GPGPU VIRTUALIZATION TECHNIQUES A COMPARATIVE SURVEY

M. Alyas¹, Hamid Hassan¹

¹Department of Computer Sciences, School of Systems and Technology, University of Management & Technology, Lahore, Pakistan.
f2016279013@umt.edu.pk

Abstract. The Graphic Processing Units (GPU) are being adopted in many High Processing Computing (HPC) facilities because of their massively parallel and extraordinary computing power, which makes it possible to accelerate many general purpose implementations from different domains. A general-purpose GPU (GPGPU) is a GPU that performs computations that were traditionally handled by central processing unit (CPU) to accelerate applications along with handling traditional computations for graphics rendering. However, GPUs have some limitations, such as increased acquisition costs as well as larger space requirements, more powerful energy supplies, and their utilization is usually low for most workloads. That results in the need of GPU virtualization to maximize the use of an acquired GPU to share between the virtual machines for optimal use, to reduce power utilization and minimize the costs. This study comparatively reviews the recent GPU virtualization techniques including API remoting, para, full and hardware based virtualization, targeted for general-purpose accelerations.

Keywords: GPU; Virtualization; GPGPU.

1. Introduction & Background. Since the start of 21st century, HPC programmers and researchers have embraced a new computing model combining two architectures: (i) multi-core processors with powerful and general-purpose cores, and (ii) many-core application accelerators. The dominant example of accelerators is GPU, with a large number of processing elements/cores, which can boost performance of HPC applications using higher level parallel processing paradigm [3]. Because of the high computational cost of current compute-intensive implementations, gpus are considered as an efficient mean of accelerating the executions of such application by utilizing the parallel programming paradigm. Present-day gpus are excellent at rendering graphics, and their highly parallel architecture gives them an edge over traditional cpus to be more efficient for a variety of different compute-intensive algorithms [4]. High-end computing units comes with gpus that include very large number of small computing units (cores) supported with a high bandwidth to their private embedded memory [1]. HPC has become a must have technology for most demanding applications in scientific fields (high-energy physics, computer sciences, weather, climate, computational chemistry, medical, bio-informatics and genomics), engineering (computational fluid dynamics, energy and aerospace), crypto, security, economy (market simulations, basket analysis and predictive analysis), creative arts and designs (compute-intensive image processing, very large 3d rendering and motion creative) and graphics acceleration [2]

Traditionally, the general-purpose computations are performed by central processing unit (CPU) like additions, subtractions, multiplications, divisions, shifts, matrix and other similar operations, but with the growth of GPU programming languages such as compute unified device architecture (CUDA), openacc, [5] opengl and opencl and high computation power of GPU [5] has made it preferred choice of HPC programmers. In GPGPU-accelerated implementations, the performance is usually boosted by dividing the application parts into compute-intensive and the rest, and compute-intensive portion is off-loaded to GPU for parallel execution [1], to carry out this operation, programmers have to define which portion of the application will be executed by CPU and which functions (or kernels) will be executed by the GPGPU [1].

Fig. 1. Architecture of the system with CPU and a discrete GPU.

Figure. 1. shows the architecture of a heterogeneous system with a CPU and discrete GPGPU. A GPGPU has many streaming multiprocessors (SM). Each SM has 32 (can vary) processing cores, L1 cache and a low latency shared memory. Every computing core has its own local registers, an integer arithmetic logic unit (ALU), a floating point unit (FPU), and several special function units (SFUs), which executes special set of instructions e.g. special math & scientific operations. Memory management unit (MMU) of a GPU offers virtual memory spaces for GPGPU-accelerated implementations. MMU resolves GPU memory address to the physical address by using its own page table of the application. This ensures that application can only access its own address space.

Discrete GPU is connected to the physical (host) machine through PCI Express interface. The interaction of GPU and CPU is done through memory mapped I/O (MMIO). The CPU can access the GPU's registers and memory via MMIO interface. The application's required GPU operations are submitted into buffer associated to the application's command submission channel (hardware unit in GPU), which is accessible to CPU through MMIO. Direct memory access (DMA) engine can be used to transfer data between host and GPU memories [3]. Unlike traditional multi-core processors, GPUs exhibits a basically different approach to execute parallel applications [6]. GPUs are throughput-oriented, with numerous simple processing cores, and a high-bandwidth memory architecture. This architecture empower maximizing the throughput of applications with a high level of concurrent processing, which are split into a large amount of threads executing on different ends in the allocated program space. This architecture allows hiding the latency of the other queued threads by the hardware scheduler, when some threads are waiting in long latency to complete operations (arithmetic or memory access) [7].

Regardless of increasingly more cores, multi-core processors architecture still focus on decreasing latency in sequential applications by means of utilizing state-of-the-art control logics and larger cache memories. In contrast, GPUs using parallelism paradigm, speed-up the execution of applications with heaps of simple cores and excessive memory bandwidth structure. Heterogeneous systems, using multi-core CPUs and GPUs together, can boost performance of HPC applications, offering better control and parallelism. Traditional CPUs, generally use control logics and larger caches to effectively handle conditional branches, deadlocks, pipelining stalls, and poor data locality, while present-day GPGPUs can process intensive workloads, have larger Static Random Access Memory (SRAM) based local memories, and have some extra functionalities as of conventional processors, but mainly focus on ensuring higher level of parallel processing and memory bandwidth [3].

Cloud infrastructure can utilize heterogeneous systems to lower the overall operational cost of acquisition with advantage of better performance and power efficiency [8, 9]. A cloud platform allows users to run compute-intensive implementations over heterogeneous compute-nodes without acquiring large-scale clusters, which also save them from maintenance hassle and huge-costs. Additionally, heterogeneous nodes gives an edge over homogeneous nodes, with the freedom to have compute-intensive programs computed by either traditional CPUs or highly-parallel GPUs depending on the level of parallelism required. Collectively, these

attractive advantages are encouraging cloud platform service providers to add GPUs to the cloud instances and offer heterogeneous programming facilities to the users to achieve higher performance as needed [10, 11, 12].

