Low-overhead dynamic sharing of graphics memory space in GPU virtualization environments

Minwoo Gu, Younghun Park, Youngjae Kim & Sungyong Park

Cluster Computing
The Journal of Networks, Software Tools and Applications

ISSN 1386-7857

Cluster Comput
DOI 10.1007/s10586-019-02967-5

Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science+Business Media, LLC, part of Springer Nature. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

Cluster Computing
https://doi.org/10.1007/s10586-019-02967-5

Low-overhead dynamic sharing of graphics memory space in GPU virtualization environments

Minwoo Gu[1] · Younghun Park[1] · Youngjae Kim[1] · Sungyong Park[1]

Received: 1 January 2019 / Revised: 5 June 2019 / Accepted: 23 July 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
The proliferation of GPU intensive workloads has created a new challenge for low-overhead and efficient GPU virtualization solutions over GPU clouds. *gVirt* is a full GPU virtualization solution for Intel's integrated GPUs that share system's on-board memory for graphics memory. In order to solve the inherent scalability limitation on the number of simultaneous virtual machines (VM) in *gVirt*, *gScale* proposed a dynamic sharing scheme for global graphics memory among VMs by copying the entries in a private graphics translation table (GTT) to a physical GTT along with a GPU context switch. However, copying entries between private GTT and physical GTT often causes significant overhead, which becomes worse when the global graphics memory space shared by each VM is overlapped. This paper identifies that the copy overhead caused by GPU context switch is one of the major bottlenecks in performance improvement and proposes a low-overhead dynamic memory management scheme called DymGPU. DymGPU provides two memory allocation algorithms such as size-based and utilization-based algorithms. While the size-based algorithm allocates memory space based on the memory size required by each VM, the utilization-based algorithm considers GPU utilization of each VM to allocate memory space. DymGPU is also dynamic in the sense that the global graphics memory space used by each VM is rearranged at runtime by periodically checking idle VMs and GPU utilization of each runnable VM. We have implemented our proposed approach in *gVirt* and confirmed that the proposed scheme reduces GPU context switch time by up to 53% and improved the overall performance of various GPU applications by up to 39%.

Keywords GPU virtualization · Memory management · GPU scheduling · Integrated GPU

1 Introduction

With the recent advances in computing and hardware technologies, various types of GPUs [2, 3] have recently been used for performance acceleration in graphics-intensive applications such as 2D and 3D rendering. This has led to a situation where cloud service providers (CSP) start offering GPU instances over virtualized clouds. In order to provide high performance GPU services over virtualized clouds, many GPU virtualization techniques have been proposed. Application programming interface (API) remoting [4–6] is a technique that intercepts high-level client's API calls and forwards them to the host for processing. Although this approach is simple to implement, it depends on the version of API library or GPU driver, which lacks flexibility and cannot provide full GPU features. Direct pass-through [7] dedicates a GPU to a single virtual machine (VM) and allows it to use GPU directly without hypervisor intervention. This technique provides high

A preliminary version of this article [1] was presented at the 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 2018.

✉ Sungyong Park
parksy@sogang.ac.kr

Minwoo Gu
mwgu@sogang.ac.kr

Younghun Park
parkyh93@sogang.ac.kr

Youngjae Kim
youkim@sogang.ac.kr

[1] Department of Computer Science and Engineering, Sogang University, 35, Baekbeom-ro, Mapo-gu, Seoul, Republic of Korea

Author's personal copy

performance at the cost of prohibiting sharing of GPU among VMs. To alleviate the problems of aforementioned approaches, full GPU virtualization solutions at the hypervisor level such as *gVirt* [8] and *GPUvm* [9] are introduced.

Among these, *gVirt* is a full GPU virtualization solution for Intel's integrated GPUs that provides mediated pass-through capability. The original *gVirt* could support only up to three VMs owing to insufficient GPU memory. *gScale* [10] solved this problem by partitioning global graphics memory space into fixed size slots and allocating them to each VM so that multiple VMs can share the global graphics memory space. The accesses to global graphics memory space are then translated to those to system memory by using a physical graphics translation table (GTT). Since each VM needs to see the whole view of global graphics memory space, it also maintains a private GTT so that the entries in a private GTT are copied to a physical GTT whenever a VM is scheduled to run. From an in-depth analysis of GPU context switch, which will be discussed in Sect. 2, we found that GPU context switch incurs non-trivial overhead and the cost of copying GTT entries is extremely high. This leads to VM throughput degradation.

There have been few research efforts to address the performance problems resulting from the costs of GPU context switch. GPUswap [11] and GPrioSwap [12] proposed swapping policies to solve memory shortage problems on NVIDIA GPU. They transfer part of an application with low priority in internal graphics memory to system memory. Since the Intel's integrated GPU uses system memory as graphics memory, it is difficult to apply their schemes directly to *gVirt*. Also, they mainly focus on maintaining fairness between clients rather than reducing the GPU context switch overhead that occurs during memory swap. *gScale* recently proposed a proactive approach [13] that copies a private GTT to a physical GTT before context switch. However, this approach requires the change in scheduler in order to optimize performance, which is not portable and cannot be used in general.

In this paper, we first show that GPU context switch creates a bottleneck in performance improvement. Based on this inference, we propose a dynamic memory management scheme called DymGPU. DymGPU includes two dynamic memory allocation algorithms that minimize the overlapping of global graphics memory space used by each VM and thus reduce GPU context switch overhead. The selection of an appropriate algorithm is currently made by users before starting DymGPU.

In summary, this paper makes the following specific contributions:

- DymGPU provides two low-overhead memory management algorithms: size-based and utilization-based. The size-based algorithm allocates global graphics memory space based on the memory size required by each VM such that the memory space shared among VMs is minimized. This is because when part of global graphics memory space is shared by two or more VMs, the copying of the entries in private GTT to physical GTT during GPU context switch is an unavoidable step, regardless of the number of VMs involved. On the other hand, in a utilization-based algorithm, if VMs with higher GPU utilization share global graphics memory space with other VMs, it is more likely that the copies can be made several times. DymGPU reduces the number of copies made as far as possible by ensuring that VMs with higher GPU utilization do not share memory with other VMs.
- DymGPU is dynamic in the sense that the global graphics memory space used by each VM is rearranged at runtime by periodically checking idle VMs and GPU utilization of each runnable VM.
- We have implemented our approach in the 2016Q4 version of *gVirt* on a Xen hypervisor [14]. We also incorporated *gScale*'s GPU memory sharing technique into *gVirt* in order to scale up to 15 Linux VMs. The benchmarking results show that DymGPU reduces GPU context switch time by up to 53% and improves the overall performance of various graphics applications by up to 39%.

The rest of this paper is organized as follows. Section 2 briefly introduces *gVirt* and discusses the motivations related to our proposed approach. Section 3 explains the overall architecture of DymGPU and its implementation issues in detail. Section 4 evaluates DymGPU against *gVirt*. Section 5 presents related works and Sect. 6 concludes this paper with possible future works.

