GPU Virtualization and Scheduling Methods: A Comprehensive Survey

CHEOL-HO HONG, IVOR SPENCE, and DIMITRIOS S. NIKOLOPOULOS,
Queen's University Belfast

The integration of graphics processing units (GPUs) on high-end compute nodes has established a new accelerator-based heterogeneous computing model, which now permeates high-performance computing. The same paradigm nevertheless has limited adoption in cloud computing or other large-scale distributed computing paradigms. Heterogeneous computing with GPUs can benefit the Cloud by reducing operational costs and improving resource and energy efficiency. However, such a paradigm shift would require effective methods for virtualizing GPUs, as well as other accelerators. In this survey article, we present an extensive and in-depth survey of GPU virtualization techniques and their scheduling methods. We review a wide range of virtualization techniques implemented at the GPU library, driver, and hardware levels. Furthermore, we review GPU scheduling methods that address performance and fairness issues between multiple virtual machines sharing GPUs. We believe that our survey delivers a perspective on the challenges and opportunities for virtualization of heterogeneous computing environments.

CCS Concepts: • **General and reference** → *Surveys and overviews*; • **Networks** → *Cloud computing*; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → *Scheduling;*

Additional Key Words and Phrases: GPU virtualization, GPU scheduling methods, cloud computing, CPU-GPU heterogeneous computing

ACM Reference Format:
Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. 2017. GPU virtualization and scheduling methods: A comprehensive survey. ACM Comput. Surv. 50, 3, Article 35 (June 2017), 37 pages.
DOI: http://dx.doi.org/10.1145/3068281

1. INTRODUCTION

Since the early 2000s, high-performance computing (HPC) programmers and researchers have adopted a new computing paradigm that combines two architectures: namely multi-core processors with powerful and general-purpose cores and many-core accelerators, the leading example of which is graphics processing units (GPUs), with a massive number of simple cores that accelerate algorithms with a high degree of data parallelism. Despite an increasing number of cores, multi-core processor designs still aim at reducing latency in sequential programs by using sophisticated control logic and large cache memories. Conversely, GPUs seek to boost the execution throughput of parallel applications with thousands of simple cores and a high memory bandwidth

This work is supported by the European Commission under the Horizon 2020 program (RAPID project H2020-ICT-644312).
Authors' addresses: C.-H. Hong, I. Spence, and D. S. Nikolopoulos, Data Science and Scalable Computing, School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, University Road, Belfast BT7 1NN, Northern Ireland, United Kingdom; emails: {c.hong, i.spence, d.nikolopoulos}@qub.ac.uk.

DOI: http://dx.doi.org/10.1145/3068281

architecture. Heterogeneous systems combining multi-core processors and GPUs can meet the diverse requirements of a wide range of high-performance computing applications with both control-intensive components and highly data-parallel components. The success of heterogeneous computing systems with GPUs is evident in the latest Top500 list [Top500 2016], where more than 19% of supercomputers adopt both CPUs and GPUs.

Cloud computing platforms can leverage heterogeneous compute nodes to reduce the total cost of ownership and achieve higher performance and energy efficiency [Crago et al. 2011; Schadt et al. 2011]. A cloud with heterogeneous compute nodes would allow users to deploy computationally intensive applications without the need to acquire and maintain large-scale clusters. In addition to this benefit, heterogeneous computing can offer better performance within the same power budget compared to systems based on homogeneous processors, as computational tasks can be placed on either conventional processors or GPUs depending on the degree of parallelism. These combined benefits have been motivating cloud service providers to equip their offerings with GPUs and heterogeneous programming environments [Lee and Katz 2011; Expósito et al. 2013; NVIDIA 2016b]. A number of HPC applications can benefit from execution on heterogeneous cloud environments. These include particle simulation [Green 2010], molecular dynamics simulation [Glaser et al. 2015], and computational finance [Dixon et al. 2014], as well as two-dimensional (2D) and 3D graphics acceleration workloads, which exhibit high efficiency when exploiting GPUs.

System virtualization is a key enabling technology for the Cloud. The virtualization software creates an elastic virtual computing environment, which is essential for improving resource utilization and reducing cost of ownership. Virtualization systems are invariably underpinned by methods of multiplexing system resources. Most of the system resources including processors and peripheral devices can be completely virtualized nowadays and there is ample research in this field dating from the early 1960s [Goldberg 1974]. However, virtualizing GPUs is a relatively new area of study and remains a challenging endeavor. A key barrier to this has been the implementations of GPU drivers, which are not open for modification due to intellectual property protection reasons. Furthermore, GPU architectures are not standardized, and GPU vendors have been offering architectures with vastly different levels of support for virtualization. For these reasons, conventional virtualization techniques are not directly applicable to virtualizing GPUs.

Contributions: In this article, we present an extensive and in-depth survey of GPU virtualization techniques and their scheduling methods. We first review background research related to GPU virtualization techniques in Section 2. We then classify GPU virtualization techniques according to their implementation methods in Section 3. We further compare the different GPU virtualization approaches in Table II of Section 3. We proceed to introduce the most relevant advances in the GPU virtualization literature based on our classification. First, we study methods using API remoting, which virtualizes GPUs at the library level, in Section 4. Next, we study approaches that adopt para and full virtualization techniques in Section 5. These methods enable virtualization at the driver level. Finally, we review hardware-assisted GPU virtualization in Section 6. To address performance and fairness issues between multiple tenants in cloud computing, fair and effective GPU scheduling is essentially required in conjunction with the virtualization techniques. Section 7 provides a classification of GPU scheduling methods in the literature and discusses the detailed scheduling algorithms. Section 8 suggests remaining challenges and future work that advance the state of practice in GPU virtualization. We conclude the article in Section 9.

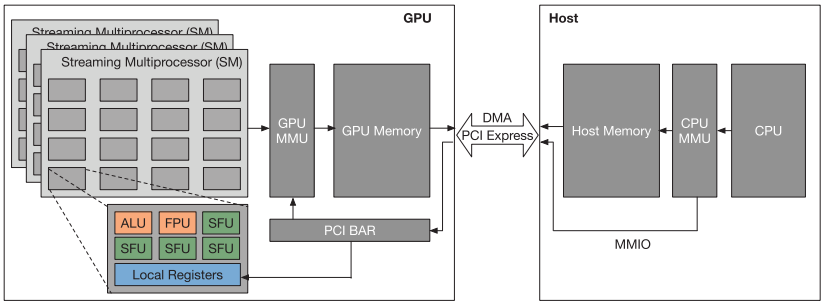

Fig. 1. Architecture of a heterogeneous system equipping a discrete GPU.

Scope of the article: Our survey excludes articles of yet-unimplemented micro-architectural techniques for GPU virtualization but includes articles that analyze the performance of current hardware extensions for GPU virtualization such as NVIDIA GRID [Herrera 2014]. When surveying GPU scheduling methods, we include work performed in a single OS environment, because the same studies can be applied to virtualized environments without significant changes to the underlying GPU virtualization layer.

2. BACKGROUND

We introduce background research related to GPU virtualization. This section explores GPU architectures, GPU application programming interfaces (APIs) and programming models, common GPU benchmarking applications, and the concept of system virtualization, all of which cover basic knowledge from hardware to applications in the GPU virtualization stack.

2.1. GPU Architecture

GPUs adopt a fundamentally different design for executing parallel applications compared to conventional multi-core processors [Kirk and Wen-mei 2012]. GPUs are based on a throughput-oriented design and offer thousands of simple cores and a high bandwidth memory architecture. This design enables maximizing the execution throughput of applications with a high degree of data parallelism, which are expected to be decomposable into a large number of threads operating on different points in the program data space. In this design, when some threads are waiting for the completion of arithmetic operations or memory accesses with long latency, other threads can be scheduled by the hardware scheduler to hide the latency [Patterson 2009]. This mechanism may lengthen the respective execution time of individual threads but improve total execution throughput. On the contrary, the design of conventional processors is optimized for reducing the execution time of sequential code on each core, thus adding complexity to each core at the cost of offering fewer cores in the processor package. Conventional processors typically use sophisticated control logic and large cache memories to efficiently deal with conditional branches, pipeline stalls, and poor data locality. Modern GPUs also handle complex control flows, have large SRAM-based local memories, and adopt some additional features of conventional processors but preserve the fundamental properties of offering a higher degree of thread-level parallelism and higher memory bandwidth.

Figure 1 shows the architecture of a traditional heterogeneous system equipping a discrete GPU. The GPU part is based on the Fermi architecture of NVIDIA [Wittenbrink et al. 2011] but is not limited to NVIDIA architectures, as recent GPUs adopt a similar

high-level design. A GPU has several streaming multiprocessors (SMs), each of which has 32 computing cores. Each SM also has an L1 data cache and a low latency shared memory. Each core has local registers, an integer arithmetic logic unit (ALU), a floating point unit (FPU), and several special function units (SFUs) that execute transcendental instructions such as sine and cosine operations. A GPU memory management unit (MMU) provides virtual address spaces for GPU applications. A GPU memory reference by an application is resolved into a physical address by the MMU using the application's own page table. Memory accesses from each application therefore cannot refer to other applications' address spaces.

The host connects the discrete GPU using the PCI Express (PCIe) interface. The CPU in the host interacts with the GPU via memory mapped input/output (MMIO). The GPU registers and device memory can be accessed by the CPU through the MMIO interface. The MMIO region is configured at boot time based on the PCI base address registers (BARs), which are memory windows that can be used by the host for communication. GPU operations issued by an application are submitted into a ring buffer associated with the application's command submission channel, which is a GPU hardware unit and visible to the CPU via MMIO. Large data can be transferred between the host memory and the GPU device memory by the direct memory access (DMA) engine.

The discrete GPU architecture shown in Figure 1 can cause data transfer overhead over the PCIe interface, because the maximum bandwidth that current PCIe can offer is low (i.e., 16GB/s) compared to the internal memory bandwidth of the GPU (i.e., hundreds of GB/s). Furthermore, the architecture incurs large programming effort to manage data manipulated by both the CPU and the GPU. To address these issues, GPUs have been integrated into the CPU chip. Intel's GPU architecture [Hammarlund et al. 2014] and AMD's HSA architecture [Kyriazis 2012] integrate the two processors on the same bus with shared system memory. These architectures enable a unified virtual address space and eliminate data copying between the devices. They can also reduce a programmer's burden to manage the separate data address spaces.

2.2. GPU APIs and Programming Models

We introduce OpenGL, Direct3D, CUDA, and OpenCL as GPU APIs and programming models, because the API remoting and other GPU virtualization approaches have mainly focused on virtualizing the aforementioned libraries and models.

OpenGL [Woo et al. 1999] is a library for accessing GPU hardware to accelerate graphics. The library is specialized for implementing video games, image processing, and visualization tasks for scientific applications. OpenGL provides a hardware-independent application programming interface (API) implemented on different graphics cards, regardless of their underlying system software.

Direct3D [Blythe 2006] is a proprietary graphics API for Microsoft Windows. Direct3D is a low-level API used to render 3D graphics for performance intensive applications such as games. Direct3D provides a coherent and general abstraction in front of specific GPU hardware implementations, exposing advanced graphics capabilities such as Z-buffering, W-buffering, stencil buffering, and spatial anti-aliasing.

CUDA [Nvidia 2007b] is a programming model developed by NVIDIA for parallel computing platforms. It allows software developers to exploit CUDA-enabled GPUs to perform general-purpose data parallel computation, thus converting graphics cards into general-purpose graphics processing units (GPGPU). CUDA is tightly coupled with programming languages such as C and C++ and extends these languages with a small set of primitives for device memory allocation, data transfer, GPU kernel execution, event handling, and atomic and synchronization operations.

OpenCL [Group et al. 2008] is a framework for parallel applications that execute on heterogeneous platforms. OpenCL specifies a C-like programming language called OpenCL C for writing compute kernels. It also defines APIs to launch kernels into an OpenCL device and to manage memory transfer between the host and the device. The key difference between CUDA and OpenCL is that CUDA can be run only on NVIDIA GPUs, but OpenCL applications can be executed on both CPUs and accelerators regardless of the manufactures.

2.3. GPU Applications

GPU programs are categorized into graphics acceleration and general-purpose computing workloads. The former category includes 2D and 3D graphics workloads. The latter includes a wide range of general-purpose data parallel computations.

Graphics acceleration workloads: 3DMark [Futuremark 1998] is a GPU benchmark test application developed by Futuremark Corporation for measuring the performance of 3D graphics rendering capabilities. 3DMark evaluates various Direct3D features including tessellation, compute shaders, and multi-threading.

The Phoronix Test Suite (PTS) [Larabel and Tippett 2011] is a set of open source benchmark applications developed by Phoronix Media. Phoronix performs comprehensive evaluation for measuring the performance of computing systems. GPU software developers usually utilize Phoronix for testing the performance of OpenGL games such as Doom 3, Nexuiz, and Enemy Territory.

General-purpose computing workloads: Rodinia [Che et al. 2009] is a benchmark suite focusing on the performance evaluation of compute-intensive applications implemented by CUDA, OpenMP, and OpenCL. Each application or kernel covers different types of behavior of compute-intensive applications, and the suite broadly covers the features of the Berkeley Seven Dwarfs [Asanovic et al. 2006].

The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite [Danalis et al. 2010] is a set of benchmark programs evaluating the performance and stability of GPGPU computing systems using CUDA and OpenCL applications. The suite supports the evaluation of both cluster-level parallelism with the Message Passing Interface (MPI) [Gropp et al. 1996] and node-level parallelism using multiple GPUs in a single node. The application scope of SHOC includes the Fast Fourier Transform (FFT), linear algebra, and molecular dynamics among others.

Parboil [Stratton et al. 2012] is a collection of compute-intensive applications implemented by CUDA, OpenMP, and OpenCL to measure the throughput of CPU and GPU architectures. Parboil provides collected benchmark applications from diverse scientific and commercial fields. They include bio-molecular simulation, fluid dynamics, image processing, and astronomy.

The CUDA SDK benchmark suite [Nvidia 2007a] is released as a part of CUDA Toolkit. It covers a diverse range of GPGPU applications performing data-parallel algorithms used in linear algebra operations, computational fluid dynamics (CFD), image convolution, and Black-Scholes & binomial option pricing.

2.4. System Virtualization

System virtualization allows several operating systems (OSs) to run simultaneously on a single physical machine, thus achieving effective sharing of system resources in personal and shared (e.g., cloud) computing platforms. The software for system virtualization includes a hypervisor, also known as a virtual machine monitor (VMM), and virtual machines (VMs). A hypervisor virtualizes physical resources in the system such as the CPU, memory, and I/O devices. A VM is composed of these virtualized

resources and is provided to a guest OS. The guest OS can run on the VM as though the VM were a real physical machine. Popular hypervisors used widely for personal and cloud computing include VMware ESXi [Chaubal 2008], KVM [Kivity et al. 2007], Hyper-V [Velte and Velte 2009], and Xen [Barham et al. 2003].

