Design of an adaptive GPU sharing and scheduling scheme in container-based cluster

Qichen Chen, Jisun Oh, Seoyoung Kim & Yoonhee Kim

Cluster Computing
The Journal of Networks, Software Tools and Applications

ISSN 1386-7857

Cluster Comput
DOI 10.1007/s10586-019-02969-3

Springer

Cluster Computing
https://doi.org/10.1007/s10586-019-02969-3

Design of an adaptive GPU sharing and scheduling scheme in container-based cluster

Qichen Chen[1] · Jisun Oh[2] · Seoyoung Kim[2] · Yoonhee Kim[2]

Received: 1 February 2019 / Revised: 15 June 2019 / Accepted: 23 July 2019

Abstract
Container based virtualization is an innovative technology that accelerates software development by providing portability and maintainability of applications. Recently, a growing number of workloads such as high performance computing (HPC) and Deep Learning(DL) are deployed in the container based environment. However, GPU resource management issues especially the GPU memory over subscription issue in container-based clusters, which brings substantial performance loss, is still challenging. This paper proposes an adaptive fair-share method to share effectively in container-based virtualization environment as well as an execution rescheduling method to manage the execution order of each container for acquiring maximum performance gain. We also proposed a checkpoint based mechanism especially for DL workload running with TensorFlow, which can efficiently solve the GPU memory over subscription problem. We demonstrate that our approach contributes to overall performance improvement as well as higher resource utilization compared to default and static fair-share methods with homogeneous and heterogeneous workloads. Compared to two other conditions, their results show that the proposed method reduces by 16.37%, 15.61% in average execution time and boosts approximately by 52.46%, 10.3% in average GPU memory utilization, respectively. We also evaluated our checkpoint based mechanism by running multiple CNN workloads with TensorFlow at the same time and the result shows our proposed mechanism can ensure each workload executing safely without out of memory (OOM) error occurs.

Keywords GPU resource sharing · GPU management · GPU scheduling · GPU virtualization

1 Introduction

Recently high performance computing (HPC) applications and deep learning (DL) applications play key roles in many different research fields. The common feature exists in these applications is that all of them require massive computation power, which is in accordance with the high parallelism characteristics of the graphics processing unit (GPU).

A GPU is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device [1]. However, due to their powerful structure which supports massive and energy-efficient parallelism as well as high computational bandwidth, they have been recently utilized more for intensive-parallel computing than for general-purpose CPUs.

In the last decade, the increment of problem size and model complexity of HPC and DL applications has led to deploy GPUs in a wide range of platforms such as cloud

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

✉ Yoonhee Kim
yulan@sookmyung.ac.kr

Qichen Chen
charliecqc@dcslab.snu.ac.kr

Jisun Oh
jsoh8088@gmail.com

Seoyoung Kim
sssyyy77@gmail.com

[1] Department of Computer Science and Engineering, Seoul National University, Seoul, South Korea

[2] Department of Computer Science, Sookmyung Women's University, Seoul, South Korea

Published online: 31 July 2019

computing, large-scale distributed computing environments. To efficiently deploy these workloads, virtual machine (VM) has then emerged, offering various options for users to choose, which helped to maximize their applications' performance. For example, one of the most popular container-based virtualization solution: Docker [2] which is an open-source virtualization technology of container is adopted as an alternative. Each docker container encapsulates an application and can be run on different machines on the top of a docker engine. In addition, it isolates each independent container running on the same instance of operating system by making use of Linux kernel features like *cgroups* and *namespaces* to control CPU time, memory, and network bandwidth individually. However, GPU had not been available in any virtualization environment for a long time. Recently, NVIDIA has presented the NVIDIA Docker [3], to share GPU drivers from host to containers without having them installed on each container individually. However, NVIDIA Docker simply assigns the whole physical GPU to a single container to prevent any performance degradation since the allocated GPU is freely available. The problem occurs when multiple containers trying to share the same GPU at the same time. Since the NVIDIA Docker is not aware of how GPU memory is used by the application inside the container, concurrent allocating of GPU memory by different applications, leading to an out-of-memory(OOM) error due to exceeding total GPU memory amount. Several works for providing virtualized GPU(vGPU) in virtualization environment has been proposed, such as GViM, gVirtus and vCUDA, However, they are not support a container-based virtualized environment and required to use a specific device, which does not meet the concept of container based virtualization. As mentioned above, however, effective methods to improve the utilization of GPUs and to share them properly in the virtualized environment is required.

In this paper, we proposed an adaptive fair sharing algorithm that can determine GPU memory for each container at the initialization time. We also proposed a scheduling method based on the profiling ,which manage the execution order of each container to avoid OOM error occurs. Our proposed approach has been compared with the original method provided by NVIDIA-Docker and static fair share method in the evaluations.

Besides, We also proposed a solution that share the physical GPU among containers that running Machine Learning workload on TensorFlow [4] in container-based virtualized environment. Our proposed system contains two major parts: (1) A check-point module that intercepts the GPU memory allocation calls transfer them to the scheduler. It also checkpoint current containers contents that in the GPU memory to Host memory to claim the occupied space and then bring them back when it is necessary. (2) A scheduler that make a decision of whether approve each GPU memory allocation request or reject it by the intercepted GPU memory allocation calls from each check-pointer and current system status. We apply and implement the techniques on TensorFlow in Linux Kernel v4.4.

