DymGPU: Dynamic Memory Management for Sharing GPUs in Virtualized Clouds

Younghun Park, Minwoo Gu, Sungju Yoo, Youngjae Kim, Sungyong Park
Department of Computer Science and Engineering
Sogang University
Seoul, Republic of Korea
{parkyh93, mwgu, wysh123, youkim, parksy}@sogang.ac.kr

***Abstract*—*gVirt* is a full GPU virtualization technique for Intel's integrated GPUs that alleviates the problems of other GPU virtualization techniques such as API remoting and direct pass-through. The original *gVirt* is known to have an inherent scalability limitation on the number of simultaneous virtual machines (VM). *gScale* solved this problem by allowing each VM to share a global graphics memory space and copy the entries in a private graphics translation table (GTT) to a physical GTT along with a GPU context switch. However, it still suffers from a large overhead of copying entries between private GTT and physical GTT, which becomes worse when the global graphics memory space allocated for each VM is overlapped.**

In this paper, we identify that the copy overhead caused by GPU context switch is the major bottleneck in performance improvement and propose a dynamic memory management scheme, called DymGPU, that 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 the 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%.

***Index Terms*—Cloud Computing, GPU Virtualization, GPU Scheduling**

I. Introduction

With the advances in computing and hardware technologies, various types of GPUs [1] [2] have recently been used for performance acceleration in compute-intensive applications. 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. API remoting [3] [4] [5] 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 the API library or GPU driver, which lacks flexibility and can't provide full GPU features. Direct pass-through [6] dedicates a GPU to a single virtual machine (VM) and allows it to use the GPU directly without hypervisor intervention. This technique provides high performance at the cost of prohibiting the sharing of GPU among VMs. To alleviate the problems of aforementioned approaches, full GPU virtualization solutions at the hypervisor level such as *gVirt* [7] and *GPUvm* [8] are introduced.

Among them, *gVirt* is an open source, full GPU virtualization solution for Intel's integrated GPUs that are integrated into the CPU and utilize system memory instead of dedicated graphics memory. In *gVirt*, the performance-critical resources can be directly accessed by a VM, while the hypervisor intervenes for privileged operations. The original *gVirt* could only support up to three VMs owing to insufficient GPU memory. *gScale* [9] 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 Section II, we found that GPU context switch incurs non-trivial overhead and the cost of copying GTT entries is extremly 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 [10] and GPrioSwap [11] 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 [12] 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

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

978-1-5386-5175-9/18/$31.00 ©2018 IEEE
DOI 10.1109/FAS-W.2018.00025

creates a bottleneck in performance improvement. Based on this inference, we propose a dynamic memory management scheme called DymGPU. DymGPU provides two memory allocation algorithms such as size-based and utilization-based algorithms that can be selected by users. The size-based algorithm allocates global graphics memory space to each VM 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 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.

To demonstrate the performance improvement by the use of DymGPU, we have implemented our approach in the 2016Q4 version of *gVirt* on a Xen hypervisor [13]. Furthermore, 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 II briefly introduces *gVirt* and discusses the motivations related to our proposed approach. Section III explains the overall architecture of DymGPU and its implementation issues in detail. Section IV evaluates DymGPU against *gVirt*. Section V concludes this paper with possible future works.

II. Background and Motivation

A. Background of gVirt

gVirt (also called Intel GVT-g) is a high-performance, full GPU virtualization technique for Intel's integrated GPUs [7]. 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 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. Each vGPU is allotted a 16-ms time quantum

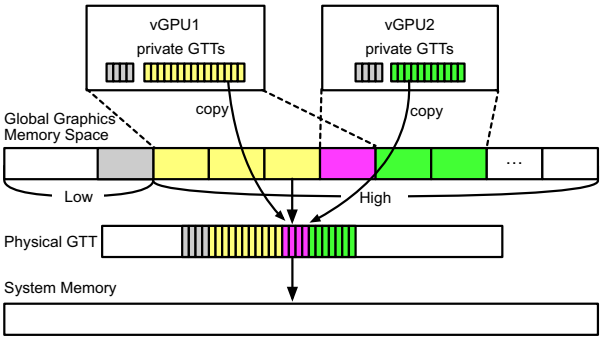


Fig. 1: Global Graphics Memory Space and Mapping in *gVirt*

that takes into account high GPU context switch cost and speed. After the assigned time is elapsed, the vGPU state is saved and the state of the next vGPU is restored.