System virtualization can be categorized into three major classes: full, para, and hardware-supported virtualization. Full virtualization completely emulates the CPU, memory, and I/O devices to provide a guest OS with an environment identical to the underlying hardware. Privileged instructions of a guest OS that modify the system state are trapped into the hypervisor by a binary translation technique that automatically inserts trapping operations in the binary code of the guest OS. The advantage of this approach is that guest OSs run in the virtualization environment without modification. However, full virtualization usually exhibits high-performance penalties due to the cost for emulation of the underlying hardware.

Para virtualization addresses the performance limitations of full system virtualization by modifying the guest OS code to support more efficient virtualization. Privileged instructions of a guest OS are replaced with hypercalls, which provide a communication channel between the guest OS and the hypervisor. This optimization eliminates the need for binary translation. Para virtualization offers a guest OS an environment similar but not identical to the underlying hardware. The advantage of this approach is that it has lower virtualization overhead than full virtualization. The limitation is that it requires modification to guest OSs, which can be tedious when new versions of an OS kernel or device driver are released.

Hardware-supported virtualization requires hardware capabilities such as Intel VT-x [Uhlig et al. 2005] to trap privileged instructions from guest OSs. These capabilities typically introduce two operating modes for virtualization: guest (for an OS) and root (for the hypervisor). When a guest OS executes a privileged instruction, the processor intervenes and transfers the control to the hypervisor executing in the root mode. The hypervisor then emulates the privileged instruction and returns the control to guest mode. The mode change from guest to root is called a VM Exit. The reverse action is called a VM Entry. The advantage of this approach is that it does not have to modify a guest OS and that it exhibits higher performance than full virtualization.

3. CLASSIFICATION OF GPU VIRTUALIZATION TECHNIQUES

Table I classifies GPU virtualization techniques in terms of their implementation. We classify the techniques based on three approaches:

- API remoting: This approach virtualizes GPUs at a higher level in the GPU execution stack. As GPU vendors do not provide the source code of their GPU drivers, it is difficult to virtualize GPUs at the driver level; other devices such as disks are often virtualized at this level. To address this issue, API remoting provides a GPU wrapper library to a guest OS to intercept GPU calls. The intercepted calls are forwarded to the host OS or a remote machine with GPUs. The requests are processed remotely and the results are returned to the guest OS. This approach can overcome the limitation that black-box GPU drivers incur by virtualizing GPUs at the library level.
- Para & full virtualization: Para and full virtualization offer GPU virtualization at the driver level in the GPU stack. Recently, architecture documentation has been made available for some GPU models by vendors [AMD 2009] or via reverse engineering [X.OrgFoundation 2011; PathScale 2012; Menychtas et al. 2013]. This approach uses a custom GPU driver based on the available documentation to realize GPU virtualization at the driver level. Para virtualization slightly modifies the custom driver in the guest for delivering sensitive operations directly to the host

Table I. Classification of GPU Virtualization Techniques Based on the Implementation Manners

Classification	References
API remoting	[Becchi et al. 2012; Castelló et al. 2015; Duato et al. 2009, 2010a, 2010b, 2011, 2011; Giunta et al. 2010, 2011; Gupta et al. 2009; Gupta et al. 2011; Hansen 2007; Humphreys et al. 2002; Jang et al. 2013; Kato et al. 2012; Kuzkin and Tormasov 2011; Laccetti et al. 2013; Lagar-Cavilla et al. 2007; Lama et al. 2013; Lee et al. 2016; Li et al. 2011, 2012; Liang and Chang 2011; Merritt et al. 2011; Montella et al. 2011, 2014, 2016a, 2016b; Niederauer et al. 2003; Oikawa et al. 2012; Peña et al. 2014; Pérez et al. 2016; Prades et al. 2016; Qi et al. 2014; Ravi et al. 2011; Reaño et al. 2012; 2013; ,2015a, 2015b; Reaño and Silla 2015; Rossbach et al. 2011; Sengupta et al. 2013, 2014; Shi et al. 2009, 2011, 2012; Tien and You 2014; Vinaya et al. 2012; Xiao et al. 2012; You et al. 2015; Zhang et al. 2016]
Para & full virtualization	[Dalton et al. 2009; Dong et al. 2015; Dowty and Sugerman 2009; Gottschlag et al. 2013; Guan et al. 2015; Huang et al. 2016; Qi et al. 2014; Shan et al. 2013; Song et al. 2014; Suzuki et al. 2014, 2016; Tian et al. 2014; Wang et al. 2016; Xue et al. 2016; Zhang et al. 2014]
Hardware virtualization	[Abramson et al. 2006; Amazon 2010; Expósito et al. 2013; Herrera 2014; Hong et al. 2014; Jo et al. 2013a, 2013b; Ou et al. 2012; Shainer et al. 2011; Shea and Liu 2013; Vu et al. 2014; Walters et al. 2014; Yang et al. 2012a, 2012b, 2014; Yeh et al. 2013; Younge and Fox 2014; Younge et al. 2014, 2015]

driver for improving performance, whereas full virtualization does not require this modification because it fully emulates GPUs.
- Hardware-supported virtualization: In this approach, a guest OS is given direct access to GPUs with hardware extension features provided by either motherboard chipset or GPU manufacturers. This GPU pass-through access is enabled by remapping DMAs and interrupts to each guest OS. Intel VT-d [Abramson et al. 2006] and AMD-Vi [Van Doorn 2006] chipsets support this mechanism, but they cannot support sharing of a single GPU between multiple guest OSs. Recently, NVIDIA GRID [Herrera 2014] addressed this limitation and allows multiplexing in recent NVIDIA GPUs targeting cloud environments.

Table II shows a comparison of representative GPU virtualization solutions in the literature. We identify the techniques based on the following factors:

- Category: This denotes which implementation method the solution adopts among API remoting, para & full virtualization, and hardware-supported virtualization.
- Acceleration target: This part indicates the target of acceleration of the solution between graphics acceleration and GPGPU computing.
- Hypervisor: This indicates on which hypervisor the solution is building for GPU virtualization.
- Remote acceleration: Some solutions offload GPU tasks to a remote machine to implement a virtual GPU device. This indicates whether the solution supports remote offloading.
- Programming model: This property refers to the GPU programming language and model that the solution supports.
- Open source: This property indicates whether the solution adopts an open source policy.
- GPU hardware: This property indicates the GPU hardware where the solution is evaluated or the hardware architecture that the solution can support.
- Multiplexing: This indicates whether the solution can support sharing of a single GPU between multiple VMs.

Table II. Comparison of GPU Virtualization Techniques

Name	Category	Acceleration target	Hypervisor					Remote acceleration	Programming model				Open source	GPU hardware (Architecture)				Multiplexing	GPU scheduling
			KVM	Xen	VMware	Parallels	Others		OpenGL	Direct3D	CUDA	OpenCL		NVIDIA	AMD	Intel	Others		
Chromium [Humphreys et al. 2002]	API remoting	Graphics						O	O					GeForce 3 (Kelvin)				O	
VMGL [Lagar-Cavilla et al. 2007]	API remoting	Graphics		O	O				O				O		Radeon X600			O	
Blink [Hansen 2007]	API remoting	Graphics		O					O						Radeon 9600SE			O	
Parallels Desktop [Kuzkin and Tormasov 2011]	API remoting	Graphics				O		O	O	O								O	
VADI [Lee et al. 2016]	API remoting	Graphics					SASP		O								Vivante GC2000	O	
GViM [Gupta et al. 2009]	API remoting	GPGPU		O							O			GeForce 8800, 9800 GTX (Tesla)				O	O
vCUDA (1*) [Shi et al. 2009]	API remoting	GPGPU	O	O							O			GeForce 8600 GT (Tesla)				O	
rCUDA [Duato et al. 2010b]	API remoting	GPGPU						O			O		O (2*)	Tesla C1060 (3*) (Tesla 2.0)				O	
GVirtuS [Giunta et al. 2010]	API remoting	GPGPU	O	O	O			O			O		O	Tesla C1060 (3*) (Tesla 2.0)				O	
GVM [Li et al. 2011]	API remoting	GPGPU					Proprietary hypervisor				O			Tesla C2070 (Fermi)				O	
Pegasus [Gupta et al. 2011]	API remoting	GPGPU		O							O			GeForce 9800 GTX (Tesla)				O	O
Shadowfax [Merritt et al. 2011]	API remoting	GPGPU		O				O			O			GeForce 8800, 9800 GTX (Tesla)				O	O
VOCL [Xiao et al. 2012]	API remoting	GPGPU						O				O		Tesla M2070 (Fermi)				O	
DS-CUDA [Oikawa et al. 2012]	API remoting	GPGPU						O			O		O	GeForce GTX 560 Ti (Fermi)				O	
VMware SVGA II [Dowty and Sugerman 2009]	Para virtualization	Graphics			O				O	O					Radeon HD2600 (TeraScale)			O	
LoGV [Gottschlag et al. 2013]	Para virtualization	GPGPU	O								O	O		GeForce GTX 480 (Fermi)				O	
VGRIS [Qi et al. 2014]	Para virtualization	Graphics			O					O					Radeon HD6750 (TeraScale 2)			O	O
[Huang et al. 2016]	Para virtualization	GPGPU	O									O			Radeon R7 (GCN 1.1)			O	
GPUvm [Suzuki et al. 2014]	Full and para virtualization	GPGPU		O							O		O	Quadro 6000 (Fermi)				O	O
gVirt [Tian et al. 2014]	Full virtualization	Graphics		O					O	O		O	O			HD Graphics 4600 (4*) (4th generation)		O	O
KVMGT [Song et al. 2014]	Full virtualization	Graphics	O						O	O		O	O			HD Graphics 4600 (4*) (4th generation)		O	O
gHyvi [Dong et al. 2015]	Full virtualization	Graphics		O					O	O		O	O			HD Graphics 4600 (4*) (4th generation)		O	O
gScale [Xue et al. 2016]	Full virtualization	Graphics		O					O	O		O	O			HD Graphics P4700 (4*) (4th generation)		O	O
GPU pass-through [Abramson et al. 2006]	Hardware virtualization	Graphics and GPGPU	O	O	O	O			O	O	O	O		GPUs that support GPU pass-through					
vmCUDA [Vu et al. 2014]	Hardware virtualization with API remoting	GPGPU			O						O			Quadro 4000 (Fermi)				O	
NVIDIA GRID [Herrera 2014]	Hardware virtualization	Graphics and GPGPU	O	O	O	O			O	O	O	O		NVIDIA GPUs that support NVIDIA GRID				O	

(1*) The final version of vCUDA only supports KVM.
(2*) Currently, rCUDA is distributed in a binary format.
(3*) Current GVirtuS and rCUDA can support all NVIDIA GPUs beyond the Tesla architecture.
(4*) Current gVirt, KVMGT, gHyvi, and gScale can support all Intel GPUs beyond the 4th generation.

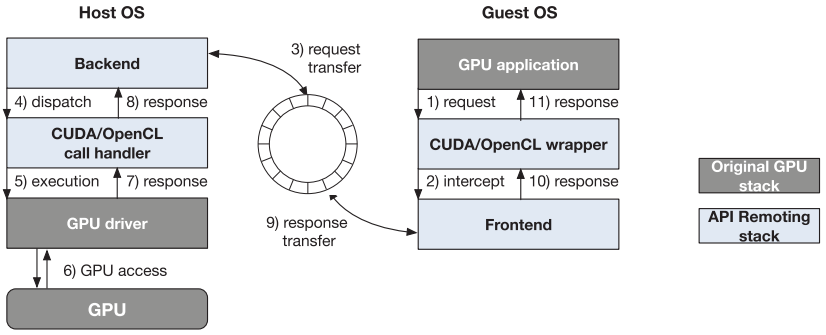

Fig. 2. Architecture of the API remoting approach.

- GPU scheduling: This indicates whether the solution provides GPU scheduling for fair or SLA-based sharing on GPUs. Details about GPU scheduling are discussed in Section 7.

The introduced solutions will be discussed in depth in the following sections.

4. API REMOTING

Virtualizing GPUs has been regarded as more difficult than virtualizing I/O devices such as network cards or disks. Several reasons add complexity to multiplexing and sharing GPU resources between VMs. First, GPU vendors tend not to reveal the source code and implementation details of their GPU drivers for commercial reasons. Such technical specifications are essential for virtualizing GPUs at the driver level. Second, even when driver implementations are unveiled, for example, by reverse engineering methods [X.OrgFoundation 2011; Menychtas et al. 2014], GPU vendors still introduce significant changes with every new generation of GPUs to improve performance. As a consequence, specifications revealed by reverse engineering become unusable. Finally, some OS vendors provide proprietary GPU drivers for virtualization, but the proprietary drivers cannot be used across all OSs. In summary, there are no standard interfaces for accessing GPUs, which are required for virtualizing these devices.

The API remoting approach overcomes the aforementioned limitations and is now the most prevalent approach to GPU virtualization. The premise of API remoting is to provide a guest OS with a wrapper library that has the same API as the original GPU library. The wrapper library intercepts GPU calls (e.g., OpenGL, Direct3D, CUDA, and OpenCL calls) from an application before the calls reach the GPU driver in the guest OS. The intercepted calls are redirected to the host OS in the same machine through shared memory or a remote machine with available GPUs. The redirected calls are processed remotely and only the results are delivered to the application through the wrapper library. The API remoting approach can emulate a GPU execution environment without exposing physical GPU devices in the guest OS.

Figure 2 illustrates an example of a system that adopts the API remoting approach, which forwards GPU calls in the guest to the host in the same machine. The architecture adopts a split device model where the frontend and backend drivers are placed in the guest and host OSs, respectively. The wrapper library installed in the guest OS intercepts a GPU call from the application and delivers it to the frontend driver. The frontend packs the GPU operation with its parameters into a transferable message and sends the message to the backend in the host OS via shared memory. In the host OS, the backend driver parses the message and converts it into the original GPU call.

The call handler executes the requested operation on the GPU through the GPU driver. The call handler returns the result back to the application via the reverse path.

The key advantage of this approach is that it can support applications using GPUs without recompilation in most cases. The wrapper library can be dynamically linked to existing applications at runtime. In addition, it incurs negligible virtualization overhead as the virtualization architecture is simple and bypasses the hypervisor layer [Gupta et al. 2011]. Finally, as the virtualization layer is usually implemented in user space, this approach can be agnostic on underlying hypervisors [Giunta et al. 2010], specifically if it does not use hypervisor-specific inter-VM communication methods. The limitation is that keeping the wrapper libraries updated can be a daunting task as new functions are gradually added to vendor GPU libraries [Menychtas et al. 2014]. In addition, as GPU requests bypass the hypervisor, it is difficult to implement basic virtualization features such as execution checkpointing, live migration, and fault tolerance [Dowty and Sugerman 2009].

4.1. Methods for Graphics Acceleration

Chromium [Humphreys et al. 2002] is an early example of API remoting. In the past, graphics processors could not be fully utilized by a number of applications in the same machine because the hosts were using slow serial interfaces to the graphic cards. The goal of Chromium is to aggregate GPU calls from different machines and to process them in a powerful cluster rendering system with multiple graphics accelerators. For this purpose, Chromium provides four OpenGL wrapper libraries that encapsulate frequently used operations: stream packing, stream unpacking, point-to-point connection-based networking abstractions, and complete OpenGL state tracker libraries. These libraries intercept OpenGL operations and transfer them to a rendering cluster.