Fig. 2. Host System Virtualization.

System virtualization is a model that allows to concurrently run diverse operating systems on a single physical machine, with the goal to attain optimal resource sharing of physical machine in private and shared computing environments, popular example is cloud computing. The virtualization software is called hypervisor (a.k.a. virtual machine monitor (VMM)), that virtualizes physical machine resources i.e. CPU, memory, and I/O resources. A Virtual Machine (VM) use virtualized resources and have a guest OS installed. The guest operating system runs on VM similarly as the VM were a physical machine. Some well-known hypervisors being used in production environments for private and shared cloud are VMware ESXi [13], Kernel-based virtual machine (KVM) [14], Hyper-V [15], and Xen [16]. Fig. 2. shows the virtualization of a host machine through hypervisor.

System resources virtualization can be classified in three main categories: (i) full-virtualization (ii) para-virtualization (iii) hardware based virtualizations. In full-virtualization setup, the guest OS doesn't know that it's a guest OS, hence directly issues the resources related calls to underlying hardware including CPU, I/O and memories. The hypervisor translates those privileged calls into binary format for the guest OS. The benefit of full-virtualization is that guest OS doesn't need to be modified in order to run in virtual environment, but may have performance bottlenecks due to direct interaction with host machine hardware. In para-virtualization approach, the guest OS is modified for system calls and knows that it's a guest OS, hence issues hypercalls (to communicate with hypervisor) when needs to interact with hardware resources. As compared to full-virtualization, para-virtualization has lower overheads and better performance. The downside of this approach is that it requires changes in guest operating system, which can be hectic as drivers and OS updates are released quite often. Hardware-based virtualizations needs to be capable to run privileged system calls from guest OS. Generally, two modes for virtualization are there: (i) guest (ii) root, where guest mode is for OS and root mode for hypervisor. Upon a privileged call from OS, the control is transferred to hypervisor running as root mode, that process the instruction and control is returned to the guest. The mode changes known as VM Exit (guest to root) and VM Entry (root to guest). This approach doesn't need to modify the guest OS and offers better performance as compared to full-virtualization [3].

Resources virtualization plays key role in Cloud computing technology. Virtualization programs enable creation of virtual environment as VM, which gives freedom of operating system choice, ensures optimal use of resources at reduced cost. Virtualized systems are always supported with techniques to multiplex available physical machine resources. Full virtualization solutions are already available for common physical resources including CPUs, memories, and peripheral devices, since there has been huge amount of research in this area since early 1960s. [17]. On the other hand, GPU virtualization is a relatively new field of research and a challenging task. The main barrier to GPU virtualization is the implementations of GPU drivers, which are not available open for customizations because of intellectual property protections. Moreover, there are no GPU design standards are regulated, and GPGPU manufacturers have been providing variety of architectures, which supports different degrees of virtualization. Because of such reasons, usual virtualization methods are not directly applicable for GPU virtualization [3].

The GPU programming models/APIs include CUDA [18], OpenGL, OpenCL and Direct3D, and most virtualization methods target these models/APIs for GPU virtualization. CUDA [18] is NVIDIA owned programming model for parallel computations, which enables programmers to utilize CUDA-enabled GPUs for general-purpose parallel computations, ultimately converting graphics cards into GPGPU. OpenGL [19] is an application programming interface (API) library, which is used to access GPU hardware for graphics acceleration. Its special use includes video games, images processing & rendering, and visualization needs for diverse applications. OpenGL offers hardware independent API, which can be used to interact with variety of graphics cards, despite of vendor system software. OpenCL [20] is a library for parallel computations which works over heterogeneous environments. OpenCL programming language offers syntax similar to C language known as OpenCL C to code computing kernels, and set of APIs to launch kernels into an OpenCL device (e.g. GPU) and facilitates data transfer management between device and host memories. The main difference in OpenCL and

CUDA is that CUDA is only supported by NVIDIA GPUs, while OpenCL can execute applications over variety of accelerators (e.g. GPU, regardless of vendor) and CPUs. Direct3D [21] is a Microsoft owned graphics API for Windows. This API can be used to accelerate 3D graphics for performance hungry applications e.g. games. It provides general abstraction layer to interact with GPU hardware, and offers advanced graphics features such as buffering and anti-aliasing. The GPU applications can be classified into two categories a) conventional graphic-accelerations b) general purpose computing. Graphic acceleration includes rendering of 2D, 3D graphics and simulations, while general-purpose computing involves parallel computations. The rapidly increasing demand of GPU for general-purpose computing requires the availability of GPU instances in cloud infrastructure services. This study comparatively reviews the recent available GPU virtualization techniques & strategies for general-purpose computing i.e. GPGPU Virtualization techniques. **Gpgpu Virtualization Techniques:** This section briefly describes the recent GPU virtualization techniques. In terms of implementation approach, the GPU virtualization can be classified into three classes: (i) API Remoting (ii) Para & Full (iii) Hardware-based virtualization methods.

2.1.1 Api Remoting Gpgpu Virtualization: Application Programming Interface (API) is a method to interact with remote providers for different types of requests fulfillments. In GPU virtualization perspective, API Remoting is a higher level frontend approach where a GPGPU related request is forwarded to remote (or host) server equipped with GPGPU, that process the request and send the results back to VM. Without source-code of GPU drivers, it's difficult to virtualize them at driver level, thus API remoting allows virtualization at libraries level. Fig. 3. illustrates the process of interaction between VM and remote server for GPGPU related requests, the steps shows that a request is initiated by GPU application, is intercepted by wrapper of programming model on VM to frontend layer, that is transferred to remote host OS, that dispatch the request to programming model handler, that further is transferred to GPU driver to be executed on GPU, and in reverse process the response is returned to GPU application.

Fig. 3. API Remoting GPU Interaction.

