FairGV: Fair and Fast GPU Virtualization

FairGV: Fair and Fast GPU Virtualization

Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos, *Senior Member, IEEE*

Abstract—Increasingly high performance computing (HPC) application developers are opting to use cloud resources due to higher availability. Virtualized GPUs would be an obvious and attractive option for HPC application developers using cloud hosting services. Unfortunately, existing GPU virtualization software is not ready to address fairness, utilization, and performance limitations associated with consolidating mixed HPC workloads. This paper presents FairGV, a radically redesigned GPU virtualization system that achieves system-wide weighted fair sharing and strong performance isolation in mixed workloads that use GPUs with variable degrees of intensity. To achieve its objectives, FairGV introduces a trap-less GPU processing architecture, a new fair queuing method integrated with work-conserving and GPU-centric coscheduling polices, and a collaborative scheduling method for non-preemptive GPUs. Our prototype implementation achieves near ideal fairness (≥ 0.97 Min-Max Ratio) with little performance degradation (≤ 1.02 aggregated overhead) in a range of mixed HPC workloads that leverage GPUs.

Index Terms—GPU virtualization; trap-less architecture; fair queuing, coscheduling and hybrid scheduling strategies.

1 Introduction

Recent advances in heterogeneous computing, best exemplified by the ubiquity of systems with graphics processing units (GPUs) and multi-core CPUs, have catalyzed high performance computing (HPC) including scientific, engineering, data-intensive, and financial applications. GPUs in particular have facilitated high performance and energy efficiency in HPC systems via making massive multithreading easily accessible to programmers. Heterogeneous computing with GPUs has also accelerated performance-sensitive components of system software dramatically, in areas such as network and cybersecurity, database management, and file systems [1], [2].

Increasingly HPC application developers are moving their applications to cloud hosting services such as Amazon EC2 and the Google Cloud platform due to higher availability [3]. Application developers can benefit from these cloud platforms without having to maintain large in-house HPC facilities or queuing for long times to access external facilities. In 2010, Amazon EC2 announced the Cluster GPU Instance. This virtual machine (VM) instance type provides access to NVIDIA GPUs with up to 1,536 cores, and supports OpenGL, DirectX, CUDA, and OpenCL libraries. Leveraging virtualized GPUs in cloud computing is obviously an attractive choice for HPC developers, while virtualizing and sharing GPUs for higher utilization and lower cost of ownership is an attractive choice for cloud hosting data centers.

Unfortunately, efficient virtualization and sharing of GPUs between HPC applications is challenging. As more HPC applications move to the Cloud, the diversity of HPC workloads running in cloud servers also increases. Current GPU virtualization software is not ready to support effective sharing of GPUs between HPC workloads, particularly workloads with varying intensity of GPU access requests. When such mixed workloads are consolidated, tenants may experience unfairness and unpredictable performance variation due to inefficient virtualization stacks, synchronization bottlenecks, and non-preemptive scheduling. Such limitations prevent cloud hosting service providers from giving access to GPUs on a pay-per-use basis.

Among the limitations that prevent effective sharing of GPUs, non-preemptive scheduling can be addressed by adopting most recent GPUs that support hardware-based preemption [4]. These GPUs can save and restore the context of GPUs, which include the contents of register files and on-chip memory, upon requests from the system software. However, as a GPU is composed of massive computation cores each of which has its own context, the amount of data to be saved and restored at a single time reaches to several hundreds of KB. Unfortunately, this causes significant throughput degradation up to 35% in GPU applications [5]. Therefore, we believe that non-preemptive GPUs are still relevant for performance-sensitive applications, and addressing non-preemptive scheduling remains an important issue.

Prior research in GPU virtualization falls short of addressing the limitations regarding fairness and performance. First, small (e.g., sub-millisecond level) but frequent GPU requests, which are common in a wide range of classical HPC applications and emerging applications in real-time analytics, can burden virtualization stacks by frequent context switching between user and hypervisor spaces. Previous research invokes system or hypervisor calls on every GPU request [6], [7], [8], [9], [10], [11], [12], [13], which causes significant per-request trapping costs for small GPU requests. Second, workloads with high CPU-GPU interactivity can cause synchronization bottlenecks between the CPU and GPU schedulers. Previous research suggests that coscheduling of a VM and its corresponding virtual GPU can improve performance [10]. However, co-scheduling in itself fails to achieve good fairness because of interference between the CPU and GPU schedulers. Finally, variable

• *Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos are with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast BT7 1NN, United Kingdom. E-mail: {c.hong, i.spence, d.nikolopoulos}@qub.ac.uk.*

Manuscript received November 23, 2016; revised May 04, 2017.

request sizes in mixed workloads, which are challenging to handle on non-preemptive GPUs, are either ignored [6], [10], or addressed with reverse engineering methods which are not supported by many GPUs [8], [9], [12], [14]. None of the existing GPU virtualization solutions takes all these factors into account, thus failing to attain acceptable fairness combined with strong performance isolation.

This paper presents new methods to achieve nearly *ideal fairness* and *high utilization* for workloads with mixed GPU access intensity running on virtualized GPUs. We achieve two objectives: Under ideal fairness, each tenant receives a weighted fair share of the GPU resource, according to the price paid for this resource. Under high utilization, each tenant experiences predictable performance with strong performance isolation. For realizing fair and efficient GPU virtualization, we introduce FairGV, a new system that combines a highly optimized GPU virtualization framework with a novel fair-share scheduler. FairGV implements a radical redesign of GVirtuS [7] to improve performance and scalability for HPC workloads with mixed intensity of GPU requests. FairGV also introduces a fine-grain fair queuing algorithm that considers the diverse traits of mixed workloads. FairGV can be used on both non-preemptive and preemptive GPUs and addresses the limitations that previous research in the area has encountered, regardless of support for preemption. We demonstrate that FairGV can achieve near ideal fairness (≥ 0.97 Min-Max Ratio) and high utilization (≤ 1.02 aggregated overhead) in a broad range of mixed HPC workloads.

The contributions of this paper are summarized as follows:

- We introduce a trap-less GPU processing architecture to significantly improve the performance and scalability of mixed workloads with small, short-running, and repetitive GPU requests. This new architecture enables FairGV to process GPU requests directly from user space without trapping to the OS kernel or the hypervisor.
- We propose a new fair queuing method to achieve near ideal fairness without performance degradation in GPU virtualization. This method uses GPU-centric coscheduling to effectively tackle the challenge of running workloads with high CPU-GPU interactivity, while maintaining strong work-conserving properties both in GPU and CPU schedulers. This policy can be applied to both non- and preemptive GPUs.
- We develop a collaborative scheduling method that is combined with a novel and accurate accounting mechanism for achieving fairness between short- and long-running GPU kernels on non-preemptive GPUs. The accounting mechanism is not dependent on reverse engineering and uses a simple interposition technique to measure the request size with an error of less than 3%.
- We implement existing GPU schedulers including Credit and Strict-co scheduling in the same framework, to thoroughly evaluate and analyze the fairness and performance impact on a wide range of mixed HPC workloads that use GPUs.

The remainder of this paper is structured as follows: Section 2 describes background and related work. Section 3 elaborates on the design and algorithms of FairGV. Section 4 provides the implementation details. Section 5 shows our experimental results. Section 6 presents a discussion point. Section 7 concludes the paper.

2 Background and Related Work

GPU virtualization techniques use three approaches: API remoting, para & full virtualization, and hardware-supported virtualization [15]. We elaborate on the three approaches and introduce representative solutions in each approach. We then compare GPU scheduling methods. Table 1 shows a comparison of GPU virtualization solutions in terms of the architecture and scheduling method.

API remoting: This approach virtualizes GPUs at the library level in the GPU execution stack. As GPU vendors tend to not provide the source code of their GPU drivers, API remoting offers a guest OS a GPU wrapper library in order to intercept GPU calls before the calls reach the GPU driver. The intercepted calls are forwarded to the host OS and processed remotely.

GViM [6], GVirtuS [7], rCUDA [13], Pegasus [10], vCUDA [11], and VADI [16] are based on API remoting. They adopt a split device model where the frontend and the backend are located in the guest and the host respectively. A wrapper library in the guest intercepts GPU calls and delivers them to the frontend. The frontend transfers the intercepted calls to the backend in the host, and the backend executes the GPU calls on behalf of the guest. Most of the aforementioned solutions provide a shared memory mechanism for communication between the guest and the host. An exception is rCUDA, which aims at utilizing remote GPUs and uses TCP/IP-based communication for both local and remote GPU virtualization [17].

Existing API remoting solutions adopt communication modules provided by the hypervisor. These modules can burden the virtualization stack with frequent context switching between user and hypervisor spaces. FairGV substantially improves the performance of API remoting by introducing the trap-less architecture presented in Section 3.1.

We select GVirtuS [7] for our exploration of ideal fairness and high utilization of virtualized accelerators because it is the only GPU virtualization framework available that is both open source and supports the latest version of CUDA and OpenCL [18]. In GVirtuS, when the connection between the frontend and the backend is first established, the backend spawns a child process to differentiate the GPU context from those of other applications. We refer to this spawned process as a vGPU (virtual GPU) in the rest of this paper. Also, we will refer to a VM that runs GPU kernels as a GPU VM.

Para & full virtualization: This approach enables GPU virtualization at the driver level utilizing either a custom GPU driver based on reverse engineering [19], [20] or an open source driver [21]. In this approach, the host exposes emulated virtual GPUs to the guest driver, which regards them as real GPUs. Para virtualization modifies the guest GPU driver for improved performance while full virtualization does not modify the guest GPU driver and fully emulates GPUs instead.