VMGL [Lagar-Cavilla et al. 2007] implements the API remoting approach for accelerating OpenGL applications in recent hypervisors including Xen and VMware. It provides hardware accelerated rendering abilities to OpenGL applications in each VM. VMGL consists of the following three modules: the VMGL library, the VMGL stub, and the VMGL X server extension. The VMGL library is an OpenGL wrapper library that replaces standard implementations. The VMGL stub is created in the host for each VMGL library instance to receive and process GPU requests from the library. OpenGL commands are delivered by a network transport, which makes VMGL agnostic of underlying hypervisors. The VMGL X server extension runs in the guest OS side and is used to register the size and visibility of OpenGL-enabled windows. Through the registered information, the VMGL stub only processes a region that can be visible in the guest OS's desktop. VMGL additionally supports suspend and resume functionalities by keeping track of the entire OpenGL state in the guest and restoring the state in a new stub.

Blink [Hansen 2007] offers accelerated OpenGL rendering abilities to applications inside a VM similarly to VMGL but focuses more on performance optimization. Blink provides the BlinkGL wrapper library for guest OSs, which is a superset of OpenGL. The wrapper library provides stored procedures, each of which is a sequence of serialized BlinkGL commands to eliminate the performance overhead of additionally (de-)serializing GL command streams during communication between the guest and the host. The Blink Server in the host interprets the transferred stored procedure by using a Just-In-Time (JIT) compiler. The host and the guest OSs communicate with each other through shared memory to reduce the overhead of using a network transport on large texture or frame buffer objects.

The Parallels Desktop [Kuzkin and Tormasov 2011] offers a proprietary GPU driver for guest OSs to offload their OpenGL and Direct3D operations onto remote devices with

GPUs. The proprietary GPU driver can be installed only on Parallels products such as Parallels Desktop and Parallels Workstation. The server module in the remote device receives access requests from a number of remote VMs and chooses a next VM to use the GPU. The module then sends a token to the selected guest and allows it to occupy the GPU for a specific time interval. The guest OS and the remote OS use a remote procedure call (RPC) protocol for delivering GPU commands and the corresponding results.

VADI [Lee et al. 2016] implements GPU virtualization for vehicles by multiplexing a GPU device used by the digital cluster of a car. VADI works on a proprietary hypervisor for vehicles called the Secure Automotive Software Platform (SASP) [Kim et al. 2013]. This hypervisor exploits the ARM TrustZone technology [Winter 2008], which can accommodate two guest OSs in the secure and normal "worlds", respectively. VADI implements the GL wrapper library and the V-Bridge-normal in the normal world and the GL stub and the V-Bridge-secure in the secure one. GPU commands executed in the normal world are intercepted by the wrapper library and processed in the secure world by the GL stub. Each V-Bridge is connected by shared memory and is responsible for communication between the two worlds.

4.2. Methods for GPGPU Computing

Following NVIDIA's launch of CUDA in 2006, general-purpose computing on GPUs (GPGPU) became more popular and practical. NVIDIA's CUDA conceals the underlying graphics hardware architecture from developers and allows programmers to write scalable programs without learning new programming languages. Research on GPU virtualization has focused more on GPGPUs since the introduction of CUDA to accelerate compute-intensive applications running in the Cloud.

GViM [Gupta et al. 2009] implements GPU virtualization at the CUDA API level in the Xen hypervisor [Barham et al. 2003]. To enable a guest VM to access the GPU located in the host, GViM implements the Interposer CUDA Library for guest OSs, and the frontend and backend drivers for communication between the host and the guest. GViM focuses on efficient sharing of large data volumes between the guest and the host when a GPU application is data intensive. For this purpose, GViM furnishes shared memory allocated by Xenstore [Cho and Jeon 2007] between the frontend and backend, instead of using a network transport. It further develops the one-copy mechanism that maps the shared memory into the address space of the GPU application. This removes data copying from user space to kernel space in the guest OS and improves communication performance.

vCUDA [Shi et al. 2009] also implements GPU virtualization in the Xen hypervisor. vCUDA provides a CUDA wrapper library and virtual GPUs (vGPUs) in the guest and the vCUDA stub in the host. The wrapper library intercepts and redirects API calls from the guest to the host. vGPUs are created per application by the wrapper library and give a complete view of the underlying GPUs to applications. The vCUDA stub creates an execution context for each guest OS and executes remote GPU requests. For communication between VMs, vCUDA adopts XML-RPC [Cerami 2002], which supports high-level communication between the guest and the host. In the latest version [Shi et al. 2012], vCUDA is ported to KVM using VMRPC [Chen et al. 2010] with VMCHANNEL [Patni et al. 2015]. VMRPC utilizes a shared memory zone between the host OS and the guest OS to reduce the overhead of using XML-RPC transmission with TCP/IP. VMCHANNEL enables an asynchronous notification mechanism in KVM to reduce the latency of inter-VM communication. vCUDA also develops Lazy RPC that performs batching specific CUDA calls that can be delayed (e.g., *cudaConfigureCall()*). This prevents frequent context switching between the guest OS and the hypervisor occurred by repeated RPCs and improves communication performance.

rCUDA [Duato et al. 2010b] focuses on remote GPU-based acceleration, which offloads CUDA computation parts onto GPUs located in a remote host. rCUDA recognizes that prior virtualization research based on emulating local devices is not appropriate for HPC applications because of unacceptable virtualization overhead. Instead of device emulation, rCUDA implements virtual CUDA-compatible devices by adopting remote GPU-based acceleration without the hypervisor layer. More concretely, rCUDA provides a CUDA API wrapper library to the client side, which intercepts and forwards GPU calls from the client to the GPU server, and the server daemon in the server side, which receives and executes the remote GPU calls. The client and the server communicate with each other using a TCP/IP socket. rCUDA points out network performance bottlenecks when several clients concurrently access the remote GPU cluster. To overcome this issue, rCUDA provides a customized application-level communication protocol [Duato et al. 2010a]. Current rCUDA supports EDR 100G InfiniBand using Mellanox adapters for providing higher network bandwidth [Reaño et al. 2015b].

GVirtuS [Giunta et al. 2010] implements a CUDA wrapper library, the frontend and backend drivers, and communicators supporting various hypervisors including KVM, Xen, and VMware. The frontend and backend drivers are placed in the guest and the host, respectively. The two drivers communicate with each other by a communicator specific to each hypervisor. GVirtuS identifies that the performance of GPU virtualization depends on communication throughput between the frontend and the backend. To address this issue, GVirtuS implements pluggable communication components that utilize high performance communication channels provided by the hypervisors. The communicators for KVM, Xen, and VMware employ VMSocket, XenLoop [Wang et al. 2008], and the VMware Communicator Interface (VMCI) [Gebhardt and Tomlinson 2010] as communication channels, respectively. In the latest version, the VMShm communicator [Di Lauro et al. 2012], which leverages shared memory, was introduced for better communication performance. It allocates a POSIX shared memory chunk on the host OS and allows both the backend and frontend to map the memory for communication. GVirtuS also provides a TCP/IP-based communicator for remote GPU-based acceleration.

GVM [Li et al. 2011] is based on a model that predicts the performance of GPU applications. GVM validates this model by introducing its own virtualization infrastructure, which consists of the user process APIs, the GPU Virtualization Manager (GVM), and the virtual shared memory. The user process APIs expose virtual GPU resources to programmers. The source code then needs to be modified to contain the APIs to utilize the virtual GPUs. GVM runs in the host OS and is responsible for initializing the virtual GPUs, receiving requests from guest OSs, and passing them to physical GPUs. The virtual shared memory is implemented as POSIX shared memory by which the guest and host OSs can communicate with each other.

Pegasus [Gupta et al. 2011] advances its predecessor, GViM [Gupta et al. 2009], by managing virtualized accelerators as first class schedulable and shareable entities. For this purpose, Pegasus introduces the notion of an accelerator virtual CPU (aVCPU), which embodies the state of a VM executing GPU calls on accelerators, similarly to the concept of virtual CPUs (VCPUs). In Pegasus, an aVCPU is a basic schedulable entity and consists of a shared call buffer per domain, a polling thread in the host OS, and the CUDA runtime context. GPU requests from a guest OS are stored in the call buffer shared between the frontend and backend drivers. A polling thread selected by the GPU scheduler then fetches the GPU requests from the buffer and passes them to the actual CUDA runtime in the host OS. The scheduling methods Pegasus adopts will be introduced in Section 7.2.2.

Shadowfax [Merritt et al. 2011] enhances its predecessor, Pegasus [Gupta et al. 2011]. Shadowfax tackles the problem that under Pegasus applications requiring significant

GPU computational power are limited to using only local GPUs, although remote nodes may boast additional GPUs. To address this issue, Shadowfax presents the concept of GPGPU assemblies, which can configure diverse virtual platforms based on application demands. This virtual platform concept allows applications to run across node boundaries. For local GPU execution, Shadowfax adopts the GPU virtualization architecture of Pegasus. For remote execution, Shadowfax implements a remote server thread that creates a fake guest VM environment, which consists of a call buffer and a polling thread per VM in the remote machine. To reduce remote execution overhead, Shadowfax additionally does batching of GPU calls and their data.

VOCL [Xiao et al. 2012] presents a GPU virtualization solution for OpenCL applications. Similarly to rCUDA [Duato et al. 2010b], VOCL adopts remote GPU-based acceleration to provide virtual devices supporting OpenCL. VOCL provides an OpenCL wrapper library on the client side and a VOCL proxy process on the server side. The proxy process receives inputs from the library and executes them on remote GPUs. The wrapper library and the proxy process communicate via MPI [Gropp et al. 1996]. The authors claim that MPI can provide a rich communication interface and dynamically establish communication channels compared to other transport methods.

DS-CUDA [Oikawa et al. 2012] provides a remote GPU virtualization platform similarly to rCUDA [Duato et al. 2010b]. It is composed of a compiler, which translates CUDA API calls to respective wrapper functions, and a server, which receives GPU calls and their data via an InfiniBand IBverb or RPC socket. Compared to other similar solutions, DS-CUDA implements redundant calculations to improve reliability where two different GPUs in the cluster perform the same calculation to ensure that the result from the cluster is correct.

5. PARA AND FULL VIRTUALIZATION

The API remoting approach can virtualize GPUs without significant performance penalty by intercepting and emulating GPU requests with GPU wrapper libraries for graphics acceleration and GPGPU computing. However, the wrapper libraries should be updated when new functions are added to vendor GPU libraries, which can be a daunting task. This means that many API remoting solutions cannot be used currently on new graphics hardware and the most recent GPU libraries. To overcome this limitation, full and para virtualization methods where GPUs are virtualized at the driver level are introduced. Full virtualization uses unmodified GPU drivers, whereas para virtualization makes adjustments to GPU drivers for performance improvement. Recently, AMD has released open documentation regarding their GPU architectures for certain models [AMD 2009]. Furthermore, some developers have inferred the interfaces between NVIDIA GPUs and the host by using reverse engineering techniques [X.OrgFoundation 2011; PathScale 2012; Menychtas et al. 2013]. Due to these efforts, custom GPU drivers that can control AMD or NVIDIA GPUs have been released. This movement motivated a number of research on para and full virtualization techniques.

Figure 3 illustrates an example of a system adopting the full or para virtualization approach. The architecture adopts an unmodified or modified custom GPU driver in the guest. GPU requests coming from the guest driver are delivered to the QEMU GPU device in the host through shared memory in the hypervisor. The QEMU device emulates the underlying GPU hardware while exposing MMIO PCIe BARs (explained in Section 2.1) to the guest driver; the guest driver therefore regards the QEMU device as a real GPU. The vGPU control block keeps the state of each virtual GPU and maintains a queue to buffer GPU requests issued from the QEMU device. The GPU scheduler selects a next virtual GPU to run and fetches GPU requests from the associated queue. The GPU requests are then processed in the GPU, and the results are returned to the application through the reverse path.

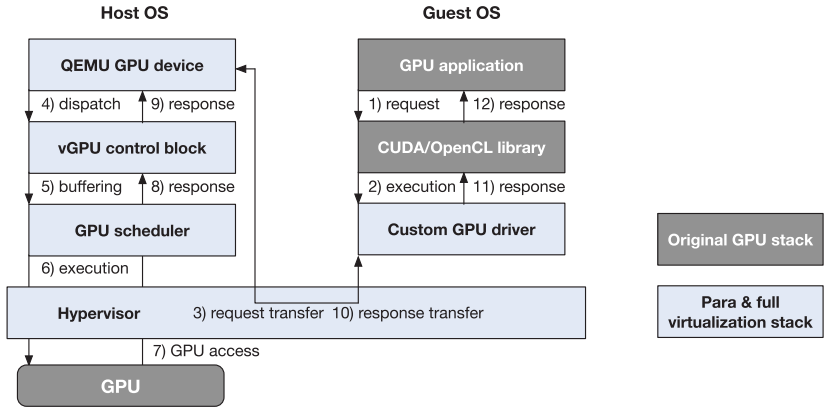

Fig. 3. Architecture of the full or para virtualization approach.

The advantage of this approach is that it can reuse existing GPU libraries and is prepared to cope with future library changes because the implementation point for GPU virtualization is relocated to the lower driver layer in the guest OS. In addition, as GPU requests from a guest OS can be monitored and mediated by the hypervisor, essential virtualization features such as live migration can be implemented without difficulty compared to the API remoting approach. The disadvantage is that this approach is heavily dependent on custom GPU drivers, which in turn rely on reverse engineering or open documentation. The release of a new GPU microarchitecture may impose significant burdens on the development of its corresponding custom driver.

5.1. Para Virtualization

VMware SVGA II [Dowty and Sugerman 2009] is a GPU virtualization approach provided by VMware's hosted products including VMware Workstation and VMware Fusion. The authors point out that API remoting is straightforward to implement but completely surrenders interposition that allows the hypervisor to arbitrate hardware access between a VM and the physical hardware. This makes it difficult for the hypervisor to implement basic virtualization features such as suspend-to-disk and live migration. VMware provides the VMware SVGA Driver built based on the open documentation of AMD GPUs [AMD 2009]. This driver replaces the original GPU driver in the guest. The SVGA Driver is given access to a virtual GPU called VMware SVGA II created by the hypervisor. This is not a physical graphics card but acts like a physical one by providing three virtual hardware resources: registers, Guest Memory Regions (GMRs), and a FIFO command queue. The registers are used for hardware management such as mode switching and IRQ acknowledgment. The GMRs emulate physical GPU VRAM and are allocated in the host memory. The FIFO command queue, which adopts a lock-free data structure to eliminate synchronization overhead, receives GPU commands from the guest. The backend in the host, called Mouse-Keyboard-Screen (MKS), then fetches and issues the GPU requests asynchronously from the FIFO queue. VMware SVGA II focuses on supporting graphics acceleration rather than GPGPU computing.