The API remoting approach allows to write portable GPU based applications and is easier to setup and integrate [22]. The advantages of this approach includes; easy setup, highly-portable applications, dynamic linking and wide range of supported GPU models and architectures. Generally this virtualization layer runs in user-space, in result, such library calls may bypass hypervisor. The restraints are to keep wrapper libraries updated in order to comply with vendor updates [24], and difficult to have fundamental virtualization features including fault-tolerance and live migration [25]. Since launch of NVIDIA's CUDA, GPUs are being used widely as GPGPU to accelerate applications as CUDA allows programmers to exploit GPU's power for general-purpose computations, which increased the need of GPGPU's virtualization to be used in cloud environment and sharing between VMs [3].

GVIM [26] enables virtualization at CUDA API level which can be implemented on Xen-based hypervisor. This allows a guest machine to use GPU attached to the host, an Interpose Library to access CUDA from guest OS, a frontend driver to communicate with the host backend driver. GVIM utilize memory allocated by Xenstore [27] instead of using network transfer for data-intensive application, and concentrates on efficient sharing of heaps of data between host and guest VM. Furthermore, it use shared memory concept to share address spaces of application on guest VM & host GPU, which eliminates the need to copy data between user and kernel spaces, that ultimately boost the processing performance. vCUDA [28] allows to virtualize GPU in the Xen hypervisor. It offers a CUDA library wrapper and virtual GPU (vGPU) to the VM and vCUDA library at host level. The guest

OS use CUDA wrapper to generate API call to the host as a client, and wrapper library creates vGPUs to give full view of host GPUs. vCUDA stub at host level works as server to execute API requests for GPU access. It use XML-RPC [29] channel for efficient communicate between host vCUDA and guest VM. In recent release [30], vCUDA is deployed in KVM using VMRPC [31] with VMCHANNEL [32]. Due to XML-RPC network transmission overhead, VMRPC use shared-memory space between VM and host OS. To minimize latency between virtual machines, VMCHANNEL allows an asynchronous message system in KVM. vCUDA utilize Lazy RPC for GPU calls that can be delayed, and process them in a batch to boost performance by reducing context switching overheads. CUDA [34] targets remote GPGPU acceleration to have GPU related computations performed over a remote host. It implements virtual CUDA-complied layers to execute GPU related calls over remote host without involving hypervisor. More precisely, rCUDA offers wrapper library for CUDA API on guest VM to generate and send GPU related calls to the remote GPU host. The guest VM and GPU host use TCP/IP communication protocol to interact with each other. rCUDA performance may be limited by network overloads when large number of VMs are accessing remote GPU server concurrently. To eliminate network issues, rCUDA offers application-level interaction mechanism [33]. Recent improvements in rCUDA provides support multithreaded applications, lower overheads to use GPU within local cluster, and allows an application running on a VM to utilize all the available GPUs within the cluster to, which truly maximize the performance of HPC applications [34].

GVirtuS [36] offers support for many hypervisors e.g. VMWare, Xen and KVM by establishing a transparent layer between VM and host. It uses usual approach as CUDA wrapper, frontend and backend drivers. The VM have frontend driver, while backend driver operates in host machine, both drivers communicates through a hypervisor specific communicator. Since the GPU virtualization efficiency depends on communication between guest VM and GPU host, to gain maximum performance, GVirtuS utilize communication channel offered by hypervisors i.e. VMSocket is used for KVM, XenLoop [37] for Xen and VMCI [38] for VMware. Later, GVirtuS introuced VMShm communicator [39] to improve communication using shared memory paradigm, which reserves POXIS shared memory block on the host to allow memory mapping for communication between VM and backend host. GVirtuS also support remote GPU accelerations through TCP/IP-based channel. It now also support x86 and ARM CPUs over cloud clusters and appliances as well as local work stations [40].

GVM [41] model estimates the performance of GPU-based implementation and verify it by own virtualization framework consisting of (i) user process APIs, (ii) virtual shared memory and (iii) GPU Virtualization Manager (GVM). The guest OS is modified to include APIs, which programmers use to make calls virtual GPUs. The GVM operates in host which initialize the vGPUs, receives guest requests and pass them to discrete GPU. POSIX shared memory is used for communication between host and guest operating systems.

Pegasus [42] is an advances the GViM [26], which operates at hypervisor level and share accelerators among multiple VMs. It introduced the concept of an accelerator virtual CPU (aVCPU) which is similar to virtual CPUs, and it manifests the state of a guest executing calls over GPGPU. aVCPU is first-class schedulable component that have call buffer at guest VM, polling process on GPU host, and runtime API for CUDA. Guest OS calls for GPU are stored in shared buffer between guest and host, a polling thread selects the GPU calls from buffer and pass to CUDA to execute on physical GPU. By making its interface similar to CUDA API, Pegasus provide support for existing application for GPGPU needs [42].Shadowfax [43] further advances Pegasus [42] by handling the limitations that Pegasus powered application face when in need of higher GPU computational powers, as Pegasus can utilize only local GPUs while combining additional remote nodes with local GPU can boost performance. Shadowfax offers concept of GPGPU clusters, which can be used by variety of virtual solutions according to GPGPU application need. It use Pegasus's concept to empower applications for local GPU, and for remote host, it creates fake VM through remote server thread, which have a buffer to queue calls, and for each VM, a polling process in remote host. Shadowfax use batching to minimize remote communication overheads for GPU requests and data. VOCL [44] offers GPGPU virtualization similar to rCUDA [34] but for OpenCL implementations. It utilize remote GPUs to accelerate virtual devices that support OpenCL. It implements wrapper library on guest VM and VOCL process on GPU host. The library on client end sends the request to remote host, where it's processed by VOCL proxy process. A rich and dynamic MPI [45] channel is used for communication between client and remote host. DS-CUDA [46] too offers remote GPU virtualization similar to rCUDA [34]. DS-CUDA consists of a compiler and a server. Compiler is used to generate wrapper functions for CUDA API calls, and server to receive calls/data and execute them on remote host. It uses RPC or IniniBand channels for

communication. In contrast to other similar remote virtualization approaches, DS-CUDA performs redundant calculations over two different GPUs in a cluster, to ensure the integrity that the computed outcome is correct. Enhanced XMLRPC [47] model is based on XML-RPC and CUDA. It's similar approach as of rCUDA [34] for remote GPU virtualization but with optimized data encapsulation concept. In this paradigm, the XMLRPC data is optimized using XMLRPC-String method before sending over the network to the remote GPU host. It focus on keeping the number of packets at minimum with optimized packet size. It claims to boost performance by 4.5X to 7X, with and without pre-processing respectively.

