gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space

Mochi Xue[1,2], Kun Tian[2], Yaozu Dong[1, 2], Jiajun Wang[1], Zhengwei Qi[1], Bingsheng He[3], Haibing Guan[1]
{*xuemochi, jiajunwang, qizhenwei, hbguan*}*@sjtu.edu.cn*
{*kevin.tian, eddie.dong*}*@intel.com BSHE@ntu.edu.sg*
[1]Shanghai Jiao Tong University, [2]Intel Corporation, [3]Nanyang Technological University

Abstract

With increasing GPU-intensive workloads deployed on cloud, the cloud service providers are seeking for practical and efficient GPU virtualization solutions. However, the cutting-edge GPU virtualization techniques such as gVirt still suffer from the restriction of scalability, which constrains the number of guest virtual GPU instances.

This paper introduces *gScale*, a scalable GPU virtualization solution. By taking advantage of the GPU programming model, gScale presents a dynamic sharing mechanism which combines partition and sharing together to break the hardware limitation of global graphics memory space. Particularly, we propose three approaches for gScale: (1) the private shadow graphics translation table, which enables global graphics memory space sharing among virtual GPUs, (2) ladder mapping and fence memory space pool, which forces CPU access host physical memory space (serving the graphics memory) bypassing global graphics memory space, (3) slot sharing, which improves the performance of vGPU under a high density of instances.

The evaluation shows that gScale scales up to 15 guest virtual GPU instances in Linux or 12 guest virtual GPU instances in Windows, which is 5x and 4x scalability, respectively, compared to gVirt. At the same time, gScale incurs a slight runtime overhead on the performance of gVirt when hosting multiple virtual GPU instances.

1 Introduction

The Graphic Processing Unit (GPU) is playing an indispensable role in cloud computing as GPU efficiently accelerates the computation of certain workloads such as 2D and 3D rendering. With increasing GPU intensive workloads deployed on cloud, cloud service providers introduce a new computing paradigm called *GPU Cloud* to meet the high demands of GPU resources (e.g., Amazon EC2 GPU instance [2], Aliyun GPU server [1]).

As one of the key enabling technologies of GPU cloud, GPU virtualization is intended to provide flexible and scalable GPU resources for multiple instances with high performance. To achieve such a challenging goal, several GPU virtualization solutions were introduced, i.e., GPUvm [26] and gVirt [28]. gVirt, also known as GVT-g, is a full virtualization solution with mediated pass-through support for Intel Graphics processors. Benefited by gVirt's open-source distribution, we are able to investigate its design and implementation in detail. In each virtual machine (VM), a virtual GPU (vGPU) instance is maintained to provide performance-critical resources directly assigned. By running a native graphics driver inside a VM, there is no hypervisor intervention in performance critical paths. Thus, it optimizes resources among the performance, feature, and sharing capabilities [5].

Though gVirt successfully puts GPU virtualization into practice, it suffers from scaling up the number of vGPU instances. The current release of gVirt only supports 3 guest vGPU instances on one physical Intel GPU[1]. Such a scalability is inefficient for cloud usage. In this paper, we explore the design of gVirt, and address the weakness of its scalability. To be specific, gVirt proposes a static *resource partition* mechanism for the global graphics memory, which limits the number of vGPUs. For a virtualization solution, scalability is an indispensable feature which ensures high density of instances. High scalability improves the consolidation of resources and balances the cost.

This paper presents gScale, a practical, efficient and scalable GPU virtualization solution. To increase the number of vGPU instances, gScale targets at the bottleneck design of gVirt and introduces a dynamic sharing scheme for global graphics memory space. gScale provides each vGPU instance with a *private shadow graphics translation table* (GTT) to break the limitation of global graphics memory space. gScale copies vGPU's

[1]In this paper, Intel GPU refers to the Intel HD Graphics embedded in HASWELL CPU.

private shadow GTT to physical GTT along with the context switch. The private shadow GTT allows vGPUs to share the global graphics memory space, which is the essential design of gScale. However, it is non-trivial to make the global graphics memory space sharable, for that global graphics memory space is both accessible to CPU and GPU. gScale implements a novel *ladder mapping* and a fence memory space pool to let CPU access host physical memory space serving for the graphics memory which bypasses the global graphics memory space. At the same time, gScale proposes *slot sharing* to improve the performance of vGPUs under a high density of instances.

This paper implements gScale based on gVirt, which comprises of about 1000 LoCs. The source code is now available on Github[2]. In summary, this paper overcomes various challenges, and makes the following contributions:

- A private shadow graphics translation table for each vGPU, which makes the global graphics memory space sharable. It keeps a specific copy of the physical GTT for the vGPU. When the vGPU becomes the render owner, its private shadow graphics translation table will be written on the physical graphics translation table by gScale to provide correct translations.

- The ladder mapping mechanism, which directly maps guest physical address to host physical address serving the guest graphic memory. With ladder mapping mechanism, CPU can access the host physical memory space serving guest graphic memory, which bypasses the global graphics memory space.

- Fence memory space pool, a dedicated memory space reserved in global graphics memory space with dynamic management. It guarantees that the fence registers operate correctly when a certain range of global graphics memory space is unavailable for CPU.

- Slot sharing, which optimizes the performance of vGPUs. It could reduce the overhead of private shadow GTT copying under a high instance density.

- The evaluation shows that gScale can provide 15 guest vGPU instances for Linux VMs or 12 guest vGPU instances for Windows VMs on one physical machine, which is 5x and 4x scalability, respectively, compared to gVirt. It achieves up to 96% performance of gVirt under a high density of instances.

[2]https://github.com/inkpool/XenGT-Preview-kernel/tree/gScale

The rest of paper is organized as follows. Section 2 describes the background of GPU programming model, and Section 3 reveals gVirt's scalability issue and its bottleneck. The design and implementation of gScale are presented in Section 4. We evaluate gScale's performance in Section 5 with the overhead analysis. We discuss the architecture independency in Section 6 and the related work is in Section 7. Finally, in Section 8 we conclude our work with a brief discussion of future work.

2 Background and Preliminary

GPU Programming Model Driven by high level programming APIs like OpenGL and DirectX, graphics driver produces GPU commands into primary buffer and batch buffer while GPU consumes the commands accordingly. The primary buffer is designed to deliver the primary commands with a ring structure, but the size of primary buffer is limited. To make up for the space shortage, batch buffer is linked to the primary buffer to deliver most of the GPU commands.

GPU commands are produced by CPU and transferred from CPU to GPU in batches. To ensure that GPU consumes the commands after CPU produces them, a notification mechanism is implemented in the primary buffer with two registers. The *tail* register is updated when CPU finishes the placement of commands, and it informs GPU to get commands in the primary buffer. When GPU completes processing all the commands, it writes the *head* register to notify CPU [28].