LoGV [Gottschlag et al. 2013] implements para virtualization in KVM using a modified PathScale GPU driver (pscnv) [PathScale 2012] in the guest OS. This is an open source driver implemented based on reverse engineering of NVIDIA drivers. The key point of LoGV's virtualization is to partition the GPU device memory into several pieces and to grant a guest OS direct access to its own portion. Modern GPUs have their own memory management unit (MMU) to map the partitioned GPU memory into

the GPU application's address space. By managing and configuring each GPU page table referred to by the GPU MMU, LoGV allows each VM to access the mapped region without involvement from the hypervisor. LoGV only mediates memory allocation or mapping operations to ensure that any VM does not establish mapping to other VMs' address spaces. The GPU driver in the guest OS is modified for this purpose to send these sensitive operations to the hypervisor. The virtual device in the hypervisor is then responsible for validation of these requests. After any necessary checks, the virtual device delivers the allocation and mapping requests to the GPU driver in the host, which performs the actual allocation. A command submission channel is virtualized in the same way; GPU applications can send commands to GPUs without intervention from the hypervisor after a request for creating a virtual command channel is validated.

VGRIS [Qi et al. 2014] adopts the para virtualization technique implemented in VMware SVGA II [Dowty and Sugerman 2009] for gaming applications in virtualization environments. Using VMware player 4.0, VGRIS further develops an agent for each VM in the host, which is responsible for monitoring the performance of each individual VM and sending this information to the GPU scheduler. The scheduling method will be explained in Section 7.2.2.

Huang et al. [2016] developed a para virtualization solution for the Heterogeneous System Architecture (HSA) [Kyriazis 2012] proposed by AMD. HSA combines a CPU and a GPU on the same silicon die to relieve the communication overhead between them. The authors try to address this architectural change in KVM-based virtualization. First, HSA realizes shared virtual memory where the CPU and the GPU share the same virtual address space. To virtualize this concept, the authors just assign the page tables used by CPU MMU to GPU IOMMU to reuse them as GPU shadow page tables. Second, HSA generates an interrupt when a GPU memory access generates a page fault; by this trigger, the CPU can modify the corresponding page table for the GPU. To virtualize this feature, the authors modify the shadow page table on the interrupt and also notify the corresponding guest OS to modify its guest page table. Shadow page tables are actually used for address translation, but the guest page table modification is also required for maintaining system integrity. The GPU driver in the guest OS is modified to deal with this notification. Finally, HSA allows the CPU and the GPU to have a shared buffer in user space to store GPU commands. To virtualize this feature, the authors simply let the GPU know the address of a shared buffer in the guest so the GPU can fetch GPU commands directly from the guest.

5.2. Full Virtualization

GPUvm [Suzuki et al. 2014] implements both full and para virtualization in the Xen hypervisor by using a Nouveau driver [X.OrgFoundation 2011] in the guest OS side. To isolate multiple VMs on a GPU in full virtualization, GPUvm partitions both physical GPU memory and the MMIO region into several pieces and assigns each portion to an individual VM. A GPU shadow page table per VM enables access to the partitioned memory by translating the virtual GPU addresses to the physical GPU addresses of the partitioned memory. Each shadow page table is updated on TLB flush. In CPU virtualization, the hypervisor updates shadow page tables when page faults occur. However, GPUvm cannot deal with page faults from the GPU because of a limitation of the current NVIDIA GPU design. Therefore, GPUvm should scan the entire page table on every TLB flush. The partitioned MMIO region is configured as read-only so every GPU access from a guest can generate a page fault. The OS then intercepts and emulates the access in the driver domain of Xen. Because the number of command submission channels (explained in Section 2.1) is limited in hardware, GPUvm also virtualizes them by creating shadow channels and mapping a virtual channel to a shadow channel. Actually, this full virtualization technique shows poor performance

for the following reasons: (1) the interception of every GPU access and (2) the scanning of the entire page table on every TLB flush. GPUvm addresses the first limitation with BAR Remap, which only intercepts GPU calls related to accesses to GPU channel descriptors. A possible isolation issue caused by passing through other GPU accesses is addressed by utilizing shadow page tables, which isolate BAR area accesses among VMs. For the second limitation, GPUvm suggests a para virtualization technique. Similarly to Xen [Barham et al. 2003], GPUvm constructs guest GPU page tables and allows VMs to use these page tables directly instead of shadow page tables. The guest driver issues hypercalls to GPUvm when its GPU page table needs to be updated. GPUvm then validates these requests for isolation between VMs.

gVirt [Tian et al. 2014] is based on its previous work, XenGT [Shan et al. 2013], and implements full GPU virtualization for Intel on-chip GPUs in the Xen hypervisor. It focuses on graphics acceleration rather than GPGPU computing. gVirt asserts that the frame and command buffers are the most performance-critical resources in GPUs. It allows each VM to access the two buffers directly (pass-through) without intervention from the hypervisor. For this purpose, the graphics memory resource is partitioned by the gVirt Mediator so each VM can have its own frame and command buffers in the partitioned memory. At the same time, privileged GPU instructions are trapped and emulated by the gVirt Mediator in the driver domain of Xen. This enables secure isolation among multiple VMs without significant performance loss. The whole process is called mediated pass-through. KVMGT [Song et al. 2014] is a ported version of gVirt for KVM and has been integrated into the mainline Linux kernel since version 4.10.

gHyvi [Dong et al. 2015] points out that its predecessor, gVirt, suffers from severe performance degradation when a GPU application in a VM performs frequent updates on guest GPU page tables. This modification causes excessive VM exits [Har'El et al. 2013], which are expensive operations in hardware-based virtualization. This is also known as the massive update issue. gHyvi introduces relaxed page table shadowing, which removes the write-protection of the page tables to avoid excessive trapping. The technique rebuilds the guest page tables at a later point of time when rebuilding is required. This lazy reconstruction is possible because the modification to the guest page tables will not take effect before relevant GPU operations are submitted to the GPU command buffer.

gScale [Xue et al. 2016] solves gVirt's scalability limitation. gVirt partitions the global graphics memory (2GB) into several fixed size regions and allocates them to vGPUs. Due to the recommended memory allocation for each vGPU (e.g., 448MB in Linux), gVirt limits the total number of vGPUs to 4. gScale overcomes this limitation by making the GPU memory shareable. For the high graphics memory in Intel GPUs, gScale allows each vGPU to maintain its own private shadow graphics translation table (GTT). Each private GTT translates the vGPU's logical graphics address to any physical address in the high memory. On context switching, gScale copies the next vGPU's private GTT to the physical GTT to activate the vGPU's graphics address space. For the low memory, which is also accessible by the CPU, gScale introduces Ladder mapping combined with private shadow GTTs. As virtual CPUs and GPUs are scheduled asynchronously, a virtual CPU may access illegal memory if it refers to the current graphics address space. Ladder mapping modifies the Extended Page Table (EPT) used by the virtual CPU so it can bypass the graphics memory space. With these schemes, gScale can host up to 15 vGPUs for Linux VMs and 12 for Windows VMs. The scheduling method that gScale adopts is discussed in Section 7.2.2.

6. HARDWARE-SUPPORTED VIRTUALIZATION

In the hardware-supported virtualization approach, each VM is given direct access to the GPU instead of using an API remoting library or an emulated device. This

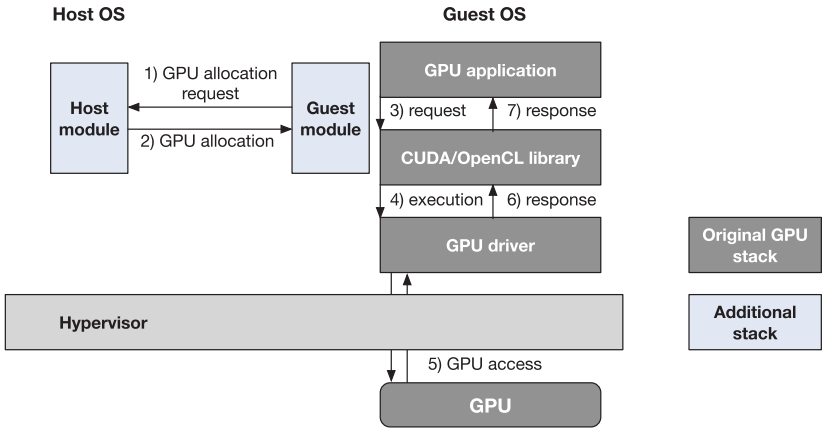

Fig. 4. Architecture of the hardware-supported approach.

approach exploits hardware extension features for I/O virtualization provided by chip manufactures or GPU vendors, which include Intel VT-d [Abramson et al. 2006], AMD-Vi [Van Doorn 2006], and NVIDIA GRID [Herrera 2014]. The auxiliary features virtualize device transfers and interrupts by using a remapping technology; the DMA channels and interrupts of the device are directly mapped to the guest OS. Therefore, data can flow from the GPU device to the memory space of the VM without hypervisor involvement while the corresponding interrupts are directly delivered to the VM. Intel VT-d and AMD-Vi only support a single VM for I/O virtualization; a GPU is dedicated to a designated VM when the VM boots. However, NVIDIA GRID overcomes this limitation and supports multiplexing of a GPU between VMs.

Figure 4 illustrates a system that adopts hardware-supported virtualization implemented by Intel VT-d or AMD-Vi. In this architecture, the original GPU stack in the VM can utilize the GPU without any modification to the library or driver. The guest OS bypasses the hypervisor while communicating with the GPU because data transfers and interrupts are remapped to the VM. Since Intel VT-d or AMD-Vi only support a single VM for GPU virtualization, a hot plug functionality has been proposed [Jo et al. 2013a], which can dynamically install or remove a GPU device in the VM to share the GPU in a coarse-grained manner. In this case, the host and guest modules interact with each other to process GPU allocation requests from each VM.

The advantage of this approach is that it can realize GPU virtualization without an additional software layer while achieving near-native performance due to hardware support. The limitation is that imposing GPU scheduling policies, as discussed further in Section 7, may be impossible, because GPU operations do not pass the host OS or the hypervisor. In addition, it is difficult to implement execution checkpointing, live migration, and fault-tolerant execution features as in the API remoting approach.

6.1. Methods Supporting a Single VM

Amazon Elastic Compute Cloud (Amazon EC2) [Amazon 2010] is the first cloud hosting service that supports GPUs for cloud tenants by using the Intel GPU pass-through technology [Ou et al. 2012]. In 2010, Amazon EC2 introduced Cluster GPU Instances (CGIs), which provide two NVIDIA Tesla GPUs per VM [Yeh et al. 2013]. CGIs can support HPC applications requiring massive parallel processing power by exposing native GPUs to each guest OS directly.

Expósito et al. [2013] explored the performance of a cluster of 32 CGIs in Amazon EC2. They tested the SHOC and Rodinia benchmark suites as synthetic kernels,

NAMD [Phillips et al. 2005] and MC-GPU [Badal and Badano 2009] as real-world applications in science and engineering, and the HPL benchmark [Petitet 2004] as a widely used implementation of Linpack [Dongarra et al. 2003]. They measured the performance both in virtualization using Amazon EC2 CGIs and in a native environment using their own cluster. The authors show that computationally intensive programs can generally take full advantage of GPUs in the cloud setting. However, memory-intensive applications can experience a small penalty because Amazon EC2 CGIs enable ECC memory error check features, which can limit memory bandwidth. Also, network-intensive GPU applications may suffer from virtualized network access, which reduces scalability.

Yang et al. [2012a, 2012b, 2014] implemented a GPU pass-through system using Xen and KVM and performed performance analysis of CUDA applications. The authors explain how to enable GPU pass-through technically in both hypervisors and evaluate the performance of the CUDA SDK benchmark suite. The authors claim that the GPU performance by using the Intel pass-through technology in both hypervisors is similar to the performance in a native environment.

Shea and Liu [2013] explored the performance of cloud gaming in a GPU pass-through environment. They found that some gaming applications perform poorly when they are deployed in a VM using a dedicated GPU. This is because the virtualized environment cannot secure enough memory bandwidth while transferring data between the host and the GPU compared with a native environment. The authors identify that the performance in KVM is less than 59% of that of their bare-metal system. By detailed profiling, some gaming applications are observed to generate frequent context switching between the VM and the hypervisor to process memory access requests during memory transfers, which brings memory bandwidth utilization down.

Younge and Fox [2014] and Younge et al. [2014] evaluated the performance of a Xen VM infrastructure using a PCI pass-through technology and the SHOC benchmark. The authors found that there is only a 1.2% performance penalty in the worst case in the Kepler K20m GPU-enabled VM, whereas the API remoting approach incurs performance overhead up to 40%. In more recent work [Younge et al. 2015], they evaluated HPC workloads in a virtualized cluster using PCI pass-through with SR-IOV [Dong et al. 2012] and GPUDirect [Shainer et al. 2011]. SR-IOV is a hardware-assisted network virtualization technology that provides near-native bandwidth on 10-Gigabit connectivity within VMs. GPUDirect reduces the overhead of data transfers across GPUs by supporting direct RDMA between GPUs on an InfiniBand interconnect. For evaluation, they used two molecular dynamics (MD) applications: large-scale atomic/molecular massively parallel simulator (LAMMPS) [Plimpton et al. 2007] and highly optimized object-oriented molecular dynamics (HOOMD) [Anderson et al. 2010]. The authors observe that the MD applications using MPI and CUDA can run at near-native performance with only 1.9% and 1.5% overheads for LAMMPS and HOOMD, respectively, compared to their execution in a non-virtualized environment.

Walters et al. [2014] characterized the performance of VMWare ESXi, KVM, Xen, and Linux Containers (LXC) using the PCI pass-through mode. They tested the CUDA SHOC and OpenCL SDK benchmark suites as microbenchmarks and the LAMMPS molecular dynamics simulator [Plimpton et al. 2007], GPU-LIBSVM [Athanasopoulos et al. 2011], and the LULESH shock hydrodynamics simulator [Karlin et al. 2013] as application benchmarks. The authors observe that KVM consistently yields near-native performance in all benchmark programs. VMWare ESXi performs well in the Sandy Bridge microarchitecture [Kanter 2010], but not in the Westmere microarchitecture [Kurd et al. 2010]. The authors speculate that VMWare ESXi is optimized for more recent microarchitectures. Xen consistently shows average performance among the hypervisors. Finally, LXC performs closest to the native environment because LXC guests share a single Linux kernel.

6.2. Methods Supporting Multiplexing

Jo et al. [2013a, 2013b] tried to overcome the inability of GPU pass-through to support sharing of a single GPU between multiple VMs. In GPU pass-through environments, a GPU can be dedicated only to a single VM when the VM boots. The GPU cannot be de-allocated until the VM is shutdown. The authors implemented coarse-grained sharing by utilizing the hot plug functionality of PCIe channels, which can install or remove GPU devices dynamically. To realize this implementation, a CUDA wrapper library is provided to VMs to monitor the activity of GPU applications. If an application requires a GPU, then a GPU allocation request is sent to Virtual Machine 0 (VM0), which is the host OS in KVM or domain 0 in Xen, by the wrapper library. The GPU-Admin in VM0 mediates this request and attaches an available GPU managed by the GPU pool to the VM. When the application finishes its execution, the wrapper library sends a de-allocation request to the GPU-Admin. The GPU-Admin then returns the GPU into the GPU pool.

vmCUDA [Vu et al. 2014] combines the API remoting approach with the hardware-assisted approach to implement GPU virtualization for an environment where the host OS does not exist (e.g., VMware ESX hypervisor). In such an environment, only the hypervisor has the right to access the GPU. However, communicating by API remoting with the hypervisor, which resides in a deep privileged protection level, can impose significant burdens on the system. To address this issue, vmCUDA launches a management VM called an appliance VM that can act as a driver domain with a backend driver. vmCUDA provides a CUDA wrapper library and a frontend driver to a guest OS, which can intercept and deliver GPU requests to the backend. For communication between the split drivers, vmCUDA uses VMware VMCI [Burtsev et al. 2009], VMware vRDMA [Ranadive and Davda 2012], or TCP/IP. The appliance VM adopts a pass-through technology to process the intercepted calls with little overhead.