FairGV [48] model focus on weighted fair sharing and utilization of the GPU in mixed workloads. It introduce trap-less architecture for GPU processing, queuing methods and co-scheduling policies. Trap-less architecture helps boosting the performance, since trapping in OS kernel adds execution overhead and impact the performance. FairGV can interact with GPU calls directly from user space without hypervisor or kernel trapping. In FairGV, the guest VM sends request to host to be processed and data is shared through shared memory between guest and host OS. The VM frontend polls the response ring for the result of request. The major difference between FairGV and other similar solution is that FairGV is trap-less, it also offers queuing and scheduling policies in combination to boost performance.

2.2 Para & Full Gpgpu Virtualization: API Remoting gives ability to virtualize GPU with less effort and acceptable performance, but requires to update API libraries as soon as the underlying GPU vendor libraries update or new functionalities may not be available and it may break existing functionalities if vendor decides to remove certain feature calls from libraries. To keep libraries updated is a tedious task, to eliminate these limitations, para & full virtualization approaches are used, which allows to virtualize the GPUs at driver level. Para virtualization requires driver's modification, while full virtualization doesn't need driver modification. Generally, vendor doesn't provide source code for GPU drivers, but AMD has released GPU architecture documentation for their models [49], also some programmers has reverse engineered [50, 51, 52] NVIDIA GPU interfaces. Collectively, due to the efforts, custom drivers has been built for AMD and NVIDIA GPUs that leads to para & full virtualization methods. Fig. 3. explains the architecture of the para & full GPU virtualization techniques, in which guest OS is equipped with modified (para) or unmodified (full) GPU driver. QEMU GPU device on the host end receives the GPU calls from guest through a shared memory at hypervisor level. QEMU device pass the GPU requests to physical GPU and result is returned to guest OS. The vGPU manifests the requests for each VM. The guest driver thinks QEMU device as actual GPU.

This approach has an edge that GPU libraries doesn't need to be modified and existing applications can also run over this virtualization architecture. Additionally, since hypervisor is involved in this approach, so the GPU calls can be controlled, monitored and also live migration is supported. The downside of this approach is that it relies on custom GPU drivers, which can be produced only through open documentation provided by the GPU vendors or reproduction by inference. LoGV [53] is a para-virtualization solution for KVM virtualized platform where guest VM is equipped with custom PathScale [51] GPU driver. The PathScale driver is a reverse engineered solution for NVIDIA GPUs and is available as open-source. The basic function of the LoGV architecture is to partition GPU memory into various parts, and a VM is only allowed to interact with own part. GPU's partitioned memory and GPU accelerated application's address space is mapped together using memory management unit (MMU) available in today's GPUs. LoGV support guest to interact with mapped region without a role of hypervisor by configuring GPU page tables referenced by MMU. LoGV intervenes memory allocation to prevent a VM from mapping in other guest VMs spaces. The driver at guest end is responsible to send such operations to hypervisor, and these requests are validated by virtual device at hypervisor level. After validations, upon receiving mapping requests from virtual device, GPU driver in the host completes the memory allocations. A command submission channel is established similarly, where GPU application can send requests directly to the GPU without involving hypervisor.

Fig. 4. Para & Full Virtualization Architecture.

KVM-based hypervisor [54] paravirtualization approach was developed for Heterogeneous System Architecture (HSA). HSA by AMD architecture puts CPU and GPU on same chip to eliminate communication overheads and enables both GPU/CPU to use shared virtual memory. The KVM-based hypervisor solution implements the assignment of page tables by CPU's MMU to IOMMU so GPU can use them as shadowtables. Furthermore, an interrupt is generated by HSA when a page fault is occurred in GPU, which triggers CPU to modify the referenced page for GPU. This KVM-based approach asks the guest OS to update guest page table upon interrupt from HSA. GPUs use shadow page tables for addressing but for the sake of integrity, the guest page tables are updated too. The guest VMs are equipped with a custom driver, responsible to these operations. Lastly, HSA architecture facilitates a shared buffer between CPU and GPU in user space to queue GPU commands. This KVM-based approach just inform GPU about this buffer address residing at guest OS, so that guest VM can directly interact with physical GPU. VGVM [55] is another para-virtualization solution, consist of a VGVM library, a frontend driver and a backend driver. Existing applications using CUDA Runtime API remains compatible, and VGVM library transfer the routine arguments to the host through frontend driver, and receive results. It intercepts the routine arguments, bundle them GPU request along with other data and forward to the frontend driver. VGVM frontend driver works as a middle agent between library and host, which sends execution requests to the GPU host backend driver. Frontend driver is implemented in guest VM's kernel, and also manage the memory allocations and copy the arguments from user-space to kernel-space, upon receiving result from backend, the process is reversed by copying results from kernel to user space. Finally, a backend driver as virtual CUDA device, which acts like a dispatcher to handle multiple requests from VMs and have them executed over GPU. The results of execution are dispatched back to requesting VM's frontend driver that further deliveries to GPU application. GPUvm [56,57] offers both para & full virtualization solutions over Xen hypervisor using Nouveau [50] driver on virtual machine. In full-virtualization scenario, GPUvm partitions GPU's physical memory and MMIO space in several parts, where each part is assigned to different VM, which helps keeping the VMs isolated. By using dedicated GPU shadow page table for each VM, Virtual GPU memory addresses are translated to physical GPU address of allocated part of partitioned memory. GPUvm cannot handle page faults because of limitations in NVIDIA GPU architecture, thus GPUvm scans whole page table on every TLB flush to update shadow page table. Every GPU request from guest VM creates a page fault, as parted MMIO space is setup as read-only, thus OS intervenes and surpass the access to Xen driver space. Hardware has limited number of command submission channels, GPUvm virtualize them. It then creates shadow channels that are mapped to virtual channels. Overall, GPUvm full virtualization solution is not efficient because (i) it need to scan whole page table on each TLB flush (ii) it needs to intercept each GPU request. GPUvm handles the second limitation by using BAR Remap, intercepting only GPU related calls that requires access GPU channel descriptors, and other possible isolation problems are handled by using shadow page tables. The first GPUvm limitation can be solved with para virtualization approach offered by GPUvm. Similar to Xen [16], GPUvm creates guest page tables, and guests use these page tables