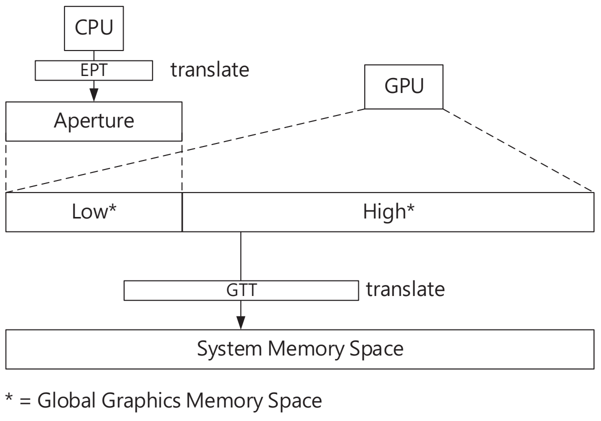


Figure 1: Graphics Translation Table

Graphics Translation Table and Global Graphics Memory Graphics translation table (GTT), sometimes known as *global graphics translation table*, is a memory-resident page table providing the translations from logical graphics memory address to physical memory address, as Figure 1 shows. It is worth noting that the physical memory space served by GTT is assigned to be the global graphics memory space, especially for GPUs without dedicated memory, such as Intel's GPU. However, through the Aperture [6], a range defined in the graphics memory mapping input/output (MMIO), CPU could also access the global graphics memory space. And this CPU's visible part of global graphics memory is called *low global graphics memory*, while the rest part is called *high global graphics memory*. For example, the GTT size of Intel GPU is 2MB, mapping to a 2GB graphics memory space, and the Aperture range could maximally be 512KB which maps to 512MB graphics memory space visible by CPU. Accordingly, the low graphics memory space is 512MB, while the high graphics memory space is 1536MB.

gVirt gVirt is the first product level full GPU virtualization solution with mediated pass-through [28]. So the VM running native graphics driver is presented with a full featured virtualized GPU. gVirt emulates virtual GPU (vGPU) for each VM, and conducts context switch among vGPUs. vGPUs are scheduled to submit their commands to the physical GPU continuously, and each vGPU has a 16ms time slide. When time slide runs out, gVirt switches the render engine to next scheduled vGPU. To ensure the correct and safe switch between vGPUs, gVirt saves and restores vGPU states, including internal pipeline state and I/O register states.

By passing-through the accesses to the frame buffer and command buffer, gVirt reduces the overhead of performance-critical operations from a vGPU. For global graphics memory space, resource partition is applied by gVirt. For local graphics memory space, gVirt implements per-VM local graphics memory [28]. It allows each VM to use the full local graphics memory space which is 2GB in total. The local graphics memory space is only accessible by vGPU, so gVirt can switch the graphics memory spaces among vGPUs when switching the render ownership.

3 Scalability Issue

The global graphics memory space can be accessed simultaneously by CPU and GPU, so gVirt has to present VMs with their global graphics memory spaces at any time, leading to the *resource partition.* As shown in Figure 2, when a vGPU instance is created with a VM, gVirt only assigns the part of host's low global graphics memory and the part of host's high global graphics memory to the vGPU, as its *low* global graphics memory and *high* global graphics memory, respectively. These two parts together comprise the vGPU's global graphics memory space. Moreover, the partitioned graphics memory spaces are mapped by a shared shadow GTT, which is maintained by gVirt to translate a guest graphics memory address to a host physical memory address.

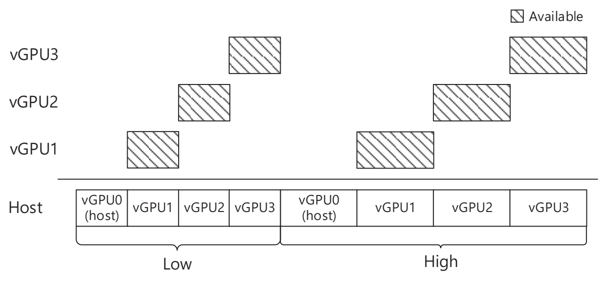


Figure 2: Global Graphics Memory Space with Partition

To support simultaneous accesses from CPU and GPU, the shared shadow GTT has to carry translations for all the VMs, which means the guest view of shadow GTT is exactly the same with host, as shown the left side in Figure 5. gVirt introduces an address space ballooning mechanism to balloon the space that does not belong to the VM. gVirt exposes the partition information to VM's graphics driver, and graphics driver marks the space which does not belong to the VM as "ballooned" [28]. Note here, gVirt's memory space ballooning mechanism is for resource isolation, which is different with traditional memory ballooning technique [29]. Though guests have the same view of shadow GTT with the host, with ballooning mechanism, guest VM can only access the global graphics memory space allocated for its own.

Due to the *resource partition* mechanism for global graphics memory space, with a fixed size of global graphics memory, the number of vGPUs hosted by gVirt is limited. If gVirt wants to host more vGPUs, it has to configure vGPUs with less global graphics memory. However, it sacrifices vGPU's functionality if we increase the number of vGPUs by shrinking the global graphics memory size of vGPUs. Moreover, the graphics driver reports errors or even crashes when it cannot allocate memory from global graphics memory space [4]. For instance, a vGPU with deficient global graphics memory size may lose functionality under certain workloads which need the high requirements of global graphics memory. In fact, more global graphics memory does not bring performance improvement for vGPUs, for that this memory only serves frame buffer and ring buffer which are in limited sizes, while the massive rendering data reside in local graphics data [28]. Specifically, for vGPU in Linux VM, the 64MB low global graphics memory and 384MB high global graphics memory are recommended. For vGPU in Windows VM, the recommend configuration is 128MB low global graphics memory and 384MB high global graphics memory [11]. In the scalability experiment of gVirt [28], it hosted 7 guest vGPUs in Linux VMs. However, the global graphics memory size of vGPU in that experiment is less than recommended configuration. Such configuration cannot guarantee the full functionality of vGPU, and it would incur errors or crashes for vGPU under certain workloads for the deficiency of graphics memory space [4]. In this paper, the vGPUs are configured with recommended configuration.