2 Background and motivation

In this section, we briefly introduce *gVirt* and analyze its overhead incurred by GPU context switch and memory sharing scheme.

2.1 Background

gVirt, also known as Intel GVT-g, is a high-performance, full GPU virtualization technique for Intel's integrated GPUs [8]. This technique provides mediated pass-through capability that runs the native graphics driver in the guest. Therefore, the performance-critical resources can be directly accessed by a VM, while the hypervisor intervenes

Author's personal copy

only for privileged operations. Currently, two implementations based on both Xen hypervisor (called XenGT) and KVM hypervisor (KVMGT) are available. The initial *gVirt* implementation was restricted to run only up to 3 vGPU (virtual GPU) instances. *gScale* overcame this limitation by allowing global graphics memory space to be shared among multiple vGPU instances and scaled up to 15 vGPU instances in Linux and 12 vGPU instances in Windows. In this paper, we use *gVirt* as a GVT-g implementation for Xen (XenGT) in which *gScale* modifications are added.

In *gVirt*, a mediator in Dom0 schedules vGPUs in a round-robin manner for fair scheduling. Each vGPU is allotted a 16-ms time quantum that takes into account high GPU context switch cost and speed. After the assigned time is elapsed, vGPU state is saved and the state of next vGPU is restored. Since current Intel's integrated GPU cannot preempt GPU kernel, *gVirt* transmits a command to each vGPU so that it terminates execution within the time quantum by tracking command submission.

The 4 GB global graphics memory space in Intel's integrated GPU is divided into low global graphics memory that both CPU and GPU can access, and high global graphics memory that only GPU can access. The original *gVirt* partitions low global graphics memory for each vGPU such that it is not shared with other VMs. This restricts the number of vGPUs running at the same time. *gScale* modified the layout of low global graphics memory and creates a slot such that multiple vGPU can share the slot and swap related contents whenever each vGPU is ready to run. High global graphics memory is divided into 64 MB slots, and multiple contiguous slots can be assigned to each vGPU. When a new VM is created, *gVirt* computes and assigns a score for every slot based on whether the slot is shared by other vGPUs or not. For example, if a slot is not occupied by any vGPU, nothing is added to the score. Otherwise, weight, which is deliberately set to a very large value, is added to the score to minimize sharing of the slot with other vGPUs. After analyzing the scores from each slot, contiguous slots with the lowest scores are allocated to the vGPU. If there are multiple minimum scores, the leftmost slots are chosen. For example, if there are 5 slots and slots 1, 2, 3, and 4 are occupied by one VM, then the vGPU of a newly created VM that requires 2 slots is allocated to slots 4 and 5.

Figure 1 depicts how global graphics memory is mapped to system memory in *gVirt*. The logical address in global graphics memory is converted to a physical address in system memory using a physical GTT. In order to load a large number of vGPUs in a small global graphics memory space, *gScale* proposes a *slot sharing* mechanism that allows vGPUs to share global graphics memory space. Each vGPU has private GTTs for low and high global graphics memories. To activate a vGPU to switch in, private GTT entries of the vGPU that do not exist in the physical GTT are copied. Whenever a vGPU modifies the physical GTT, its private GTT is also synchronized. However, after the vGPU is switched out, the physical GTT which is mapped to low global graphics memory has entries of another vGPU and the CPU can't access the VM through aperture. To solve this problem, *gScale* implements *ladder mapping* and *fence memory space pool*. Ladder mapping maps guest physical address to host physical address directly. Fence memory space pool allows fence register to operate correctly.

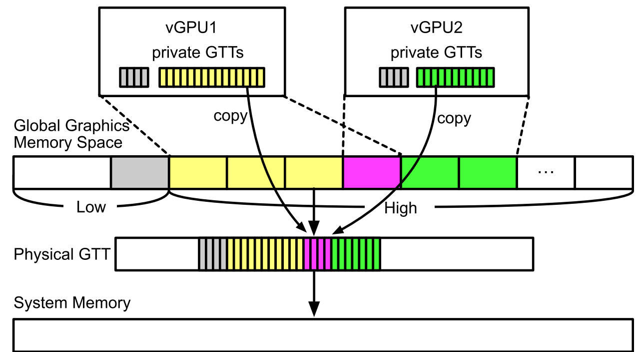


Fig. 1 Global graphics memory space and mapping in *gVirt*

2.2 GPU context switch overhead analysis

Global graphics memory includes frame and command buffers that store the pixel information of display and the graphics commands produced by CPU. In Linux VM, it is generally known that 64 MB and 384 MB in size are sufficient for low and high global graphics memory respectively, while 128 MB and 384 MB are recommended in Windows VM [15]. However, we found that small high global graphics memory can affect the performance of GPU workloads and can sometimes lead to a crash especially when a VM runs GPU workloads with many rendering operations or involves a high-resolution display environment such as quad-high-definition (QHD) and ultra-high-definition (UHD).

To confirm this, we measured average frames per second (FPS) of two 3D benchmarks, Unigine Valley [16] and Unigine Superposition [17], by varying high global

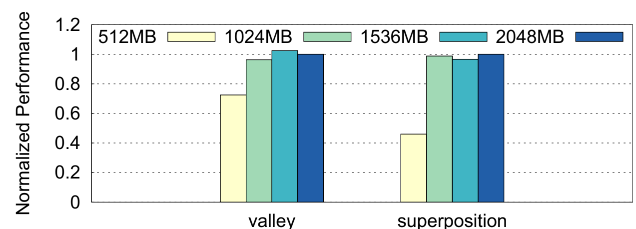


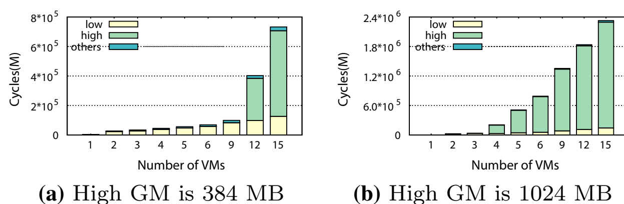
Fig. 2 Performance of 3D workloads

Author's personal copy

graphics memory size from 512 to 2048 MB. Figure 2 shows normalized performance with respect to the performance with 2048 MB size. As shown in Fig. 2, when high global graphics memory size is larger than 1024 MB, similar performance is observed. Whereas, when high global graphics memory size is 512 MB, the performance degrades severely because of insufficient high global graphics memory. These results indicate that large high global graphics memory can increase the performance of GPU workloads. However, it should be noted that the possibility of overlapping address spaces among vGPU instances also gets bigger, which incurs large overhead in a GPU context switch.