directly instead of shadow tables. VM driver perform hypercalls to GPUvm when there is a need to update GPU page table, which then are validated by GPUvm for isolation between guests. G-KVM [58] is a full GPGPU virtualization approach on KVM hypervisor which is inspired by its predecessor GPUvm [56, 57] (which can only run on Xen). Both hypervisors Xen and KVM are different in architecture, Xen is implemented at bare-level hardware which can manage memory and tasks scheduling for VMs like an OS, while KVM is implemented in kernel to offer virtualizations. G-KVM, in addition to GPUvm features, uses an aggregator and a QEMU device design.

2.3 Hardware-Based Gpgpu Virtualization: Hardware-based virtualization techniques allows VMs to directly access the GPU instead of using APIs or emulators. This approach utilize the hardware virtualization extensions offered by the GPU vendors e.g. NVIDIA GRID [65], AMD-Vi [68] and Intel VT-d [67]. The Direct Memory Access (DMA) channels and interrupts are mapped to VM directly, which allows direct data transfer from GPU memory to VM without hypervisor intervention and interrupts are directly sent to VM. Hardware virtualization is further divided into two categories (i) single VM per GPU support (ii) multiple VMs per GPU support. AMD-Vi and Intel VT-d falls under first category, while NVIDIA GRID allows multiplexing and falls under second category.

Fig. 5. Hardware-based Virtualization Architecture.

Fig. 5 illustrates the hardware-based architecture for AMD-Vi or VT-d. The VM can interact with GPU directly without any custom modified driver or library and hypervisor involvement is not needed. As AMD-Vi and VT-d just support single virtual machine per GPU, so a pluggable approach, which dynamically add/remove the GPU device in VM, is used to share GPU between multiple VMs. Hardware-based virtualization gives the maximum performance since no middleware (libraries) are involved and address spaces are directly mapped with VM and host, but implementation of live-migration and fault-tolerance is relatively difficult over this architecture.

Amazon Elastic Compute Cloud (Amazon EC2) [59] is the first platform that introduced GPUs for cloud users, and used Intel's pass-through technology [60]. Amazon introduced Cluster GPU Instances (CGI), which gives couple of NVIDIA GPGPUs to every VM [61]. HPC applications in need of massive parallelism can exploit CGIs, which offers direct GPU access to each VM. The virtualization performance of the CGIs are measured and benchmarks shows that compute-intensive applications can exploit GPUs in cloud for performance boost, while memory-intensive applications may have performance penalty due to EC2 structure that implements ECC memory error detection, which can ultimately cap the memory bandwidth.

vmCUDA [62] is a hybrid solution that use API Remoting and Hardware-based approaches together that enable GPU virtualization in VMware ESX hypervisor. It introduces the concept of appliance VM, which acts like middle server to serve multiple VMs, and pass-through the intercepted requests to actual GPU. It offers CUDA applications from different VMs to utilize GPU and ensure optimal usage. It's compatible with vMotion, which supports live migration, that means the guest VM can either be on the same node as of appliance VM or can be remotely located. vmCUDA doesn't need any modification to the hypervisor or existing CUDA applications. It offers API libraries and a frontend driver to the guest VM to interact with appliance VM. vmCUDA utilize vRDMA [63], VMCI [64] or TCP/IP channels in order to have communication between frontend / backend drivers.

NVIDIA GRID [65] eliminates the limitations of GPU sharing between the VMs that may be caused in pass-through methods. It implements Input/output Memory Management Unit (IOMMU), which translates VM's virtual address space to physical GPU's address space. Furthermore, GRID offers dedicated input buffer to each VM to isolate commands from other VMs. These changes to architecture has enabled GRID to sense virtualization and isolate every VM's GPU interaction, but still provides performance as of pass-through design. Cloud infrastructure can best utilize the NVIDIA GPUs including GRID Tesla M6, M60, K1 and K2, they all support the GRID, that make the NVIDIA GPUs better choice for multi-VM services. Hong et al. [66] benchmarked the performance of NVIDIA GRID-enabled GPGPUs for cloud gaming platforms, which shows that GRID-enabled GPUs gives higher performance than usual pass-through based GPUs due to optimized GPU hardware.

Comparison & Discussion: In this section all the virtualization techniques discussed above are summarized and compared for their features and architecture. Fig. 6 describes the list of supported hardware by each virtualization method. The hardware detail column show the GPU vendor and model that was used to test the corresponding technique, GPU models other than in following table may be supported as well. This would help in quickly scanning through the techniques for your needed hardware and available GPU models. Fig. 7 shows the detailed comparison of the GPGPU virtualization approaches discussed in previous section. The solutions are compared on the following metrics;

Virtualization Approach	Hardware Details
GViM [26]	GeForce 8800, 9800 GTX (Tesla)
vCUDA[28]	GeForce 8600 GT (Tesla)
rCUDA [34]	All NVIDIA GPUs from Tesla architecture
GVirtuS [36]	All NVIDIA GPUs from Tesla architecture
GVM [41]	Tesla C2070 (Fermi)
Pegasus [42]	GeForce 9800 GTX (Tesla)
Shadowfax [43]	GeForce 8800, 9800 GTX (Tesla)
VOCL [44]	Tesla M2070 (Fermi)
DS-CUDA [46]	GeForce GTX 560 Ti (Fermi)
Enhanced XMLRPC [47]	Tesla K20c
FairGV [48]	TitanX
LoGV [53]	GeForce GTX480
KVM-based [54]	AMD Radeon R7
VGVM [55]	Tesla M2050
GPUvm [56,57]	Quadro 6000 (Fermi)
G-KVM [58]	Geforce GTX 480
Intel [67]	Any GPU with pass-through support
Amazon EC2 [59]	Variety of NVIDIA GPUs
vmCUDA [62]	Quadro 4000 (Fermi)
NVIDIA GRID [65]	Any GPU with GRID support

Fig. 6. Hardware Details for Virtualization Techniques.