Actually, the current source code (2015Q3) of gVirt sets the maximal vGPU number as 4. For platform with Intel GPU, there is 512MB low global graphics memory space and 1536MB high global graphics memory space in total. While gVirt can only provide 3 guest vGPUs (64MB low global graphics memory, and 384MB high global graphics memory) for Linux VMs or 3 guest vGPUs (128MB low global graphics memory, and 384MB high global graphics memory) for Windows VMs, because the host VM also occupies one vGPU. As a GPU virtualization solution, gVirt is jeopardized by its scalability issue. Also, the static partition of global graphics memory space is the root cause of this scalability issue. In this paper, we attempt to break the limitation of static resource partition and sufficiently improve the scalability for gVirt.

4 Design and Implementation

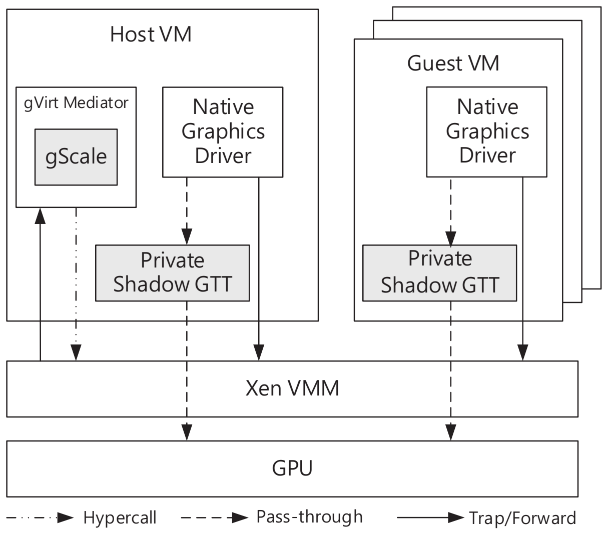


Figure 3: Architecture

The architecture of gScale is shown in Figure 3. To break the limitation of global graphics memory, gScale proposes a dynamic sharing scheme which combines partition and sharing together as Figure 4 illustrates. For the access of GPU, we introduce private shadow GTT to make global graphics memory space sharable. For the access of CPU, we present ladder mapping to force CPU directly access host physical memory space serving the graphics memory, which bypasses the global graphics memory space. For concurrent access of CPU and GPU, gScale reserves a part of low global graphics memory as *the fence memory space pool* to ensure the functionality of fence registers. gScale also divides the high global graphics memory space into several slots to leverage the overhead caused by private shadow GTT copying.

In this section, the design of gScale addresses three technical challenges: (1) how to make global graphics memory space sharable among vGPUs, (2) how to let CPU directly access host memory space serving the graphics memory, which bypasses global graphics memory space, and (3) how to improve the performance of vGPUs under a high instance density.

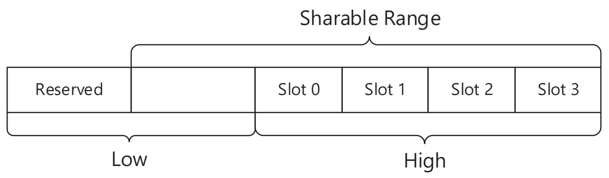
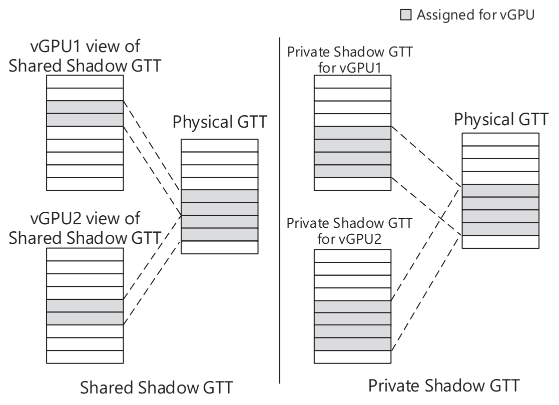


Figure 4: Dynamic Sharing Scheme of gScale

4.1 Private Shadow GTT

It is a non-trivial task to make the global graphics memory space sharable among vGPUs, for that CPU and GPU access the low global graphics memory space simultaneously, as we mentioned in Section 2. However, high global graphics memory space is only accessible to GPU, which makes it possible for vGPUs to share high global graphic memory space. Taking advantages of GPU programming model, vGPUs are scheduled to take turns to be served by render engine, and gScale conducts context switch before it changes the ownership of render engine. This inspires us to propose the private shadow GTT for each vGPU.

Figure 5 shows the gVirt's shared shadow GTT and gScale's private shadow GTT. Specifically, shared shadow GTT is introduced to apply the resource partition on global graphics memory space. It provides every vGPU with a same view of physical GTT, and each vGPU is assigned with a different part of shadow GTT. Accordingly, each vGPU occupies the different ranges of global graphics memory space. However, gScale's private shadow GTT is specific for each vGPU, and it provides vGPU with a unique view of global graphics memory space. Moreover, the translations that private shadow GTT contains are only valid for its corresponding vGPU. And gScale copies vGPU's private



Assigned for vGPU
vGPU1 view of Shared Shadow GTT
Physical GTT
vGPU2 view of Shared Shadow GTT
Shared Shadow GTT
Private Shadow GTT for vGPU1
Physical GTT
Private Shadow GTT for vGPU2
Private Shadow GTT

Figure 5: Private Shadow GTT

shadow GTT onto the physical GTT along with the context switch to ensure that translations of physical GTT are correct for this vGPU. When vGPU owns the physical engine, gScale synchronizes the modifications of physical GTT to vGPU's private shadow GTT.

By manipulating the private shadow GTTs, gScale could allow vGPUs to use an overlapped range of global graphics memory, which makes the high global graphics memory space sharable, as shown in Figure 6. However, low graphics memory space is still partitioned among the vGPUs, for that it is also visible to CPU.

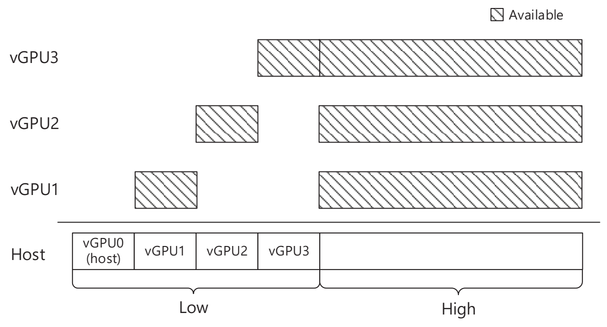


Figure 6: Sharable Global Graphics Memory Space

On Demand Copying Writing private shadow GTT onto physical GTT incurs the overhead. gScale introduces *on demand copying* to reduce this unnecessary copying overhead. Theoretically, gScale could assign the whole sharable global graphics memory space to a vGPU. But gScale only configures vGPU with the sufficient global graphics memory, for that more global graphics memory does not increase the performance of vGPU while it could increase the overhead of copying shadow GTT. When gScale switches the vGPU which owns the render engine, it conducts the context switch for vGPUs and copies the entries of its private shadow GTT to the physical GTT. The size of private GTT is exactly the same with physical GTT, but vGPU is configured with a portion of available global graphics memory space (corresponding to only part of vGPU's private shadow GTT). By taking advantage of this characteristic, gScale only copies the demanding part of vGPU's private shadow GTT to the physical GTT.