Our experiment results show that the proposed scheduling method reduces average execution time by 16.37%, 15.61% and boosts approximately by 52.46%, 10.3% in average GPU memory utilization. In addition, our checkpoint based methods enables multiple containers that running DL workloads with TensorFlow share the GPU safely without OOM error occurs. Specifically, this paper's contributions are as follows:

- We analyzed the GPU memory over subscription problem in the current Docker container based virtualization system.
- We proposed an algorithm to share GPU memory effectively for GPU containers
- We design and implement a GPU sharing solution for DL workloads running in TensorFlow to solve the GPU memory over subscription problem that based on checkpoint module and scheduler.
- We demonstrate our proposed solution can efficiently solve the GPU memory over subscription problem through our evaluation results.

The organization of this paper is as follows: "Background and motivation" describes the background and motivation. "Design and Implementation" presents the design of our proposed algorithm, checkpoint based mechanism, and running scenario of each case. "Evaluation" shows the experimental results and "Related Work" discuss the related work and lastly, "Conclusion" concludes the paper.

2 Background and motivation

2.1 GPU virtualization and containerization

Virtualization is a combination of hardware and software solutions that support the creation and operation of virtual versions of devices or resources, such as servers, storage devices, networks, or OS [5]. The virtualization platform such as Virtual Machines allows one to divide the physically unified hardware system into a logical set of independent computing resources [6]. And virtualization of computing resources allowed to solve the problem of increasing the efficiency of scheduling in cluster computing systems [7].

As HPC or DL tasks usually involve GPU resources in their solution, there are many attempts that have been made to introduce virtualized GPU into the virtualization

environment. For example, there are such approaches including GViM [8], gVirtuS [9], and vCUDA [10] that are based on creating copies of the CUDA API for virtualizing GPU and providing them to virtualization environment [5] and in case of rCUDA solution [11], it proposed the technology of remote GPU usage. However, the above approaches degrade the performance of the GPU during the virtualization process [12]. There is another approach such as NVIDIA GRID [13] to virtualize the GPU at the hardware level. However, the solution can be adapted for a specific type of GPU like NVIDIA GRID K1 and K2 [14].

On the other hand, containerization technology such as Docker provides a run-time environment for the application at the OS level, reducing the overhead compared to the virtualization [15]. In detail, the container provides virtualization solutions: each guest OS uses the same kernel (and in some cases other parts of the OS) as the host server. Therefore, they much more in common with the host server, so that they are smaller and more compact than virtualization approaches [5].

2.2 NVIDIA Docker

Docker container is widely used because it can provide both hardware and software encapsulation on the same system at the same time. However, there exists a problem when using specialized hardware such as NVIDIA GPUs that require kernel modules and user level-libraries. The previous way to handle this problem is to fully install the NVIDIA drivers inside the container, however it will drastically reduce the portability of containers, which weakening the most important advantage of docker. To solve the problem mentioned above while enabling the portability, NVIDIA provides the NVIDIA docker, a project that can easily share GPU driver from host to containers without installing NVIDIA drivers individually.

Nevertheless, the above solutions for GPU containerization and orchestration are still attempted to virtualize GPU entirely wich is opposed to the original concept of the container.

2.3 TensorFlow

TensorFlow is an open source software library for numerical computation using data flow graphs.The graph nodes represent mathematical operations, while the graph edges represent the multidimensional data arrays that flow between them. This flexible architecture enables you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code. Especially for machine learning applications, it uses CUDA extension to help them perform general operations on the GPU. Even the applications are not written in CUDA, it is still possible to operate them with TensorFlow on GPUs without code modification. However, if the applications are executed without specifying specific GPU memory range, TensorFlow will allocate most of the GPU memory for the application and cause lacking of GPU memory when other applications need to be executed. On the other hand, with the option

config.gpu_options.per_process_gpu_memory_fraction, GPU memory can be allocated for each application.

2.4 HPC & DL workload execution pattern

We select HPC and the TensorFlow applications provided by NVIDIA GPU cloud (NGC),which is a GPU-accelerated cloud platform optimized for scientific computing, to understand the GPU resource usage pattern during container execution time. The applications we selected to be analyzed are LAMMPS, GROMACS, QMCPAC that belong to HPC application and CNN (convolution neural network)-MNIST that belongs to TensorFlow application. Nvidia-smi is used to moitor the resource usage every 5 s, and using the result to analyze the GPU resource usage variation over the time.

2.4.1 LAMMPS

LAMMPS is a classical molecular dynamics code with a focus on materials modeling. It's an acronym for Large-scale atomic/molecular massively parallel simulator. LAMMPS has potentials for solid-state materials (metals, semiconductors) and soft matter (biomolecules, polymers) and coarse-grained or mesoscopic systems. It can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, meso, or continuum scale. LAMMPS runs on single processors or in parallel using message-passing techniques and a spatial-decomposition of the simulation domain. Many of its models have versions that provide accelerated performance on CPUs, GPUs, and Intel Xeon Phis. The code is designed to be easy to modify or extend with new functionality.