To analyze which operation takes the longest time during a GPU context switch and the effect of high global graphics memory size in VMs, we measured consumed CPU cycles taken for each of the following three activities in a GPU context switch by varying the number of VMs from 1 to 15: (1) private GTT copies in low global graphics memory, (2) private GTT copies in high global graphics memory, (3) other activities such as restoring render context, restoring ring buffers, and switching local page table. We conducted two experiments for which high global graphics memory size of each VM is set to 384 MB and 1024 MB, respectively. All VMs excluding Dom0 share 64 MB low global graphics memory and 3456 MB high global graphics memory. The testbed is described in greater detail in Sect. 4.1.

Figure 3 shows average consumed CPU cycles for private GTT copies in low and high global graphics memory. It also depicts the CPU cycles consumed by other activities in a GPU context switch. Since all VMs share a single low global graphics memory, a private GTT copy of the same size in low global graphics memory is made for every GPU context switch. As shown in Fig. 3a, when the number of VMs is less than or equal to 9, private GTT copies in low global graphics memory take most of the CPU cycles (approximately 84%). In this case, private GTT copy in high global graphics memory is made only the first time, because the VMs do not share high global graphics memory. However, when the number of VMs is greater than 9, they begin to share high global graphics memory and private GTT copies in low and high global graphics memory take about 17% and 79% of the total CPU cycles, respectively. On the other hand, if the size of high global graphics memory is increased to 1024 MB as shown in Fig. 3b, the consumed CPU cycles in high global graphics memory is more than 90% of the total CPU cycles. This explains that private GTT copy overhead forms a very large portion of GPU context switch overhead. Thus, private GTT copies that occur in low and high global graphics memory should be reduced to improve the performance of VMs.



(a) High GM is 384 MB **(b)** High GM is 1024 MB

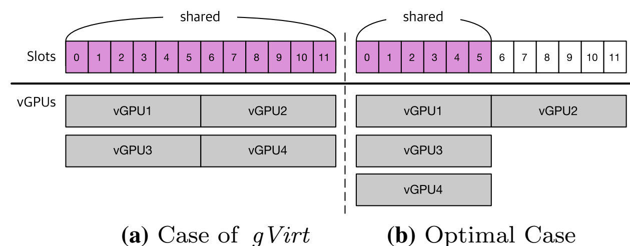
Fig. 3 Consumed CPU cycles of each context switch steps when high global graphics memory (GM) size is 384 MB and 1024 MB

2.3 Global graphics memory mapping analysis

As explained in Sect. 2.1, *gVirt* maps a vGPU to contiguous slots in high global graphics memory such that the number of shared vGPU per each slot is minimized. However, when one slot is shared by two or more vGPUs, GTT copy always occurs whenever vGPUs are switched in regardless of how many vGPUs share the slot.

Figure 4 compares the *gVirt*'s allocation scheme with that of optimal case, where 12 slots are initially available in high global graphics memory and each vGPU from vGPU1 to vGPU4 requires six slots. Assume that vGPUs are scheduled in a round-robin manner from vGPU1 to vGPU4. At the first round of scheduling in *gVirt*, vGPUs are allocated as shown in Fig. 4a and private GTTs from all vGPUs are copied to physical GTT because all physical GTT entries are initially empty. When vGPUs are scheduled again, private GTT copies should also be made since the maximum number of vGPUs that can be allocated exclusively is 2. Hence, when each vGPU is scheduled n times, the total number of GTT entry copies for the slots is $24 + 24(n - 1)$, which is the sum of initial GTT entry copies and the GTT entry copies at every iteration.

However, if vGPU4 is allocated to the same memory space that vGPU3 is assigned as shown in Fig. 4b, vGPU2 always occupies its own slots and no private GTT copies are needed for vGPU2. This reduces the total number of GTT entry copies to $24 + 18(n - 1)$. Considering that private GTT copies constitute a considerably large amount of CPU cycles in a GPU context switch, the difference of $6(n - 1)$ in private GTT copies affects the overall performance of VM, especially when n becomes larger.



(a) Case of *gVirt* **(b)** Optimal Case

Fig. 4 Comparison of GTT copy between *gVirt* and optimal case with an example

Author's personal copy

3 Design and implementation

In this section, we present the overall architecture of DymGPU and explore how to minimize private GTT copy overhead.

3.1 Overall architecture

Figure 5 presents the overall architecture of DymGPU. DymGPU consists of two modules: *monitor* and *memory allocator*. *Monitor* collects necessary information for each vGPU such as memory size required by vGPU, vGPU status (idle or active), GPU usage, and maintains a status table so that *memory allocator* uses for dynamic reallocation. For example, *monitor* periodically checks every vGPU and discovers idle vGPUs that have not been used for a certain period of time (threshold). The threshold value is configurable and currently set to 20 s. If an idle GPU occupies a slot in global graphics memory alone, memory space is wasted, which may affect the GPU throughput of the VMs running at a physical machine. In addition, *monitor* periodically collects GPU utilization of each vGPU every second. Instead of using GPU usage at a certain point in time, we average GPU utilization values for 20 s to determine the GPU utilization of each vGPU. The 20 s value is also a configuration parameter and can be set with different values.

Memory allocator allocates or reallocates global graphics memory space for each vGPU. In addition to adjusting memory space when a VM is created or destroyed, *memory allocator* dynamically allocates memory space based on the information obtained from *monitor*. *Memory allocator* provides two memory allocation algorithms such as size-based and utilization-based algorithms. In an environment where most VMs execute GPU applications with similar GPU utilization, the sized-based algorithm is preferred. In this case, DymGPU allocates memory space for each vGPU to minimize the number of shared slots based on the memory size required by each VM, which results in maximizing the number of slots that are used by only one vGPU. Whereas, if each vGPU has different GPU utilization patterns, the utilization-based algorithm is preferred. If vGPUs with high GPU utilization share the same slots, more private GTT copies are likely to happen for the shared slots. Therefore, DymGPU prevents vGPUs with high GPU utilization from sharing slots with other vGPUs as much as possible. In the rest of this section, we describe the detailed design of *memory allocator*.

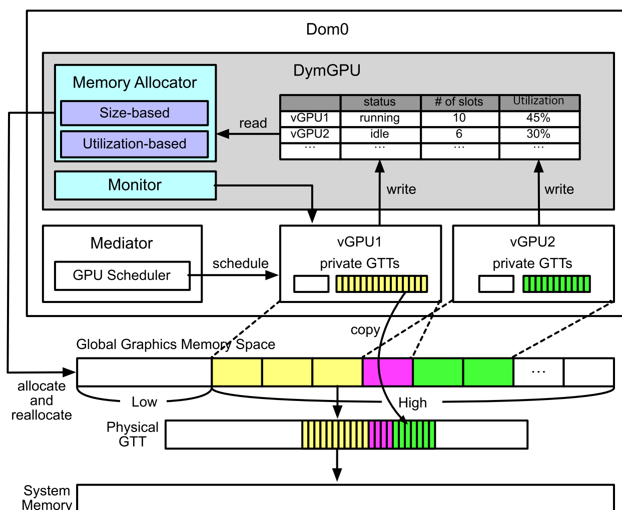


Fig. 5 Architecture of DymGPU

3.2 Memory allocator

3.2.1 Size-based allocation algorithm

In order to minimize the number of private GTT copies, we suggest a greedy algorithm, which minimizes the number of shared slots in high global graphics memory.