4.2 Ladder Mapping

It is not enough to only let high global graphics memory space sharable because the static partition applied to low global graphics memory space still constrains the number of vGPUs. Low global graphics memory space is accessible to both CPU and GPU, while CPU and GPU are scheduled independently. gScale has to present VMs with their low global graphics memory spaces at all time. Intel GPU does not have dedicated graphics memory, while the graphics memory is actually allocated from system memory. The graphics memory of VM actually resides in host physical memory. gScale proposes *the ladder mapping* to force CPU directly access the host memory space serving the graphics memory which bypasses the global graphics memory space.

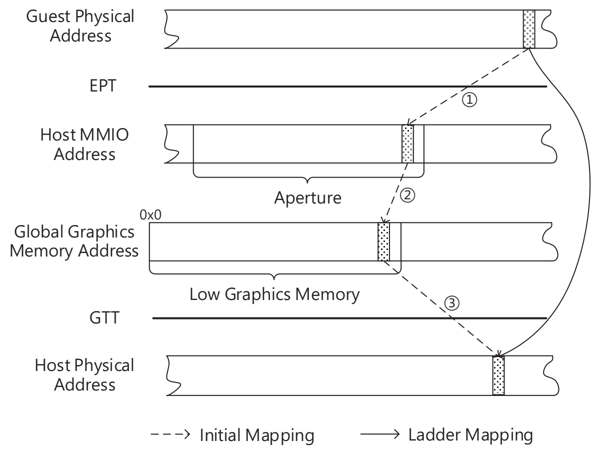


Figure 7: Ladder Mapping

When a VM is created, gScale maps VM's guest physical memory space to host physical memory space by Extended Page Table (EPT). EPT is a hardware supported page table for virtualization, which translates guest physical address to host physical address [22]. Through the aperture, a range of MMIO space in host physical memory space, CPU could access the low part of global graphics memory space. With the translations in GTT, the global graphics memory address is translated into host physical address served for graphics memory. Finally, CPU could access the graphics data residing in host physical memory space.

Figure 7 shows the initial mapping we mentioned above, and through the Step 1, 2 and 3, guest physical address is translated into host physical address. When the process is completed, a translation between guest physical address and host physical address serving the graphics memory is established. After that, gScale modifies the translation of EPT to directly translate the guest physical address to host physical address serving the graphics memory without the reference of global graphics memory address. We call this mechanism the ladder mapping. Ladder mapping is constructed when CPU accesses global graphics memory space by referring to the GTT. gScale monitors the GTT at all time, and builds ladder mapping as long as the translation of GTT is modified by CPU. In a nutshell, the goal of ladder mapping is to force CPU access host memory space serving the graphics memory which bypasses the global graphics memory space. After that, gScale could make low global graphics memory space sharable with private shadow GTT.

Fence Memory Space Pool Although we use ladder mapping to force CPU bypass the global graphics memory space, there is one exception that CPU could still access global graphics memory space through *fence registers*. Fence register contains the information about tiled formats for a specific region of graphics memory [6]. When CPU accesses this region of global graphics memory recorded in a fence register, it needs the format information in the fence to operate the graphics memory. However, after we enable ladder mapping, the global graphics memory space is no longer available for CPU. The global graphics memory address in fence register is invalid for CPU.

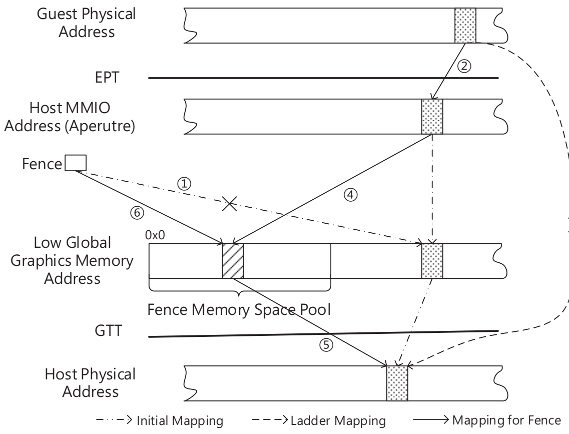


Figure 8: Fence Memory Space Pool

To address the malfunction of fence registers, gScale reserves a dedicated part of low global graphics memory to work for fence registers, and enables dynamic management for it. We call this reserved part of low global graphics memory, the *fence memory space pool*. Figure 8 illustrates the workflow of how fence memory space pool works:

Step 1, when a fence register is written by graphics driver, gScale acquires the raw data inside the register. By analyzing the raw data, gScale gets the format information and the global graphics memory space range served by this fence register.

Step 2, by referring to the initial mapping of EPT, gScale finds the guest physical memory space range which corresponds to the global graphics memory space range in the register. Though the initial mapping of EPT is replaced by ladder mapping, it is easy to restore the original mapping with a backup, because the initial mapping is continuous with clear offset and range [6]. After that, this range of guest physical memory space is again mapped to a range of physical memory space within the aperture.

Step 3, gScale suspends the ladder mapping for this range of guest physical memory space, and allocates a range of memory space in the fence memory space pool with same size.

Step 4, gScale maps the host physical memory space in aperture to the memory space newly allocated in fence memory space pool.

Step 5, gScale copies the entries of GTT serving the graphics memory space in fence register to the part of GTT corresponding to the new graphics memory space allocated in fence memory space pool.

Step 6, gScale writes the new graphics memory space range along with untouched format information into the fence register. To this end, gScale constructs a temporary mapping for fence register, and CPU could finally use the information in fence register correctly.

When a fence register is updated, gScale restores the ladder mapping for the previous range of global graphics memory space that fence register serves, and frees its corresponding memory space in the fence memory space pool. After that, gScale repeats the procedure as we mentioned above to ensure the updated register work correctly with fence memory space pool.

4.3 Slot Sharing

In real cloud environments, the instances hosted by cloud may not remain busy at all time, while some instances become idle after completing their tasks [23]. gScale implements *slot sharing* to improve the performance of vGPU instance under a high instance density. Figure 9 shows the layout of physical global graphics memory space, gScale divides the high global graphics memory space into several slots, and each slot could hold one vGPU's high graphics memory. gScale could deploy several vGPUs in the same slot. As we mentioned in Section 2, high global graphics memory space provided by Intel GPU is 1536MB, while 384MB is sufficient for one VM. However, gScale only provides slots for VMs in high graphics memory space, for that the amount of low global graphics memory space is 512MB which is much smaller than high global graphics memory space. There is no free space in low graphics memory space spared for slots.