GPUvm [12] provides both full and para virtualization in the Xen hypervisor using a custom GPU driver. In full

	GVirtuS	rCUDA	vCUDA	Pegasus	GPUvm	gVirt	NVIDIA GRID	FairGV
Category	API remoting	API remoting	API remoting	API remoting	Para-virt	Full-virt	Hardware-virt	API remoting
Supported GPUs	NVIDIA	NVIDIA	NVIDIA	NVIDIA	NVIDIA	Intel	NVIDIA	NVIDIA
Trap-less architecture	No	No	No	No	No	Partially yes	Yes	Yes
Scheduling discipline	–	–	–	Credit	Credit	Round-robin	–	Fair queuing
GPU-CPU coscheduling	–	–	–	Yes (CPU-centric)	No	No	–	Yes (GPU-centric)
Non-preemptive scheduling	–	–	–	No	Yes	Yes	–	Yes

TABLE 1
Comparison of GPU virtualization solutions in terms of the architecture and scheduling method.

virtualization, GPUvm makes every GPU access generate a page fault so that the hypervisor can emulate the access. This approach shows poor performance because of frequent trapping. GPUvm improves performance by adopting a para-virtualization method that utilizes batch execution.

gVirt [21] implements full virtualization for Intel on-chip GPUs in the Xen hypervisor, to accelerate 2D and 3D graphics workloads. gVirt allows each VM to access the frame and command buffers in the GPU without intervention from the hypervisor. At the same time, privileged instructions are trapped and emulated by the hypervisor for isolation. This approach is called mediated pass-through. A KVM version gVirt called KVMGT also exists [22].

GPU instructions of para & full virtualization are internally processed by the hypervisor, which causes frequent context switching between user and hypervisor spaces. Although gVirt implements mediated pass-through that allows non-privileged instructions to bypass the hypervisor, Tian et al. [21] report that certain applications still suffer from mediation overhead with frequent trapping events.

Hardware-supported virtualization: In this approach, a guest is allowed to access GPUs directly with hardware features for I/O virtualization, which remap direct memory accesses and interrupts to the guest. Intel VT-d and AMD-Vi only support a single VM to exploit a GPU. NVIDIA GRID [23] can support sharing of a single GPU between multiple guests and is implemented in a few NVIDIA GPUs that target cloud computing environments.

Hardware-supported virtualization achieves near-native performance. However, imposing GPU scheduling policies on this approach is difficult because GPU operations bypass the hypervisor. For the same reason, important virtualization features such as execution checkpointing, live migration, and fault-tolerant execution are hard to implement [24].

Scheduling methods: GPU scheduling methods are essential to fair and effective distribution of GPU resources between tenants in a shared computing environment.

Pegasus [10] achieves high performance for CPU-GPU interactive applications under a CPU-centric coscheduling approach. However, the CPU-centric policy can hamper fairness because of frequent interference between the CPU and GPU schedulers. Furthermore, Pegasus does not consider fairness on non-preemptive GPUs. FairGV improves the fairness of Pegasus with GPU-centric coscheduling (Section 3.2.3) and non-preemptive scheduling (Section 3.3).

GPUvm [12] adopts the BAND scheduler of Gdev [9], which is based on Credit scheduling. The BAND scheduler considers non-preemptive GPUs by waiting for the completion of GPU kernels and assigning a credit value to the task based on its GPU usage. gVirt [21] also waits for the ring buffer to be emptied by the GPU to support non-preemptive GPUs. These implementations are dependent on a reverse-engineered or open source driver, which are not supported by many GPUs. FairGV's accounting mechanism is not dependent on a custom or open source driver thanks to its interposition technique explained in Section 3.3.1.

3 Design

In this section, we introduce the design of FairGV, which is our proposed model for strong fairness and high utilization of virtualized GPUs. First, we introduce a trap-less GPU processing architecture to significantly improve the performance and scalability of short-running but repetitive GPU requests. Next, we develop a fine-grain fair queuing algorithm to fairly and effectively schedule a diverse range of GPU applications in terms of GPU computation intensity and CPU-GPU interactivity. Finally, we introduce a collaborative scheduling method combined with a novel and accurate accounting mechanism for fairness between short- and long-running kernels on non-preemptive GPUs.

FairGV is based on GVirtuS. However, our design is not specific to GVirtuS and can be applied to other GPU virtualization frameworks, as well as different hypervisors. In this section, we present our design in more detail in the context of GVirtuS and the KVM hypervisor.

3.1 Trap-Less Architecture

To achieve high performance, GPUs process requests directly from user space using memory-mapped I/O. System calls are hardly used by GPUs, typically just for maintenance and initialization purposes. This is because trapping to the OS kernel in a system call carries significant instruction execution overhead and the indirect cost of cache pollution, which can be thousands of CPU cycles [25]. For a range of GPU applications that issue short-lived and frequent GPU kernel execution requests, these overheads can significantly degrade system efficiency.

Existing API remoting approaches including GViM [6], GVirtuS [7], Pegasus [10], and vCUDA [11] provide a wrapper library that invokes system or hypervisor calls on every GPU request and reply. For example, GViM and Pegasus issue hypervisor calls to use Xenbus and Xenstore [26], which provide shared ring buffers and event channels. vCUDA also adopts the VMchannel residing in the KVM hypervisor for implementing notification channels. OS kernel-based approaches including Gdev [9], GPUvm [12], gVirt [21], and KVMGT [22] use custom open-source drivers such

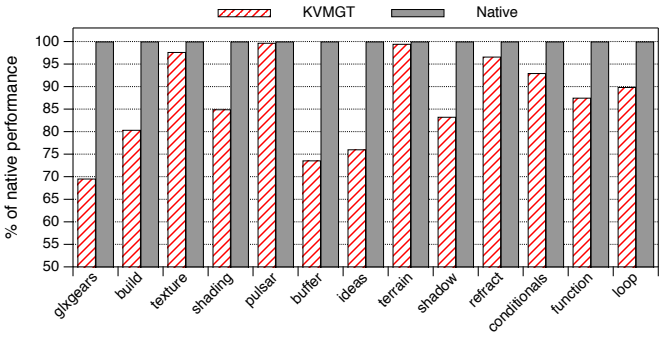

Fig. 1. Performance of KVMGT running glxgears and a suite of scenes in glmark2.

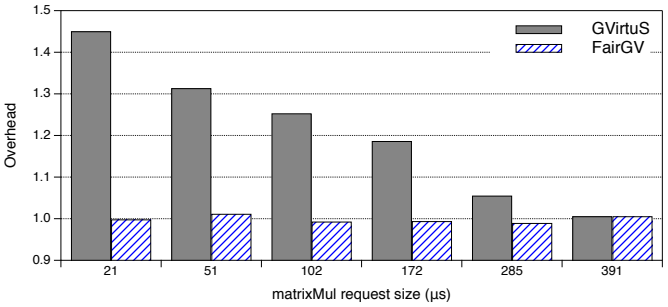

Fig. 2. Overhead of matrixMul measured by GVirtuS and FairGV with different kernel sizes. The corresponding matrix sizes for each kernel size are (160,160)(160,160) for 21 μs, (160,320)(320,160) for 51 μs, (160,480)(480,160) for 102 μs, (160,640)(640,160) for 172 μs, (160,800)(800,160) for 285 μs, and (160,960)(960,160) for 391 μs.

as Nouveau [19] or Intel GPU drivers for virtualization. However, they also heavily use the kernel-level drivers for each GPU request and reply. Figure 1 shows the relative performance of KVMGT running the glxgears and glmark2 OpenGL applications in an Intel HD Graphics 4600. Owing to KVMGT's mediated pass-through where non-privileged instructions bypass the hypervisor (Section 2), the texture, pulsar, and terrain scenes in glmark2 achieve near native performance. However, other applications still suffer from performance degradation with frequent trapping events as reported by Tian et al. [21]. Because of frequent trapping into the OS and the hypervisor, these approaches can be problematic for small and repetitive GPU requests.

Figure 2 shows the overhead of matrixMul in the NVIDIA SDK measured by GVirtuS with its shared memory module and FairGV with different kernel sizes in an NVIDIA TitanX. The overhead is calculated by dividing the execution time in a VM by the time taken in native Linux. The kernel size was adjusted by changing the size of the matrices of matrixMul. When the kernel size is 391 μs, GVirtuS does not incur any overhead. However, with small kernel sizes, it causes significant overhead (from 1.05 at 285 μs to 1.45 at 21 μs) because it executes four system calls on each GPU request for buffer synchronization. This result validates that with small GPU requests, frequent trapping can become a major bottleneck. A significant number of important GPU applications is known to execute small (10 – 250 μs) and frequent GPU kernels [14]. With the advance of GPU micro-architectures, request execution time is expected

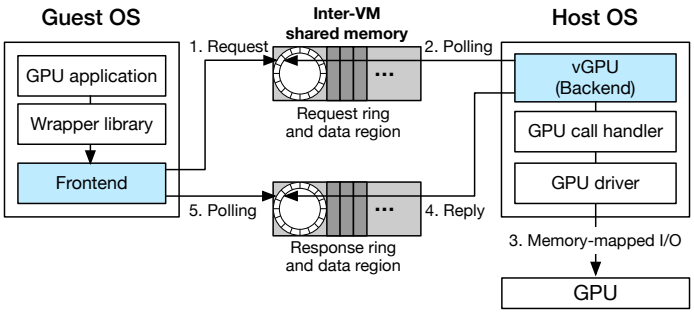

Fig. 3. Trap-less GPU processing architecture of FairGV.

to drop even further, which will in turn inflate the trapping overhead.