The 4 GB global graphics memory space of Intel's integrated GPU is divided into low global graphics memory that both the CPU and GPU can access, and high global graphics memory that only the 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* modifies the layout of low global graphics memory and creates a slot such that multiple vGPU can share the slot and swap the related contents whenever each vGPU is ready to run. High global graphics memory is divided into 64 MB slots, and multiple continuous slots are assigned to each vGPU. When a new VM is created, *gVirt* computes a score for every case where the vGPU of the VM can be allocated. *gVirt* assigns a score per slot for each case. If the 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 VMs. After summarizing the scores of each slot, continuous 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.

Fig. 1 depicts the global graphics memory and its mapping 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* allows vGPUs to share global graphics memory. Each vGPU has a private GTT of low global graphics memory and high global graphics memory. To activate the 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 vGPU is switched out, the physical GTT which is mapped to low global graphics memory has entries of another vGPU, so the CPU can't access the VM through aperture. To solve

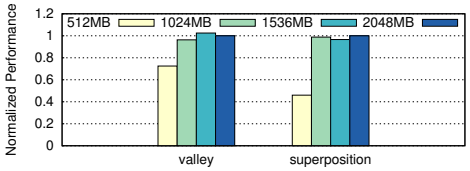
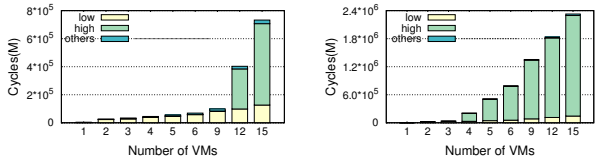


Fig. 2: Performance of 3D Workloads



(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

this problem, *gScale* implements ladder mapping that map directly guest physical address to host physical address and fence memory space pool that allows fence register to operate correctly.

B. GPU Context Switch Overhead Analysis

Global graphics memory mostly handles the frame buffer and command buffer, which store pixel information regarding display and commands produced by the CPU. In case of Linux VM, 64 MB and 384 MB are considered to be enough and generally recommended for low global graphics memory and high global graphics memory, respectively [14]. Nowadays, Intel's integrated graphics processors have 4 GB of graphics memory. So when a VM is set to 384 MB high global graphics memory and there are 15 VMs, only 2304 MB, which is 36 slots, are shared. However, we found that the small size in high global graphics memory can affect the performance of GPU workloads and can sometimes lead to a crash especially when the 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 the average frames per second (FPS) of two benchmarks, Unigine Valley [15] and Unigine Superposition [16], by varying high global graphics memory size from 512 MB to 2048 MB. Fig. 2 shows the normalized performance with respect to the performance with 2048 MB size. As shown in Fig. 2, when the high global graphics memory size is larger than 1024 MB, similar performance is observed, but when high global graphics memory size is 512 MB, the performance degrades severely because of the insufficient high global graphics memory. It should be noted that larger size in high global graphics memory can increase not only the performance of GPU workloads but also the possibility of overlapping address spaces among vGPU instances. It is therefore necessary to devise an efficient solution to minimize the overhead incurred 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 3 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. We conducted two experiments for which high global graphics memory size of each VM is set to 384 MB and 1024 MB. The testbed is described in greater detail in Section IV-A.