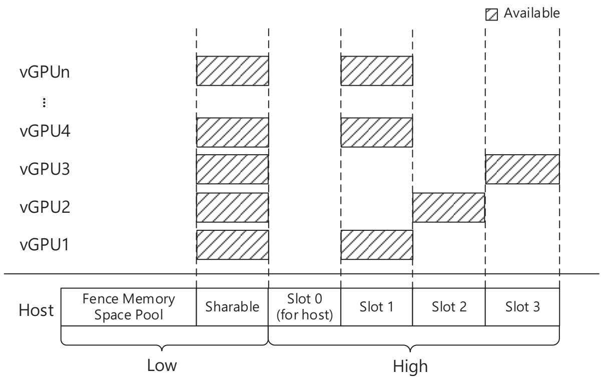


Figure 9: Slot Sharing

Optimized Scheduler gScale does not conduct context switch for idle vGPU instances, and this saves the cost of context switch and private shadow GTT copying. For vGPU instances without workloads, they do not submit commands to physical engine. gScale skips them, and focuses on serving the instances with heavy workloads. At the same time, gScale does not copying entries from idle vGPU's private shadow GTT to physical GTT. With slot sharing, if there is only one active vGPU in a slot, this vGPU will own the slot. Actually, gScale keeps its high global memory part of private shadow GTT on physical GTT without entry copying. In fact, slot sharing reduces the overhead of private shadow GTT copying, and we have a micro overhead analysis in Section 5.4.

gScale currently has 4 slots (1536MB/384MB = 4): one is reserved for host vGPU, while the rest 3 are shared by guest vGPUs. Slot sharing helps gScale improve guest vGPU's performance under a high instance density while only a few vGPUs are busy. We believe slot sharing could be utilized if the cloud provider deploys the guest VMs meticulously. For example, cloud providers let a busy vGPU share one slot with a few idle vGPUs.

5 Evaluation

In this section, we evaluate the scalability of gScale when it hosts increasing quantity of guest vGPUs with intensive workloads. gScale scales well for GPU intensive workloads, which achieves up to 81% performance of gVirt when it scales to 15 vGPUs. We compare the performance of gScale with gVirt, and it turns out gScale brings negligible performance change. Also, the performance of gScale and its basic version (without slot sharing) under a high density of instances is compared. In our experiments, slot sharing improves the performance of gScale up to 20%, and mitigates the overhead caused by copying private shadow GTT entries up to 83.4% under certain circumstances.

5.1 Experimental Setup

Host Machine Configuration	
CPU	Intel E3-1285 v3 (4 Cores, 3.6 GHz)
GPU	Intel HD Graphics P4700
Memory	32GB
Storage	SAMSUNG 850Pro 256GB * 3
Host VM Configuration	
vCPU	4
Memory	3072 MB
Low Global GM	64 MB
High Global GM	384 MB
OS	Ubuntu 14.04
Kernel	3.18.0-rc7
Linux/ Windows Guest VM Configuration	
vCPU	2
Memory	1800 MB/ 2048MB
Low Global GM	64 MB/ 128 MB
High Global GM	384 MB
OS	Ubuntu 14.04/ Windows 7

Table 1: Experimental Configuration

Configurations All the VMs in this paper are run on one server configured as Table 1, and gScale is applied on gVirt's 2015Q3 release as a patch. To support higher resolution, fence registers have to serve larger graphics memory range. In our test environment, gScale reserves 300MB low global graphics memory size to be the fence memory space pool, and this is enough for 15 VMs under the 1920*1080 resolution.

Benchmarks We mainly focus on the 3D workloads, for that in cloud environment, 3D computing is still the general GPU intensive workload. Some 2D workloads are covered too. However, we only use 2D workloads to prove the full functionality of vGPUs hosted by gScale, because 2D workloads can also be accelerated by CPU. For Linux 3D performance, we choose the Phoronix Test Suit 3D marks[3], including Lightsmark,

[3]Phronix Test Suit, `http://www.phoronix-test-suite.com/`

Nexuiz, Openarena, Urbanterror, and Warsow. Cairo-perf-trace[4] which contains a group of test cases is picked to evaluate Linux 2D performance. For Windows, we use 3DMark06[5] to evaluate 3D performance. PassMark[6] is chosen to evaluate 2D functionality. All the benchmarks are run under the 1920*1080 resolution.

Methodology We have implemented a test framework that dispatches tasks to each VM. When all the tasks are completed, we collect the test results for analysis. When gScale hosts a large amount of VMs, I/O could be a bottleneck. So we install 3 SSD drives in our server and distribute the virtual disks of VMs in these SSD drives to meet VM's I/O requirement. For 3DMark06, the loading process takes a great amount of time, which leads to an unacceptable inaccuracy. Moreover, VMs start loading at the same time, but they cannot process rendering tasks simultaneously due to the different loading speed. To reduce the inaccuracy caused by loading, we run the 3DMark06 benchmark by splitting it into single units and repeat each unit for 3 times. The single units in 3DMark06 are GT1-Return To Proxycon, GT2-Firefly Forest, HDR1-Canyon Flight and HDR2-Deep Freeze, and they are for SM2.0 and SM3.0 performance.

5.2 Scalability

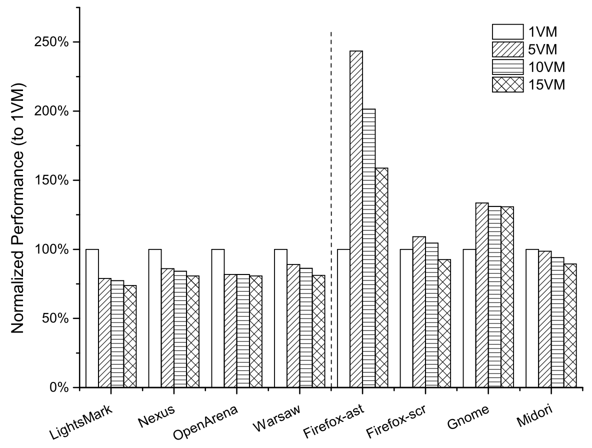


Figure 10: Scalability of gScale in Linux

We present the scalability results on Linux and Windows. Figure 10 shows the 2D and 3D performance of Linux VMs hosted by gScale, scaling from 1 to 15, and the results of all the tests are normalized to 1VM. All the 3D performance in this paper is measured by value of frame per second (FPS) given by benchmarks. For most of our test cases, there is a clear performance degradation when the number of VMs is over 1, due to the overhead from copying private shadow GTT entries. The maximal degradations of Lightsmark, Nexuiz, Openarena, and Warsow are 26.2%, 19.2%, 19.3%, and 18.9%, respectively. In Section 5.4, we will give a detailed analysis about the overhead of private shadow GTT copying. For 3D workload Lightsmark, Nexuiz, Openarena, and Warsow, scaling from 5VM to 15VM, gScale achieves a negligible performance change. It demonstrates that GPU resource is efficiently shared among multiple VMs. For 2D workload, firefox-ast and gnome increase their performance from 1VM to 5VM, for that 2D workloads are also accelerated by CPU.