FairGV makes the GPU processing architecture trap-less by using three mechanisms:

First, FairGV uses a dedicated inter-VM shared memory region in user space to remove the necessity of trapping to the OS or the hypervisor for communication. For this purpose, FairGV allocates a shared memory segment on the host and dynamically maps the memory region into the virtual address space of each vGPU and its GPU application. As the page tables of each entity are modified at the initialization phase, there is no need to trap the OS or the hypervisor on each GPU request and reply.

Second, a bounded lock-free queue [27] is adopted as a shared memory data structure. Shared memory is typically managed by locks. However, in order to manage locks, system calls that can hold the status of locks are required. Context switching by such system calls is known to take on the order of micro-seconds for a contested lock [28]. To remove such overheads, FairGV creates two lock-free rings for both communication directions, the request and response rings, as depicted in Figure 3. Each ring is composed of descriptors, and each descriptor contains a request (or reply) command and an offset to the data buffer where the actual data to be transferred is stored. Through the lock-free data structure, neither vGPUs nor GPU VMs will read or write the same buffer concurrently, which eliminates the overhead of system calls that manage locks.

Finally, FairGV adopts a polling mechanism when a vGPU or a GPU VM checks whether the shared ring has a new message. FairGV avoids an event notification mechanism such as a virtual interrupt because a notification interrupt causes an expensive VM Exit operation [29], which transfers the control from the guest OS to the hypervisor. During the execution of programs with small and repetitive GPU requests, the notification rate is expected to be high and reduce overall performance. Therefore, FairGV adopts polling, which continuously inspects the shared ring in user space.

The whole process of accessing the GPU is illustrated in Figure 3. Throughout this process, FairGV can deal with GPU requests directly from user space without trapping to the OS kernel or the hypervisor. Therefore, in Figure 2, the overheads of FairGV are shown to be nearly zero regardless of the GPU kernel size, which means the execution time in FairGV is quite close to the execution time of the same GPU request in a native environment. The three mechanisms in

FairGV require additional processing time for transferring and checking messages, but the overhead can be hidden because the vGPU and the virtual CPU (vCPU) of the GPU VM can execute on different cores.

3.2 FairGV Scheduling Method

FairGV combines three scheduling policies including fair queuing, work-conserving, and GPU-centric coscheduling to deal with workloads with mixed intensity of GPU requests. Fair queuing is a basic scheduling policy of FairGV, which attains fairness when sharing a limited resource. However, fair queuing alone is not sufficient to achieve high fairness and utilization with mixed workloads. Therefore, FairGV adaptively activates work-conserving and/or GPU-centric coscheduling according to the workload characteristics.

Work-conserving is applied to non GPU-intensive workloads in order to improve GPU utilization. Work-conserving keeps the GPU busy by allowing a vGPU temporarily having no GPU request to yield its allocated GPU to another vGPU ready to be scheduled. GPU-centric coscheduling is adaptively applied to CPU-GPU interactive workloads to improve the fairness. In this policy, the GPU scheduler has the capability to request coscheduling, instead of the CPU scheduler. GPU-centric coscheduling preserves the fairness policy of fair queuing by eliminating the interference from the CPU scheduler. In summary, the three policies adaptively work together to achieve high fairness and utilization for various workloads. The three policies can be applied to both non-preemptive and preemptive GPUs.

3.2.1 Fair Queuing Algorithm

FairGV is based on a standard fair queuing algorithm, which is widely used for sharing CPUs and I/O devices [30]. We choose fair queuing for two reasons: First, it does not require a priori knowledge of the length of the time slice. Given non-preemptive GPUs, a vGPU may overrun its time slice, which would render scheduling decisions based on that time slice inaccurate. Standard fair queuing overcomes this problem. Second, standard fair queuing prevents a vGPU that wakes up from a lengthy sleep period from accumulating significant unspent GPU time and monopolizing the GPU.

FairGV assigns a weight value, a start tag, and a finish tag to each vGPU and schedules vGPUs in increasing order of start tags. A weight value reflects the vGPU's relative use of GPU resources; it is assigned by the system administrator according to the price paid for GPUs. The start tag and the finish tag represent accumulated virtual run time before and after using the GPU respectively; virtual time is weighted run time and indicates computational progress based on the weight. When a vGPU has finished its requests, its start tag is updated as the value of its finish tag. The finish tag of vGPU i, F_i, after the jth time quantum is calculated as follows: $F_i = S_i + \frac{L_{i,j}}{\omega_i}$ where S_i denotes the start tag, $L_{i,j}$ indicates the execution length (measured in time units, i.e., milliseconds) at the jth time quantum, and ω_i represents the weight. The finish tag is increased in inverse proportion to the weight of the vGPU. A vGPU with a high weight value will therefore have a start tag increasing relatively slowly. As FairGV schedules vGPUs in increasing order of start tags, this vGPU will be selected more frequently in proportion to the weight. The algorithm also maintains a global virtual time in order to assign a new start tag to a vGPU that wakes up. The start tag of an unblocked vGPU is updated as the current virtual time, which is the minimum of the start tags of active vGPUs. With this method, the vGPU cannot monopolize the GPU while it catches up on the start tags of other runnable vGPUs.

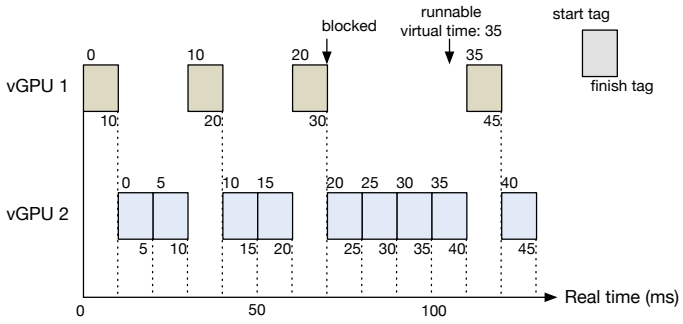

Fig. 4. Example of start tag, finish tag, and global virtual time calculation.

Figure 4 illustrates an example of start tag, finish tag, and global virtual time calculation in FairGV. Consider two vGPUs, 1 and 2, with weight values 1 and 2 respectively. Let us suppose that the length of every time slice is 10 ms, and vGPU 1 is scheduled first when vGPUs have the same start tags. After the first quantum, the start tag of vGPU 1 becomes 10 because the finish tag is calculated as 10 (i.e., $F_1 = S_1 + \frac{L_{1,1}}{\omega_1} = 0 + \frac{10}{1} = 10$), and the start tag is updated as the value of the finish tag. The scheduler proceeds to select vGPU 2 because its start tag is less than that of vGPU 1. After the first run, the start tag of vGPU 2 becomes 5 (i.e., $F_2 = S_2 + \frac{L_{2,1}}{\omega_2} = 0 + \frac{10}{2} = 5$). As the start tag of vGPU 2 is less than that of vGPU 1, vGPU 2 is selected again. This procedure is repeated, and each vGPU receives GPU time in proportion to its weight. After vGPU 1 is blocked at time 70, it becomes runnable at time 105. Then, the start tag of vGPU 1 is forced to have the current virtual time, 35, which is the minimum of the start tags in the system. vGPU 1 will be scheduled in the next turn because of having the minimum start tag, but it cannot claim resources from its idle period.

On preemptive GPUs, a short time slice value in fair queuing (e.g., 1 ms) will cause considerable loss of throughput because of the cost of context switching. To mitigate this overhead, FairGV sets the time slice value to 30 ms when preemptive GPUs are used. For non-preemptive GPUs, this restriction can be relaxed, and we set the value to 6 ms to improve the responsiveness of interactive applications. The both time slice values can be adjusted by the cloud administrator.

3.2.2 Work-Conserving Scheduling

To improve the total GPU utilization, the GPU scheduler in FairGV supports work-conserving where a vGPU voluntarily yields the GPU during its time slice when it has no requests to issue. Such a vGPU accommodates a non-saturating workload that sleeps frequently or is CPU-intensive. To implement work conservation, when a vGPU checks the request ring and finds no requests, it yields

```
next_vcpu ← next vCPU to be run;
next_vgpu ← find_vgpu(next_vcpu);
if next_vgpu ≠ NULL then
    send_schedule_info_to_CPU(next_vgpu);
end
```

Algorithm 1: CPU scheduler part of Pegasus.

```
curr_vgpu ← current running vGPU;
target_vgpu ← target vGPU informed by the CPU;
max_credit ← find_max_credit_value(target_vgpu);
target_credit ← credit_value(target_vgpu);
if target_credit ≥ max_credit then
    GPU_context_switch(curr_vgpu, target_vgpu);
else
    continue_running(curr_vgpu);
end
```

Algorithm 2: GPU scheduler part of Pegasus.

```
prev_vgpu ← previous vGPU to be preempted;
next_vgpu ← next vGPU to run;
GPU_context_switch(prev_vgpu, next_vgpu);
if CPU_GPU_interactive(next_vgpu) = true then
    next_vcpu ← find_vcpu(next_vgpu);
    send_schedule_info_to_CPU(next_vcpu);
end
```

Algorithm 3: GPU scheduler part of FairGV.

```
next_vcpu ← vCPU informed by GPU scheduling;
cpu ← current cpu number of next_vcpu;
if next_vcpu = IDLE then
    wakeup_vcpu(next_vcpu);
end
resched_cpu(cpu);
```

Algorithm 4: CPU scheduler part of FairGV.

the GPU after a predefined spinning time. We will refer to this policy as hybrid spinning. In our prototype, the vGPU releases the GPU after spinning 100K times. The time is configured after considering the communication latency between the host and the guest. Fair queuing changes the status flag of this vGPU to *blocked* and later unblocks it when a new request is available in the request ring. FairGV also helps the CPU scheduler implement a work-conserving policy. When a GPU VM does not have a new message in the response ring because its vGPU is not online, the VM voluntarily releases the CPU instead of continuously polling the ring. The same predefined spinning time as that of the GPU scheduler is used in this hybrid spinning.

