for scheduling VMs in an interval. Last, the request submission queue has requests from two or more VMs just after a GPU kernel completes, which differs from that in the 2VM case.

Note that BAND cannot achieve fairness among the VMs in a fine-grained manner on the current GPUs. Fig. 14 shows the GPU usages of the VMs over 100 msec. Even with BAND, the GPU usages fluctuated over time, because the GPU is a non-preemptive device. We need a novel mechanism inside the GPU that effectively switches the GPU kernels to achieve a finer-grained GPU fair-share scheduling.

5.4 Scheduling Overhead

We compare the overheads of FIFO, CREDIT, and BAND schedulers, using madd and long-madd benchmarks in the previous section. Madd and long-madd are executed on 1, 2, 4, and 8 VMs, respectively.

Fig. 15 exhibits the results. The x-axis is the combination of the selected task, the number of VMs and the schedulers. The y-axis is the average of the execution time. The overhead of FIFO is the smallest in our schedulers since FIFO simply issues commands to GPUs just after the commands from madd or long-madd arrived. The execution times of CREDIT and BAND are at most 39% and 30% longer than FIFO in the madd case, while their execution times in the long-madd cases are at most 5% and 3% longer.

In most of the cases, execution time on CREDIT is longer than that on BAND in 2, 4, and 8 VMs. This is because the context switches occur more frequently in CREDIT. Since BAND waits for a small amount of time to receive requests from the previously executed GPU context, BAND tends to execute one GPU context in a batch manner.

Fig. 15 also indicates that the scheduling overheads are larger in madd than long-madd under the same configuration (the same number of VMs and the same scheduler). Since the madd workload consists of multiple shorter command streams than long-madd, the schedulers run more frequently in the madd cases. In addition to the frequently invoked schedulers, the ratio of the context switches to the whole execution in the madd cases is larger than that in the long-madd cases. Since time for one context switch is constant [46], [47] and madd's execution is much shorter than long-madd, the performance penalty in the madd cases becomes larger.

6 Related Work

Some vendors have invented different GPU virtualization techniques. NVIDIA has announced NVIDIA VGX [48],