Fig. 7 shows the detailed comparison of the GPGPU virtualization approaches discussed in previous section. The solutions are compared on the following metrics;

Virtualization category: This field compares the available techniques for their approach type, whether they offer API Remoting, para-virtualization, full-virtualization or hardware-based GPGPU virtualization.

Hypervisor: This field compares the discussed solutions from their compatible hypervisor perspective, hypervisors options include KVM, Xen, VMware, Parallels or other.

Remote acceleration: This metric shows that whether an approach support the usage of a GPGPU installed on a remote node, either in same cluster or totally different network based cluster.

Programming model: This field compares the approaches from their programming model point, to whether an approach support CUDA or OpenCL programming library, this metric can help programmers to pick the right library to develop GPGPU empowered applications.

GPU Hardware: This field lists the GPGPU vendors for their support for particular virtualization method, Fig. 6 describes this column in detail with GPU models and vendor details.

Multiplexing: This field shows that whether an approach supports or not, the GPU sharing between multiple VMs. Fig. 7 shows that most of the techniques falls under API Remoting category, because of its relatively easy implementation and maintenance attracts more programmers. Additionally, API Remoting can be used on the environments where GPGPU virtualization is not supported by the hardware natively. API Remoting also allows to develop portable GPGPU application, which can be deployed on an environment equipped with the dependent library, regardless of underlying hardware. Para & Full virtualizations are also trendy due to other benefits over API Remoting, which allows live-migrations and also lower communication overheads makes this approach faster than API Remoting. Hardware-based methods provides ultimate performance because of almost none communication overheads, as shared memory paradigms are used and VMs can directly interact with host without interference of hypervisors or network layers. Further, the comparison reveals that KVM and Xen are the popular hypervisors being used for virtualization approaches from all three categories, VMware support is available only hardware-based approaches. Remote acceleration is supported by majority of API Remoting approaches, which allows the sharing of a GPU installed over a remote node either on the same cluster or total different network.

GPGPUs support two programming models (i) CUDA (ii) OpenCL. Comparison table shows that CUDA is the most popular programming model supported by majority of the virtualization approaches. It also shows that which availability of particular architecture as open-source, list reveals that not many of the architectures are available as open-source. GPU vendors and compared which shows that NVIDIA is the leading provider of the GPGPUs and are being supported by the wide range of the virtualization techniques. Multiplexing all ticks shows that almost approaches support the sharing of a single physical GPU between multiple VMs. This survey reveals that though there has been good amount of research dealing with GPGPU virtualization and many challenges has been addressed, but still GPGPU virtualization has not reached its adulthood. There are many areas that needs improvements including scalability, security, portability, power efficiency, shared spaces, live migration, communication between guest VMs and host.

Conclusion: Since the rise of HPC, heterogeneous systems are being exploited to improve the efficiency of the applications by exploiting the parallel paradigms to achieve higher computational performance but at lower cost. This study has reviewed the GPU virtualization techniques that focus on virtualization for general-purpose accelerations. Cloud is a heterogeneous environment, and GPGPU virtualization allows to share a physical GPU between multiple heterogeneous virtual machines to save costs, ensure optimal usage of the GPGPU devices, and offer its customers high-performance platform. In this survey, the available GPGPU virtualization solutions are explored and compared for their possible features and supported frameworks. The study reveals that API Remoting, para, full, and hardware-based solutions has been presented to perform the GPGPU virtualization. Furthermore, NVIDIA is the leading vendor among GPGPU providers and CUDA is the most supported programming language. Each virtualization solution has own benefits and limitations, which can be adopted according to the need and available hardware resources. Future work may involve exploring the GPGPU virtualization along with scheduling methods, and benchmarking these techniques to get real numbers and comparisons.

Name	Virtualization Category	Hypervisor					Remote acceleration	Programming model		Open source	GPU hardware			Multiplexing
		KVM	Xen	VMware	Parallels	Other		CUDA	OpenCL		NVIDIA	AMD	Intel	
GVIM [26]	API Remoting		✓					✓			✓			✓
vCUDA[28]	API Remoting	✓	✓					✓			✓			✓
rCUDA [34]	API Remoting						✓	✓		✓	✓			✓
GVirtuS [36]	API Remoting	✓	✓	✓			✓	✓		✓	✓			✓
GVM [41]	API Remoting					Proprietary		✓			✓			✓
Pegasus [42]	API Remoting		✓					✓			✓			✓
Shadowfax [43]	API Remoting		✓				✓	✓			✓			✓
VOCL [44]	API Remoting						✓		✓		✓			✓
DS-CUDA [46]	API Remoting						✓	✓		✓	✓			✓
Enhanced XMLRPC [47]	API Remoting						✓	✓			✓			✓
FairGV [48]	API Remoting						✓	✓			✓			✓
LoGV [53]	Para	✓						✓	✓		✓			✓
KVM-based [54]	Para	✓							✓			✓		✓
VGVM [55]	Para							✓						✓
GPUvm [56,57]	Para & Full		✓					✓		✓				✓
G-KVM [58]	Full	✓						✓						✓
Intel [67]	Hardware	✓	✓	✓	✓			✓	✓		any GPU support pass-through			
Amazon EC2 [59]	Hardware	✓	✓					✓	✓					✓
vmCUDA [62]	Hardware			✓			✓	✓			✓			✓
NVIDIA GRID [65]	Hardware	✓	✓	✓	✓			✓	✓		any GPU support NVIDIA GRID			✓