3.2.3 GPU-Centric Coscheduling Policy

Some GPU applications continuously submit asynchronous kernel launch functions to the GPU. In these applications, synchronous calls are used rarely, mainly just for copying data between the host and the device before and after consecutive kernel launch and completion. In FairGV, asynchronous GPU calls are queued at the request ring (Section 3.1) and executed asynchronously. Therefore, the GPU VM scheduled by the CPU scheduler can proceed regardless of whether or not its vGPU is currently online. However, other applications execute at least one synchronous call per kernel launch. They typically perform a device to host copy operation after a kernel launch, and this copy is a synchronous operation. We refer to these applications as *CPU-GPU interactive* applications. A synchronous call requires the GPU VM to wait for the result returned from the GPU. Therefore, if the corresponding vGPU is not scheduled immediately, the GPU VM cannot make progress for a significant amount of time, leading to severe performance degradation.

Coscheduling of a GPU VM and its corresponding vGPU can solve this problem. Coscheduling was originally proposed in [31], and is widely used in several CPU schedulers for hypervisors [32] in order to address the performance implications of synchronization between vCPUs. Pegasus [10] applies a Strict coscheduling policy for virtualized GPU scheduling. The Strict coscheduling policy makes a GPU VM and its corresponding vGPU run on physical cores simultaneously. Therefore, Strict coscheduling allows the GPU VM to progress immediately when a synchronous call is submitted. In this scheduling scheme, the CPU scheduler sends a coscheduling request to the GPU scheduler when the next vCPU has a runnable vGPU as shown in Algorithm 1. The GPU scheduler then preempts the current running vGPU and schedules the next vGPU informed by the CPU scheduler. Strict coscheduling prevents unconditional coscheduling because unfairness can occur between vGPUs when low weight vGPUs are scheduled frequently. For this purpose, the GPU scheduler performs a fairness condition check shown in Algorithm 2. The condition is that the credit value of the target vGPU should be higher than the maximum credit value of other vGPUs. If this condition is met, the current vGPU context is preempted.

Strict coscheduling in Pegasus can improve the performance of workloads with frequent CPU-GPU interaction, but suffers from two limitations. First, Strict coscheduling and its sibling, AugC [10], cannot achieve good fairness. This is because the CPU scheduler frequently interferes in the scheduling policy of the GPU scheduler. The GPU scheduler part in Pegasus is based on Credit scheduling [26], and this policy can maintain good fairness only when the time slice of each vGPU is fully and exactly consumed. When the CPU scheduler preempts the current running vGPU, the descheduled vGPU is put at the tail of its priority list without fully spending its time slice. This situation can compromise overall fairness [33]. The coscheduling condition check in Algorithm 2 prevents excessive and unconditional vGPU preemption, but it cannot promise good fairness because it allows a certain amount of preemption. Second, multiple coordination requests issued from different cores at the same time cannot be accepted. Under the coscheduling condition check policy, only one GPU VM that has the maximum credit value can be accepted by the GPU scheduler. Unselected GPU VMs should then wait even though they need their vGPUs.

To address these issues, we propose a new GPU-centric coscheduling policy. The key idea is to pass the capability to request coscheduling to the GPU scheduler, instead of the CPU scheduler. The GPU scheduler can then retain the primary responsibility for preserving the fair share of each vGPU while protecting the performance of workloads with frequent CPU-GPU interaction. Furthermore, as the GPU scheduler requests only one vCPU candidate for coscheduling at each instance, the execution of GPU VMs is balanced

and not interleaved in time. This eliminates unnecessary waiting of VMs that require synchronized GPU executions.

The GPU scheduler part in FairGV is based on the fair queuing policy explained in Section 3.2.1. The GPU scheduler sends a coordination request to the CPU scheduler when it changes the current vGPU context as shown in Algorithm 3. The request is delivered to the CPU scheduler in KVM via a new system call that receives the next vCPU as a parameter. Upon the request, the CPU scheduler unblocks the requested vCPU and tickles the core where the vCPU has become runnable in order to schedule the vCPU as shown in Algorithm 4.

The vCPU associated with the next vGPU should be scheduled immediately by the CPU scheduler for effective coscheduling. The CPU scheduler in KVM, the Completely Fair Scheduler (CFS), implements a class of weighted fair queuing [34], similar to that of FairGV. When FairGV tickles the core where the target vCPU becomes runnable, CFS selects a vCPU with the lowest virtual time in the run queue. To ensure that the target vCPU is selected, FairGV takes advantage of hybrid spinning (Section 3.2.2) where a vCPU of a GPU VM is blocked when the GPU VM does not have an item to process in the response ring. When the next vGPU (next_vgpu in Algorithm 3) was de-scheduled in its previous preemption point, the next vCPU (next_vcpu in Algorithm 4) would be also blocked by hybrid spinning, because its vGPU became offline. When the next vCPU is unblocked at the current scheduling point, it can have the lowest virtual time allocated by the fair queuing algorithm of the CPU scheduler. Therefore, the target vCPU can be scheduled immediately for coscheduling. This mechanism is non-intrusive, because FairGV does not force the CPU scheduler to schedule the target vCPU. Otherwise, the CPU scheduler may hamper fairness between CPU workloads.

In addition, FairGV implements a hybrid scheduling policy where CPU-GPU interactive workloads are selectively coscheduled (Algorithm 3). Coscheduling is not effective and may cause some overhead against non CPU-GPU interactive applications, because such applications mainly issue asynchronous GPU calls. To selectively apply coscheduling, FairGV characterizes GPU workloads in the host by measuring the frequency of synchronous calls. When a vGPU submits more than 10 synchronous calls per 10 ms, we classify the vGPU as CPU-GPU interactive and apply coscheduling. To deal with fluctuating phase changes in GPU workloads, FairGV continuously monitors communication between the CPU and the GPU. FairGV feeds this information to the GPU scheduler in order to decide whether to coschedule a vGPU and its corresponding vCPU.

When there is more than one vCPU in the GPU VM, FairGV associates a vGPU with its corresponding vCPU by tracking the CPU a request is coming from, between all vCPUs in the GPU VM. The frontend sends its CPU number (i.e., virtual CPU number) to the backend through a descriptor when it submits GPU requests. The GPU scheduler knows which vCPU to coschedule by tracking this information.

3.3 Non-Preemptive Scheduling

FairGV targets supporting fair GPU scheduling for both non-preemptive and preemptive GPUs. GPUs were non-preemptive until recently, which implies that GPU requests are processed serially in a GPU on a first come, first served basis. GPUs could only preempt executions at the boundary of GPU kernel calls. The problem with this approach is that certain kernels are composed of an amount of computation work, and such kernels can cause unfairness and poor responsiveness in a shared environment. To overcome this limitation, GPU architectures that support hardware-based preemption were suggested, and finally preemptive GPUs have emerged in the market recently [4]. However, context switching in such GPUs is very expensive and recent research [5] reports that hardware-based preemption decreases the total throughput up to 35% in a wide range of GPU applications. Owing to this reason, we expect that certain HPC clouds will still employ non-preemptive GPUs for throughput-sensitive applications. Therefore, supporting non-preemptive GPUs for fair sharing in virtualization is still an important issue and remains a challenging endeavor.

```
void Backend::GpuExecute(Handler *handler, ...) {
    unsigned long long startTime = NOW();
    Result *result = handler->Dispatch(request, parameter);
    cudaDeviceSynchronize();
    unsigned long long endTime = NOW();
    unsigned long long executionTime = endTime - startTime;
    ...
}
```

Fig. 5. Pseudo code of the accounting mechanism in FairGV.

3.3.1 Accounting Mechanism

For non-preemptive GPUs, a scheduling method needs to precisely measure how long each request occupies a GPU and to take this into account when scheduling vGPUs. Scheduling vGPUs without respective request accounting causes unfairness in a time slice-based scheduler because of interference between short- and long-running kernels inside the GPU. Let us suppose that a VM continuously executes short running GPU kernels (e.g., 10 μs) whereas another VM issues long running GPU kernels (e.g., 100 ms). As a kernel launch operation is asynchronous, each VM can issue as many GPU kernels as possible during its time slice. The issued GPU kernels are then buffered in the hardware queue of the GPU and are processed sequentially. However, as the GPU is non-preemptive, the long running kernels will be uninterrupted and monopolize the GPU, thus causing severe unfairness between the two VMs. To achieve fairness, we need to account for each GPU request and deliver this information to the GPU scheduler.