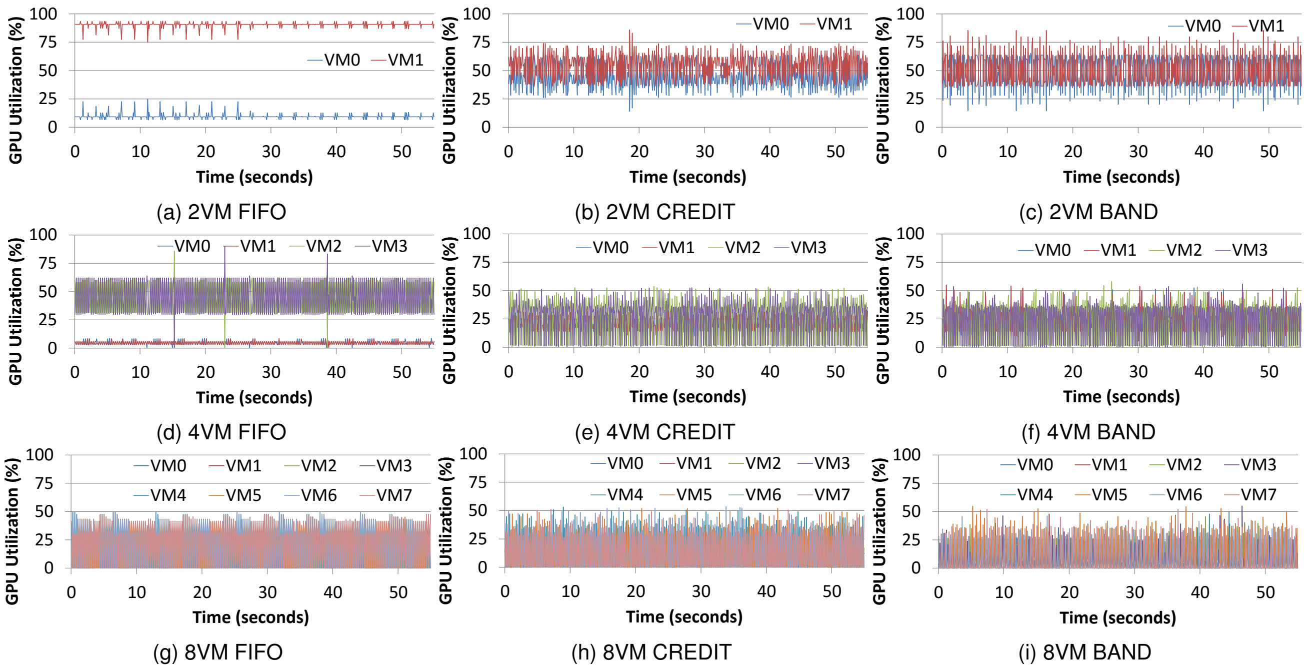

Fig. 14. GPU usage of VMs (over 100 ms) for three GPU schedulers.

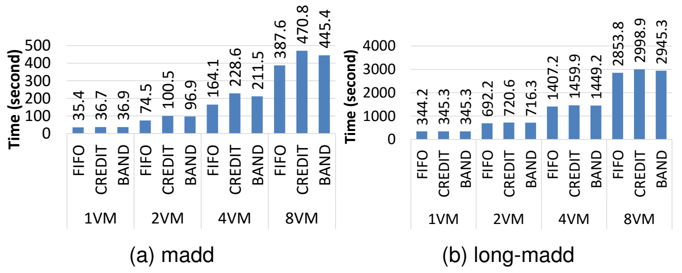

Fig. 15. Execution time of madd and long-madd with different VMs and scheduling policies.

which exploits the virtualization supports of Kepler generation GPUs. They are proprietary, so their details are closed. To the best of our knowledge, GPUvm is the first open architecture for the GPU virtualization targeting discrete GPUs offered by the hypervisor. GPUvm carefully selects the resources to virtualize and the GPU page tables and channels to maintain its applicability to various GPU architectures.

VMware SVGA2 [27] para-virtualizes GPUs to mitigate the overhead of virtualizing the GPU graphics features. The SVGA2 handles graphics-related requests using an architecture-independent communication to efficiently perform 3D rendering and to hide the GPU hardware. While this approach is specific to graphics acceleration, GPUvm coordinates the interactions between the GPUs and guest device drivers.

Gottschalk et al. proposed a low-overhead GPU virtualization, named LoGV, for GPGPU applications [35]. Their approach is categorized into a para-virtualization where the device drivers in the VMs send requests for resource allocation and mapping memory into the system RAM to the hypervisor. Similar to that in our work, this work exhibits para-virtualization mechanisms to minimize the GPGPU virtualization overhead. Our work reveals which virtualization technique for the GPUs is efficient in a quantitative way.

API remoting, in which the API calls are forwarded from the client to the server that has the GPU, have been widely studied. GViM [22], vCUDA [25], and rCUDA [20] forward CUDA APIs. VMGL [24] achieves the API remoting of OpenGL. gVirtuS [21] supports the API remoting of CUDA, OpenCL, and part of OpenGL. In these approaches, the applications are inherently limited to the APIs the wrapper-libraries offer. Keeping the wrapper-libraries compatible to the original ones is not a trivial task because new functionalities are frequently integrated into the GPU libraries, including CUDA and OpenCL. Moreover, API remoting requires that the all the GPU software stacks, including the device drivers and runtimes, become part of the TCB.