Figure 7. Comparison of GPGPU Virtualization Solutions

REFERENCES

- [1] Reaño, C., Peña, A. J., Silla, F., Duato, J., Mayo, R., & Quintana-Ortí, E. S. (2012, December). Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. In High Performance Computing (HiPC), 2012 19th International Conference on (pp. 1-10). IEEE.
- [2] Montella, R., Giunta, G., & Laccetti, G. (2014). Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. Cluster computing, 17(1), 139-152.
- [3] Hong, C. H., Spence, I., & Nikolopoulos, D. S. (2017). GPU virtualization and scheduling methods: A comprehensive survey. ACM Computing Surveys (CSUR), 50(3), 35..
- [4] Yang, C. T., Liu, J. C., Wang, H. Y., & Hsu, C. H. (2014). Implementation of GPU virtualization using PCI pass-through mechanism. The Journal of Supercomputing, 68(1), 183-213.
- [5] Hsu, H. C., & Lee, C. R. (2016, December). G-KVM: A Full GPU Virtualization on KVM. In Computer and Information Technology (CIT), 2016 IEEE International Conference on (pp. 545-552). IEEE.
- [6] Kirk, D. B., & Wen-Mei, W. H. (2016). Programming massively parallel processors: a hands-on approach. Morgan kaufmann.
- [7] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., ... & Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley (Vol. 2). Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
- [8] Crago, S., Dunn, K., Eads, P., Hochstein, L., Kang, D. I., Kang, M., ... & Walters, J. P. (2011, September). Heterogeneous cloud computing. In Cluster Computing (CLUSTER), 2011 IEEE International Conference on (pp. 378-385). IEEE.

- [9] Schadt, E. E., Linderman, M. D., Sorenson, J., Lee, L., & Nolan, G. P. (2011). Cloud and heterogeneous computing solutions exist today for the emerging big data problems in biology. *Nature Reviews Genetics*, 12(3), 224.
- [10] Lee, G., & Katz, R. H. (2011, June). Heterogeneity-Aware Resource Allocation and Scheduling in the Cloud. In *HotCloud*.
- [11] Hong, C. H., Spence, I., & Nikolopoulos, D. S. (2017). GPU virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3), 35.
- [12] Chaubal, C. (2008). The architecture of vmware esxi. *VMware White Paper*, 1(7).
- [13] Kivity, A., Kamay, Y., Laor, D., Lublin, U., & Liguori, A. (2007, July). kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium (Vol. 1, pp. 225-230)*.
- [14] Velte, A., & Velte, T. (2009). *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc..
- [15] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., ... & Warfield, A. (2003, October). Xen and the art of virtualization. In *ACM SIGOPS operating systems review (Vol. 37, No. 5, pp. 164-177)*. ACM.
- [16] Goldberg, R. P. (1974). Survey of virtual machine research. *Computer*, (6), 34-45.
- [17] Nvidia, C. U. D. A. (2007). *Compute unified device architecture programming guide*.
- [18] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. 1999. Addison-Wesley Longman Publishing Co., Inc.
- [19] Khronos OpenCL Working Group et al. The opencl specification. 2008. Version 1, 29 (2008), 8.
- [20] Blythe, D. (2006). The direct3d 10 system. *ACM Transactions on Graphics (TOG)*, 25(3), 724-734.
- [21] Eftekharypour, Y., Ong, H., & Yong, K. K. (2016, August). Enhanced XMLRPC-based GPU virtualization. In *Computer and Information Sciences (ICCOINS), 2016 3rd International Conference on (pp. 459-464)*. IEEE.
- [22] Li, T., Narayana, V. K., El-Araby, E., & El-Ghazawi, T. (2011, September). GPU resource sharing and virtualization on high performance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on (pp. 733-742)*. IEEE.
- [23] Menychtas, K., Shen, K., & Scott, M. L. (2014, February). Disengaged scheduling for fair, protected access to fast computational accelerators. In *ACM SIGPLAN Notices (Vol. 49, No. 4, pp. 301-316)*. ACM.
- [24] Dowty, M., & Sugerman, J. (2009). GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review*, 43(3), 73-82.
- [25] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., & Ranganathan, P. (2009, March). GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing (pp. 17-24)*. ACM.
- [26] Cho, Y. C., & Jeon, J. W. (2007, October). Sharing data between processes running on different domains in paravirtualized xen. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on (pp. 1255-1260)*. IEEE.
- [27] Shi, L., Chen, H., Sun, J., & Li, K. (2012). vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6), 804-816.
- [28] Cerami, E. (2002). *Web services essentials: distributed applications with XML-RPC, SOAP, UDDI & WSDL*. "O'Reilly Media, Inc."
- [29] Shi, L., Chen, H., Sun, J., & Li, K. (2012). vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6), 804-816.
- [30] Chen, H., Shi, L., & Sun, J. (2010, June). VMRPC: A high efficiency and light weight RPC system for virtual machines. In *Quality of Service (IWQoS), 2010 18th International Workshop on (pp. 1-9)*. IEEE.
- [31] Patni, S., George, J., Lahoti, P., & Abraham, J. (2015, September). A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial. In *Next Generation Computing Technologies (NGCT), 2015 1st International Conference on (pp. 6-9)*. IEEE.
- [32] Duato, J., Pena, A. J., Silla, F., Mayo, R., & Quintana-Ortí, E. S. (2010, January). Modeling the CUDA remoting virtualization behaviour in high performance networks. In *First Workshop on Language, Compiler, and Architecture Support for GPGPU*.
- [33] Jos'e 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.
- [34] Reaño, C., Silla, F., Castelló, A., Peña, A. J., Mayo, R., Quintana Ortí, E. S., & Duato, J. (2015). Improving the user experience of the rCUDA remote GPU virtualization framework. *Concurrency and Computation: Practice and Experience*, 27(14), 3746-3770.
- [35] Giunta, G., Montella, R., Agrillo, G., & Coviello, G. (2010, August). A GPGPU transparent virtualization component for high performance computing clouds. In *European Conference on Parallel Processing (pp. 379-391)*. Springer, Berlin, Heidelberg.