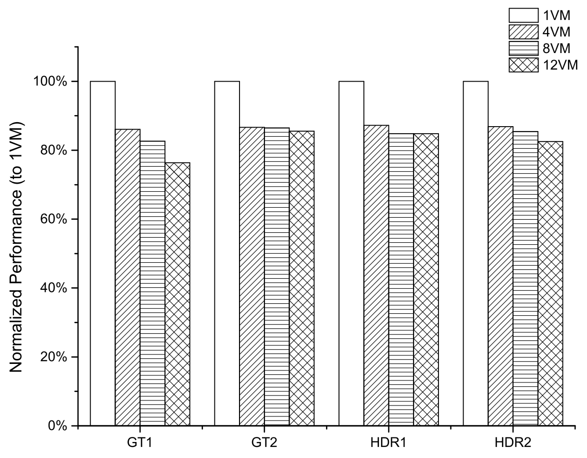


Figure 11: Scalability of gScale in Windows

The 3D performance of Windows VMs hosted by gScale scaling from 1 to 12 is in Figure 11, and all the test results are normalized to 1 VM. Similar with Linux, there is a visible performance degradation for each case when the number of VMs is over 1, and the maximal degradations of GT1, GT2, HDR1, and HDR2 are 23.6%, 14.5%, 15.2%, and 17.5%, respectively. The cause of degradation is the same with Linux VMs, which will be analyzed in Section 5.4. The performance scales well from 5VMs to 12VMs, and it proves that GPU resource is efficiently utilized when the number of VMs increases.

5.3 Performance

Comparison with gVirt We compare the performance of gScale with gVirt in Figure 12, and the performance of gScale is normalized to gVirt. We examine the settings of 1-3 VMs. Note, gVirt can only support 3 guest vGPUs. For Linux, gScale achieves up to 99.89% performance of gVirt, while for Windows, gScale archives up to 98.58% performance of gVirt. There is a perfor-

[4]Cairo, `http://http://cairographics.org/`

[5]Cario, `http://www.futuremark.com`

[6]PassMark, `http://www.passmark.com`

mance drop which is less than 5% of normalized performance when the number of instances is over 1. The performance decrease is due to copying the part of private shadow GTT for low graphics memory, and we will have a micro analysis in Section 5.4. This overhead is inevitable, for that global graphics memory space sharing will incur the overhead of copying private shadow GTT.

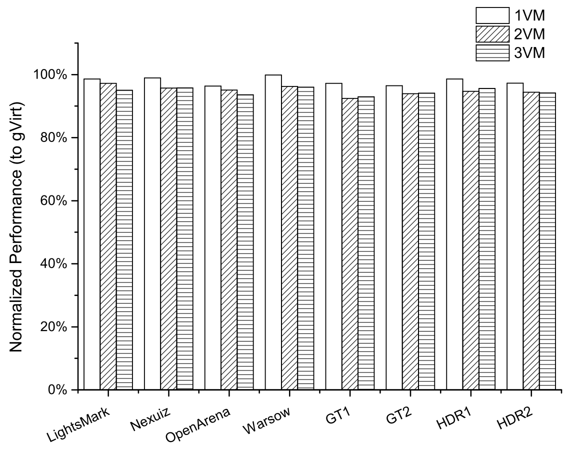


Figure 12: Performance Comparison

Performance Impact of Slot Sharing In this experiment, we want to evaluate the slot sharing of gScale under a high instance density. We launch 15 VMs at the same time. However, we only run GPU intensive workloads in a few of them, while the rest VMs remain idle. Note here, an idle VM means a launched VM without GPU workload. We increase the number of busy VMs from 1 to 15, and observe the performance change. We use gScale-Basic to represent the gScale without slot sharing.

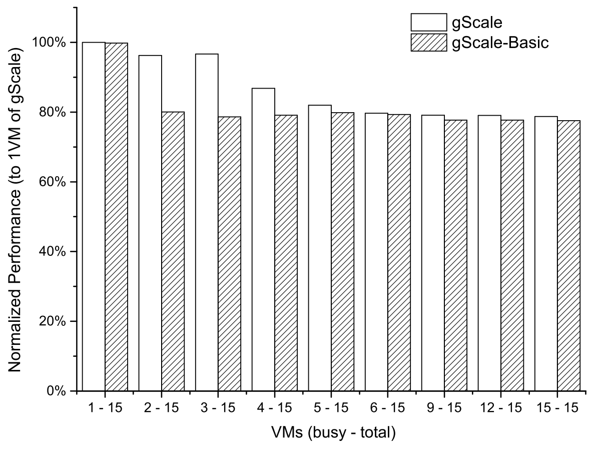


Figure 13: 3D Performance of Linux VMs

For 3D performance of gScale in Linux, we pick Nexuiz as a demonstration, and the case is run in an increasing number of VMs while gScale hosts 15 VMs in total, as shown in Figure 13. gScale and gScale-Basic has the same performance when the busy VM is only one. When the number of busy VMs is over 1, private shadow GTT copying happens. There is a 20% performance decrease for gScale-Basic. However, gScale has little performance degradation when the number of busy VMs is less than 4, and slot sharing mitigates the performance degradation when the number of active VMs is less than 6. However, when active VMs exceed 6, the slot sharing does not help with the overhead, and the performance is stable around 80% of normalized performance.

For 3D performance of gScale in Windows, GT1 is chosen to run in the rising number of VMs while gScale hosts 12 VMs in total. gScale shows the same performance with gScale-Basic when there is only one busy VM. However, similar to Linux, when the number of busy VMs is over 1, there is a 16.5% performance degradation for gScale-Basic. gScale achieves a flat performance change when the number of busy VMs is less than 4, and the results show that slot sharing mitigates the performance degradation before the number of busy VMs reaches 6. When busy VMs exceed 6, the performance of gScale and gScale-Basic is very close.