Amazon EC2 G1 Instance [19] provides GPU instances. It makes use of the 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 machine.

gVirt [28] fully virtualizes the Intel integrated GPUs at the hypervisor. Although gVirt uses similar techniques to GPUvm such as the shadow page tables, gVirt is not designed for the architecture on which multiple active channels can coexist; it is required to switch the render contexts on the driver side. Thus, this paper describes original techniques such as shadow channels, and quantitatively evaluates our GPU virtualization solution. In addition, GPUvm supports vanilla device drivers for the current NVIDIA GPUs while gVirt has to integrate an extension of the specifications for the Intel GPUs into the device driver, and thus, requires driver modification.

GPUvm is complementary to the GPU command scheduling methods. VGRIS [26] enables us to schedule GPU commands in SLA-aware, proportional-share, or the hybrid scheduling. Pegasus [23] coordinates GPU command queuing and CPU dispatching so that multi-VMs can effectively share the CPU and GPU resources. Disengaged Scheduling [42] uses a fair queuing scheduling with a probabilistic extension to the GPU, and provides protection and fairness without compromising efficiency.

Some work aims at the efficient management of GPUs on the operating system layer such as the GPU command scheduler [43], kernel-level GPU runtime [40], oversubscription mechanism for the GPU memory [36], OS abstraction of GPUs [17], [49] and file system and networking for the GPUs [5], [16]. These mechanisms can be incorporated into GPUvm.

7 Conclusion

This paper presented GPUvm, an open architecture for GPU virtualization. GPUvm supports the full- and naive para- and high-performance para-virtualization using optimization techniques. The experimental results using our prototype showed that the full-virtualization incurs a non-trivial overhead largely due to the MMIO handling, and naive para-virtualization provides a two or three times slower performance than the pass-through and native approaches. The high-performance para-virtualization with the high-level interface significantly reduces the overhead.

Table 3 summarizes the tradeoffs between the GPUvm hypervisor-level GPU virtualization modes. Although the full-virtualization mode does not require any modification of the software stack on the VMs, its performance penalty is the highest (297–5113%). The overhead of the naive para-virtualization is lower than that of the full-virtualization at the expense of some modification to the code of the device driver in order to use hypercalls (540 LOC). PVDRM offers the DRM interfaces to the VMs and incurs at most a 9% overhead. The drawbacks of PVDRM are supporting only applications using the DRM and requiring device driver modification (3474 LOC).

Our suggestion to the GPU hardware design is that a nested page table support in GPUs, similar to Intel EPT, is effective to reduce overhead of GPU virtualization. Since the hardware extension translates guest virtual addresses to machine addresses by setting a physical-to-machine mapping table to a special register in advance, we do not have to scan all page table entries in building the shadow page table. We can offer a virtual GPU by grouping several GPU channels and assigning one nested page table to the group.

For our future directions, the optimization techniques proposed in vIOMMU [50] that can be applied to GPUvm should be investigated. We hope our experience with GPUvm gives insight into designing the support for device virtualization such as SR-IOV [51].

Acknowledgments

Partially supported by JSPS KAKENHI 25280043, 15J09761.

References

[1] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka, "Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer," in *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, pp. 3:1–3:11.

[2] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers," in *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, pp. 11:1 –11:12.

[3] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-Accelerated Software Router," in *Proc. of the ACM SIGCOMM 2010 Conf.* ACM, 2010, pp. 195–206.

[4] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors," in *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*. USENIX, 2011, pp. 1–14.

[5] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," in *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013, pp. 485–498.

[6] W. Sun, R. Ricci, and M. L. Curry, "GPUstore," in *Proc. of the 5th Annual Int'l Systems and Storage Conf.* ACM, 2012, pp. 1–12.

[7] S. R. Agrawal and A. R. Lebeck, "Rhythm : Harnessing Data Parallel Hardware for Server Workloads," in *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014, pp. 19–34.

[8] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "GPU Join Processing Revisited," in *Proc. of the 8th Int'l Workshop on Data Management on New Hardware*. ACM, 2012, pp. 55–62.