OS kernel-based approaches address this problem via reverse engineering. TimeGraph [8] and Gdev [9] replace the black-box NVIDIA driver with the Nouveau [19] or pscnv [20] driver, a reverse engineered driver for NVIDIA GPUs. They configure the custom driver to generate an interrupt after a group of requests is processed in the GPU. Disengaged scheduling [14] infers the direct-mapped interface by making the candidate memory region read-only and catching the resulting page faults. However, reverse engineering can be a difficult task as new GPUs are gradually introduced in the market. For example, no open source

Benchmark	NVIDIA (µs)	FairGV (µs)	Error (%)
matrixMul	207	214	3
BlackScholes	377	381	1
scan	171	176	3
convolutionSeparable	391	397	2

TABLE 2
Average execution time of the first 20 kernels measured by the NVIDIA Profiler and FairGV.

driver can fully support the recent Maxwell architecture of NVIDIA since its introduction in February 2014. API remoting approaches including Pegasus [10] and GViM [6] avoid reverse engineering as they reuse the vendor GPU driver, but they just allow a vGPU to run for a time slice, which causes a severe fairness problem for mixed workloads.

To achieve fairness in the execution of short- and long-running kernels, FairGV employs a simple interposition technique to account for the time each GPU request occupies. The measured time is used by the GPU scheduler (Section 3.2.1), for selecting a vGPU to run. The technique FairGV develops is not dependent on reverse engineering and can be applied to any GPU virtualization platform. FairGV inserts an additional device synchronization function (e.g., cudaDeviceSynchronize() in CUDA and clFinish() in OpenCL) after a vGPU dispatches a request, as depicted in Figure 5. The synchronization function waits until the GPU has completed the submitted request. Therefore, when the function returns, the vGPU can measure the execution time of the submitted request. Table 2 shows the average execution time of the first 20 kernels of selected applications in the CUDA SDK, which are measured by the NVIDIA Profiler and FairGV respectively. The results indicate that the interposition technique of FairGV is quite accurate, with an error of less than 3% of the actual run time.

Because GPU software stacks are generally not open source, the operation of device synchronization functions is unclear. We infer that the function continuously checks a reference counter in the memory-mapped I/O region, and the counter tracks the completion of each request. This assumption is based on reverse engineered results that recognize the semantics of data structures in user space [8], [19], [35]. Because of this additional processing, we observe that inserting a device synchronization function after every GPU request causes a drop in performance of up to 7%. To alleviate this, we introduce a sampling technique, described in more detail in Section 3.3.2.

3.3.2 Collaborative Scheduling

When FairGV operates on non-preemptive GPUs, the scheduling discipline of FairGV is configured as non-preemptive. In this discipline, each vGPU has an accounting function (explained in Section 3.3.1) for measuring GPU usage. When the usage exceeds the length of a predefined time slice (i.e., 6 ms), the vGPU informs the GPU scheduler of the usage and voluntarily releases the GPU. The scheduler then executes its account update function and selects the next vGPU to run.

FairGV uses the interposition mechanism to measure the GPU usage of each request (Section 3.3.1). This mechanism introduces overhead up to 7%. To address this problem, we introduce a sampling technique that can predict the number of requests that may use up the entire length of the time slice. When a vGPU is selected to run, it measures the execution times of the first several requests (5 in our system) and obtains the average completion time. During this sampling period, if the sum of the lengths of the requests exceeds the length of time slice, the vGPU is forced to release the GPU. Based on the obtained value, the accounting function infers the number of remaining requests to fill the time slice value. The vGPU then runs without interposition until it reaches its last request. A device synchronization function then follows the last request to exactly measure the total execution time. As the execution times of successive GPU kernels of a program tend to be similar, this mechanism preserves accurate accounting while substantially reducing the overhead. Additional samples can improve accounting accuracy, but may sacrifice performance. Accounting inaccuracy or a considerable surplus or shortage of run time caused by sudden phase changes in the application can be compensated by fair queuing in subsequent scheduling points.

FairGV in non-preemptive scheduling specifically targets the efficient execution of fine-grain GPU kernels, with typical execution times under 500 μs, similarly to all GPU virtualization systems published in the literature [9], [10]. Many important HPC applications, such as particle simulation, molecular dynamics, and medical imaging, as well as emerging applications in the domain of real-time data analytics for computational finance, smart traffic, and cybersecurity repetitively or continuously execute short-lived kernels that require low latency processing [14]. FairGV's cooperative scheduling is beneficial for improving fairness and GPU utilization without compromising latency in these scenarios.

FairGV also copes with repetitive coarse-grain kernels exceeding the length of the time slice. When such a kernel uses up its time slice, fair queuing assigns the corresponding vGPU a high start tag value after the kernel finishes its execution. The corresponding vGPU will not be selected again until other vGPUs with fine-grain kernels catch up on the start tag. In this way, fairness is preserved between long- and short-running kernels.

Unfortunately, offending, greedy, or buggy applications can submit very long running kernels into their vGPUs (e.g., 60 seconds). This puts cooperative scheduling in jeopardy. To address this issue, FairGV implements a GPU kernel slicing technique that can split a long running GPU kernel into small ones. The literature shows this technique to be feasible with low performance overhead and significant benefits in terms of responsiveness [36]. We have adopted the implementation details from GPES [37]. When a vGPU has a long running kernel, FairGV divides the kernel into a set of sub-kernels so that a sub-kernel is executed by a specified number of thread blocks. How to decide an appropriate number of thread blocks will be discussed in more detail in Section 5.5.

4 Implementation

FairGV is implemented in the KVM hypervisor. The implementation is depicted in Figure 6. FairGV is based on

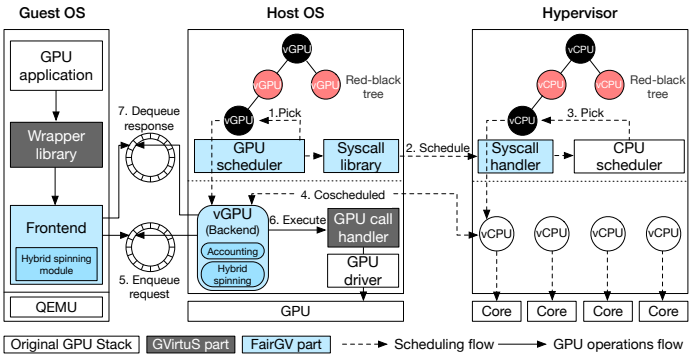

Fig. 6. Implementation detail of FairGV.

GVirtuS and reuses the GVirtuS wrapper library in the guest OS, and the GPU call handler in the host OS by which the vGPU executes CUDA/OpenCL operations. Also, FairGV performs significant changes to the frontend and the back-end of GVirtuS to implement the trap-less architecture and other support for scheduling. We additionally implement our GPU scheduling methods on top of the redesigned framework. This section provides implementation details regarding the inter-VM shared memory, GPU scheduling, and coordination mechanisms.

4.1 Inter-VM Shared Memory

As explained in Section 3.1, FairGV establishes a shared memory segment between the host and a guest when the guest boots. For this purpose, we develop a virtual device in QEMU, which is a user space interface of KVM and emulates peripherals, to map a shared memory region created by the host into the guest address space. The virtual device can be controlled by the guest with a loadable kernel module. The shared memory creation process is as follows: First, the backend creates a POSIX shared memory object using shm_open() and transfers the object handle to the frontend by a network transport. Second, the frontend informs the virtual device in QEMU about the handle so that the virtual device can open and map the same region by using shm_open() and mmap(). Finally, the frontend can use the region by mapping the virtual device into memory.

4.2 GPU Scheduling

The GPU scheduler implements fair queuing by maintaining vGPUs in a time-ordered red-black tree, similarly to the CPU scheduler of KVM [34]. A red-black tree is a self-balancing binary tree, which offers good worst-case run time. The GPU scheduler stores vGPUs having lowest start tags toward the left side of the binary tree and selects the left-most node at a scheduling point. The GPU scheduler and each vGPU are implemented as user-level processes in the host. They are also bound to a dedicated core. For context switching, the GPU scheduler requests the CPU scheduler to swap the current and next vGPUs through a new system call. The CPU scheduler then blocks the current vGPU and wakes the chosen one up to proceed. As there are no other runnable processes in the dedicated core, the next vGPU can be executed without delay.

4.3 Coordination Mechanism

As explained in Section 3.2.3, the GPU scheduler sends a coscheduling event to the CPU scheduler when the selected workload is CPU-GPU interactive. In this case, the CPU scheduler awakes the vCPU of the selected vGPU in order to coschedule the two. For this procedure, the GPU scheduler sends the target vCPU information through a new system call. Upon message reception, the system call handler unblocks the target vCPU by changing the status flag to *runnable* and tickles the target core by calling resched_cpu().

5 Evaluation

We implemented FairGV on an Intel Xeon E5-2620 v3 platform with six 2.4 GHz cores, 15 MB of L3 cache, and 32 GB of main memory. The GPU used in this evaluation is an NVIDIA TitanX with 3,072 cores, which is based on the NVIDIA Maxwell architecture and does not support hardware-based preemption. Our experiments employ the NVIDIA 352.55 driver and the CUDA 6.5 library, which is supported by the current version of GVirtuS.

5.1 Experimental Workload

We collected benchmarks from Rodinia 3.0 [38] and the CUDA SDK 6.5, which provide applications from a diverse range of HPC domains. The list of benchmarks is provided in Table 3. BS, CS, HG, MM, and SCAN are from the CUDA SDK while the remaining ones come from the Rodinia suite. We lengthened the execution time of each program to about 10 seconds by increasing the problem size or the iteration count, which minimizes the GPU initialization cost when we repeatedly execute programs for evaluation. We profiled each program in Table 3 by measuring the average kernel size and the frequency of GPU calls per unit of time between the host and the device. When a vCPU issues more than 10 synchronous calls per 10 ms, we characterize the program as *CPU-GPU interactive* as explained in Section 3.2.3. This class of applications demands a coscheduling method.