Figure 1 shows the GPU and its memory utilization when a single container runs on one physical GPU. As the

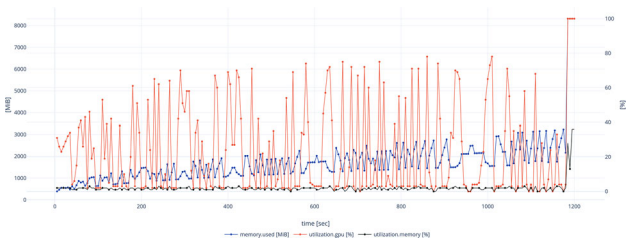


Fig. 1 LAMMPS-1GPU, 1 container

figure shows, average GPU memory usage is 1.7GB and GPU utilization is in average is 23.8%. After 1180 seconds of execution, the GPU memory usage reaches to 8.3GB and the GPU utilization rapidly increased to 100%, at last the execution time is 19 min and 55 s. Figure 2 illustrates the GPU and its memory utilization when two containers running on one physical GPU. In this case, the maximum GPU memory usage is 11.4 GB and the maximum GPU utilization is 100%. In addition, The memory usage increased rapidly after executing 1615 s and an OOM (out-of-memory) error occurs after the applications executed 1625 s. This is caused by the overlapping of sharply increased GPU memory requirement by executing two containers at the same time. As a result, the whole execution time is 27 min and 10 s with only one application terminated normally.

2.4.2 GROMACS

GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids that have a lot of complicated bonded interactions. in our experiment, a water molecule data set with the size of 1536 KiB has been tested.

Figure 3 gives out the GPU and its memory utilization when running a single application container on a physical GPU. As the figure shows, the maximum GPU memory usage is 539 MB and the whole execution time is 715 s. Figure 4 show the results of experiments where deploying 2 containers on single GPU respectively.

In the case of Fig. 4 the maximum GPU memory usage is 1743 MB and the whole execution time is 2480 s. As a result, the execution time increases in proportion to the number of containers when one GPU is shared by several containers, revealing the fact that running multiple containers on a single GPU simultaneously do not have the benefit of concurrent execution.

2.4.3 QMCPACK

QMCPACK is an open-source,

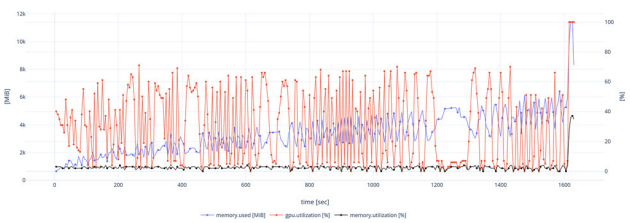


Fig. 2 LAMMPS-1GPU, 2 containers

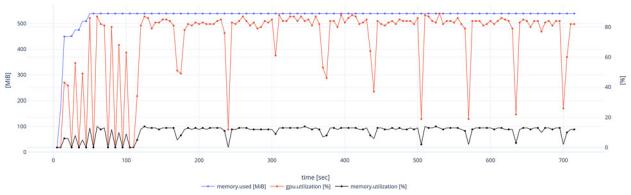


Fig. 3 GROMACS-1GPU, 1 container

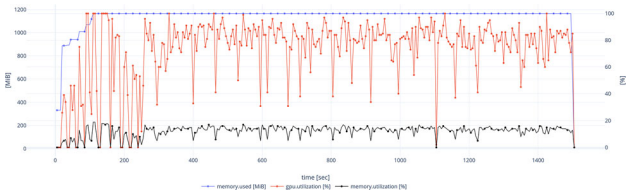


Fig. 4 GROMACS-1GPU, 2 containers

high-performance electronic structure code that implements numerous Quantum Monte Carlo algorithms. Its main applications are electronic structure calculations of molecular, periodic 2D and periodic 3D solid-state systems. Figures 5 and 6 demonstrates the experiment result of running one and two QMCPACK containers on a single physical GPU. In the case of Fig. 5, the total execution time is 190 s with the maximum GPU memory usage of 5389 MB, and under the circumstance of Fig. 6, the total execution time reached to 385 s while the maximum memory usage increased to 10,768 MB. From the experiments, We can draw a conclusion that running multiple containers sequentially is as same as running them concurrently on a single physical GPU. The reason is even we launched two QMCPACK application concurrently, due to its high GPU core usage, the contention of GPU cores will give a huge impact on the concurrent performance.

2.4.4 CNN-MNIST

CNN-MNIST is an image analysis algorithm for composite images. CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, RELU layer i.e. activation function, pooling layers, fully connected layers and normalization layers. Figure 7 shows the result of runing TensorFlow workload in 4 containers on 1 GPU. During the execution on TensorFlow, if GPU memory