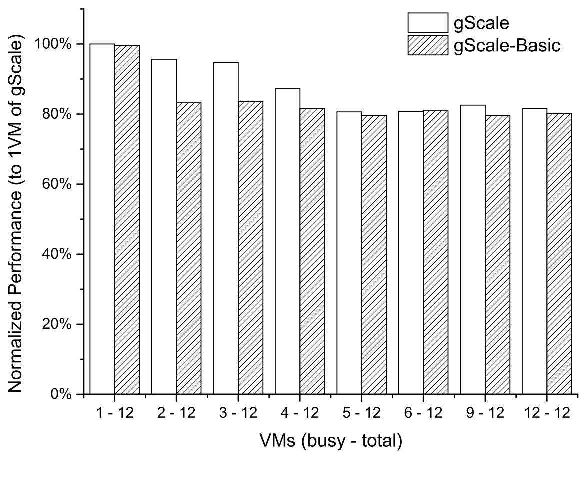


Figure 14: 3D Performance of Windows VMs

5.4 Micro Analysis

Overhead of Private Shadow GTT We evaluate that overhead caused by copying private shadow GTT to confirm the performance optimized by slot sharing. Lightsmark and HDR2 are chosen to be the workloads in Linux and Windows VMs, respectively. Actually, we study the difference of overhead between gScale and gScale-Basic. For Linux, we launch 15 VMs, and run workloads in 3 of them. For Windows, run workloads in 3 VMs while total 12 VMs are launched. We measure the time of private shadow GTT copying and the time vGPU owns the physical engine in each schedule. Then, we collect the data from about 3000 schedules, and calculate the percentage of how much time gScale uses to do the private shadow GTT copying in each schedule.

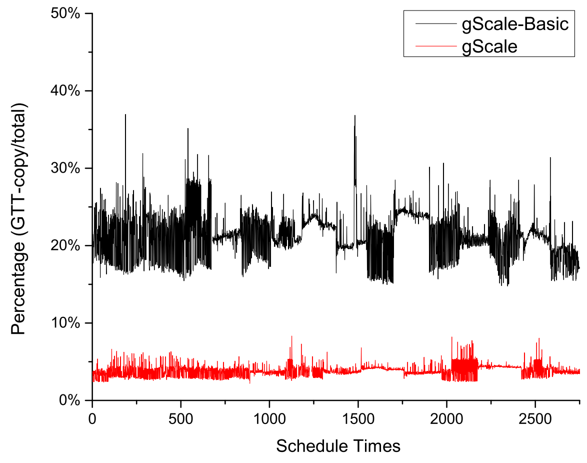


Figure 15: Overhead of Private Shadow GTT Copying in Linux

Figure 15 shows the overhead of gScale in Linux, for gScale-Basic (without slot sharing), the average overhead is 21.8%, while the average overhead of gScale is only 3.6%. In this case, slot sharing reduces the overhead of private shadow GTT copying by 83.4%. The overhead is dithering around the average value, for that shadow GTT copying needs memory bandwidth and CPU resource, which are also occupied by 3D workload.

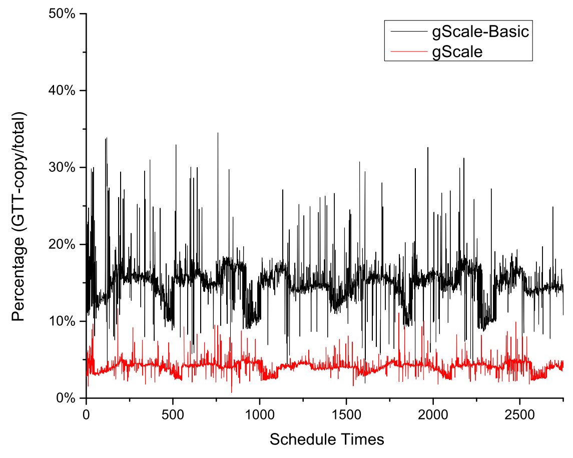


Figure 16: Overhead of Private Shadow GTT Copying in Windows

Figure 16 shows the overhead of private shadow GTT copying in Windows, for gScale-Basic, the average overhead is 15.35%, while the average overhead of gScale is only 4.16%. In this case, slot sharing reduces the overhead of private shadow GTT copying by 72.9%. The slot sharing works better for Linux, because it only optimizes the overhead from high global graphics memory part of private shadow GTT copying, while we configure vGPU with twice the amount of low global graphics memory in Windows of that in Linux. However, the overhead caused by the low graphics memory part of private shadow GTT copying is less than 5%, which is acceptable.

	LightsMark	Nexuiz	Openarena	Warsow	SM2.0	HDR
L Mapping(k)	18.8	4.67	4.9	6.6	10.2	8.1
GTT Modify(k)	455.3	313.5	228.5	1629.9	1134.2	1199.7
Percentage	4.13%	1.49%	2.14%	0.40%	0.90%	0.68%

Table 2: Frequency of Ladder Mapping

Frequency of Ladder Mapping Ladder mapping is constructed by gScale when CPU modifies the entry of GTT. We try to figure out the frequency of ladder mapping when 3D workloads are running. We count the total times of GTT modifications and the times of ladder mapping to calculate the percentage as shown in Table 2. For Windows workloads, the ladder mapping happens very rarely, which is less than 1%, which explains the flat change of performance in our scalability evaluation. For Linux, the percentage of ladder mapping frequency is higher than Windows, and we believe the reason is that the total amount of GTT modifications in Windows is a lot more than in Linux (up to 8x). At the same time, we observe a phenomenon that the ladder mapping mostly happens when workloads are loading, and it seldom happens when workloads are being processed.

6 Discussion

Currently, gScale only supports Intel Processor Graphics. However, the principle of our design can be applied to other architectures. In addition to Intel, vendors like AMD, Qalcomm and Samsung also have integrated CPU/GPU systems and the graphics memory is also served by system memory [24]. Our ladder mapping could be applied to their solutions if they have similar requirements. Some GPUs may have dedicated graphics memory, but they also use graphics translation tables to do address translation. We believe the concept of gScale's private shadow GTT could also help them share the graphics memory space. Similar to shadow page table, which is also a general method to virtualize memory, this sharing scheme of gScale could also inspire them to increase the user density and resource consolidation.

7 Related Work

Using modern GPUs in a shared cloud environment remains challenge with a good balance among performance, features and sharing capability [28]. A lot of research efforts have been done to enable GPUs in virtual machines (i.e., Device emulation, API forwarding, Device Pass-through, and full GPU virtualization).

Device emulation is considered impractical because GPU hardware is vendor-specific and modern GPUs are complicated. Thus, QEMU [13] has emulated a legacy VGA device with a low performance to support only some basic functionality.