Let the set of vGPUs be $V = \{V_0, V_1, \ldots, V_{N-1}\}$, where N is the number of vGPUs. Each V_i has two parameters: S_i which is the number of required slots, and P_i which is the start slot index of V_i. Suppose that we place V on high global graphics memory with M slots. Figure 6 depicts the vGPU mapping scheme. *Memory allocator* sorts V in non-increasing order of required slots and places vGPUs in order of slot size beginning from the leftmost free slots until the vGPU does not share slots with another. In Fig. 6, V_0, ..., V_{K-1} belong to that case. And then V_K is placed at the rightmost end of the high global graphics memory. Finally, the remaining vGPUs are allocated starting from the leftmost V_K, so that V_{K+1}, ..., V_{N-1} are allocated on top of the overlapping L slots formed by V_{K-1} and V_K. For example, if M is 5 and there are four vGPUs, V_0, V_1, V_2, V_3 require 4, 4, 3, and 2 slots respectively. V_0, requiring the largest slot, is placed from slot 0 to slot 3. And V_1 is placed at the rightmost slot, because V_1 cannot be placed in the remaining one slot. And then remaining vGPUs, V_2, V_3 are placed on top of three shared slots made by V_0 and V_1. Therefore, P_0, P_1, P_2, and P_3 are 0, 1, 1, and 1, respectively. In this case, L is 3 and the private GTT copy occurs only on slot 2, 3, and 4 in a GPU context switch.

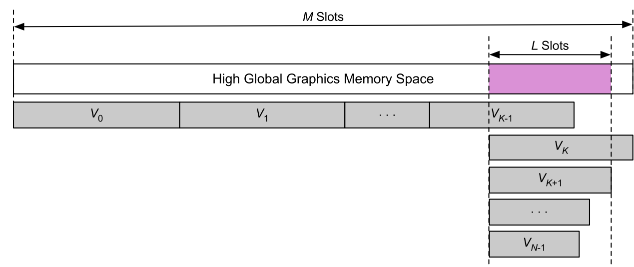


Fig. 6 Size-based allocation algorithm

Author's personal copy

This algorithm is activated when a new vGPU instance is created or an existing vGPU instance is terminated. If the number of required slots for new vGPU is less than or equal to L, the vGPU is mapped to P_K which is on top of shared slots. Otherwise, *memory allocator* compares the number of shared slots when the vGPU is mapped to P_K with the number of shared slots when all vGPUs are reallocated. Then *memory allocator* chooses the case where the number of shared slots is smaller. Likewise, when an existing vGPU is terminated, *memory allocator* reallocates the memory space used by other vGPUs if the terminated vGPU has slots that have not been shared with other vGPUs. *Memory allocator* also checks idle vGPUs every second, and processes them as if the vGPU were terminated. Algorithm 1 is the pseudo-code of size-based allocation algorithm.

Algorithm 1 `Size-based Algorithm`

Require:
M: the number of the slots of high GM
N: the number of vGPUs
$V = \{V_0, V_1, ..., V_{N-1}\}$: a set of vGPUs
$S = \{S_0, S_1, ..., S_{N-1}\}$: a set of the number of required slots of vGPUs
Ensure:
$P = \{P_0, P_1, ..., P_{N-1}\}$: a set of start slot index of vGPUs

1: Sort V and S in non-increasing order of S
2: $sum \leftarrow 0$
3: $index \leftarrow 0$
4: **while** $sum < M$ **do**
5: **if** $sum + S_{index} < M$ **then**
6: $P_{index} \leftarrow sum$
7: **else**
8: $P_{index} \leftarrow M - S_{index}$
9: $K \leftarrow index$
10: **end if**
11: $sum \leftarrow sum + S_{index}$
12: $index \leftarrow index + 1$
13: **end while**
14: **while** $index < N$ **do**
15: $P_{index} \leftarrow P_K$
16: $index \leftarrow index + 1$
17: **end while**
18: **return** P

3.2.2 Utilization-based allocation algorithm

As mentioned before, if vGPUs with low GPU utilization occupy slots alone, the memory space used by those vGPUs is wasted, resulting in performance degradation. The performance degradation increases as the difference in GPU utilization between VMs increases. Therefore, we suggest an additional greedy algorithm that considers the GPU utilization of vGPUs as shown in Fig. 7.

In this algorithm, DymGPU sorts V in non-increasing order of GPU utilization and places them in order of large GPU usage starting from leftmost free slots until the vGPU doesn't share the slots with other vGPUs. $V_0, V_1, \ldots, V_{K-1}$ are in that case. Remaining vGPUs, $V_K, V_{K+1}, \ldots, V_{N-1}$, are allocated to rightmost slots in high global graphics memory.

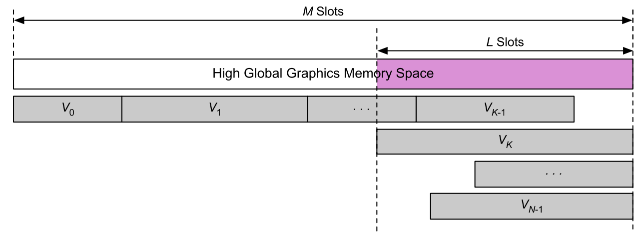


Fig. 7 Utilization-based allocation algorithm

Memory allocator reallocates memory space every 20 s based on GPU utilization. When each vGPU is reallocated, the slots occupied by the vGPU are invalidated and the private GTT entries mapped to the slots are copied to a new position. In the worst case, all slots in high global graphics memory are invalidated and many private GTT entry copies can be made. For this reason, *memory allocator* uses a relatively large interval to mitigate invalidation overhead. Since utilization-based allocation algorithm doesn't consider the number of slots allocated to vGPUs, it is possible that a large number of slots can be shared. However, this algorithm can reduce context switch overhead further than size-based allocation algorithm, as the frequency of GPU context switch generally depends on the GPU utilization of each vGPU. Algorithm 2 is the pseudo-code of the utilization-based allocation algorithm.