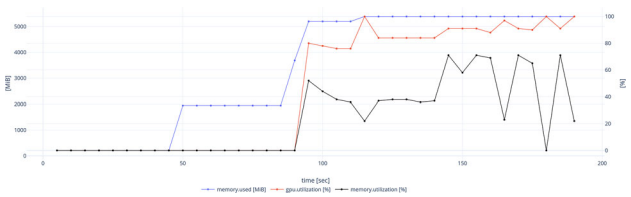


Fig. 5 QMCPACK-1GPU, 1 container

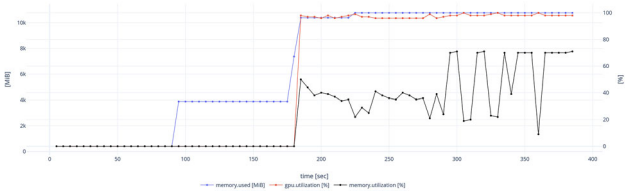


Fig. 6 QMCPACK-1GPU, 2 containers

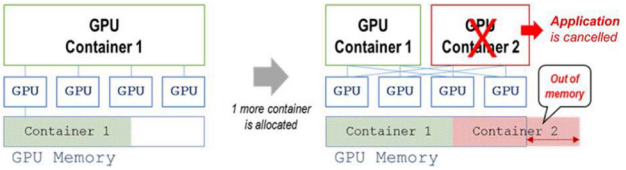


Fig. 8 Out-of-memory error occurs when multiple containers share the GPU

address space is not specified, the maximum GPU memory used is almost 8518 MB and the execution time comes to 3 min and 24 s. However, if we limit the GPU memory to 2207 MB, then the execution time will still maintain to 3 min and 42 s. As a result, reducing unnecessary GPU memory usage of containers will increase GPU resource utilization while not give an huge impact on the performance.

2.5 Motivation

GPUs can accelerate the performance of many applications; however, the capacity of device memory on GPUs limits the size of data that can be processed. Therefore, GPU memory is one of the most important GPU resources that needed to be shared in a proper manner. However, as mentioned above, existing solutions for GPU containerization and resource management still virtualize GPU entirely. Therefore, they can cause *over subscription* problem if clients' request exceeds the available GPU memory.

Generally, GPU memory is one of the most important GPU resources that needed to be shared. Lacking of GPU memory when containers try to allocate memory usually causes out-of-memory error. Figure 8 depicts the details. Suppose GPU container 1 has already allocated certain amount of GPU memory. Meanwhile another GPU container is started and trying to allocate its own GPU memory, however, as the Fig. 8 shows, within the part being used, the amount of GPU memory that required by the second GPU container exceeds the size of physical GPU memory thus resulting in the second container gets a out-of-memory error and being stopped by short of memory, running counter to the original purpose that sharing the GPU resources among containers. As the problem shows in the figure, currently it is still difficult to share the GPU memory among several containers within appropriate ratio. Even though the creation of container is aimed to fully utilize the restricted resources while providing acceptable performance, how to share the GPU resources efficiently is still a problem need to be taken into consideration.

To solve these issues, We proposed an adaptive fair-share method for HPC workloads and a check point based mechanism for DL workloads running with the TensorFlow in the GPU containerization environment.

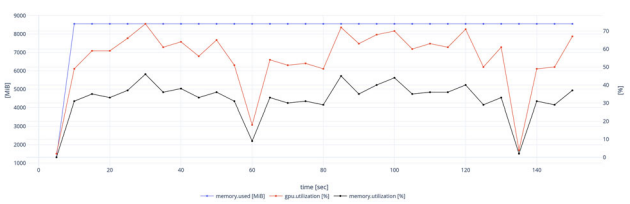


Fig. 7 CNN-1GPUs, 4 container

3 Design and implementation

In this section, we introduce the design and implementation of our proposed mechanisms. The overall design is aiming at sharing the GPU resources especially the GPU memory resource among several containers. The proposed system consists of three main components, The *checkpoint module* captures each GPU memory allocation call transfer it to the *scheduler*. Then it waits for the signal from the *scheduler* to continue allocating in the GPU memory or transferring the corresponding container's GPU contents to CPU host memory.In addition, it also uses profiled information to manage GPU resources and GPU memory while the other one is the *monitor*, which surveillance GPU resource in run-time and using the information to adjust the scheduling algorithm. Basically, this scheduler keeps checking GPU resources used by each application and their memory limit to determinate the execution order of them.

3.1 System design

Figure 9 demonstrates the overall system design. As the figure shows, for those DL workloads that running on TensorFlow that inside a container, since the TensorFlow platform provides the option to partially use the GPU memory, we designed the check point module inside the platform by using this feature to solve the GPU memory subscription problem. On the other hand. due to the fact that nvidia-docker does not support GPU memory

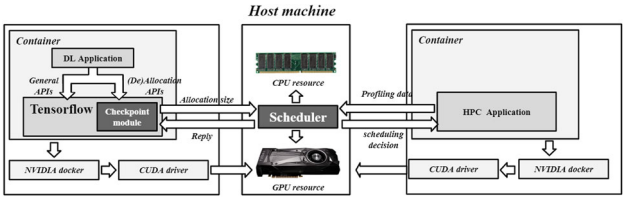


Fig. 9 Overall system design

isolation, we designed a profiled based pause/restart solution to solve the problem mentioned above.

In the case of DL workloads, when they send a GPU memory allocation call to our customized TensorFlow framework that running in a container, it redirect the command to the scheduler that located in host through TCP/IP request. The docker essentially forbidden the inter-process communication(IPCs) between containers and host, we proposed to use TCP/IP socket to establish the connection between container and host. The alternative ways have considered are using shared file between container and host or choosing UNIX socket. However, our proposed system are aiming at solving GPU virtualization problem in a cluster environment thus using TCP/IP socket will be the most suitable one.