Fig. 3 shows average consumed CPU cycles for the 3 activities mentioned above. As shown in Fig. 3(a), the rate at which private GTT copies are made in low global graphics memory is approximately 84% when the number of VMs is less than or equal to 9. 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 the rates at which private GTT copies are made in low and high global graphics memory are 17% and 79% respectively. The rate for high global graphics memory is higher when GPU memory contention is greater as shown in Fig. 3(b). When the number of VMs is greater than 5, the number of consumed cycles in high global graphics memory is more than 90% of the number of total cycles. Therefore, the private GTT copy overhead forms a very large portion of GPU context switch overhead. Thus, the GTT copy that occurs in low and high global graphics memory should be reduced to improve the performance of VMs.

III. Design and Implementation

In this section, we present the overall architecture of DymGPU and explore how to minimize the private GTT copy overhead in Intel's integrated GPU environment.

Currently, same as *gVirt* and *gScale*, DymGPU supports only Intel's integrated GPUs. However, our concept is applicable to other architectures which use system memory as graphics memory.

A. Overall Architecture

Fig. 4 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 seconds. If an idle GPU occupies a slot in the global graphics memory alone, the 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

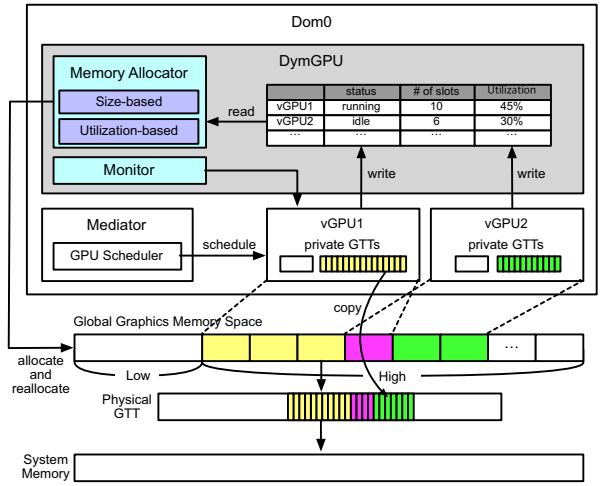


Fig. 4: Architecture of DymGPU

the GPU usage at a certain point in time, we average the GPU utilization values for 20 seconds to determine the GPU utilization of each vGPU. The 20 second 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 the memory space based on the information obtained from the *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*.

B. Memory Allocator

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 $V = \{V_0, V_1, ..., V_{N-1}\}$, 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. Fig. 5 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

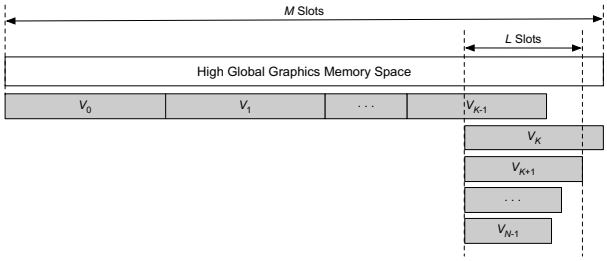


Fig. 5: Size-based Allocation Algorithm

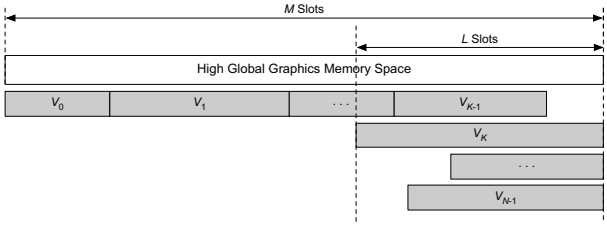
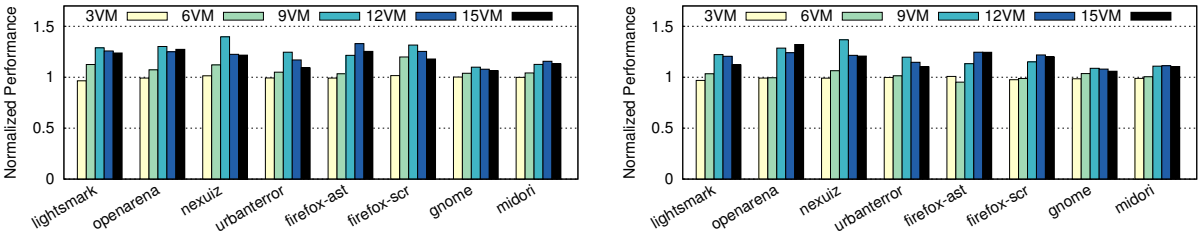


Fig. 6: Utilization-based Allocation Algorithm

leftmost free slots until the vGPU does not share slots with another. In Fig. 5, 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.