Algorithm 2 `Utilization-based Algorithm`

Require:
M: the number of the slots of high GM
N: the number of vGPUs
$V = \{V_0, V_1, ..., V_{N-1}\}$: a set of vGPUs
$S = \{S_0, S_1, ..., S_{N-1}\}$: a set of the number of required slots of vGPUs
$U = \{U_0, U_1, ..., U_{N-1}\}$: a set of the GPU utilization of vGPUs
Ensure:
$P = \{P_0, P_1, ..., P_{N-1}\}$: a set of start slot index of vGPUs

1: Sort V, S and U in non-increasing order of U
2: $sum \leftarrow 0$
3: $index \leftarrow 0$
4: **while** $sum < M$ **do**
5: **if** $sum + S_{index} < M$ **then**
6: $P_{index} \leftarrow sum$
7: **else**
8: $P_{index} \leftarrow M - S_{index}$
9: **end if**
10: $sum \leftarrow sum + S_{index}$
11: $index \leftarrow index + 1$
12: **end while**
13: **while** $index < N$ **do**
14: $P_{index} \leftarrow M - S_{index}$
15: $index \leftarrow index + 1$
16: **end while**
17: **return** P

Author's personal copy

3.3 Discussion

The main purpose of this paper is to propose a low-overhead dynamic sharing mechanism of graphics memory space for integrated GPUs and prove the proposed ideas over a publicly available software platform. Because the proposed mechanism should be implemented at the driver level and the availability of source code was very important, we chose the open source-based *gVirt* as a software platform to verify the proposed ideas.

Although DymGPU is currently targeted only at Intel's integrated GPUs, its design principle can be applied to other architectures such as AMD GPU as well, where the system memory is also used as GPU memory.

Whereas, in the dedicated GPUs that are equipped with separate graphics memory such as NVIDIA, we believe that DymGPU can help as they also use graphics translation table for address translation. However, the lack of zero-copy mechanism and unavailability of source code make it difficult to experiment our ideas over this platform.

4 Evaluation

This section compares the performance of two algorithms proposed in DymGPU with that of *gVirt* and shows how much overhead DymGPU can reduce in a GPU context switch. In order to conduct the experiments with up to 15 Linux VMs, we have extended *gVirt* so that it contains the scalability features provided by *gScale* such as *ladder mapping* and *fence memory space pool*.

For the extensive comparison with different workload patterns, we use Phoronix Test Suite [18] and Cairo-perf-trace [19], where various 3D and 2D workloads are included. Among them, we use *lightsmark*, *openarena*, *nexuiz*, *urbanterror* for 3D workloads and *firefox-asteroids* (*firefox-ast*), *firefox-scrolling* (*firefox-scr*), *gnome-system-monitor* (*gnome*), *midori* for 2D workloads. The performance of 3D benchmark is measured by average frame per second (FPS) and the performance of 2D benchmark is measured by execution time, and normalized with respect to that of single VM.

4.1 Experimental setup

Table 1 summarizes the configurations of a physical machine (PM) and virtual machines (VM) running on the PM used for experiments. The size of global graphics memory in the PM is set to 4 GB, where the low and high global graphics memory sizes are set to 256 MB and 3840 MB, respectively. While Dom0 uses global graphics memory alone, DomU shares low and high global graphics memory excluding the area reserved by Dom0. For experiments, we vary the size of high global graphics memory size in each VM from 384 to 1024 MB.

4.2 Overall performance

4.2.1 Performance Comparison with Similar GPU Utilization

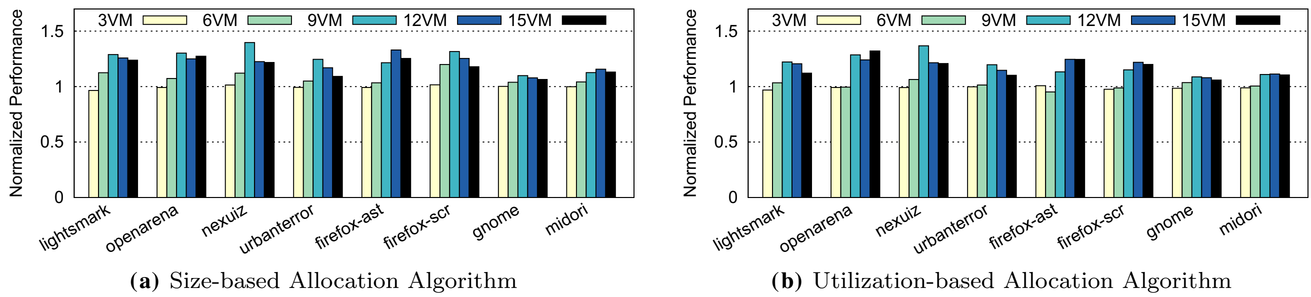
In order to evaluate the performance of DymGPU where the GPU utilization of VMs is similar, we run the same workload in each VM and check the performance as the number of VMs is increased by 3. Furthermore, the size of high global graphics memory in VMs participating in the experiment is varied among 384 MB, 704 MB, and 1024 MB with the same ratio (i.e., 1:1:1). It is worth mentioning that the performance of VMs with different high global graphics memory sizes varies as discussed in Sect. 2.2. The performance of DymGPU is normalized to that of *gVirt*.

Figure 8a, b show the normalized performance of two algorithms when all VMs execute the same workloads. When the number of VMs is 3, the performance of

Table 1 Experimental setup

Physical machine	
Processor	Intel Core i7-6700 3.40GHz (4 cores / 8 threads) / Intel HD Graphics 530
Memory	32 GB
Disk	Samsung SSD 850 PRO 256GB * 3
Host virtual machine (Dom0)	
vCPU / Memory	8/4 GB
Hypervisor	Xen version 4.6.0
OS	Ubuntu 16.04.1 (kernel version 4.3.0)
Low/high GM	64 MB / 384 MB
Guest virtual machine (DomU)	
vCPU / Memory	2/2 GB
OS	Ubuntu 16.04 (Kernel version 4.3.0)
Low GM	64 MB

Author's personal copy



(a) Size-based Allocation Algorithm **(b)** Utilization-based Allocation Algorithm

Fig. 8 Performance comparison with similar GPU utilization (3D and 2D workloads)

DymGPU is similar to that of *gVirt* because the total required size for high global graphics memory is still less than available size. However, as we increase the number of VMs, DymGPU outperforms *gVirt* for all workloads. It should be noted that DymGPU reduces the number of shared slots in the size-based algorithm and it also considers GPU utilization when allocating memory space. Moreover, the average performance improvement of size-based algorithm relative to the utilization-based algorithm for all benchmarks is 7%, 4%, 2%, and 1% for 6, 9, 12, and 15 VMs, respectively. That is, the performance of the size-based algorithm is better than that of the utilization-based algorithm when all VMs actively use GPUs. This is because reducing the number of shared slots is more important in a situation where all VMs have similar GPU usages. In case of *gnome* and *midori*, DymGPU shows little improvement than other workloads because the workloads submit few GPU commands and thus generate infrequent GPU context switches.

4.2.2 Performance comparison with various GPU utilization

To evaluate the performance with various GPU workloads, we define five sets of benchmarks by mixing workloads with different GPU utilization as shown in Table 2. That is, *lightsmark* is classified as a workload with low GPU utilization, while the *firefox-asteroids* and *urbanterror* are workloads with medium and high GPU utilization, respectively. For experiments, we run a total of 15 VMs and the number in the table represents the number of instances for each workload.