NVIDIA GRID [Herrera 2014] overcomes the inter-VM GPU sharing limitations of GPU pass-through. The NVIDIA GRID architecture implements a new I/O Memory Management Unit (IOMMU) that can individually map and translate the virtual address space of each guest OS to the physical address space of the GPU. In addition, NVIDIA GRID allows each guest OS to have its own separate input buffer for an isolated command stream. Through these two architectural changes, NVIDIA GRID-enabled GPUs can realize GPU sharing and isolate each guest OS's GPU execution environment while preserving the performance that a pass-through mechanism would offer. Currently, NVIDIA GPUs targeted for cloud environments, which include NVIDIA GRID K1, K2, NVIDIA Tesla M6, and M60, support the GRID feature.

Hong et al. [2014] utilized one of NVIDIA GRID-enabled GPUs and re-evaluated the performance impact of cloud gaming formerly evaluated by Shea and Liu [2013]. The authors observe that the new GPU can outperform traditional pass-through-based GPUs because of hardware-specific optimizations. In addition, they reveal that frequent context-switching between VMs and the hypervisor does not contribute to the performance loss, which is a fundamentally different conclusion from Shea and Liu [2013]. They claim that although the VMs incur more context switches, GRID-enabled GPUs achieve a net performance gain in terms of frames per second.

7. SCHEDULING METHODS

GPU scheduling methods are required to fairly and effectively distribute GPU resources between tenants in a shared computing environment. However, GPU virtualization software faces several challenges on applying GPU scheduling polices because of the following reasons. First, GPUs normally do not provide the information of how long a GPU request occupies the GPU, which creates a task accounting problem [Dwarakinath

Table III. Comparison of Scheduling Methods Based on the Scheduling Discipline, Support for Load-balancing, and Software Platform

Name	Scheduling discipline							Load balancing	Software platform	
	FCFS	Round-robin	Priority-based	Fair queuing	Credit-based	Affinity-based	SLA-based		Single OS	Hypervisor
GERM [Dwarakinath 2008]				O					O	
GViM [Gupta et al. 2009]		O			O					O
[Jim´enez et al. 2009]	O							O	O	
TimeGraph [Kato et al. 2011d]			O						O	
RGEM [Kato et al. 2011c]			O						O	
PIX [Kato et al. 2011b]			O						O	
Pegasus [Gupta et al. 2011]	O				O		O			O
[Ravi et al. 2011]						O				O
PTask [Rossbach et al. 2011]	O		O					O	O	
Gdev [Kato et al. 2012]					O				O	
Rain [Sengupta et al. 2013]			O		O			O	O	
Strings [Sengupta et al. 2014]			O		O			O	O	
Disengaged scheduling [Menychtas et al. 2014]		O		O					O	
GPUvm [Suzuki et al. 2014]					O					O
gVirt [Tian et al. 2014]		O								O
VGRIS [Qi et al. 2014]			O				O			O
VGASA [Zhang et al. 2014]							O			O
gScale [Xue et al. 2016]		O								O
Libra [Farooqui et al. 2016]						O		O	O	

2008; Menychtas et al. 2014]. Second, system software often regards GPUs as I/O devices rather than full processors and hides the methods of multiplexing the GPU in the device driver. This prevents GPU virtualization software from directly imposing certain scheduling policies on GPUs [Gupta et al. 2011; Panneerselvam and Swift 2012]. Finally, GPUs were non-preemptive until recently, which means a long-running GPU kernel cannot be preempted by software until it finishes. This will cause unfairness between multiple kernels and severely deteriorate the responsiveness of latency-critical kernels [Tanasic et al. 2014]. Currently, a new GPU architecture to support GPU kernel preemption has emerged in the market [NVIDIA 2016a], but it is expected that existing GPUs will continue to suffer from this issue. In this section, we introduce representative GPU scheduling polices and mechanisms proposed in the literature to address these challenges.

7.1. Classification of GPU Scheduling Methods

Table III shows a comparison of representative GPU scheduling methods in the literature. We classify the methods in terms of the scheduling discipline, support for load-balancing, and software platform.

Scheduling discipline is an algorithm to distribute GPU resources among processes or virtual GPUs (vGPUs). We classify the GPU scheduling methods based on a commonly used classification [Silberschatz et al. 1998] as follows:

- FCFS: First-come, first-served (FCFS) serves processes or vGPUs in the order that they arrive.
- Round-robin: Round-robin is similar to FCFS but assigns a fixed time unit per process or vGPU, referred to as a time quantum, and then cycles through processes or vGPUs.
- Priority-based: Priority-based scheduling assigns a priority rank to every process or vGPU, and the scheduler executes processes or vGPUs in order of their priority.
- Fair queuing: Fair queuing is common in network and disk scheduling to attain fairness when sharing a limited resource [Demers et al. 1989; Park and Shen 2012]. Fair queuing assigns start tags to processes or vGPUs and schedules them in increasing order of start tags. A start tag denotes the accumulated usage time of a GPU.
- Credit-based: Credit-based scheduling is a computationally efficient substitute to fair queuing [Bensaou et al. 2001]. The scheduler periodically distributes credits to every process or vGPU, and each process or vGPU consumes credits when it is served on the CPU for exploiting the GPU. The scheduler selects a process or vGPU with a positive credit value.
- Affinity-based: This scheduling algorithm produces affinity scores for a process or vGPU to predict the performance impact when it is scheduled on a certain resource.
- Service Level Agreement– (SLA) based: SLA is a contract between a cloud service provider and a tenant regarding Quality of Service (QoS) and the price. SLA-based scheduling tries to meet the service requirement when distributing GPU resources [Wu et al. 2011].

Load balancing indicates whether the scheduling method supports the distribution of workloads across multiple processing units. The software platform denotes whether the scheduling method is developed in a single OS or hypervisor environment. We include GPU scheduling research performed in a single OS environment because the same research can be also applicable to virtualized environments without significant modifications to the system software.

The GPU scheduling methods in Table III will be discussed in depth in the following section.

7.2. Algorithms for Scheduling a Single GPU

7.2.1. Single OS Environment. GERM [Dwarakinath 2008] is a GPU scheduling policy that utilizes Deficit Round Robin fair queuing [Shreedhar and Varghese 1996], which is a network scheduler for switching packets with multiple flows. GERM maintains per-process queues for GPU commands and allows each queue to send commands to the GPU during a predefined time quantum. A queue's deficit or surplus time compared to the time quantum will be compensated or reimbursed in the next round. This scheme is suitable for non-preemptive GPUs where a GPU request cannot be preempted and the size of each request can vary significantly. Regarding the accounting of each request, GERM cannot measure the request size exactly because GPUs generally do not interrupt the CPU after a request is processed. Therefore, it adopts heuristics to estimate how long a group of commands will occupy the GPU on average. GERM injects a special GPU command that increases a scratch register containing the number of processed requests in the GPU. By reading this register periodically, GERM infers how much time is taken for a GPU command.

TimeGraph [Kato et al. 2011a, 2011d] focuses on GPU scheduling for soft real-time multi-tasking environments. It provides two scheduling polices: Predictable-Response-Time (PRT) and High-Throughput (HT). The PRT policy schedules GPU applications based on their priorities, so important tasks can expect predictable response times. When a group of GPU commands is issued by a process, the group is buffered in the wait queue, which resides in kernel space. TimeGraph configures the GPU to generate

an interrupt to the CPU after each group's execution is completed. This is enabled by using pscnv [PathScale 2012], an open source NVIDIA GPU driver. The PRT scheduler is triggered by each interrupt and fetches the highest-priority group from the wait queue. As the scheduler is invoked every time a group of GPU commands finishes its execution, it incurs non-negligible overhead. The HT scheduler addresses this issue by allowing the current task occupying the GPU to execute its following groups without buffering into the wait queue, when there are no other higher priority groups waiting.

RGEM [Kato et al. 2011c] develops a responsive GPGPU execution model for GPGPU tasks in real-time multi-tasking environments, similarly to TimeGraph [Kato et al. 2011d]. RGEM introduces two scheduling methods: Memory-Copy Transaction scheduling and Kernel Launch scheduling. The former policy splits a large memory copy operation into several small pieces and inserts preemption points between the separate pieces. This prevents a long-running memory copy operation from occupying the GPU boundlessly, which will block the execution of high priority tasks. The latter policy follows the scheduling algorithm of the PRT scheduler in TimeGraph, except that Kernel Launch scheduling is implemented in user space.

PIX [Kato et al. 2011b] applies TimeGraph [Kato et al. 2011d] to GPU-accelerated X Window systems. When employing the PRT scheduler in TimeGraph, PIX solves a form of the priority inversion problem where the X server task (X) with low priority can be preempted by a medium-priority task (A) on the GPU while rendering the frames of a high-priority task (B) (i.e., $P_B > P_A > P_X$). The high priority task is then blocked for a long time while the X server task deals with the frames of the medium-priority task. PIX suggests a priority inheritance protocol where the X server inherits the priority of a certain task while rendering the frames of the task. This eliminates the priority inversion problem raised by the existence of the additional X server task.

Gdev [Kato et al. 2012] introduces a bandwidth-aware non-preemptive device (BAND) scheduling algorithm. The authors found that the Credit scheduler [Gupta et al. 2009; Barham et al. 2003] fails to achieve good fairness in GPU scheduling because the Credit scheduler assumes that it will run preemptive CPU workloads, whereas GPUs do not support hardware-based preemption. To address this issue, Gdev performs two heuristic modifications to the Credit scheduler. First, the BAND scheduler does not degrade the priority of a GPU task after the credit value of the task becomes zero. The BAND scheduler lowers the priority when the task's actual utilization exceeds the assigned one. This modification compensates for credit errors caused by non-preemptive executions. Second, the BAND scheduler waits for the completion of GPU kernels of a task and assigns a credit value to the task based on its GPU usage. This modification contributes to fairer resource allocations.

Disengaged scheduling [Menychtas et al. 2014] provides a framework for scheduling GPUs and introduces three algorithms to achieve both high fairness and high utilization. The framework endeavors to employ the original NVIDIA driver and the libraries; it uses neither the API remoting approach nor a custom GPU driver to mediate GPU calls. The framework makes the GPU MMIO region of each task read-only so every GPU access can generate a page fault. The OS then intercepts and buffers GPU calls in kernel space. Disengaged scheduling offers three scheduling policies. First, the Timeslice with Overuse Control scheduling algorithm implements a standard token-based time slice policy. A token is passed to a certain task and the task can use the GPU during its time slice. The scheduler accounts for overuse by waiting for all submitted requests of the token holder to be completed at the end of each time slice. Since the GPU requests of both the token holder and other tasks generate page faults, this policy causes significant overhead due to frequent trapping to the OS. In addition, it does not implement work-conserving because the GPU can be underutilized if applications

are not GPU intensive. Second, Disengaged Timeslice reduces this overhead by allowing the token holder to issue GPU commands without buffering in kernel space. However, this scheduling is still not work-conserving. Finally, Disengaged Fair Queueing executes several tasks concurrently without trapping in the common case. Only in the accounting period does the scheduler enable the trapping mechanism and is each task run sequentially. In this period, the scheduler samples the request size of each task and feeds this information to fair queuing to approximate each task's cumulative GPU usage. The scheduler then selects several tasks that have low start tags to run them without trapping until the next accounting period. The scheduler is work conserving because several tasks can exploit the GPU simultaneously; from the Kepler microarchitecture, NVIDIA allows multiple GPU kernels from different tasks to run concurrently [NVIDIA 2012].

7.2.2. Virtualization Environment. GViM [Gupta et al. 2009] uses both simple Round-Robin scheduling and Credit scheduling of the Xen hypervisor for scheduling tasks on GPUs. As GViM operates on top of the driver level, GViM controls the rate of GPU request submissions for scheduling before the requests reach the driver. GViM implements Round Robin– (RR) and XenoCredit– (XC) based scheduling. RR selects a vGPU sequentially for every fixed time slice and monitors the vGPU's call buffer during the period. XC uses the concept of credit, which represents the allocated GPU time of each vGPU. XC processes the vGPU's call buffer for a variable time in proportion to the credit amount, which enables weighted fair-sharing between guest VMs.

Pegasus [Gupta et al. 2011] addresses one of the challenges in GPU scheduling that a GPU virtualization framework cannot impose a scheduling policy on GPUs because the method of GPU multiplexing is hidden in the device driver. Pegasus introduces the concept of an accelerator VCPU (aVCPU) to make GPUs basic schedulable entities; the components of an aVCPU are discussed in Section 4.2. Pegasus focuses on satisfying different application requirements by providing diverse methods for scheduling GPUs. Pegasus includes first-come, first-served (FCFS), proportional fair-share (AccCredit), strict co-scheduling (CoSched), augmented credit-based scheme (AugC), and SLA feedback-based (SLAF) schedulers. AccCredit adapts the Credit scheduling concept in Xen for GPU scheduling. CoSched applies co-scheduling for barrier-rich parallel applications where a VCPU of a VM and its corresponding aVCPU frequently synchronize with each other. CoSched forces both entities (i.e., the VCPU and its corresponding aVCPU) to be executed at the same time to address synchronization bottlenecks. However, this strict co-scheduling policy can hamper fairness between multiple VMs. AugC conditionally co-schedules both entities to achieve better fairness only when the target VCPU has enough credits and can lend its credits to its corresponding aVCPU. SLAF applies feedback-based proportional fair-share scheduling. The scheduler periodically monitors Service-Level Objective (SLO) violations by the feedback controller and compensates each domain by giving extra time when a violation is detected.

Ravi et al. [2011] tackled the problem that one GPU application sometimes cannot have enough parallelism to fully utilize a modern GPU. To increase overall GPU utilization, the authors try to consolidate multiple GPU kernels from different VMs in space and time. Space sharing co-schedules kernels that do not use all streaming multiprocessors (SMs) in the GPU. Time sharing allows more than one kernel to share the same SM if the cumulative resource requirements do not exceed the capability of the SM. Because the NVIDIA Fermi-based GPU used in this research only allows a set of kernels submitted from a single process to be executed concurrently, the authors let GPU kernels from different VMs be handled by a single thread. The scheduler then finds an affinity score between every two kernels to predict the performance improvement

when they are space and time shared. In addition, the scheduler calculates potential affinity scores when they are space and time shared with a different number of thread blocks and threads. The scheduler then selects n kernels to run based on the set of affinity scores.