5.2 Trap-Less Architecture Evaluation

The performance overhead (or speedup) when using GVirtuS, rCUDA, and FairGV is depicted in Figure 7. The overhead is calculated by dividing the execution time of a program in a VM by the time taken by the program in native Linux. We ran each benchmark program 30 times and obtained the average value to report the overhead.

GVirtuS with its shared memory module can suffer from performance degradation when the kernel size is under 250 μs (CFD, HS, SRD1, MM, and SCAN). Trapping to the OS kernel per request causes high overhead as explained in Section 3.1. rCUDA uses TCP/IP for inter-VM communication [17]. For high network performance, we enabled *virtio*, which is a para-virtualized network driver for KVM and offers up to 30 Gbps of inter-VM bandwidth in our system. Compared to GVirtuS, rCUDA does not show performance degradation in HS and MM, which mainly execute asynchronous calls, despite their small kernel sizes. In rCUDA, asynchronous calls are transferred to the host asynchronously and buffered in the queue while the GPU is running [13]. Therefore, the trapping overhead can be

Benchmark	Domain	CPU-GPU interactive	Kernel size (µs)	Total calls per 10 ms	Asynchronous calls per 10 ms	Synchronous calls per 10 ms	Run time in VM (s)
cfd (CFD)	Fluid Dynamics	Yes	198	136.891	49.761	87.130	8.0
heartwall (HW)	Medical Imaging	Yes	10,201	12.028	0.794	11.233	1.3
hotspot (HS)	Physics Simulation	No	133	59.534	59.520	0.014	7.8
laveMD (LMD)	Molecular Dynamics	No	1,908,357	0.064	0.002	0.061	3.4
nw (NW)	Bioinformatics	No	11,032	2.743	2.666	0.076	1.4
pathfinder (PF)	Grid Traversal	No	592	0.105	0.027	0.078	1.8
srad_v1 (SRD1)	Image Processing	Yes	46	891.115	247.525	643.589	9.3
BlackScholes (BS)	Finance Processing	Yes	377	52.673	26.318	26.355	7.6
convolutionSeparable (CS)	Image Processing	Yes	391	38.515	19.223	19.281	5.2
histogram (HG)	Image Processing	Yes	431	79.150	39.561	39.588	10.1
matrixMul (MM)	Math Processing	No	207	44.275	44.241	0.033	5.8
scan (SCAN)	Array Processing	Yes	171	109.462	54.705	54.757	8.8

TABLE 3
Description of the evaluated benchmark applications.

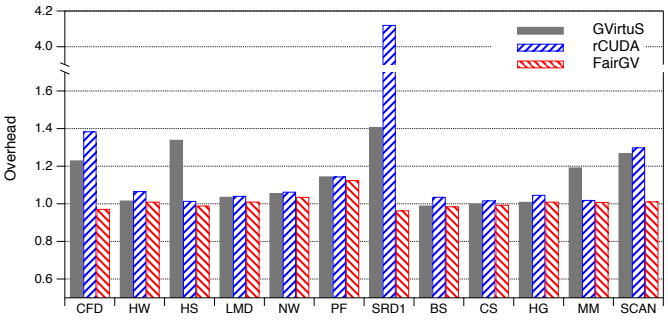

Fig. 7. Basic standalone performance.

overlapped with GPU computation. rCUDA exhibits performance degradation in applications with small kernel sizes and frequent synchronous calls (i.e., CFD, SRD1, and SCAN) because they burden the network virtualization stack. SRD1 specifically generates significant overhead due to very small (46 μs) and repetitive (644 sync calls per 10 ms) requests.

FairGV addresses this issue by introducing the trapless architecture. The overhead of FairGV in most of these programs is close to ideal (1.0). Because the vGPU and its vCPU execute in parallel, occasionally, FairGV introduces slight speedup (e.g., CFD and BS), which means that FairGV actually achieves more efficient scheduling than Linux. This result also shows that the interposition technique in FairGV does not cause performance degradation due to the sampling technique introduced in Section 3.3.2. PF in FairGV shows a little higher overhead than other benchmarks, because of frequent file access operations during its execution. This is caused by the I/O performance overhead in virtualization and addressing it is beyond the scope of this paper.

5.3 GPU Schedulers for Comparison

We implemented state of the art GPU schedulers including *Credit* and *Strict coscheduling* to evaluate their impact on our workloads in terms of fairness and performance. The implementation of Credit scheduling is adopted from the CPU scheduler of Xen [26]. We created two variants, *Credit-poll* and *Credit-hs*. *Credit-poll* adopts continuous polling when checking the response ring whereas *Credit-hs* uses FairGV's hybrid spinning for the CPU scheduler (Section 3.2.2). The algorithm of Strict coscheduling is from Pegasus [10] and is based on Credit scheduling. The policy makes a vCPU and its corresponding vGPU run on physical cores simultaneously to solve synchronization bottlenecks. It sends a coscheduling request from the CPU scheduler to the GPU scheduler when a GPU VM is selected to run whereas FairGV generates the request from the GPU scheduler (Section 3.2.3). Both the Credit scheduler and the Strict coscheduler adopt simple time-sharing based on a time slice.

5.4 FairGV Scheduling Method Evaluation

This section evaluates FairGV's basic scheduling policies including fair queuing, work-conserving, and coscheduling.

5.4.1 Fairness and Performance Metric

To quantify fairness between multi-tenants in the Cloud, we use the Min-Max Ratio (MMR), as introduced in Pisces [39], which provides a min-max fairness notation for multi-tenancy. The MMR is defined as $\frac{\min x_i}{\max x_i}$, where x_i is the normalized throughput of vGPU i. x_i is calculated as $\frac{T_i}{O_i}$, where T_i is the measured throughput of vGPU i, and O_i is the ideal fair throughput of the vGPU. The index ranges from 0 (completely unfair) to 1 (completely fair). For a performance metric, we use the aggregated overhead (or equivalently, speedup). We obtain this metric by dividing the ideal aggregated throughput by the measured aggregated throughput. In this experiment, we run the same benchmark program in all VMs: 1) to prevent existing schedulers based on simple time-sharing from introducing unfairness with different kernel-sized workloads and 2) to harmonize the performance metric that we use for the MMR, as each benchmark reports a different performance metric depending on its application domain (e.g., the execution time in seconds or processed options or pixels per second).

5.4.2 vCPU Weight Assignment

We vary the weight of each vGPU to evaluate weighted fair sharing. In this setup, the CPU weights of GPU VMs should be calculated accurately to guarantee minimum CPU allocations to the VMs. If the CPU weight is too low, the GPU VM cannot run for sufficiently long time to submit GPU requests to its vGPU. A GPU VM requires at least the

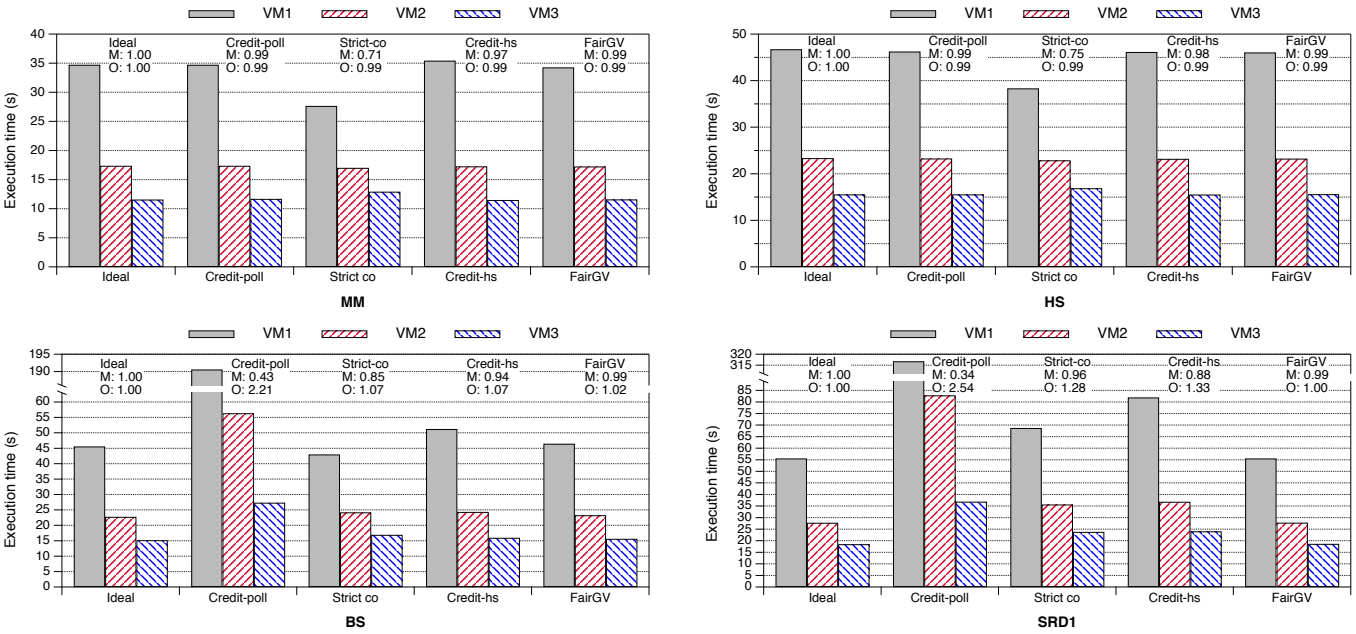

Fig. 8. Completion time for MM, HS, BS, and SRD1 deployed in three VMs. The ratio of the GPU weights for VM1–VM3 is 1:2:3. *M* and *O* denote the Min-Max Ratio (MMR) and the overhead respectively.