Figure 9 shows performance comparison with various workload sets. It is observed that the performance of DymGPU is better than *gVirt* for all five sets. Specially, the utilization-based algorithm outperforms the size-based algorithm except for set 2 and set 5. This can be explained by the fact that set 2 is mainly composed of *urbanterror*, which requires high CPU and GPU computations. In this case, GPU context switch overhead is hidden because CPU computation is already a performance bottleneck.

Table 2 Benchmark sets

Set sumber	Lightsmark	Firefox-asteroids	Urbanterror
Set 1	11	2	2
Set 2	2	2	11
Set 3	3	6	6
Set 4	6	3	6
Set 5	6	6	3

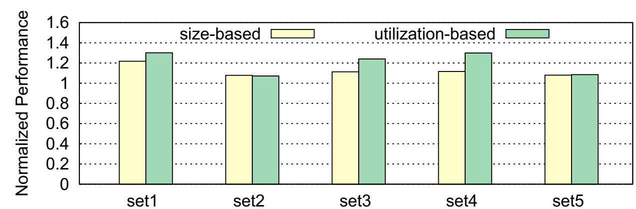


Fig. 9 Performance comparison with various GPU utilization

Furthermore, set 5 requires CPU computation more than GPU computation, resulting in less GPU context switches.

4.3 Private GTT copy overhead

In this experiment, we analyze private GTT copy overhead in high global graphics memory using the same workload types and benchmark sets explained in Sect. 4.2. The experiments are conducted by using *nexuiz* (3D workload) and *midori* (2D workload) as we increase the number of VMs.

Figure 10a, b show the number of private GTT copies normalized to that of *gVirt* when VMs execute the same workloads. As shown in Fig. 10a, both size-based and utilization-based algorithms reduce private GTT copies by up to 35% and 30% against *gVirt* for 3D workload. When the number of VMs is small, the size-based algorithm reduces private GTT copies further compared with the utilization-based algorithm. However, as we increase the number of VMs, the difference of overhead optimization

Author's personal copy

shrinks because the degree of memory sharing and the overhead of competition between two processors increase. For 2D workload, we observe similar results as shown in Fig. 10b. Figure 10c shows the number of private GTT copies when VMs execute various workloads with different GPU utilization. We observe that the utilization-based algorithm shows fewer slot copies than the size-based algorithm for all sets. Specially, the number of slot copies in the utilization-based algorithm is reduced by about 40% compared to the size-based algorithm for set 4. This is because set 4 consists of GPU workloads with various GPU utilization.

4.4 GPU context switch overhead

Reducing the number of GTT copies does not fully explain that GPU context switch overhead is definitely decreased. In order to analyze GPU context overhead further, we have conducted additional experiments using the NW, LUD, and K-MEANS applications from *Rodinia* [20] benchmark. For this, we measured the number of GPU context switches, CPU cycle consumed in each context switch, and total CPU cycles consumed in all GPU context switches by varying the number of VMs from 2 to 7. Figures 11 and 12 depict the corresponding results of the size-based algorithm and *gVirt*.

As we can see from Fig. 12, total CPU cycles consumed in all GPU context switches are significantly reduced for all three applications. The performance gap is bigger when the number of VMs is relatively small, and the gap becomes narrow as we increase the number of VMs. Interestingly, the size-based algorithm creates more GPU context switches than *gVirt*, while the number of consumed cycles per context switch is considerably reduced, as shown in Fig. 11. This is because the reduction of private GTT copies reduces the time spent in a context switch and allows CPU to schedule other VMs, which may cause more context switches.

4.5 Fairness

In DymGPU, the private GTT copy occurs only among the VMs that share the slots in the global graphics memory space. Therefore, the two memory allocation algorithms provided by the DymGPU seem to be unfair at the cost of minimizing private GTT copy overhead.

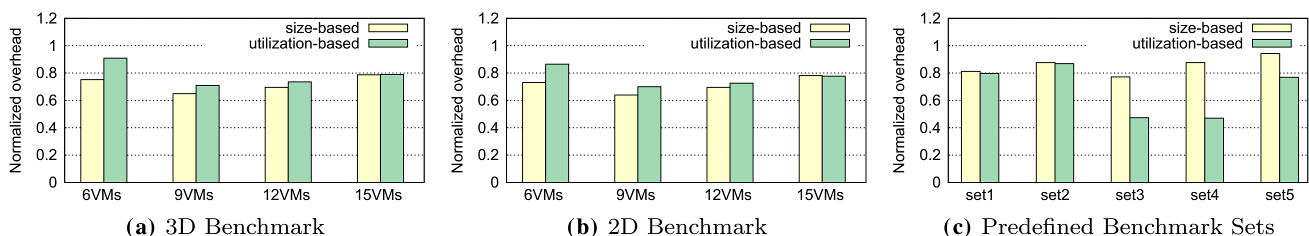
DymGPU schedules each vGPU in a round-robin manner based on the *actual execution time* which does not take the GPU context switch time into account. That is, the time spent for GPU context switch is excluded from the actual execution time and each vGPU is given a fair chance to run regardless of the location at which each vGPU resides.

To confirm this, we compare the GPU utilizations of three allocation algorithms (e.g., *gVirt*, size-based algorithm, utilization-based algorithm) when 15 VMs run *nexuiz* (3D workload). As shown in Fig. 13, the GPU utilizations of three algorithms fluctuate initially up to about 15 s and all of them finally converge to a similar utilization value (around 80–90%) after 20 s. The similar results were also observed when we run *Urbanterror* that is 2D workload with many I/O operations.

5 Related works

The main contribution of DymGPU is to propose a low-overhead dynamic memory management technique in a GPU virtualization environment. DymGPU targets at achieving high performance by reducing the number of GTT copies in a GPU context switch. Although there are few research activities to reduce the GPU context switch overhead, the rest of this section is devoted to introduce relevant research efforts related to GPU memory management techniques in a GPU virtualization environment.