3.2 Customized TensorFlow platform

Generally, The machine learning workload that running on TensorFlow needs to allocate all the GPU memory at once during the initialize time. However, the workload does not need to use whole GPU memory at a time, as a result, GPU memory are supposed to be shared among containers under certain conditions. The TensorFlow recently supports the option *–allow_growth,* which attempts to allocate only as much GPU memory based on run-time allocations, it starts out allocating very little memory, and as sessions get run and more GPU memory is needed. The TensorFlow also supports option *–per_process_gpu_memory_fraction*, which is a value between 0 and 1 that indicates what fraction of the available GPU memory to pre-allocate for each process. We applied both of these options to our customized TensorFlow platform in order to increase the potential utilization. For each container, we also profiled its run-time resource utilization, and use this information to determine appropriate *per_process_gpu_memory_fraction* for each container.

We also customized the **BFC Allocator**, which is the standard memory allocator part in TensorFlow by adding our **checkpoint module** into there. This module communicated with *scheduler* via the TCP/IP socket and the details will be described in the next subsection. We modified the original BFCAllocator class, adding TCP/IP socket information into it. During the class initialization, it connects to the scheduler and store the *socket_fd*, and they will won't be destructed until the class's destructor is called.

3.3 Checkpoint module

TensorFlow platform uses CUDA API to communicate and control NVIDIA GPU. Among these APIs, *cuMemAlloc* is used to allocate device memory and *cuMemcpyDtoH*, *cuMemcpyHtoD* are used to transfer data between CPU and GPU. To manage and schedule containers and handle the GPU memory over-commit problem, capturing these APIs is needed. We implement them in the checkpoint module. Specifically, this checkpoint module is implemented in BFC allocator. As we described above, the BFC allocator is in charge of host and device memory allocation in TensorFlow. We use the checkpoint module to store the contents in the GPU memory of previous containers into CPU memory and make empty space for the subsequent running container. The checkpoint module should make sure that the stopped container could resume its execution just as it has not been stopped. We implemented our checkpoint module in BFC allocator. In original BFC allocator, *cuMemAlloc* is called to allocate GPU memory according to applications' requirements. In our proposed checkpoint module, we capture each *cuMemAlloc* call before it is executed and transfer the GPU memory size that needed to be allocated to the Scheduler. Then the current process will be in a wait state until receiving reply from the scheduler. There are two kinds of replies from the scheduler, one approves the GPU memory allocation and another denied it. In the first case, after the checkpoint module received the reply, it executes the *cuMemAlloc* as it does in the original version. However, in the second case, where the allocation request is declined, the checkpoint module firstly allocate a new memory space in the host memory, then it transfer the contents stored in GPU memory to host memory by calling *cuMemcpyDtoH*, meanwhile it records the current region size and its corresponding host memory address. Lastly, it free its occupied GPU memory by calling *cuMemFree* and put the container itself info a wait state. When the subsequent containers has finished its execution, the previous suspended containers get the chance to continue its unfinished execution, if this requirement is approved, the checkpoint module should bring back the contents that moved to host memory via *cuMemcpyHtoD* according to previous stored region size and its address.

3.4 Scheduler

The containers should be scheduled based on the profiling running pattern based on its GPU memory usage. The **Scheduler** is written in C programming language. It runs on host machine and communicate with the checkpoint module in each container through TCP/IP socket, which is implanted in the container. The scheduler decides whether the containers should execute, reject or delay its GPU memory allocation according to the profiled information collected by the profiler and the current GPU resource utilization collected by the monitor. After current execution is finished, the scheduler also makes a decision about which waiting container should be brought back into running.

The scheduler traces every GPU memory allocation of each container as well as the current GPU memory utilization. As a result, the scheduler can be aware of how much free memory that can be allocated to a certain container. If there is no sufficient GPU memory to allocate, the scheduler will send a rejection reply to the container, and the container will be suspended after its checkpoint module transferred its GPU memory contents. There might be multiple suspended containers and each of them will be waited until the scheduler send them a continue reply. The scheduler also tracks every successful GPU memory allocation with their allocated size, and use this information to calculate the total GPU memory usage.

After the checkpoint module detects the application has finished its execution, it sends a signal to scheduler to notify its completion. Then the scheduler remove the stored stopped information of container and select a new candidate that could be executed from all the waiting containers.

3.5 GPU memory allocation algorithm

The *Adaptive Fair-Share* (after denoted as ***Adaptive F-S***) algorithm for GPU-memory allocation that we have proposed in our previous work [16]. is used to determines GPU memory allocation for each container at the initialization time.

3.5.1 Adaptive fair-share algorithm

As opposed to the conventional way, allowing all active containers to be given the fixed size of GPU-memory one by one that leads a specific container to monopolize the whole memory, our proposed method primarily takes account of sharing the memory among the multiple containers evenly. Giving all active containers relatively equal access to the memory might not always be best, because the containers considered as more important might need to be given more resources than others. In addition, the degree of importance can even fluctuate depending on the properties of jobs running inside the container or the current global workload status.

For that reasons, our ***Adaptive Fair-Share*** method aims to serve the different amount of memory to the containers according to their importance which can be determined by the count and average input size of jobs. The importance of each container is periodically adjusted by local and global updates in order to adapt the state of both overall system and applications to the scheduling (Table 1).