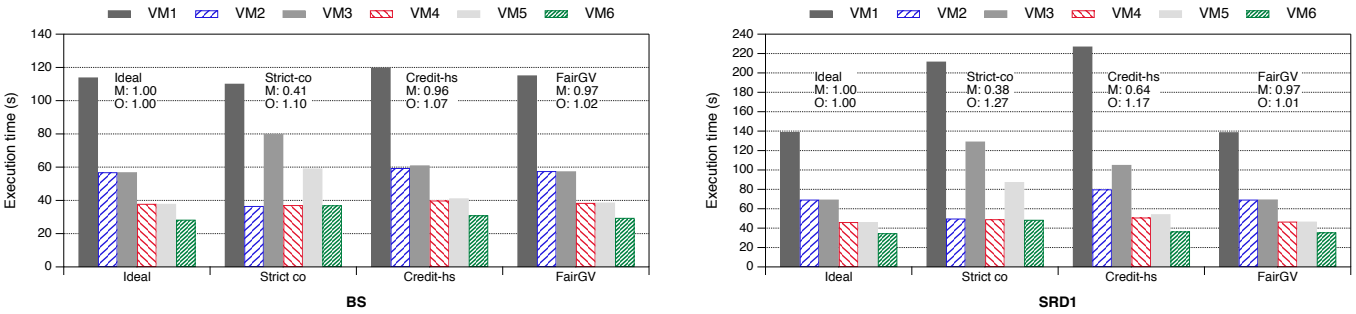

Fig. 9. Completion time for BS and SRD1 deployed in six VMs. The ratio of the GPU weights for VM1 - VM6 is 1:2:2:3:3:4.

same amount of CPU time as the corresponding vGPU will consume. If the GPU VM has an additional CPU workload, it should be given additional weight for processing that workload.

5.4.3 *Evaluation in a Non-Congested Setup*

To understand the basic scheduling capability in a non congested execution environment, we deploy three VMs on the same core. As every benchmark in Table 3 is a single-threaded application, each VM launches only one vCPU together with its vGPU. The ratio of the vGPU weights for VM1–VM3 is configured to 1:2:3. We select four applications with a long execution time from Table 3: MM, HS, BS, and SRD1. BS and SRD1 are CPU-GPU interactive applications whereas the others are not. Other programs with long running times show similar patterns to the selected ones, according to CPU-GPU interactivity. We ran each benchmark program 30 times and obtained the average value measured when all programs were run concurrently. As the relative standard deviation of the execution times is less than 7%, the obtained average value can be used to compare performance and fairness.

Figure 8 shows the completion time, MMR, and aggregated overhead of each application. For the non CPU–GPU interactive workloads (MM and HS), Credit-poll and -hs can achieve high fairness ($\geq$ 0.9 MMR) with low overhead ($\leq$ 1). The GPU VM in these applications performs mainly asynchronous calls in which the CPU and the GPU schedulers can work independently of each other. High fairness between vGPUs is then achieved by the fairness policy of Credit scheduling. Strict coscheduling fails to achieve reasonable fairness because it can only handle CPU-GPU interactive applications properly. For non-interactive applications, Stric coscheduling causes unnecessary and frequent vGPU context switching due to coordination requests from the CPU scheduler. As the GPU scheduler is mainly responsible for preserving the fair share of each vGPU, frequently changing vGPU contexts by the CPU scheduler causes unfairness. FairGV operates using non-coscheduling by hybrid scheduling for non CPU-GPU interactive applications. It achieves nearly ideal weighted fair sharing ($\geq$ 0.99 MMR), similarly to other Credit scheduling policies.

Considering CPU-GPU interactive workloads, Credit-poll suffers from fairness and performance deterioration.

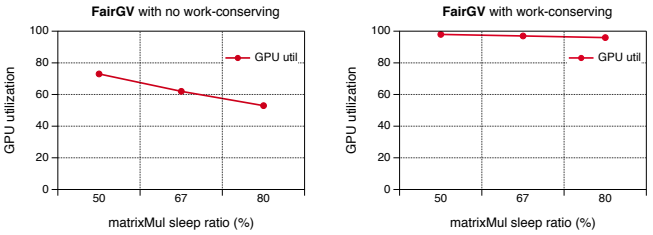

Fig. 10. GPU utilization of co-running VMs running BS and a non-GPU-saturating version of MM, using FairGV with and without the work-conserving policy

When the corresponding vGPU is not online, the GPU VM spins uselessly checking the response ring for the replies of synchronous calls. Strict coscheduling solves this problem by running the vCPU and its vGPU together, which can dramatically improve performance and fairness. Credit-hs can also deal with this situation properly because the waiting VM yields the CPU to other GPU VMs that may have pending replies. As SRD1 is far more CPU-GPU interactive than BS (644 vs 26 sync calls per 10ms), Credit-hs exhibits high overhead in SRD1; frequent synchronous calls may cause excessive vCPU context switching, which is an expensive operation. FairGV can achieve nearly ideal weighted fair sharing (≥ 0.99 MMR) with the least overhead among all schedulers because it makes accurate coscheduling decisions in the GPU scheduler. These decisions cause neither unnecessary vCPU switching, which may harm performance, nor vGPU switching, which may harm fairness.

5.4.4 Evaluation in a Congested Environment

This section evaluates the schedulers under congestion, where the GPU device is shared by multiple VMs at the same time. For this purpose, we deploy six VMs in three cores and run BS and SRD1 respectively. The weight ratio of VM1–VM6 is configured as 1:2:2:3:3:4. We pinned two adjacent VMs in the same core, thus making the total ratio across the cores (1+2):(2+3):(3+4), to observe the performance and fairness impact of the unbalanced weight combination.

Figure 9 shows the completion time, MMR, and aggregated overhead of each application. In this evaluation, Strict coscheduling cannot achieve fairness at all because GPU scheduling is significantly affected by CPU scheduling. On the same core, a vCPU with a high weight achieves better performance as the number of coscheduling requests issued by the GPU VM is in proportion to its vCPU weight (Section 5.4.2). Pegasus suggests the fairness condition check algorithm to prevent excessive vGPU switching (Algorithm 2), but it does not work properly in a congested execution environment. Credit-hs exhibits high fairness and performance in the case of BS, but not in SRD1. Because SRD1 is highly CPU-GPU interactive, yielding operations occur frequently in each GPU VM, thus spending significant amounts of time in vCPU context switching. FairGV shows quite high fairness (≥ 0.97 MMR) with low overhead (≤ 1.02) under congestion.

5.4.5 Work-Conserving Scheduling Evaluation

Figure 10 shows the total GPU utilization of co-running VMs running BS and a non-GPU-saturating version of MM, using FairGV with and without the work-conserving policy. In this evaluation, we modified MM to contain sleep functions, which decrease the percentage of time MM spends on the CPU. We adjusted the sleep to total execution time ratio from 50% to 80%. FairGV without the work-conserving policy exhibits low GPU utilization as the sleep ratio increases, compared to FairGV with the work-conserving policy. In non work-conserving execution, the GPU is wasted while a vGPU checks the request ring having no requests with continuous polling. FairGV eventually adopts hybrid spinning which preserves work-conserving scheduling on the GPU and improves GPU utilization.

Grid size	Kernel size (ms)	Number of sub-kernels	Number of thread blocks	Slowdown (%)
600 x 600	19,506	1	360,000	0.0
		64	5,625	0.0
		128	2,813	1.0
		256	1,407	4.3
		512	704	11.8
		1,024	352	23.9
		2,048	176	47.8
		3,072	118	71.1
		4,096	74	94.6
900 x 900	81,030	1	810,000	0.0
		64	12,657	0.0
		128	6,329	0.0
		256	3,165	0.0
		512	1,583	0.2
		1,024	792	6.2
		2,048	396	17.5
		3,072	264	30.7
		4,096	166	44.5

TABLE 4
Slowdowns of sliced MM with two inputs of grid sizes with 600×600 and 900×900 respectively, in terms of the number of sub-kernels and the number of thread blocks per sub-kernel.

5.5 Non-preemptive Scheduling Evaluation

This section evaluates FairGV's support for non-preemptive GPUs including collaborative scheduling and kernel slicing.