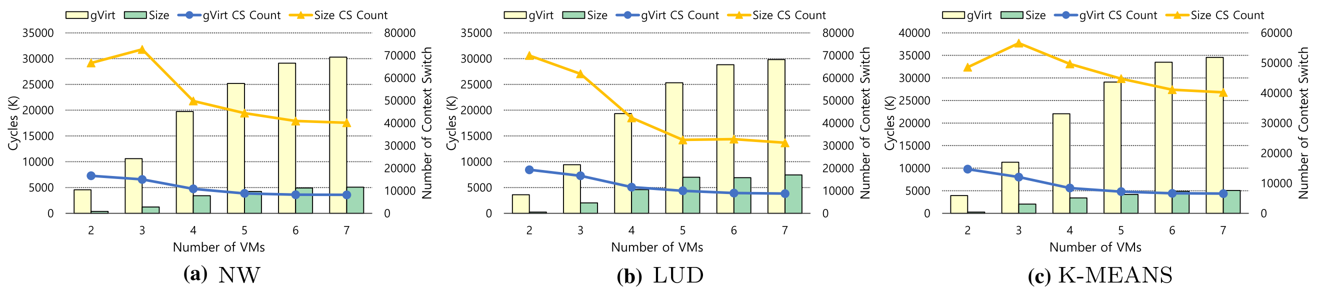
Multiple vGPUs in *gVirt* [8] can share low global graphics memory by swapping GTT entries whenever they are ready to run. However, the exclusive allocation scheme of GTT entries on high global graphics memory restricts simultaneous VMs up to 3. By adopting a sharing mechanism also in high global graphics memory, *gScale* [10] increased the number of simultaneous VMs to 15. The round robin-based allocation strategy in *gScale* still incurs a considerable overhead in a context switch. Recently, a low-overhead mechanism called predictive copy [13] that switches private GTT entries before context switch is proposed. Although this study aims to minimize context



(a) 3D Benchmark **(b)** 2D Benchmark **(c)** Predefined Benchmark Sets

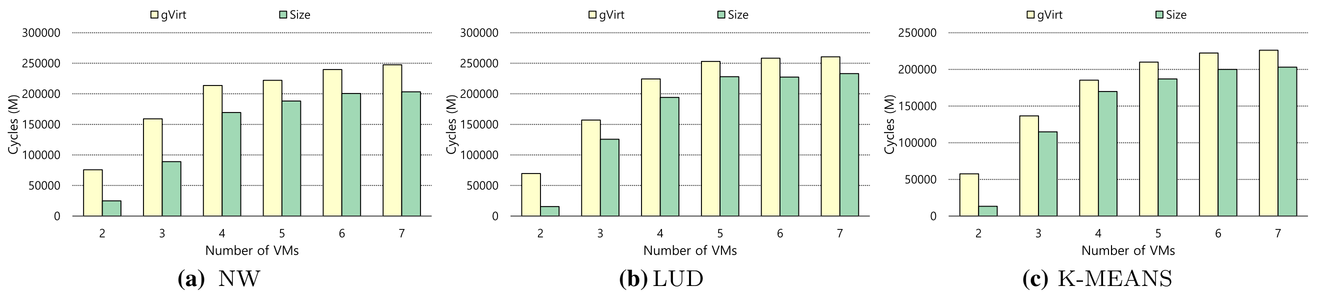
Fig. 10 Normalized number of private GTT copies

Author's personal copy



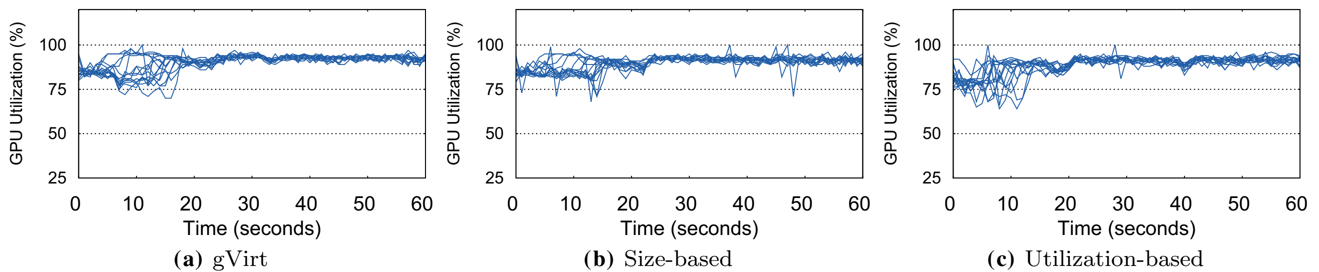
(a) NW (b) LUD (c) K-MEANS

Fig. 11 Number of context switches and consumed cycles per context switch



(a) NW (b) LUD (c) K-MEANS

Fig. 12 Total cycles consumed in context switches



(a) gVirt (b) Size-based (c) Utilization-based

Fig. 13 GPU utilization changes of 15 VMs running *nexuiz* in three algorithms

switch overhead, which is similar to ours, this approach requires a predictive copy aware scheduling and is thus dependent on a scheduler. Whereas, our approach does not require scheduler change and the number of GTT copies is the same as that in predictive copy approach.

Gdev [21], GDM [22], RSVM [23] and VMBR [24] solve a GPU memory insufficiency problem. They use system memory as backup memory, and copy data from GPU memory to system memory at runtime when applications need more memory to run. GPUswap [11] and GPrioSwap [12] propose swapping policies in NVIDIA GPU that aim to achieve resource fairness among applications and high GPU memory utilization. GPUswap divides buffer into fixed-size chunks called swap unit. When more memory is needed, GPUswap randomly chooses chunks from an application that occupied the largest memory space. Since a victim is randomly selected, the performance of the application can be degraded if the chunks include actively used or reusable data. To overcome this problem, GPrioSwap also proposes a technique to profile the number of memory accesses of GPU applications and selects a victim based on the priority defined by the number of accesses.

Traditional NVIDIA docker [25] environment simply assigns a GPU to a container, which causes program failure if multiple containers share the GPU and use GPU memory dynamically. ConVGPU [26] proposes a solution to share GPU among containers by restricting the use of GPU memory at runtime. When a container tries to use GPU memory more than allocated memory limitation, ConVGPU denies the memory request. When a container requests GPU memory within the limitation but GPU memory is not enough, the request is suspended until requested memory size is available. On the other hand, the GPU memory space used by running container is made available upon exiting to other containers that are waiting for GPU memory to be freed. ConVGPU uses a best fit

Author's personal copy

strategy to schedule containers paused for memory allocation.

GaiaGPU [27] proposes an approach to sharing GPU memory and computing resources in a container-based GPU cloud environment. GaiaGPU divides a physical GPU into several virtual GPUs and assigns them to containers using the device plugin framework in Kubernetes. GaiaGPU divides memory resources into 256MB units called *vmemory devices* and splits computing resources into 100 *vprocessor devices* with one percent of utilization each. GaiaGPU allocates multiple *vmemory devices* and *vprocessor devices* to each container. The allocation status can be changed during runtime. Two types of resource allocation schemes are proposed in GaiaGPU: elastic resource allocation and dynamic resource allocation. Elastic resource allocation modifies the resource allocation status of a container temporarily, while dynamic resource allocation changes it permanently. In GaiaGPU, elastic resource allocation is used for computing resources, and dynamic resource allocation is used for both memory and computing resources.