Adaptive F-S method basically follows these abstract steps: (1) collecting jobs' information in the queue (2) categorizing them to multiple groups by the common properties (3) deciding the ratio and assigning the memory to each group by the fair share scheduling formulation (4) assigning the memory to the groups according to the distribution rate and rebuilding applications in accordance with the distributed memory (5) evaluating jobs and updating the formulation. Details of ***Adaptive F-S***'s procedure will be explained below with its algorithm.

Algorithm 1 Adaptive Fair-Share Algorithm

```
1:  function AdaptiveFS( 𝕁 )
2:      𝔾 = Grouping(𝕁);
3:      𝕊𝔼𝕋 = CalculDists(𝔾);
4:      GPU = MonitorAvailGPUMem();
5:      for each Set_i in Set_{1...n} of 𝕊𝔼𝕋 do
6:          AssignJobsToContainer(Set_i, GPU);
7:      end for
8:      TaskExecutionPlanning;
9:      Monitoring;
10:     Sleep(interval);
11: end function
```

Algorithm 1 describes the abstract procedure of the adaptive fair-share method. For a set of incoming jobs($(\mathbb{J})$) the count of which is denoted as *window*, it categorizes them into multiple groups depending on their properties (line 2). In general, the grouping can be made depending on the application name, input size, parameter types or user id, etc. Usually, we defined the group according to the user id, because the applications launched by the same user will have potential locality. After grouping, it calculates memory distributions according to the properties such as the number of jobs per group or input size, etc. (Fig. 10).

By the function CalculDists(*array*) (line 3), where it applies the following Equation 1 for each defined group *i*.where *m* is the count of the properties that are used for classification, and *n* is the number of groups. Where *C* is the weight of the properties calculated accordingly to the number of properties, $prop_i^k$ being the group i' rank, being proportional to the performances, in the overall ranking $\sum_{j=1}^{n} prop_j^k$, representing the summary of scores through

Table 1 Notations

Notation	Description
Set_i	Per-group memory distribution
$\mathbb{SET}$	A set of Set_i ($Set_{1...n}$)
$\mathbb{G}$	A set of groups ($group_{1...n}$)
$\mathbb{J}$	A set of jobs
window	The job count that $\mathbb{J}$ has
g_i	The number of jobs for each group
$prop_i^m$	The i th group's weight in terms of a property $prop_m$ (e.g., input size, application type)
GPU	Current available GPU memory size for the $node_n$
ER	Error counts
ε_{perf}	Threshold of performance
$\varepsilon_{workload}$	Threshold of workload
ε_{error}	Threshold of error count
$\mathbb{C}$	A set of coefficients, $C_0, C_1, ..., C_m$
interval	An execution interval for the procedure AdaptiveFS

$$Set_i = C_0 * \frac{g_i}{\sum_{j=1}^{n} g_j} + C_1 * \frac{prop_i^1}{\sum_{j=1}^{n} prop_j^1} + ... + C_{m-1} * \frac{prop_i^{m-1}}{\sum_{j=1}^{n} prop_j^{m-1}} \quad (1)$$

Fig. 10 Equation 1 for each defined group i

all the groups. For example, (*m-1*)th property score of a group i can be defined as $prop_i^{m-1}$. During the first few executions, C_0 is set as 1 and the other coefficients $C_{1 \le x \le m}$ are 0 until it has enough profiles to analyze, meaning only taking account of the count of jobs for the first several executions. Once enough records are collected in the system, all the coefficients $C_{0 \le x \le m}$ become 1 / m and they are adjusted by the procedure *AdaptiveUpdate* which will be explained in detail with the next algorithm. Regarding the decision of *prop* value, it is generated based on the relative rank the group has. For instance, suppose that the system generated five groups depending on three–four kinds of properties which are application name, input size, user id as well as job counts, as depicted on the left part of the Fig. 11. In the group 1, 2, 5 cases, they include four kinds of properties which induce all coefficients($C_{0 \le x \le m}$) to 1 / m, that is 0.25.

Each group's memory distribution ratio Set_i (after several executions) can be defined as shown on the right part

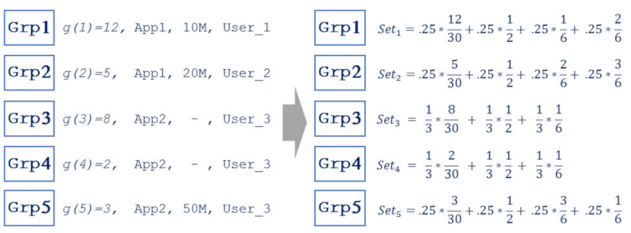


Fig. 11 An example of grouping and scoring, the properties of each group(left), the calculated dist. ratio(right)

of Fig. 11. For the first group, the property scores are 12 (job counts), 1 (1st rank among two applications), 1 (1st rank among three applications), 2 (among three applications), respectively. In the third group case, the coefficient is 1 / 3, since it has only three properties, and it calculates only for three properties. In this way, the system calculates the distribution ratios for groups according to the weights.

For all the groups, it prepares to assign the jobs to the container with the part of memory. Function AssignJobsToContainer(*float*, *int*) (line 6) includes the following three steps; rebuilding jobs, creating & deploying containers, launching jobs on the container. After deploying the containers and assigning jobs, it keep observing the execution status of each containers and managing the running order of them to prevent OOM occurs, (line 8). A cycle of the whole processes in AdaptiveFS procedure repeats itself within regular *interval*

3.6 Container scheduling example

In this section, we demonstrate how HPC workloads and DL workloads being scheduled by our proposed system respectively.