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.

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



(a) Size-based Allocation Algorithm

(b) Utilization-based Allocation Algorithm

Fig. 7: Performance Comparison with Similar GPU Utilization (3D and 2D Workloads)

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. 6. 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. $V_0, V_1, ..., V_{K-1}$ are in that case. Remaining vGPUs, $V_K, V_{K+1}, ..., V_{N-1}$, are allocated to rightmost slots in high global graphics memory.

Memory allocator reallocates the memory space every 20 seconds based on the 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 the changed position. In the worst case, all slots in the 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 the 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.

IV. 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 feature provided by *gScale*.

For the extensive comparison with different workload patterns, we use Phoronix Test Suite [17] and Cairo-perf-trace [18], 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 is measured by average frames per second (FPS) and execution time.

A. Experimental Setup

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

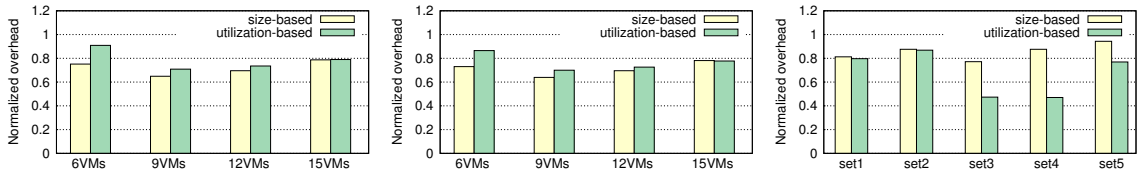
TABLE I: Experimental Setup

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 / 2GB
OS	Ubuntu 16.04 (Kernel version 4.3.0)
Low GM	64 MB

B. Performance

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 the 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 subsection II-B. The performance of DymGPU is normalized to that of *gVirt*.

Fig. 7(a)(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 DymGPU for two algorithms is similar to that of *gVirt* because the total required size for high global graphics memory is still less than the 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



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

Fig. 9: Normalized Number of Private GTT Copies

utilization when allocating a memory space. This improves the performance by reducing the number of private GTT copies. In addition, the performance of size-based algorithm is better than that of utilization-based algorithm by about 3-16% when all VMs actively use GPU. Also, 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.

TABLE II: Benchmark Sets

Set Number	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

Performance Comparison with Various GPU Utilization: To compare the performance with various GPU workloads, we define 5 sets of benchmarks by mixing workloads with different GPU utilization as shown in Table II. 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 the experiment, we run 15 VMs and the number in the table represents the number of instances for each workload.

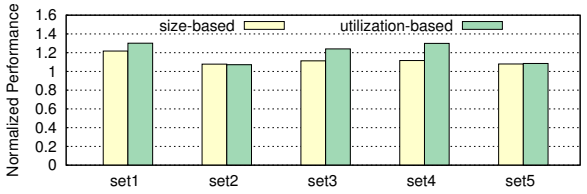


Fig. 8: Performance Comparison with Various GPU Utilization

Fig. 8 shows the comparison with various workload sets. We observe that the performance of DymGPU is better than *gVirt* for all 5 sets. Specially, the utilization-based algorithm outperforms the size-based algorithm except for set 2 and set 5. This is because set 2 is mainly composed of an urbanterror, which requires high CPU and GPU computation, so GPU context switch overhead is hidden by that the CPU computation is already bottleneck. And set 5 requires a CPU computation more than a GPU computation resulting in less GPU context switches.