GPUvm [Suzuki et al. 2014] employs the BAND scheduler of Gdev [Kato et al. 2012] and solves a flaw of Credit scheduling. The original BAND scheduler distributes credits to each VM based on the assumption that the total utilization of all vGPUs can reach 100%. However, when the GPU scheduler is active, the GPU can temporarily become idle. This situation causes each vGPU to have unused cumulative credit, which may lead to inopportune scheduling decisions. To address this issue, GPUvm first transforms the CPU time that the GPU scheduler occupies into a credit value and then subtracts the value from the total credit value of the current vGPU.

gVirt [Tian et al. 2014] implements a coarse-grained QoS policy. gVirt allows GPU commands from a VM to be submitted into the guest ring buffer during the VM's time slice. After the time slice, gVirt waits for the ring buffer to be emptied by the GPU, because the GPU is non-preemptive. To minimize this wait period, gVirt develops a coarse-grained flow control method, which ensures that the total length of submitted commands is within a time slice. gVirt also implements a gang scheduling policy where dependent graphic engines are scheduled together. The graphic engines in gVirt use semaphores to synchronize accesses to shared data. To eliminate synchronization bottlenecks, gVirt schedules the related engines at the same time.

VGRIS [Qi et al. 2014] tries to address GPU scheduling issues for gaming applications deployed in cloud computing. VGRIS introduces three scheduling policies to meet different performance requirements. The SLA-aware scheduling policy just provides minimum GPU resources to each VM to satisfy its SLA requirement. The authors observe that a fair scheduling policy provides resources evenly under contention, but non-GPU-intensive applications may obtain more resources than necessary while GPU-intensive ones may not satisfy the requirement. SLA-aware scheduling slows the execution speed of fast-running applications (i.e., non-GPU-intensive applications) so other slow applications can get more chances to occupy the GPU. For this purpose, it inserts a sleep call at the end of the frame computation code of fast-running applications before the frame is displayed. However, SLA-aware scheduling may lead to low GPU utilization when only a small number of VMs is available. The Proportional-share scheduling policy addresses this issue by distributing GPU resources fairly using the priority-based scheduling policy of TimeGraph [Kato et al. 2011d]. Finally, the Hybrid scheduling policy combines SLA-aware scheduling and Proportional-share scheduling. Hybrid scheduling first applies SLA-aware scheduling and switches to Proportional-share scheduling if a resource surplus is available.

VGASA [Zhang et al. 2014] advances VGRIS [Qi et al. 2014] by providing adaptive scheduling algorithms, which employ a dynamic feedback control loop using the proportional-integral (PI) controller [Ogata 1995]. Similarly to VGRIS, VGASA provides three scheduling policies. SLA-Aware (SA) receives the frames per second (FPS) information from the feedback controller and adjusts the length of sleep time in the frame computation code to meet the predefined SLA requirement (i.e., the rate of 30 FPS). Fair SLA-Aware (FSA) dispossesses fast running applications of their GPU resources and redistributes the resources to slow running ones. Enhanced SLA-Aware (ESA) allows all VMs to have the same FPS rate under the maximum GPU utilization. ESA improves SA by dynamically calculating the SLA requirement during runtime. ESA can address a tradeoff between deploying more applications and providing smoother experiences.

gScale [Xue et al. 2016] optimizes the GPU scheduler of gVirt. gScale develops private shadow GTTs to improve the scalability issue as explained in Section 5.2.

However, applying private GTTs requires page table copying upon every context switch. To mitigate this overhead, gScale does not perform context switching for idle vGPUs. Furthermore, it implements slot sharing, which divides the high graphics memory into several slots and dedicates a single slot to each vGPU. gScale's scheduler distributes busy vGPUs across the slots so each busy one can monopolize each slot. This arrangement can decrease the amount of page table entry copying.

The recent NVIDIA Pascal architecture [NVIDIA 2016a] implements hardware-based preemption to address the problem of long-running GPU kernels monopolizing the GPU. This situation can cause unfairness between multiple kernels and significantly deteriorate the system responsiveness. Existing GPU scheduling methods address this issue by either killing a long-running kernel [Menychtas et al. 2014] or providing a kernel split tool [Basaran and Kang 2012; Zhou et al. 2015; Margiolas and O'Boyle 2016]. The Pascal architecture allows GPU kernels to be interrupted at instruction-level granularity by saving and restoring each GPU context to and from the GPU's DRAM.

7.3. Algorithms for Load Balancing

Jiménez et al. [2009] introduced several scheduling methods for tasks that can be executed both on a CPU and a GPU. In this approach, a programmer explicitly indicates that a function can be executed on both processors, and then the compiler generates both versions of the function code (CPU and GPU). The scheduler decides which processing element (PE) is suitable for executing the function. The first-free (FF) algorithm selects the first PE that is not busy in the PE list. However, this policy does not consider the execution speed of each PE, which may cause load imbalance between different PEs. The FF Round Robin scheduler addresses this issue by allowing a high-performance PE to have a more amount of workloads. Both algorithms ignore that a particular function can run more efficiently on a specific PE. Performance History scheduling forces the first n calls of the same task to be executed on different PEs and uses this information when finding a suitable PE among non-congested ones. After a PE is selected, all policies apply a first-come, first-served (FCFS) design to choose a task in the given PE.

PTask [Rossbach et al. 2011] supports a dataflow programming model where a GPU application is modeled as a directed graph having vertices called ptasks. Each ptask is a unit of work that can run on a GPU. PTask suggests four policies for scheduling ptasks: First-available, FIFO, Priority, and Data-aware. The First-available policy does not maintain a queue for ptasks, allowing threads to compete for available GPUs. In this policy, when ptasks outnumber available GPUs, access to GPUs is arbitrated by locks that protect each GPU data structure. The FIFO policy addresses this limitation with queuing. The Priority scheduler first sorts ptasks in the queue based on their effective priorities and next assigns a suitable GPU to each task while GPUs are available. The effective priority reflects several factors, including each task's static priority, current wait time, average wait time, and average runtime. Normally, when a ptask has waited for a long time, its effective priority will increase, which prevents starvation. To assign a suitable GPU to each task, PTask introduces the concepts of fitness and strength. The fitness denotes whether a GPU supports the execution environment required by a ptask while the strength indicates the capability of a GPU including the number of cores and their clock speed. When both conditions are met, a GPU is assigned to each task in the queue until GPUs are no more available. Finally, the Data-aware policy computes effective priority values as the Priority scheduler, but its GPU selection scheme is based on data awareness. Since PTask supports dataflow programming, this scheduler assigns a certain task a GPU that has input data in the device memory, which was previously produced by related ptasks.

Rain [Sengupta et al. 2013] develops a scheduling framework for load-balancing GPU requests across multiple GPUs existing on distributed machines. Rain proposes a two-level hierarchical scheduling framework. The top level framework distributes workloads across multiple GPUs on different servers. The bottom level one applies device-level scheduling for each GPU. For workload balancing polices in the top level, Rain suggests five policies. The Global Round Robin (GRR) policy just performs round-robin assignments. The GMin policy extends GRR by maintaining a load level per device and returning a GPU with a minimum load. The Weighted-GMin policy is based on GMin and takes into account the capability of each GPU by assigning a relative weight to each GPU. The Runtime feedback (RTF) policy receives feedback from the device-level scheduler. The bottom level scheduler monitors the execution time of GPU requests and provides this information to RTF. RTF selects a GPU with minimum load in future scheduling points by utilizing the given information. The GPU utilization feedback (GUF) policy investigates the GPU utilization of each GPU application to prevent different applications with high GPU utilization from being placed in the same GPU. For device-level scheduling polices in the bottom level, Rain provides two schemes. Least Attained Service (LAS) targets decreasing the stall time of a GPU application. CPU stalls are caused by synchronization calls such as cudaThreadSynchronize(), which wait until the GPU completes the submitted requests. LAS prioritizes GPU applications that attained less GPU time to minimize the overall CPU stall time. True Fair-Share (TFS) distributes GPU resources to each application in proportion to its weight. When any long-running kernel overuses its allocated time slice, TFS penalizes it in subsequent scheduling points.

Strings [Sengupta et al. 2014] extends its previous work, Rain [Sengupta et al. 2013], by including more effective scheduling polices. For workload balancing polices, Strings additionally suggests Data Transfer Feedback (DTF) and Memory Bandwidth Feedback (MBF). DTF receives feedback about the time spent on data transfer from the device-level scheduler. This enables collocating a compute-intensive application and a memory-intensive one together, whose operations can be overlapped. MBF takes into account the approximate memory bandwidth of each application to prevent applications with high memory bandwidth from being collocated in the same GPU. For device-level scheduling, Strings further proposes the Phase Selection (PS) policy. PS leverages an additional parallelism opportunity that CUDA launch and memory copy operations from different CUDA streams can be executed concurrently by NVIDIA GPUs. PS schedules backend threads in different phases simultaneously to overlap the operations.

Libra [Farooqui et al. 2016] points out that existing work-stealing algorithms are not efficient in recent integrated CPU-GPU processors such as Intel's Broadwell and AMD's Kaveri. Existing algorithms are agnostic to the characteristics of the CPU and GPU, and this situation results in performance degradation of OpenCL applications, which can be run on both processors. To overcome this issue, Libra first applies lightweight online profiling to produce the device affinity scores of a target application. The application is assigned to a processor having the highest affinity score. During its execution, Libra applies hierarchical stealing that minimizes stealing by other devices that have lower affinity scores. This mechanism reduces cross-device stealing and thus decreases contention between the CPU and GPU incurred by steal attempts.

8. CHALLENGES AND FUTURE DIRECTIONS

Through the analysis of existing GPU virtualization techniques, we conclude that technical challenges of how to virtualize GPUs have been addressed to a significant extent. However, a number of challenges remain open in terms of performance and

capabilities of GPU virtualization environments. We discuss them in this section, along with some future research directions to address the challenges.

Lightweight virtualization: Linux-based containers are an emerging cloud technology that offers process-level lightweight virtualization [Vaughan-Nichols 2006]. Containers do not require additional wrapper libraries or front/backend driver models to virtualize GPUs, because multiple containers are multiplexed by a single Linux kernel [Haydel et al. 2015]. This feature allows containers to achieve performance that is close to that of native environments. Unfortunately, current research on GPU virtualization using containers is at an initial stage. Published work just includes performance comparisons between containers and other virtualization solutions [Walters et al. 2014]. To utilize GPU-equipped containers in cloud computing, fair and effective GPU scheduling is required. Most GPU schedulers require API extensions or driver changes in containers to mediate GPU calls, which will impose non-negligible overhead on containers. One promising option is to adapt Disengaged scheduling [Menychtas et al. 2014] in the host OS, which needs neither additional wrapper libraries nor custom drivers for GPU scheduling, as explained in Section 7.2.1.

Scalability: The primary purpose of system virtualization is to increase resource utilization and reduce cost of ownership. This goal can be attained by consolidating large numbers of VMs on each physical machine in the datacenter. For example, the Xen hypervisor is designed to create more than 500 virtual CPUs per host [Xenproject 2016]. Unfortunately, GPU virtualization does not offer such a high degree of consolidation capability for VMs that use GPUs. Xue et al. [2016] put significant effort into increasing scalability for their gScale framework, but the framework can only host up to 15 vGPUs for Linux VMs. There is still a significant imbalance between the consolidation capabilities of CPUs and GPUs, and bridging the gap remains a challenging endeavor. The performance impact of deploying a massive number of vGPUs needs to be investigated in terms of the size of GPU device memory, the frequency of GPU context switching, and the cache footprint.

Security: A critical function of the hypervisor is to provide secure isolation between VMs [Garfinkel and Rosenblum 2005]. To fulfill this task, para and full virtualization frameworks, including LoGV [Gottschlag et al. 2013], GPUvm [Suzuki et al. 2014], gVirt [Tian et al. 2014], and gScale [Xue et al. 2016], prevent a VM from mapping the GPU address spaces of other VMs. Despite this protection mechanism, GPU virtualization frameworks remain vulnerable to denial-of-service (DoS) attacks where a malicious VM uninterruptedly submits a massive number of GPU commands to the backend and thus jeopardizes the whole system. To address this issue, gVirt resets hung GPUs and kills suspicious VMs after examining each VM's execution state. Unfortunately, this can cause a service suspension time to normal VMs. To avoid a GPU reset, a fine-grained access control mechanism is required that can delay the execution speed of a malicious VM before the VM threatens the system. Methods that adopt API remoting, including vCUDA [Shi et al. 2009] and VOCL [Xiao et al. 2012], do not implement isolation mechanisms and their security features need to be reinforced.

Fused CPU-GPU chips: Conventional systems with discrete GPUs have two major disadvantages: (1) data transfer overhead over the PCIe interface, which offers a low maximum bandwidth capacity (i.e., 16GB/s), and (2) programming effort to manage the separate data address spaces of the CPU and the GPU. To address these issues, fused CPU-GPU chips furnish shared memory space between the two processors. Examples include Intel's integrated CPU-GPU [Hammarlund et al. 2014], AMD's HSA architecture [Kyriazis 2012], and NVIDIA's unified memory coupled with NVLink [Foley

2014]. These new architectures can boost the performance of big data applications that require a significant communication volume between the two processors. gVirt [Tian et al. 2014] (Section 5.2) implemented full virtualization for Intel's GPUs, while Huang et al. [2016] (Section 5.1) developed a para virtualization solution for AMD's fused chips. However, these frameworks only focus on utilizing GPUs and need to adopt sophisticated scheduling algorithms that can utilize both processors by partitioning and load-balancing workloads differently for fused CPU-GPU architectures. Montella et al. [2016b] explored NVIDIA's unified memory to simplify memory management in GPU virtualization. However, the sole use of unified memory incurs non-negligible performance degradation in data-intensive applications [Li et al. 2015] because NVIDIA maintains its discrete GPU design and automatically migrates data between the host and the GPU. NVLink enables a high-bandwidth path between the GPU and the CPU (achieving between 80 and 200GB/s of bandwidth). A combination of NVIDIA's unified memory and NVLink is required to achieve high performance for data-intensive applications in GPU virtualization.

Power efficiency: Energy efficiency is currently a high research priority for GPU platforms [Mittal and Vetter 2015; Bridges et al. 2016]. Compared to a significant volume of research studying GPU power in non-virtualized environments, there is little work related to power and energy consumption studies in virtualized environments with GPUs. One example is pVOCL [Lama et al. 2013], which improves the energy efficiency of a remote GPU cluster by controlling peak power consumption between GPUs. Besides controlling power consumption of GPUs remotely, power efficiency is also required at the host side. Runtime systems that monitor different GPU usage patterns among VMs and dynamically adjust the GPU's power state according to the workload are an open area for further research.

Space sharing: Recent GPUs allow multiple processes or VMs to launch GPU kernels on a single GPU simultaneously [NVIDIA 2012]. This space-multiplexing approach can improve GPU utilization by fully exploiting SMs with multiple kernels. However, most GPU scheduling methods are based on time-multiplexing where GPU kernels from different VMs run in sequence on a GPU, which can lead to underutilization. A combination of the two approaches is required to achieve both high GPU utilization and fairness in GPU scheduling.