API forwarding has been widely studied and has been applied to many virtualization software already. By installing a graphics library in a guest OS, graphic commands can be then forwarded to the outside host OS. Host OS can execute those commands directly using the GPU hardware. WireGL [19] and Chromium [20] intercept OpenGL commands and parallelly render them on commodity clusters. VMGL [21] makes use of Chromium to render guest's OpenGL commands on the host side. GViM [18], rCUDA [17], and vCUDA [25] virtualize GPGPU applications by forwarding CUDA commands in virtual machines. However, one major limitation of API forwarding is that the graphic stack on guest and host must match. Otherwise, host OS is not able to process guest's commands. For example, a Linux host cannot execute DirectX commands forwarded by a Windows guest. As a result, a translation layer must be built for Linux host to execute DirectX commands: Valve [8] and Wine [10] have built such translation layers, but only a subset of DirectX commands is supported; VMWare [16] and Virgil [9] implement a graphic driver to translate guests' commands to their own commands.

Device Pass-through achieves high performance in GPU virtualization. Recently, Amazon [2] and Aliyun [1] have provided GPU instances to customers for high performance computing. Graphic cards can be also passed to a virtual machine exclusively using Intel VT-d [12, 14]. However, direct pass-through GPU is dedicated, and also sacrifices the sharing capability.

Two full GPU virtualization solutions have been proposed, i.e., gVirt [28] and GPUvm [26, 27], respectively. GPUvm implements GPU virtualization for NVIDIA cards on Xen, which applies several optimization techniques to reduce overhead. However, full-virtualization will still cause non-trivial overhead because of MMIO operations. A para-virtualization is also proposed to improve performance. Furthermore, GPUvm can only support 8 VMs in their experimental setup. gVirt is the first open source product level full GPU virtualization solution in Intel platforms. It provides each VM a virtual full fledged GPU and can achieve almost native speed. Recently, gHyvi [15] uses a hybrid shadow page table to improve gVirt's performance for memory-intensive workloads. However, gHyvi inherits the resource partition limitation of gVirt, so it also suffers from the scalability issue too.

NVIDIA GRID [7] is a commercial GPU virtualization product, which supports up to 16 VMs per GPU card now. AMD has announced its hardware-based GPU virtualization solution recently. AMD multiuser GPU [3], which is based on SR-IOV, can support up to 15 VMs per GPU. However, neither NVIDIA nor AMD provides public information on technical details.

8 Conclusion and Future Work

gScale addresses the scalability issue of gVirt with a novel sharing scheme. gScale proposes the *private shadow GTT* for each vGPU instance, which allows vGPU instances to share the part of global graphics memory space only visible to GPU. A *ladder mapping* mechanism is introduced to force CPU directly access host physical memory space serving the graphics memory without referring to global graphics memory space. At the same time, fence memory space pool is reserved from low graphics memory space to ensure the functionality of fence registers. gScale also implements *slot sharing* to improve the performance of vGPU under a high instance density. Evaluation shows that gScale scales well up to 15 vGPU instances in Linux or 12 vGPU instances in Windows, which is 5x and 4x scalability compared to gVirt. Moreover, gScale archives up to 96% performance of gVirt under a high density of instances.

As for future work, we will focus on optimizing the performance of gScale, especially when gScale hosts large amount of instances with intensive workloads. To exploit the performance improvement of slot sharing, we will design a dynamic deploy policy based on the workload of instances.

References

[1] Alihpc. `https://hpc.aliyun.com/product/gpu_bare_metal/`.

[2] Amazone high performance computing cloud using gpu. `http://aws.amazon.com/hpc/`.

[3] Amd multiuser gpu. `http://www.amd.com/en-us/solutions/professional/virtualization`.

[4] Intel graphics driver. `http://www.x.org/wiki/IntelGraphicsDriver/`.

[5] Intel graphics virtualization technology (intel gvt). `https://01.org/zh/igvt-g`.

[6] Intel open source hd graphics programmer's reference manual (prm). `https://01.org/linuxgraphics/documentation/hardware-specification-prms`.

[7] Nvidia grid: Graphics-accelerated virtualization. `http://www.nvidia.com/object/grid-technology.html`.

[8] Valve togl. `https://github.com/ValveSoftware/ToGL`.

[9] Virgil 3d gpu project. `https://virgil3d.github.io`.

[10] Wine project. `http://winehq.org/`.

[11] Xengt setup guide. `https://github.com/01org/XenGT-Preview-kernel/blob/master/XenGT_Setup_Guide.pdf`.

[12] ABRAMSON, D. Intel virtualization technology for directed i/o. *Intel technology journal 10*, 3 (2006), 179–192.

[13] BELLARD, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[14] DONG, Y., DAI, J., HUANG, Z., GUAN, H., TIAN, K., AND JIANG, Y. Towards high-quality i/o virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 12.

[15] DONG, Y., XUE, M., ZHENG, X., WANG, J., QI, Z., AND GUAN, H. Boosting gpu virtualization performance with hybrid shadow page tables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 517–528.

[16] DOWTY, M., AND SUGERMAN, J. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[17] DUATO, J., PENA, A. J., SILLA, F., MAYO, R., AND QUINTANA-ORTÍ, E. S. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on* (2010), IEEE, pp. 224–231.

[18] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing* (2009), ACM, pp. 17–24.

[19] HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. Wiregl: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 129–140.

[20] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. In *ACM Transactions on Graphics (TOG)* (2002), vol. 21, ACM, pp. 693–702.

[21] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. Vmm-independent graphics acceleration. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), ACM, pp. 33–43.

[22] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal 10*, 3 (2006).

[23] PHULL, R., LI, C.-H., RAO, K., CADAMBI, H., AND CHAKRADHAR, S. Interference-driven resource management for gpu-based heterogeneous clusters. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (2012), ACM, pp. 109–120.

[24] PICHAI, B., HSU, L., AND BHATTACHARJEE, A. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. *ACM SIGPLAN Notices 49*, 4 (2014), 743–758.

[25] SHI, L., CHEN, H., SUN, J., AND LI, K. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on 61*, 6 (2012), 804–816.

[26] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. Gpuvm: why not virtualizing gpus at the hypervisor? In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference* (2014), USENIX Association, pp. 109–120.

[27] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. Gpuvm: Gpu virtualization at the hypervisor. *Computers, IEEE Transactions on PP*, 99 (2015), 1–1.

[28] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A full gpu virtualization solution with mediated pass-through. In *Proc. USENIX ATC* (2014).

[29] WALDSPURGER, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.