C. GPU Context Switch Overhead

In this experiment, we analyze the private GTT copy overhead for high global graphics memory that occurs during GPU context switch using the same workload types and benchmark sets explained in subsection IV-B. The experiments are conducted by using *nexuiz* (3D workload) and *midori* (2D workload) as we increase the number of VMs.

Fig. 9(a)(b) show the number of private GTT copies when VMs execute the same workloads. As shown in Fig. 9(a), both size-based and utilization-based algorithms reduces the context switch overhead against *gVirt* by up to 35% and 30% for 3D workload. When the number of VMs is small, the size-based algorithm reduces the private GTT copies further compared with the utilization-based algorithm. However, as we increase the number of VMs, the difference of overhead optimization shrinks because of the overhead that arises as the degree of memory sharing increases and the overhead of competition between the two processors. For 2D workload, we observe similar results as shown in Fig. 9(b).

Fig. 9(c) 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 size-based algorithm for all sets. Specially, the number of slot copies in utilization-based algorithm is reduced by about 40% compared to size-based algorithm for set 4. This is because set 4 consists of GPU workloads with various GPU utilization.

V. Conclusion and Future Work

We have observed that the GPU context switch overhead in *gVirt* is the major bottleneck in improving the performance of GPU VM due to the large number of private GTT copies. This paper explored this issue and proposed an dynamic memory management scheme called DymGPU that provides two memory allocation algorithms: size-based algorithm and utilization-based algorithm. Size-based algorithm is based on the required GPU memory size of the vGPUs, and preferred when the vGPU utilization is similar. Utilization-based algorithm is based on the vGPU utilization, and preferred when deviation of the vGPU utilization is large. 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 mechnism to dynamically adjust the memory size assigned to each vGPU at runtime.

REFERENCES

[1] The compute architecture of Intel processor graphics Gen9. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf

[2] Pascal GPU architecture | NVIDIA. [Online]. Available: https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/

[3] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE, 2010, pp. 224–231.

[4] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *European Conference on Parallel Processing*. Springer, 2010, pp. 379–391.

[5] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng, "VOCL: An optimized environment for transparent virtualization of graphics processing units," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–12.

[6] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed I/O." *Intel technology journal*, vol. 10, no. 3, 2006.

[7] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through." in *USENIX Annual Technical Conference*, 2014, pp. 121–132.

[8] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why not virtualizing GPUs at the hypervisor?" in *USENIX Annual Technical Conference*, 2014, pp. 109–120.

[9] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan, "gScale: Scaling up GPU virtualization with dynamic sharing of graphics memory space." in *USENIX Annual Technical Conference*, 2016, pp. 579–590.

[10] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of GPU memory through transparent swapping," in *ACM SIGPLAN Notices*, vol. 50, no. 7. ACM, 2015, pp. 65–77.

[11] J. Kehne, M. Hillenbrand, J. Metter, M. Gottschlag, M. Merkel, and F. Bellosa, "GPrioSwap: towards a swapping policy for GPUs," in *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM, 2017, p. 10.

[12] M. Xue, J. Ma, W. Li, K. Tian, Y. Dong, J. Wu, Z. Qi, B. He, and H. Guan, "Scalable GPU virtualization with dynamic sharing of graphics memory space," *IEEE Transactions on Parallel & Distributed Systems*, no. 1, pp. 1–1, 2018.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.

[14] Intel GVT-g setup guide. [Online]. Available: https://github.com/intel/Igvtg-kernel/blob/2016q4-4.3.0/iGVT-g_Setup_Guide.txt

[15] Valley benchmark | UNIGINE benchmarks. [Online]. Available: https://benchmark.unigine.com/valley

[16] Superposition benchmark | UNIGINE benchmarks. [Online]. Available: https://benchmark.unigine.com/superposition

[17] Phoronix Test Suite - linux testing & benchmarking platform, automated testing, open-source benchmarking. [Online]. Available: http://phoronix-test-suite.com/

[18] cairographics.org. [Online]. Available: https://www.cairographics.org/