Live migration: The emergence of NVIDIA GRID [Herrera 2014] enables true hardware-based virtualization on GPUs. NVIDIA GRID allows a GPU device to expose itself as multiple separate virtual devices at the hardware level. However, as this technique bypasses the virtualization layer, it imposes challenges on live VM migration, which is one of the most important virtualization features in cloud computing. Novel mechanisms that can capture the hardware state and restore it in a remote machine are required to enable live migration in hardware-based virtualization.

Communication optimization: The API remoting approach offers GPU virtualization through intercepting GPU calls in a guest OS and forwarding them to the host OS or a remote machine. The API remoting approach has low overhead when applications are not GPU intensive. However, a range of recent GPU applications executes small (e.g., with a duration under 100μs) and repetitive GPU kernels [Menychtas et al. 2014]. These may introduce severe communication bottlenecks while forwarding GPU calls. With the advance of GPU microarchitectures, the kernel execution time is expected to drop even further, which will in turn inflate the communication overhead. Fully optimized communication methods are required to accommodate GPU-intensive applications in the API remoting approach.

9. CONCLUSION

Over the past few years, heterogeneous computing has gained significant attention as a new computing paradigm with potential to provide higher performance, higher resource utilization, and lower operational costs for HPC and cloud platforms. In cloud datacenters with heterogeneous nodes, GPU virtualization is a key enabling technology for the effective sharing of GPU devices between multiple tenants.

In this article, we presented an intensive survey of the research work on GPU virtualization techniques and their scheduling methods. We introduced the key research contributions in this area with representative studies in GPU virtualization, which range among API remoting, para and full virtualization, and hardware-assisted virtualization. In addition, we discussed GPU scheduling methods that can realize fair and effective GPU sharing in heterogeneous cloud computing. Finally, we suggested some future research directions that can address remaining challenges and advance the state of practice in GPU virtualization.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel virtualization technology for directed I/O. *Intel Technol. J.* 10, 3 (2006).

EC Amazon. 2010. Amazon elastic compute cloud (Amazon EC2). https://aws.amazon.com/ec2/.

AMD. 2009. R6xx_3D_Registers.pdf. Retrieved from http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/10/R6xx_3D_Registers.pdf. (2009).

Joshua Anderson, Aaron Keys, Carolyn Phillips, Trung Dac Nguyen, and Sharon Glotzer. 2010. HOOMD-blue, general-purpose many-body dynamics on the GPU. In *APS Meeting Abstracts*, Vol. 1. 18008.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and others. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. EECS Department Technical Report UCB/EECS-2006-183. University of California, Berkeley.

Andreu Badal and Aldo Badano. 2009. Accelerating monte carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit. *Med. Phys.* 36, 11 (2009), 4878–4880.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operat. Syst. Rev.* 37, 5 (2003), 164–177.

Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. 2011. GPU acceleration for support vector machines. In *12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS'11)*. TU Delft; EWI; MM; PRB, Delft, The Netherlands.

Can Basaran and Kyoung-Don Kang. 2012. Supporting preemptive task executions and memory copies in gpgpus. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 287–296.

Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. 2012. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 97–108.

Brahim Bensaou, Danny H. K. Tsang, and King Tung Chan. 2001. Credit-based fair queueing (CBFQ): A simple service-scheduling algorithm for packet-switched networks. *IEEE/ACM Trans. Network.* 9, 5 (2001), 591–604.

David Blythe. 2006. The direct3d 10 system. In *ACM Transactions on Graphics*, Vol. 25. ACM, 724–734.

Robert A. Bridges, Neena Imam, and Tiffany M Mintz. 2016. Understanding GPU power: A survey of profiling, modeling, and simulation methods. *ACM Comput. Surv.* 49, 3 (2016), 41.

Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Kaladhar Voruganti, and Garth R. Goodson. 2009. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the USENIX Annual Technical Conference*.

Adrián Castelló, Antonio J. Peña, Rafael Mayo, Pavan Balaji, and Enrique S. Quintana-Ortí. 2015. Exploring the suitability of remote GPGPU virtualization for the OpenACC programming model using rCUDA. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*. IEEE, 92–95.

Ethan Cerami. 2002. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc.

Charu Chaubal. 2008. The architecture of vmware esxi. *VMware White Pap.* 1, 7 (2008).

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization, 2009 (IISWC'09)*. IEEE, 44–54.

Hao Chen, Lin Shi, and Jianhua Sun. 2010. VMRPC: A high efficiency and light weight RPC system for virtual machines. In *Proceedings of the 2010 18th International Workshop on Quality of Service (IWQoS'10)*. IEEE, 1–9.

Yun Chan Cho and Jae Wook Jeon. 2007. Sharing data between processes running on different domains in para-virtualized xen. In *Proceedings of the International Conference on Control, Automation and Systems, 2007 (ICCAS'07)*. IEEE, 1255–1260.

Steve Crago, Kyle Dunn, Patrick Eads, Lorin Hochstein, Dong-In Kang, Mikyung Kang, Devendra Modium, Karandeep Singh, Jinwoo Suh, and John Paul Walters. 2011. Heterogeneous cloud computing. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*. IEEE, 378–385.

Chris I. Dalton, David Plaquin, Wolfgang Weidner, Dirk Kuhlmann, Boris Balacheff, and Richard Brown. 2009. Trusted virtual platforms: A key enabler for converged client devices. *ACM SIGOPS Operat. Syst. Rev.* 43, 1 (2009), 36–43.

Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 63–74.

A. Demers, S. Keshav, and S. Shenker. 1989. Design and analysis of a fair queuing algorithm. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'89)*, Vol. 89.

Roberto Di Lauro, Flora Giannone, Luigia Ambrosio, and Raffaele Montella. 2012. Virtualizing general purpose GPUs for high performance cloud computing: An application to a fluid simulator. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA'12)*. IEEE, 863–864.

Matthew Dixon, Sabbir Ahmed Khan, and Mohammad Zubair. 2014. Accelerating option risk analytics in R using GPUs. In *Proceedings of the High Performance Computing Symposium*. Society for Computer Simulation International, 24.

Yaozu Dong, Mochi Xue, Xiao Zheng, Jiajun Wang, Zhengwei Qi, and Haibing Guan. 2015. Boosting GPU virtualization performance with hybrid shadow page tables. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. 517–528.

Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel Distrib. Comput.* 72, 11 (2012), 1471–1480.

Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: Past, present and future. *Concurr. Comput.: Pract. Exper.* 15, 9 (2003), 803–820.

Micah Dowty and Jeremy Sugerman. 2009. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operat. Syst. Rev.* 43, 3 (2009), 73–82.

José Duato, Francisco D. Igual, Rafael Mayo, Antonio J. Peña, Enrique S. Quintana-Ortí, and Federico Silla. 2009. An efficient implementation of GPU virtualization in high performance clusters. In *European Conference on Parallel Processing*. Springer, 385–394.

José Duato, Antonio J. Peña, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Ortí. 2011. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing (HiPC'11)*. IEEE, 1–10.

José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortı. 2010a. Modeling the CUDA remoting virtualization behaviour in high performance networks. In *Proceedings of the 1st Workshop on Language, Compiler, and Architecture Support for GPGPU*.

José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. 2010b. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the 2010 International Conference on High Performance Computing and Simulation (HPCS'10)*. IEEE, 224–231.

José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. 2011. Performance of CUDA virtualized remote GPUs in high performance clusters. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*. IEEE, 365–374.

Ashok Dwarakinath. 2008. *A Fair-Share Scheduler for the Graphics Processing Unit*. Ph.D. Dissertation. Citeseer.

Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, and Ramón Doallo. 2013. General-purpose computation on GPUs for high performance cloud computing. *Concurr. Comput.: Pract. Exper.* 25, 12 (2013), 1628–1642.

Naila Farooqui, Rajkishore Barik, Brian T. Lewis, Tatiana Shpeisman, and Karsten Schwan. 2016. Affinity-aware work-stealing for integrated CPU-GPU processors. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 30.

Denis Foley. 2014. NVLink, pascal and stacked memory: Feeding the appetite for big data. Retrieved from Nvidia.com (2014).

Futuremark. 1998. 3DMark Benchmarks—See the Current Range of this Popular PC Graphics Card Test. Retrieved from http://www.futuremark.com/benchmarks/3dmark/all?_ga=1.168926249.987441096.1470653002. (1998).

Tal Garfinkel and Mendel Rosenblum. 2005. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'05)*.

Carl Gebhardt and Allan Tomlinson. 2010. *Challenges for Inter Virtual Machine Communication*. Technical Report. Citeseer.

Francisco Giunta, Raffaele Montella, Giuliano Laccetti, Florin Isaila, and F. Blas. 2011. A GPU accelerated high performance cloud computing infrastructure for grid computing based virtual environmental laboratory. *Adv. Grid Comput.* Lecture Notes in Computer Science. Vol. 6271. Springer, Berlin, Heidelberg, 35–43.

Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. 2010. A GPGPU transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing*. Springer, 379–391.

Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. 2015. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Comput. Phys. Commun.* 192 (2015), 97–107.

Robert P. Goldberg. 1974. Survey of virtual machine research. *Computer* 7, 6 (1974), 34–45.

Mathias Gottschlag, Martin Hillenbrand, Jens Kehne, Jan Stoess, and Frank Bellosa. 2013. LoGV: Low-overhead GPGPU virtualization. In *Proceedings of the 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC'13)*. IEEE, 1721–1726.

Simon Green. 2010. Particle simulation using cuda. *NVIDIA Whitepaper* 6 (2010), 121–128.

William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 22, 6 (1996), 789–828.

Khronos OpenCL Working Group et al. 2008. The opencl specification. Version 1, 29 (2008), 8.

Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2009. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. ACM, 17–24.

Haibing Guan, Jianguo Yao, Zhengwei Qi, and Runze Wang. 2015. Energy-efficient SLA guarantees for virtualized GPU in cloud gaming. *IEEE Trans.actions on Parallel Distrib. Syst.* 26, 9 (2015), 2434–2443.

Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2011. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*. 31.

Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. 2014. Haswell: The fourth-generation intel core processor. *IEEE Micro* 34, 2 (2014), 6–20.

Jacob Gorm Hansen. 2007. Blink: Advanced display multiplexing for virtualized applications. In *Proceedings of the SIGMM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'07)*.

Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. 2013. Efficient and scalable paravirtual I/O system. In *Proceedings of the USENIX Annual Technical Conference*. 231–242.

Nicholas Haydel, Sandra Gesing, Ian Taylor, Gregory Madey, Abdul Dakkak, Simon Garcia De Gonzalo, and Wen-Mei W. Hwu. 2015. Enhancing the usability and utilization of accelerated architectures via docker.

In *Proceedings of the 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC'15)*. IEEE, 361–367.

Alex Herrera. 2014. NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. Nvidia Corp (2014). http://www.nvidia.com/content/grid/vdi-whitepaper.pdf.

Hua-Jun Hong, Tao-Ya Fan-Chiang, Che-Run Lee, Kuan-Ta Chen, Chun-Ying Huang, and Cheng-Hsin Hsu. 2014. GPU consolidation for cloud games: Are we there yet?. In *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games*. IEEE Press, 3.

Yu-Ju Huang, Hsuan-Heng Wu, Yeh-Ching Chung, and Wei-Chung Hsu. 2016. Building a KVM-based hypervisor for a heterogeneous system architecture compliant system. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 3–15.

Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. 2002. Chromium: A stream-processing framework for interactive rendering on clusters. In *ACM Transactions on Graphics*, Vol. 21. ACM, 693–702.

Su Min Jang, Won Hyuk Choi, and Won Young Kim. 2013. Client rendering method for desktop virtualization services. *ETRI J.* 35, 2 (2013), 348–351.

Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. 2009. Predictive runtime code scheduling for heterogeneous architectures. In *High Performance Embedded Architectures and Compilers*. Springer, 19–33.

Heeseung Jo, Jinkyu Jeong, Myoungho Lee, and Dong Hoon Choi. 2013a. Exploiting GPUs in virtual machine for biocloud. *BioMed Res. Int.* 2013 (2013).

Hee Seung Jo, Myung Ho Lee, and Dong Hoon Choi. 2013b. GPU virtualization using PCI direct pass-through. In *Applied Mechanics and Materials*, Vol. 311. Trans Tech Publ, 15–19.

David Kanter. 2010. Intels sandy bridge microarchitecture. http://www.realworldtech.com/sandy-bridge/.

Ian Karlin, Jeff Keasler, and Rob Neely. 2013. Lulesh 2.0 updates and changes. Livermore, CA (2013). https://codesign.llnl.gov/lulesh.php.

Shinpei Kato, Scott Brandt, Yutaka Ishikawa, and R Rajkumar. 2011a. Operating systems challenges for GPU resource management. In *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. 23–32.

Shinpei Kato, Karthik Lakshmanan, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011b. Resource sharing in GPU-accelerated windowing systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*. IEEE, 191–200.

Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011c. RGEM: A responsive GPGPU execution model for runtime engines. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium (RTSS'11)*. IEEE, 57–66.

Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. 2011d. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*. 17.

Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A. Brandt. 2012. Gdev: First-class GPU resource management in the operating system.. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'11)*. 401–412.

Se Won Kim, Chiyoung Lee, MooWoong Jeon, Hae Young Kwon, Hyun Woo Lee, and Chuck Yoo. 2013. Secure device access for automotive software. In *Proceedings of the 2013 International Conference on Connected Vehicles and Expo (ICCVE'13)*. IEEE, 177–181.

David B. Kirk and W. Hwu Wen-mei. 2012. *Programming Massively Parallel Processors: A Hands-on Approach*. Newnes.

Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: The Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Vol. 1. 225–230.

Nasser A. Kurd, Subramani Bhamidipati, Christopher Mozak, Jeffrey L. Miller, Timothy M. Wilson, Mahadev Nemani, and Muntaquim Chowdhury. 2010. Westmere: A family of 32nm IA processors. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference (ISSCC'10)*.

Maxim A. Kuzkin and Alexander G. Tormasov. 2011. Method and system for remote device access in virtual environment. (issued date: July 5 2011). Patent No. 7,975,017. Filed date: Feb 25, 2009.

George Kyriazis. 2012. Heterogeneous system architecture: A technical review. In *Proceedings of the AMD Fusion Developer Summit* (2012).

Giuliano Laccetti, Raffaele Montella, Carlo Palmieri, and Valentina Pelliccia. 2013. The high performance internet of things: Using GVirtuS to share high-end GPUs with ARM based cluster computing nodes. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 734–744.

H. Andrés Lagar-Cavilla, Niraj Tolia, Mahadev Satyanarayanan, and Eyal De Lara. 2007. VMM-independent graphics acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*. ACM, 33–43.

Palden Lama, Yan Li, Ashwin M. Aji, Pavan Balaji, James Dinan, Shucai Xiao, Yunquan Zhang, Wu-chun Feng, Rajeev Thakur, and Xiaobo Zhou. 2013. pVOCL: Power-aware dynamic placement and migration in virtualized GPU environments. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS'13)*. IEEE, 145–154.

Michael Larabel and M. Tippett. 2011. Phoronix test suite. https://www.phoronix-test-suite.com.

Chiyoung Lee, Se-Won Kim, and Chuck Yoo. 2016. VADI: GPU virtualization for an automotive platform. *IEEE Trans. Industr. Inf.* 12, 1 (2016), 277–290.