- [36] Wang, J., Wright, K. L., & Gopalan, K. (2008, June). XenLoop: a transparent high performance inter-vm network loopback. In Proceedings of the 17th international symposium on High performance distributed computing (pp. 109-118). ACM.
- [37] Gebhardt, C., & Tomlinson, A. (2010). Challenges for inter virtual machine communication. no. September, 0-17.
- [38] Di Lauro, R., Giannone, F., Ambrosio, L., & Montella, R. (2012, July). Virtualizing general purpose GPUs for high performance cloud computing: an application to a fluid simulator. In Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on (pp. 863-864). IEEE.
- [39] Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., ... & Nikolopoulos, D. S. (2017). On the virtualization of CUDA based GPU remoting on ARM and X86 machines in the GVirtuS framework. *International Journal of Parallel Programming*, 45(5), 1142-1163.
- [40] Li, T., Narayana, V. K., El-Araby, E., & El-Ghazawi, T. (2011, September). GPU resource sharing and virtualization on high performance computing systems. In Parallel Processing (ICPP), 2011 International Conference on (pp. 733-742). IEEE.
- [41] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. 2011. In Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11). 31.
- [42] Alexander M. Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. Shadowfax: Scaling in heterogeneous cluster systems via GPGPU assemblies. 2011. In Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing. ACM, 3-10.
- [43] Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. 2012. In Proceedings of the Innovative Parallel Computing (InPar'12). IEEE, 1-12.
- [44] Gropp, W., Lusk, E., Doss, N., & Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6), 789-828.
- [45] Masahiro Oikawa, Atsushi Kawai, Keigo Nomura, Koichi Yasuoka, Kenichi Yoshikawa, and Tetsu Narumi. DS-CUDA: A middleware to use many GPUs in the cloud environment. 2012. In Proceedings of the 2012 SC Companion to High Performance Computing, Networking, Storage and Analysis (SCC). IEEE, 1207-1214.
- [46] Eftekharypour, Y., Ong, H., & Yong, K. K. (2016, August). Enhanced XMLRPC-based GPU virtualization. In Computer and Information Sciences (ICCOINS), 2016 3rd International Conference on (pp. 459-464). IEEE.
- [47] Hong, C. H., Spence, I., & Nikolopoulos, D. S. (2017). FairGV: fair and fast GPU virtualization. *IEEE Transactions on Parallel and Distributed Systems*, 28(12), 3472-3485.
- [48] AMD, Radeon's next-generation Vega architecture. 2017. accessed on Feb 03, 2018. https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf.
- [49] X.OrgFoundation. Nouveau: Accelerated Open Source driver for nVidia cards. 2011. Accessed on Feb 03, 2018. <https://nouveau.freedesktop.org/wiki/>.
- [50] PathScale. pathscale/pscnv. 2012. Accessed on Feb 03, 2018. <https://github.com/pathscale/pscnv>
- [51] Menychtas, K., Shen, K., & Scott, M. L. (2013, June). Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. In USENIX Annual Technical Conference (pp. 291-296).
- [52] Gottschlag, M., Hillenbrand, M., Kehne, J., Stoess, J., & Bellosa, F. (2013, November). LoGV: Low-Overhead GPGPU Virtualization. In HPCC/EUC (pp. 1721-1726).
- [53] Huang, Y. J., Wu, H. H., Chung, Y. C., & Hsu, W. C. (2016, March). Building a kvm-based hypervisor for a heterogeneous system architecture compliant system. In ACM SIGPLAN Notices (Vol. 51, No. 7, pp. 3-15). ACM.
- [54] Vasilas, D., Gerangelos, S., & Koziris, N. (2016, July). VGVM: Efficient GPU capabilities in virtual machines. In High Performance Computing & Simulation (HPCS), 2016 International Conference on (pp. 637-644). IEEE.
- [55] Suzuki, Y., Kato, S., Yamada, H., & Kono, K. (2014, June). GPUvm: Why not virtualizing GPUs at the hypervisor?. In USENIX Annual Technical Conference (pp. 109-120).
- [56] Suzuki, Y., Kato, S., Yamada, H., & Kono, K. (2016). Gpuvm: Gpu virtualization at the hypervisor. *IEEE Transactions on Computers*, 65(9), 2752-2766.
- [57] Hsu, H. C., & Lee, C. R. (2016, December). G-KVM: A Full GPU Virtualization on KVM. In Computer and Information Technology (CIT), 2016 IEEE International Conference on (pp. 545-552). IEEE.
- [58] EC Amazon. Amazon elastic compute cloud (Amazon EC2). <https://aws.amazon.com/ec2/>. 2010. Accessed on 4th Feb. 2018.
- [59] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Yl'a-J'a" aski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. 2012. Presented in the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud).

- [60] Yeh, C. Y., Kao, C. Y., Hung, W. S., Lin, C. C., Liu, P., Wu, J. J., & Liu, K. C. (2013, May). GPU virtualization support in cloud system. In *International Conference on Grid and Pervasive Computing* (pp. 423-432). Springer, Berlin, Heidelberg.
- [61] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. GPU virtualization for high performance general purpose computing on the ESX hypervisor. 2014. In *Proceedings of the High Performance Computing Symposium*. Society for Computer Simulation International, 2.
- [62] Ranadive, A., & Davda, B. (2012). Toward a paravirtual vRDMA device for VMware ESXi guests. *VMware Technical Journal*, Winter 2012, 1(2).
- [63] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Kaladhar Voruganti, and Garth R. Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. 2009. In *Proceedings of the USENIX Annual Technical Conference*.
- [64] Herrera, A. (2014). NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. Nvidia Corp.
- [65] Hua-Jun Hong, Tao-Ya Fan-Chiang, Che-Run Lee, Kuan-Ta Chen, Chun-Ying Huang, and Cheng-Hsin Hsu. GPU consolidation for cloud games: Are we there yet? 2014. In *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games*. IEEE Press, 3.
- [66] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., ... & Wiegert, J. (2006). Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3).
- [67] Leendert Van Doorn. Hardware virtualization trends. 2006. In *Proceedings of the 2nd International ACM/Usenix Conference on Virtual Execution Environments*, Vol. 14. 45-45.