[9] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs," in *Proc. of the 2010 Int'l Conf. on Management of Data*. ACM, 2010, pp. 339–350.

[10] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *Proc. of the 2010 Int'l Conf. on Management of Data*. ACM, 2010, pp. 351–362.

[11] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational Joins on Graphics Processors," in *Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data*. ACM, 2008, pp. 511–524.

[12] N. Rath, J. Bialek, P. J. Byrne, B. DeBono, J. P. Levesque, B. Li, M. E. Mauel, D. A. Maurer, G. A. Navratil, and D. Shiraki, "High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak," *Fusion Engineering and Design*, pp. 1895–1899, 2012.

[13] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," in *Proc. of the 4th Int'l Conf. on Cyber-Physical Systems*. ACM/IEEE, 2013, pp. 170–178.

[14] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice," in *Proc. of the 2011 Int'l Conf. on Robotics and Automation*. IEEE, 2011, pp. 4889–4895.

[15] M. Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita, "GPU Implementations of Object Detection using HOG Features and Deformable Models," in *Proc. of the 1st Int'l Conf. on Cyber-Physical Systems, Networks, and Applications*. IEEE, 2013, pp. 106–111.

[16] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*. USENIX, 2014, pp. 201–216.

[17] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a Compiler and Runtime for Heterogeneous Systems," in *Proc. of the 24th Symp. on Operating Systems Principles*. ACM, 2013, pp. 49–68.

[18] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Proc. of the 2011 Int'l Parallel & Distributed Processing Symp.* IEEE, 2011, pp. 1068–1079.

[19] Amazon.com, "EC2: Amazon Elastic Compute Cloud."

[20] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proc. of the 2010 Int'l Conf. on High Performance Computing & Simulation*. IEEE, 2010, pp. 224–231.

[21] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Proc. of the 16th Int'l Euro-Par Conf. on Parallel Processing*. Springer-Verlag, 2010, pp. 379–391.

[22] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-Accelerated Virtual Machines," in *Proc. of the 3rd Workshop on System-level Virtualization for High Performance Computing*. ACM, 2009, pp. 17–24.

[23] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated Scheduling for Virtualized Accelerator-

TABLE 3
Comparison of GPUvm virtualization modes.

	Level of Interface	Overhead	Guest Driver Modification	Software Stack Limitation
Full-virtualization	MMIO	297 – 5113%	Nothing	No Limitation
Naive Para-virtualization	MMIO & Hypercalls	11 – 1150%	Extended for Hypercalls, 540 LOC[1, 2]	No Limitation
PVDRM	Hypercalls	-3 – 9%	Full Scratch, 3474 LOC[2]	Limited for Linux DRM APIs

[1] Counted additions and modifications related to naive para-virtualization.
[2] Counted by Al Danial's `CLOC`.

based Systems," in *Proc. of the 2011 USENIX Annual Technical Conf.* USENIX, 2011, pp. 31–44.

[24] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "VMM-Independent Graphics Acceleration," in *Proc. of the 3rd ACM Int'l Conf. on Virtual Execution Environments*. ACM, 2007, pp. 33–43.

[25] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2012.

[26] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan, "VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming," in *Proc. of the 22nd Int'l Symp. on High-performance Parallel and Distributed Computing*. ACM, 2013, pp. 203–214.

[27] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 73–82, 2009.

[28] K. Tian, Y. Dong, and D. Cowperthwaite, "A Full GPU Virtualization Solution with Mediated Pass-Through," in *Proc. of the 2014 USENIX Annual Technical Conf.* USENIX, 2014, pp. 121–132.

[29] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and Art of Virtualization," in *Proc. of the 19th Symp. on Operating Systems Principles*. ACM, 2003, pp. 164–177.

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

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

[32] M. Koscielnicki, "Envytools," https://0x04.net/envytools.git.