Gunho Lee and Randy H. Katz. 2011. Heterogeneity-aware resource allocation and scheduling in the cloud.. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'11)*.

Teng Li, Vikram K. Narayana, Esam El-Araby, and Tarek El-Ghazawi. 2011. GPU resource sharing and virtualization on high performance computing systems. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*. IEEE, 733–742.

Teng Li, Vikram K. Narayana, and Tarek El-Ghazawi. 2012. Accelerated high-performance computing through efficient multi-process GPU resource sharing. In *Proceedings of the 9th Conference on Computing Frontiers*. ACM, 269–272.

Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An evaluation of unified memory technology on nvidia gpus. In *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*. IEEE, 1092–1098.

Tyng-Yeu Liang and Yu-Wei Chang. 2011. GridCuda: A grid-enabled CUDA programming toolkit. In *Proceedings of the 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA'11)*. IEEE, 141–146.

Christos Margiolas and Michael F. P. O'Boyle. 2016. 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, 82–93.

Konstantinos Menychtas, Kai Shen, and Michael L. Scott. 2013. Enabling OS research by inferring interactions in the black-box GPU stack. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. 291–296.

Konstantinos Menychtas, Kai Shen, and Michael L. Scott. 2014. Disengaged scheduling for fair, protected access to fast computational accelerators. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 301–316.

Alexander M. Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. 2011. Shadowfax: Scaling in heterogeneous cluster systems via GPGPU assemblies. In *Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing*. ACM, 3–10.

Sparsh Mittal and Jeffrey S. Vetter. 2015. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Comput. Surv.* 47, 2 (2015), 19.

Raffaele Montella, Giuseppe Coviello, Giulio Giunta, Giuliano Laccetti, Florin Isaila, and Javier Garcia Blas. 2011. A general-purpose virtualization service for HPC on cloud computing: An application to GPUs. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 740–749.

Raffaele Montella, Giulio Giunta, and Giuliano Laccetti. 2014. Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. *Cluster Comput.* 17, 1 (2014), 139–152.

Raffaele Montella, Giulio Giunta, Giuliano Laccetti, Marco Lapegna, Carlo Palmieri, Carmine Ferraro, and Valentina Pelliccia. 2016a. Virtualizing CUDA enabled GPGPUs on ARM clusters. In *Parallel Processing and Applied Mathematics*. Springer, 3–14.

Raffaele Montella, Giulio Giunta, Giuliano Laccetti, Marco Lapegna, Carlo Palmieri, Carmine Ferraro, Valentina Pelliccia, Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. 2016b. On the virtualization of CUDA based GPU remoting on ARM and X86 machines in the GVirtuS framework. *Int. J. Parallel Program.* (2016), 1–22. DOI:http://dx.doi.org/10.1007/s10766-016-0462-1

Christopher Niederauer, Mike Houston, Maneesh Agrawala, and Greg Humphreys. 2003. Non-invasive interactive visualization of dynamic architectural environments. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*. ACM, 55–58.

Nvidia. 2007a. CUDA Code Samples—NVIDIA Developer. Retrieved from https://developer.nvidia.com/cuda-code-samples.

NVIDIA. 2012. HyperQ Example. Retrieved from http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

NVIDIA. 2016a. GP100 Pascal Whitepaper. Retrieved from https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

NVIDIA. 2016b. GPU Cloud Computing Service Providers—NVIDIA. Retrieved from http://www.nvidia.com/object/gpu-cloud-computing-services.html.

CUDA Nvidia. 2007b. *Compute Unified Device Architecture Programming Guide*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

Katsuhiko Ogata. 1995. *Discrete-Time Control Systems*. Vol. 2. Prentice Hall, Englewood Cliffs, NJ.

Masahiro Oikawa, Atsushi Kawai, Keigo Nomura, Koichi Yasuoka, Kenichi Yoshikawa, and Tetsu Narumi. 2012. DS-CUDA: A middleware to use many GPUs in the cloud environment. In *Proceedings of the 2012 SC Companion to High Performance Computing, Networking, Storage and Analysis (SCC)*. IEEE, 1207–1214.

Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. 2012. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. *Presented in the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

Sankaralingam Panneerselvam and Michael M Swift. 2012. Operating systems should manage accelerators. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*.

Stan Park and Kai Shen. 2012. FIOS: A fair, efficient flash I/O scheduler. In *Proceedings of the 10th USENEX Conference on File and Storage Technologies (FAST'12)*. 13.

PathScale. 2012. pathscale/pscnv. Retrieved from https://github.com/pathscale/pscnv.

Sagar Patni, Jobin George, Pratik Lahoti, and Jibi Abraham. 2015. A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial. In *Proceedings of the 2015 1st International Conference on Next Generation Computing Technologies (NGCT'15)*. IEEE, 6–9.

David Patterson. 2009. The top 10 innovations in the new NVIDIA fermi architecture, and the top 3 next challenges. *NVIDIA Whitepaper* 47 (2009).

Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. 2014. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Comput.* 40, 10 (2014), 574–588.

Ferran Pérez, Carlos Reaño, and Federico Silla. 2016. Providing CUDA acceleration to KVM virtual machines in InfiniBand Clusters with rCUDA. In *Distributed Applications and Interoperable Systems*. Springer, 82–95.

Antoine Petitet. 2004. HPL-A portable implementation of the high-performance Linpack benchmark for distributed-memory computers. Retrieved from http://www.netlib-.org/-benchmark/hpl/.

James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kale, and Klaus Schulten. 2005. Scalable molecular dynamics with NAMD. *J. Comput. Chem.* 26, 16 (2005), 1781–1802.

Steve Plimpton, Paul Crozier, and Aidan Thompson. 2007. LAMMPS-large-scale atomic/molecular massively parallel simulator. *Sandia National Laboratories* 18 (2007). http://lammps.sandia.gov.

Javier Prades, Carlos Reaño, and Federico Silla. 2016. CUDA acceleration for Xen virtual machines in infiniband clusters with rCUDA. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 35.

Zhengwei Qi, Jianguo Yao, Chao Zhang, Miao Yu, Zhizhou Yang, and Haibing Guan. 2014. VGRIS: Virtualized GPU resource isolation and scheduling in cloud gaming. *ACM Trans. Arch. Code Optimiz.* 11, 2 (2014), 17.

Adit Ranadive and Bhavesh Davda. 2012. Toward a paravirtual vRDMA device for VMware ESXi guests. *VMware Techn. J. 2012* 1, 2 (2012).

Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. 2011. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. ACM, 217–228.

Carlos Reaño, Rafael Mayo, Enrique S. Quintana-Ortí, Federico Silla, José Duato, and Antonio J. Peña. 2013. Influence of InfiniBand FDR on the performance of remote GPU virtualization. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing (CLUSTER'13)*. IEEE, 1–8.

Carlos Reaño, A. J. Pea, Federico Silla, José Duato, Rafael Mayo, and Enrique S. Quintana-Ortí. 2012. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. In *Proceedings of the 2012 19th International Conference on High Performance Computing (HiPC'12)*. IEEE, 1–10.

Carlos Reaño and Federico Silla. 2015. A performance comparison of CUDA remote GPU virtualization frameworks. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*. IEEE, 488–489.

Carlos Reaño, Federico Silla, Adrián Castelló, Antonio J . Peña, Rafael Mayo, Enrique S Quintana-Ortí, and José Duato. 2015a. Improving the user experience of the rCUDA remote GPU virtualization framework. *Concurr. Comput.: Pract. Exper.* 27, 14 (2015), 3746–3770.

Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. 2015b. Local and remote GPUs perform similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*. ACM, 4.

Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 233–248.

Eric E. Schadt, Michael D. Linderman, Jon Sorenson, Lawrence Lee, and Garry P. Nolan. 2011. Cloud and heterogeneous computing solutions exist today for the emerging big data problems in biology. *Nat. Rev. Genet.* 12, 3 (2011), 224–224.

Dipanjan Sengupta, Raghavendra Belapure, and Karsten Schwan. 2013. Multi-tenancy on GPGPU-based servers. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*. ACM, 3–10.

Dipanjan Sengupta, Anshuman Goswami, Karsten Schwan, and Krishna Pallavi. 2014. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 513–524.

Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R. Trott, Greg Scantlen, and Paul S. Crozier. 2011. The development of Mellanox/NVIDIA GPUDirect over InfiniBanda new model for GPU to GPU communications. *Comput. Sci. Res. Dev.* 26, 3-4 (2011), 267–273.

Haitao Shan, Kevin Tian, Eddie Dong, and David Cowperthwaite. 2013. XenGT: A software based intel graphics virtualization solution. *Proceedings of the Xen Project Developer Summit*.

Ryan Shea and Jiangchuan Liu. 2013. On GPU pass-through performance for cloud gaming: Experiments and analysis. In *Proceedings of the 2013 12th Annual Workshop on Network and Systems Support for Games (NetGames'13)*. IEEE, 1–6.

Lin Shi, Hao Chen, and Jianhua Sun. 2009. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, 2009 (IPDPS'09)*. IEEE, 1–11.

Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2012. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.* 61, 6 (2012), 804–816.

Weidong Shi, Yang Lu, Zhu Li, and Jonathan Engelsma. 2011. SHARC: A scalable 3D graphics virtual appliance delivery framework in cloud. *J. Netw. Comput. Appl.* 34, 4 (2011), 1078–1087.

Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Trans. Netw.* 4, 3 (1996), 375–385.

Abraham Silberschatz, Peter B. Galvin, Greg Gagne, and A. Silberschatz. 1998. *Operating System Concepts*. Vol. 4. Addison-Wesley, Reading, MA.

Jike Song, Zhiyuan Lv, and Kevin Tian. 2014. KVMGT: A full GPU virtualization solution. In *KVM Forum 2014*. http://www.linux-kvm.org/page/KVM_Forum_2014.

John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why not virtualizing GPUs at the hypervisor?. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 109–120.

Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2016. Gpuvm: Gpu virtualization at the hypervisor. *IEEE Trans. Comput.* 65, 9 (2016), 2752–2766.

Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 193–204.

Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A full GPU virtualization solution with mediated pass-through. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*.

Tsan-Rong Tien and Yi-Ping You. 2014. Enabling OpenCL support for GPGPU in Kernel-based Virtual Machine. *Softw.: Pract. Exper.* 44, 5 (2014), 483–510.

Top500. 2016. TOP500 Supercomputer Sites. Retrieved from https://www.top500.org/list/2016/06/.

Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.

Leendert Van Doorn. 2006. Hardware virtualization trends. In *Proceedings of the 2nd International ACM/Usenix Conference on Virtual Execution Environments*, Vol. 14. 45–45.

Stephen J. Vaughan-Nichols. 2006. New approach to virtualization is a lightweight. *Computer* 39, 11 (2006).

Anthony Velte and Toby Velte. 2009. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc.

M. S. Vinaya, Naga Vydyanathan, and Mrugesh Gajjar. 2012. An evaluation of CUDA-enabled virtualization solutions. In *Proceedings of the 2012 2nd IEEE International Conference on Parallel Distributed and Grid Computing (PDGC'12)*. IEEE, 621–626.

Lan Vu, Hari Sivaraman, and Rishi Bidarkar. 2014. GPU virtualization for high performance general purpose computing on the ESX hypervisor. In *Proceedings of the High Performance Computing Symposium*. Society for Computer Simulation International, 2.

John Paul Walters, Andrew J. Younge, Dong In Kang, Ke Thia Yao, Mikyung Kang, Stephen P. Crago, and Geoffrey C. Fox. 2014. GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications. In *Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing (CLOUD'14)*. IEEE, 636–643.

Bin Wang, Ruhui Ma, Zhengwei Qi, Jianguo Yao, and Haibing Guan. 2016. A user mode CPU–GPU scheduling framework for hybrid workloads. *Future Gener. Comput. Syst.* 63 (2016), 25–36.

Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. 2008. XenLoop: A transparent high performance inter-vm network loopback. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*. ACM, 109–118.

Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*. ACM, 21–30.

Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU architecture. *IEEE Micro* 2 (2011), 50–59.

Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. 1999. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc.

Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. 2011. Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'11)*. IEEE, 195–204.

Xenproject. 2016. Xen Project Release Features. Retrieved from https://wiki.xenproject.org/wiki/Xen_Project_Release_Features.

Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. 2012. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Proceedings of the Innovative Parallel Computing (InPar'12)*. IEEE, 1–12.

X.OrgFoundation. 2011. Nouveau: Accelerated Open Source driver for nVidia cards. Retrieved from https://nouveau.freedesktop.org/wiki/.

Mochi Xue, Kun Tian, Yaozu Dong, Jiajun Wang, Zhengwei Qi, Bingsheng He, and Haibing Guan. 2016. gScale: Scaling up GPU virtualization with dynamic sharing of graphics memory space. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*.

Chao-Tung Yang, Jung-Chun Liu, Hsien-Yi Wang, and Ching-Hsien Hsu. 2014. Implementation of GPU virtualization using PCI pass-through mechanism. *J. Supercomput.* 68, 1 (2014), 183–213.

Chao-Tung Yang, Hsien-Yi Wang, and Yu-Tso Liu. 2012a. Using pci pass-through for gpu virtualization with cuda. In *Network and Parallel Computing*. Springer, 445–452.

Chao-Tung Yang, Hsien-Yi Wang, Wei-Shen Ou, Yu-Tso Liu, and Ching-Hsien Hsu. 2012b. On implementation of GPU virtualization using PCI pass-through. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom'12)*. IEEE, 711–716.

Chih-Yuan Yeh, Chung-Yao Kao, Wei-Shu Hung, Ching-Chi Lin, Pangfeng Liu, Jan-Jan Wu, and Kuang-Chih Liu. 2013. GPU virtualization support in cloud system. In *International Conference on Grid and Pervasive Computing*. Springer, 423–432.

Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. 2015. VirtCL: A framework for OpenCL device abstraction and management. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 161–172.

Andrew J. Younge and Geoffrey C. Fox. 2014. Advanced virtualization techniques for high performance cloud cyberinfrastructure. In *Proceedings of the 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*. IEEE, 583–586.

Andrew J. Younge, John Paul Walters, Stephen Crago, and Geoffrey C. Fox. 2014. Evaluating GPU passthrough in Xen for high performance cloud computing. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW'14)*. IEEE, 852–859.

Andrew J. Younge, John Paul Walters, Stephen P. Crago, and Geoffrey C. Fox. 2015. Supporting high performance molecular dynamics in virtualized clusters using IOMMU, SR-IOV, and GPUDirect. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 31–38.

Chao Zhang, Jianguo Yao, Zhengwei Qi, Miao Yu, and Haibing Guan. 2014. vgasa: Adaptive scheduling algorithm of virtualized gpu resource in cloud gaming. *IEEE Trans. Parallel Distrib. Syst.* 25, 11 (2014), 3036–3045.

Youhui Zhang, Peng Qu, Jiang Cihang, and Weimin Zheng. 2016. A cloud gaming system based on user-level virtualization and its resource scheduling. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (2016), 1239–1252.

Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: A preemptive execution system for GPGPU computing. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 87–97.

Received October 2016; revised February 2017; accepted March 2017