6 Conclusion and future work

We have observed that GPU context switch overhead in *gVirt* is one of the major bottlenecks in improving the performance of GPU VM due to the large number of private GTT copies. This paper explored this issue and proposed a low-overhead dynamic memory management scheme called DymGPU that provides two memory allocation algorithms: size-based algorithm and utilization-based algorithm. The size-based algorithm is based on the GPU memory size requested from a vGPU, and preferred when the utilization of each vGPU is similar. The utilization-based algorithm is based on the GPU utilization, and preferred when the utilization of each vGPU is uneven. The benchmarking results showed that the proposed algorithms reduced the number of GTT copies by about 53% and also improved the performance of various 2D/3D workloads by up to 39% against *gVirt*.

The global graphics memory space in DymGPU is static in the sense that when a memory space is given to each vGPU, it should be kept until the workload running on the vGPU is finished. As a future work, we are currently investigating a mechanism to dynamically adjust the memory size assigned to each vGPU at runtime. In addition, current version of DymGPU can schedule only Linux VMs with low global graphics memory size of 64MB due to the limitation on low global graphics memory size. DymGPU needs to be enhanced in the future to be able to schedule not only Linux VMs but also Windows VMs that require 128 MB low global graphics memory.

Acknowledgements This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2017M3C4A7080245).

References

1. Park, Y., Gu, M., Yoo, S., Kim, Y., Park, S.: DymGPU: Dynamic Memory Management for Sharing GPUs in Virtualized Clouds. In: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W), pp. 51–57. IEEE (2018)
2. The compute architecture of Intel® processor graphics Gen9.https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf
3. Pascal GPU architecture | NVIDIA.https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/
4. Duato, J., Pena, A.J., Silla, F., Mayo, R., Quintana-Ortí, E.S.: rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In: 2010 International Conference on High Performance Computing and Simulation (HPCS), pp. 224–231. IEEE (2010)
5. Giunta, G., Montella, R., Agrillo, G., Coviello, G.: A GPGPU transparent virtualization component for high performance computing clouds. In: European Conference on Parallel Processing, pp. 379–391. Springer (2010)
6. Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J., Feng, W.C.: VOCL: an optimized environment for transparent virtualization of graphics processing units. In: Innovative Parallel Computing (InPar), 2012, pp. 1–12. IEEE (2012)
7. Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., Wiegert, J.: Intel virtualization technology for directed I/O. Intel Technol. J **10**(3), 179–192 (2006)
8. Tian, K., Dong, Y., Cowperthwaite, D.: A full GPU virtualization solution with mediated pass-through. In: USENIX Annual Technical Conference, pp. 121–132 (2014)
9. Suzuki, Y., Kato, S., Yamada, H., Kono, K.: GPUvm: why not virtualizing GPUs at the hypervisor? In: USENIX Annual Technical Conference, pp. 109–120 (2014)
10. Xue, M., Tian, K., Dong, Y., Ma, J., Wang, J., Qi, Z., He, B., Guan, H.: gScale: scaling up GPU virtualization with dynamic sharing of graphics memory space. In: USENIX Annual Technical Conference, pp. 579–590 (2016)
11. Kehne, J., Metter, J., Bellosa, F.: GPUswap: enabling oversubscription of GPU memory through transparent swapping. In: ACM SIGPLAN Notices, vol. 50, pp. 65–77. ACM (2015)
12. Kehne, J., Hillenbrand, M., Metter, J., Gottschlag, M., Merkel, M., Bellosa, F.: GPrioSwap: towards a swapping policy for GPUs. In: Proceedings of the 10th ACM International Systems and Storage Conference, p. 10. ACM (2017)
13. Xue, M., Ma, J., Li, W., Tian, K., Dong, Y., Wu, J., Qi, Z., He, B., Guan, H.: Scalable GPU virtualization with dynamic sharing of graphics memory space. IEEE Trans. Parallel Distrib. Syst. **1**, 1–1 (2018)
14. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: ACM SIGOPS Operating Systems Review, vol. 37, pp. 164–177. ACM (2003)
15. Intel® GVT-g setup guide.https://github.com/intel/Igvtg-kernel/blob/2016q4-4.3.0/iGVT-g_Setup_Guide.txt

Author's personal copy

16. Valley benchmark | UNIGINE benchmarks.https://benchmark.unigine.com/valley
17. Superposition benchmark | UNIGINE benchmarks.https://benchmark.unigine.com/superposition
18. Phoronix Test Suite - linux testing & benchmarking platform, automated testing, open-source benchmarking.http://phoronix-test-suite.com/
19. cairographics.org.https://www.cairographics.org/
20. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization, 2009. IISWC 2009. pp. 44–54. IEEE (2009)
21. Kato, S., McThrow, M., Maltzahn, C., Brandt, S.A.: Gdev: first-class GPU resource management in the operating system. In: USENIX Annual Technical Conference, pp. 401–412. Boston (2012)
22. Wang, K., Ding, X., Lee, R., Kato, S., Zhang, X.: GDM: device memory management for gpgpu computing. ACM SIGMETRICS Perform. Eval. Rev. **42**(1), 533–545 (2014)
23. Ji, F., Lin, H., Ma, X.: RSVM: a region-based software virtual memory for GPU. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, pp. 269–278. IEEE Press (2013)
24. Becchi, M., Sajjapongse, K., Graves, I., Procter, A., Ravi, V., Chakradhar, S.: A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, pp. 97–108. ACM (2012)
25. Official GitHub repository of NVIDIA Docker.https://github.com/NVIDIA/nvidia-docker
26. Kang, D., Jun, T.J., Kim, D., Kim, J., Kim, D.: ConVGPU: GPU management middleware in container based virtualized environment. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 301–309. IEEE (2017)
27. Gu, J., Song, S., Li, Y., Luo, H.: GaiaGPU: sharing GPUs in container clouds. In: IEEE International Conference on Parallel & Distributed Processing with Applications (IEEE ISPA 2018), pp. 469–476. Melbourne, Australia (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Minwoo Gu is a master's degree student in Sogang University, Seoul, Republic of Korea. He received his B.S. degree in Computer Science and Engineering from Sogang University, Seoul, Republic of Korea. His research interests include cloud computing and GPU virtualization.

Younghun Park is an employee of TmaxCloud, Seongnam-si, Gyeonggi-do, Republic of Korea. He received his B.S. and M.S. degrees in Computer Science and Engineering from Sogang University, Seoul, Republic of Korea. Now he works for TmaxCloud as a software engineer.

Youngjae Kim received his Ph.D. degree in Computer Science and Engineering from Pennsylvania State University, University Park, PA, USA in 2009. He is currently an associate professor in the department of computer science and engineering at Sogang University, Seoul, Republic of Korea. Before joining Sogang University, Dr. Kim was a staff scientist in the U.S. Department of Energy's Oak Ridge National Laboratory (2009-2015) and an assistant professor in Ajou University, Suwon, Republic of Korea (2015-2016). Dr. Kim received the B.S. degree in computer science from Sogang University, Republic of Korea in 2001, and the M.S. degree from KAIST in 2003. His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.

Sungyong Park is a professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in computer science from Sogang University, and both the M.S. and Ph.D. degrees in computer science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.