3.6.1 HPC workloads

According to the result of application execution pattern introduced above, we designed an execution plan for multiple containers to share a single physical GPU. As an example, we supposed to deploy 2 LAMMPS containers on a single GPU. In this situation, Both of the two containers' memory usage spikes at the end of the entire turn. As shown in Fig. 12, due to the lack of GPU memory, an OOM error occurs and lead to a container execution failure. However, as shown in Fig. 13, We can suspend

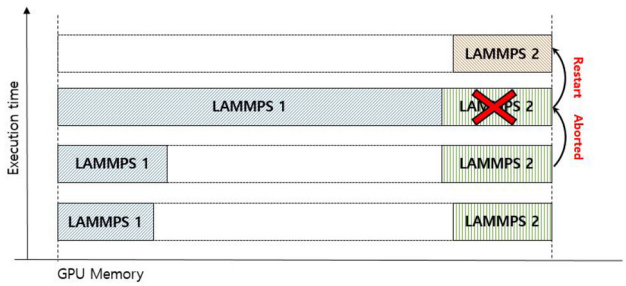


Fig. 12 Execution 2 LAMMPS containers

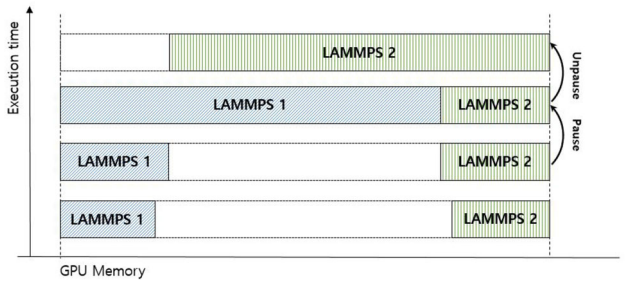


Fig. 13 Execution 2 LAMMPS containers with pause/unpause

container LAMMPS 2 before it starts to consume large amount of GPU memory and keep container LAMMPS 1 running until it terminates. Then container LAMMPS 1 will deallocate the allocated GPU memory and container LAMMPS 2 can be executed again. In addition, in the case of HPC application, the GPU memory will be reserved even the application is paused. Thus why we use the adaptive fair sharing algorithm to determine a Max GPU memory usage for each container to prevent the Deadload occurs. By following this execution plan, multiple containers can be run simultaneously without restarting the failed ones.

3.6.2 DL workloads

In this section, we use a particular scenario to describe how the check point based mechanism deals the case when multiple containers running DL workloads require to share the GPU. As described in Fig. 14a, Container 1 and 2 are running on the single physical GPU and then container 2 asks for more GPU memory that exceeds the physical capacity of the GPU. At the same time, container 3 joins with the system and waits for being scheduled. However, container 3 cannot be scheduled at once since there is no free GPU memory remained for its execution. As depicted in Fig. 14b, the scheduler detects the request and makes a decision to transferred container 2's GPU memory contents to host memory and reclaim its occupied space for container 3. Then the container 3 starts execution.

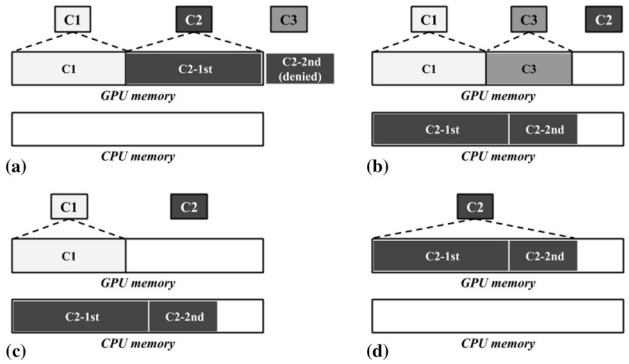


Fig. 14 Scheduling example

After container 1 and container 3 have finished their execution, as described in Fig. 14c, the size of free GPU memory becomes available for container 2's execution. Then the scheduler awake container 2 and bring it back to the GPU as shown in Fig. 14d.

4 Evaluation

In this section, we demonstrate and analysis the evaluation result of our proposed mechanism.

4.1 Experiment environment

Our experiments are conducted on the framework consisting of two node clusters. Table 2 provides the details of the clusters. In particular, the experiments are conducted on the GPU containers.

4.2 Adaptive fair sharing algorithm

demonstrate the performance improvement. We compare our scheduling method to two conditions:

1. *Default*, a baseline GPU memory distribution ratio offered by state-of-art Nvidia-docker system.
2. *Static fair-share*, a static GPU memory distribution ratio to each group in which all jobs have same application characteristics ($Set_{1...n} = GPU/n$, n is the number of groups defined).
3. *Adaptive fair-share*, the proposed method.

In this experiment, we evaluate all scenarios on both *homogeneous* and *heterogeneous* workloads. We form the homogeneous workloads using multiple copies of the same application. For heterogeneous workloads, we make them up by randomly selecting a number of applications out of two applications. and build 10 kinds of heterogeneous combination of parameters for each application. In total we evaluate 40 homogeneous and heterogeneous workloads.