5.5.1 Collaborative Scheduling Evaluation

Figure 11 shows the normalized run times of MM and HS executing in a VM with the Credit-poll and FairGV policies when another background VM executes MM while adjusting the kernel size from 21 – 1,605 μs. In this experiment, we observe the fairness impact on mixed workloads with different kernel sizes. The kernel size of background MM is adjusted by changing the size of input matrices. Because Credit-poll is a simple time sharing scheduler, the normalized run times of MM and HS are adversely affected as the kernel size of background MM increases. When the kernel size of background MM is smaller than the kernel sizes of MM (207 μs) and HS (133 μs), the normalized run times stay under 2.0 in both programs, which means that MM and HS are occupying more GPU resources than background MM. In the opposite case, background MM significantly increases the execution times of both programs. However, FairGV shows robust fairness because FairGV's collaborative scheduling is based on accurate time accounting, which accounts for each respective GPU request correctly

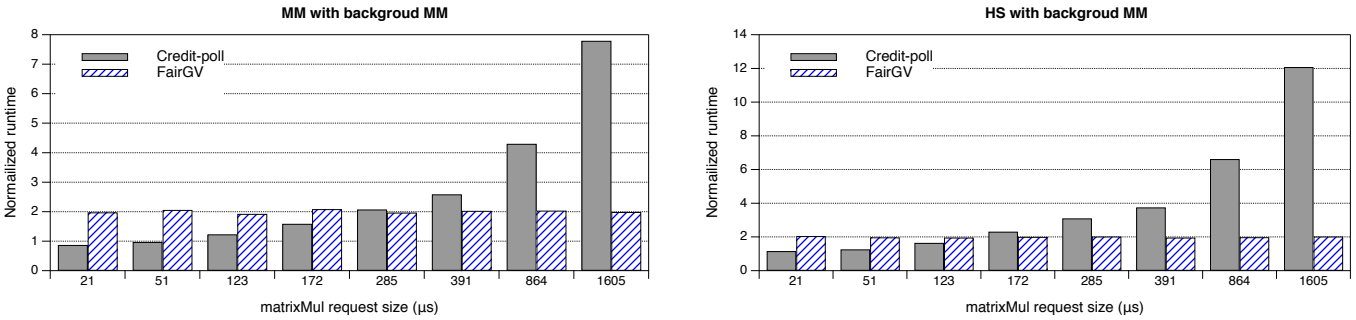

Fig. 11. Normalized run times of MM and HS executing in a VM with the Credit-poll and FairGV policies, when another background VM executes MM while adjusting the kernel size from 21 – 1,605 μs.

(Section 3.3.1). In FairGV, each vGPU has an accounting function and measures the GPU usage after the completion of a request. When the total usage reaches the time slice value, the vGPU voluntarily releases the GPU and informs the GPU scheduler of this usage. This procedure realizes fair-sharing between workloads with different kernel sizes.

5.5.2 Kernel Slicing Evaluation

FairGV implements a GPU kernel slicing technique for preventing offending, greedy, or buggy applications from monopolizing the GPU as explained in Section 3.3.2. In this section, we observe the overhead of this technique and heuristically determine an appropriate number of thread blocks per sub-kernel to minimize the overhead. For this purpose, we slice the kernel of MM into multiple sub-kernels for two inputs of grid sizes with 600×600 and 900×900 respectively. Table 4 shows the slowdowns of the two large inputs, whose execution times are about 20 and 80 seconds respectively. From this result, we can identify that 1) a larger kernel (900×900) generally incurs less overhead than a smaller kernel (600×600) given the same number of sub-kernels, 2) there exists a trade-off between fine-grained kernels and overhead, and 3) when the number of thread blocks per sub-kernel is more than about 1,500, the slowdown can be confined to less than 5%. From this observation, FairGV splits a long running kernel into small ones so that the number of thread blocks executed in each sub-kernel is no less than 1,500. FairGV identifies a kernel as very long running when the grid size is more than 600×600, but this value is configurable.

6 Discussion

Beyond the trap-less GPU processing design and scheduling methods, FairGV tries to achieve device memory partitioning for each GPU VM. As the GPU device driver is a black box, it is hard to explicitly partition the device memory between vGPUs. Programmers may write GPU kernels that use the entire device memory space, or the total memory use of all co-running vGPUs may exceed the total device memory capacity. These situations will result in blocking of vGPUs. To address this issue, FairGV prevents a vGPU from allocating memory beyond its allowed amount through inspecting the device memory (de-)allocation functions (e.g., cudaMalloc() and cudaFree() in CUDA, and clCreateBuffer() and clReleaseMemObj() in OpenCL) at the backend.

7 Conclusion

As current GPU virtualization software cannot provide acceptable fairness and performance isolation, tenants in cloud computing may experience unfairness and unpredictable performance variation due to contention with other users. In this paper, we investigated the trap-less GPU processing architecture, the new fair queuing method, and the collaborative scheduling algorithm for providing system-wide weighted fair sharing and performance isolation in GPU virtualization. Our FairGV prototype implementation achieves near ideal fairness ($\geq$ 0.97 Min-Max Ratio) with high GPU utilization ($\leq$ 1.02 aggregated overhead) in a range of mixed HPC workloads.

Acknowledgments

This work is supported by the European Commission under the Horizon 2020 program (RAPID project H2020-ICT-644312).

References

[1] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 351–362.

[2] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: integrating a file system with gpus," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 485–498.

[3] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running hpc applications in public clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 395–401.

[4] NVIDIA, "Gp100 pascal whitepaper," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[5] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 193–204.

[6] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "Gvim: Gpu-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. ACM, 2009, pp. 17–24.

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

[8] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *2011 USENIX Annual Technical Conference (USENIX ATC11)*, 2011, p. 17.
[9] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-class gpu resource management in the operating system." in *USENIX Annual Technical Conference*, 2012, pp. 401–412.
[10] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *2011 USENIX Annual Technical Conference (USENIX ATC11)*, 2011, p. 31.
[11] L. Shi, H. Chen, J. Sun, and K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," *Computers, IEEE Transactions on*, vol. 61, no. 6, pp. 804–816, 2012.
[12] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Why not virtualizing gpus at the hypervisor?" in *2014 USENIX Annual Technical Conference (USENIX ATC14). Philadelphia, PA: USENIX Association*, 2014, pp. 109–120.
[13] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE, 2010, pp. 224–231.
[14] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 301–316.
[15] C.-H. Hong, I. Spence, and D. Nikolopoulos, "Gpu virtualization and scheduling methods: A comprehensive survey," *ACM Computing Surveys (CSUR)*, in Press, 2017.
[16] C. Lee, S.-W. Kim, and C. Yoo, "Vadi: Gpu virtualization for an automotive platform," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 277–290, 2016.
[17] J. Duato, A. J. Peña, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Ortí, "Enabling cuda acceleration within virtual machines using rcuda," in *High Performance Computing (HiPC), 2011 18th International Conference on*. IEEE, 2011, pp. 1–10.
[18] R. Montella, G. Giunta, G. Laccetti, M. Lapegna, C. Palmieri, C. Ferraro, V. Pelliccia, C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "On the virtualization of cuda based gpu remoting on arm and x86 machines in the gvirtus framework," *International Journal of Parallel Programming*, pp. 1–22, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10766-016-0462-1
[19] X. Foundation, "Nouveau: Accelerated open source driver for nvidia cards," *URL https://nouveau.freedesktop.org/wiki/*, 2011.
[20] PathScale, "pathscale/pscnv," https://github.com/pathscale/pscnv, 2012.
[21] K. Tian, Y. Dong, and D. Cowperthwaite, "A full gpu virtualization solution with mediated pass-through," in *Proc. USENIX ATC*, 2014.
[22] J. Song, Z. Lv, and K. Tian, "Kvmgt: a full gpu virtualization solution," in *KVM Forum 2014*, 2014.
[23] A. Herrera, "Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation," *Nvidia Corp*, 2014.
[24] 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.
[25] L. Soares and M. Stumm, "Flexsc: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–8.
[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.
[27] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 43–52.
[28] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007, p. 2.
[29] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system." in *USENIX Annual Technical Conference*, 2013, pp. 231–242.
[30] A. G. Greenberg and N. Madras, "How fair is fair queuing," *Journal of the ACM (JACM)*, vol. 39, no. 3, pp. 568–598, 1992.
[31] J. K. Ousterhout, "Scheduling techniques for concurrent systems." in *ICDCS*, vol. 82, 1982, pp. 22–30.
[32] C. Weng, Q. Liu, L. Yu, and M. Li, "Dynamic adaptive scheduling for virtual machines," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing, San Jose*, 2011, pp. 239–250.
[33] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for i/o performance." in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 101–110.
[34] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, no. 184, p. 4, 2009.
[35] K. Menychtas, K. Shen, and M. L. Scott, "Enabling os research by inferring interactions in the black-box gpu stack." in *USENIX Annual Technical Conference*, 2013, pp. 291–296.
[36] C. Margiolas and M. F. O'Boyle, "Portable and transparent software managed scheduling on accelerators for fair resource sharing," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016, pp. 82–93.
[37] H. Zhou, G. Tong, and C. Liu, "Gpes: a preemptive execution system for gpgpu computing," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015, pp. 87–97.
[38] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.
[39] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage." in *OSDI*, vol. 12, 2012, pp. 349–362.

Cheol-Ho Hong is a Research Fellow in the Scalable Computing research center, the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast. He received the BS, MS, and PhD degrees in Computer Science from Korea University, Seoul, Korea. He worked as a Research Professor at Korea University from 2013 to 2015. His current research interests include operating systems, system virtualization, data center networks, and GPU architectures.

Ivor Spence is a Senior Lecturer and Head of Scalable Computing at Queen's University Belfast. He has a BSc in Mathematics and Computer Science and a PhD in Computer Science. He has research experience in the fields of domain specific languages, automated software generation, high performance and distributed computing, architecture description languages and configurable components. He is currently co-investigator on three major EU funded projects and has published more than 50 research papers.

Dimitrios S. Nikolopoulos is a Professor and Head of the School of Electronics, Electrical Engineering and Computer Science, at Queen's University Belfast. His research explores scalable computing systems for data-driven applications and new computing paradigms at the limits of performance, power and reliability. His research has produced over 170 top-tier outputs and has received highly competitive research funding from diverse public and private sectors. He earned a PhD (2000) in Computer Engineering and Informatics from the University of Patras.