[33] K. Menychtas, K. Shen, and M. L. Scott, "Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack," in *Proc. of the 2013 USENIX Annual Technical Conf.* USENIX, 2013, pp. 291–296.

[34] L. O.-S. Community, "Nouveau Open-Source GPU Device Driver," http://nouveau.freedesktop.org/.

[35] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa, "LoGV: Low-overhead GPGPU Virtualization," in *Proc. of the 4th Int'l Workshop on Frontiers of Heterogeneous Computing*. IEEE, 2013, pp. 1721–1726.

[36] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping," in *Proc. of the 11th ACM Int'l Conf. on Virtual Execution Environments*. ACM, 2015, pp. 65–77.

[37] AMD, "AMDGPU," http://cgit.freedesktop.org/~agd5f/linux/log/?h=amdgpu.

[38] A. Amiri Sani, K. Boos, S. Qin, and L. Zhong, "I/O Paravirtualization at the Device File Boundary," in *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014, pp. 319–332.

[39] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe Hardware Access with the Xen Virtual Machine Monitor," in *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*. ACM, 2004, pp. 1–10.

[40] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-Class GPU Resource Management in the Operating System," in *Proc. of the 2012 USENIX Annual Technical Conf.* USENIX, 2012, pp. 401–412.

[41] C. Basaran and K.-D. Kang, "Supporting Preemptive Task Executions and Memory Copies in GPGPUs," in *Proc. of the 24th Euromicro Conf. on Real-Time Systems*. IEEE, 2012, pp. 287–296.

[42] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014, pp. 301–316.

[43] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments," in *Proc. of the 2011 USENIX Annual Technical Conf.* USENIX, 2011, pp. 17–30.

[44] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why Not Virtualizing GPUs at the Hypervisor?" in *Proc. of the 2014 USENIX Annual Technical Conf.* USENIX, 2014, pp. 109–120.

[45] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. of the 2009 Int'l Symp. on Workload Characterization*. IEEE, 2009, pp. 44–54.

[46] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proc. of the 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 593–606.

[47] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proc. of the 41st Int'l Symp. on Computer Architecture*. ACM/IEEE, Jun. 2014, pp. 193–204.

[48] NVIDIA, "NVIDIA GRID VGX SOFTWARE." [Online]. Available: http://www.nvidia.com/object/grid-vgx-software.html

[49] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions To Manage GPUs as Compute Devices," in *Proc. of the 23rd Symp. on Operating Systems Principles*. ACM, 2011, pp. 233–248.

[50] N. Amit and M. Ben-Yehuda, "vIOMMU: efficient IOMMU emulation," in *Proc. of the 2011 USENIX Annual Technical Conf.* USENIX, 2011, pp. 73–86.

[51] Y. Dong, Z. Yu, and G. Rose, "SR-IOV Networking in Xen: Architecture, Design and Implementation," in *Proc. of the 1st Workshop on I/O Virtualization*. USENIX, 2008, pp. 10–16.

Yusuke Suzuki received his B.E. and M.E. degrees from Keio University in 2013 and 2015. He is currently a Ph.D. student in the Department of Information and Computer Science at Keio University. His research interests include operating systems, virtualization and programming languages on heterogeneous systems.

Shinpei Kato is an Associate Professor in the School of Information Science at Nagoya University. He received his B.S., M.S., and Ph.D. degrees from Keio University in 2004, 2006, and 2008. He has also worked at The University of Tokyo, Carnegie Mellon University, and the University of California, Santa Cruz from 2009 to 2012. His research interests include operating systems, real-time systems, and parallel and distributed systems.

Hiroshi Yamada received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006. He received his Ph.D. degree from Keio University in 2009. He is currently an associate professor at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, and cloud computing. He is a member of IEEE/CS, ACM, and USENIX.

Kenji Kono received the BSc degree in 1993, MSc degree in 1995, and PhD degree in 2000, all in computer science from the University of Tokyo. He is a professor in the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE, ACM, and USENIX.