Table 2 Experimental setting

	CPU	GPU
Architecture	Intel(R) Core(TM) i7-5820K	Nvidia GeForce TItan Xp D5x
Core clock	3.30GHz	1.58GHz
Num of cores	6 cores	128 CUDA cores
Mem. size	32 GB	12 GB
Threading API	–	Nvidia CUDA 8.0
Compiler	ICC (Intel compiler)	Nvidia C compiler (NVCC8.0)
OS	Ubuntu 16.04.3 LTS	Ubuntu 14.04.5 LTS

Overall, five thousand (5K) jobs are generated for each experiment, and we've exploited an average result from five times repetitions.

We set containers using TensorFlow [4] images for experiments. TensorFlow is a numerical computation using a data flow graph. It can use a python script without CUDA code where the number of GPUs, GPU memory usage, and etc can be specified and modified.

We employed two kinds of applications from representative domains, which are machine learning (ML) and molecular dynamics (MD), that are actively utilizing GPUs and that can generate different irregularity patterns. The details of them are as follows.

Convolution neural network (CNN) [17]—MNIST [18] is a composite product algorithm, mainly used for visual image analysis. CNN is composed of several hidden layers and is generally composed of convolution layers and pooling layers. The MNIST dataset is a numeric handwritten image. In this experiment, MNIST data of 60,000 numerical images consisting of 10 labels of 28 $\times$ 28 size was studied through CNN model composed of 3 convolution layers using TensorFlow.

AMBER [19, 20] is a suite of programs for biomolecular simulations in molecular dynamics(MD) field. It includes the collection of numerous programs that work together to setup, perform, and analyze MD simulations, from the preparation of the necessary input files, to the analysis of the results. Due to the specific characteristics that are the computational complexity and fine-grained parallelism of MD simulations of macro molecules, AMBER started to support GPU-based execution and include the adaptor program which helps to port from the existing Fortran code to the GPU platform using NVIDIA's compute unified device architecture (CUDA) language. All of the applications are executed use the default configuration provided by their organizations In this experiment, we employed its python module to carry it out with TensorFlow.

With the conditions that are addressed above, Figs. 15 and 16 depict the comparative results in terms of execution time and GPU memory utilization, respectively. The experimental environment is show in Table 2.

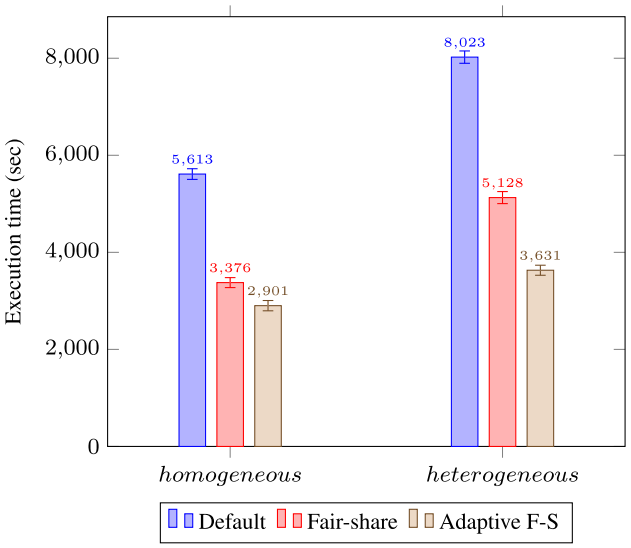


Fig. 15 Comparison of container execution time

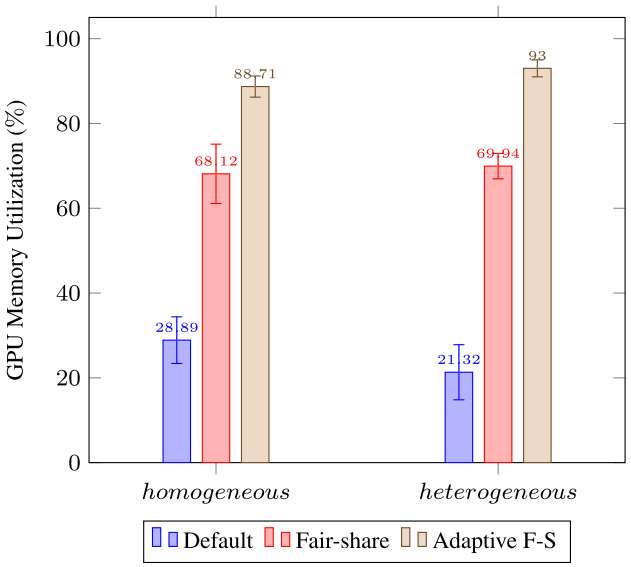


Fig. 16 Comparison of GPU Mem. utilization

4.2.1 Execution time

Figure 15 depicts the comparison results in terms of execution time. With *homogeneous* workload, the result for execution time allows us to realize (left group in the Fig.15) that the *default* condition causes the long average *makespan* time, since all jobs had to be carried out in sequential way. In the fair-share condition, the result is shorter than the *Default* condition (approximately 3376 s), since the GPU memory could be sharable and so it is possible to lead better performance than *default* one in this condition. Our adaptive fair-share method produces the shortest makespan time among the performed conditions, about 2901 s. Overall, the proposed method could improve the execution time by 16.37% compared to the static fair-share condition in the homogeneous workload group.

The experimental results with *heterogeneous* workloads (right group in the Fig.15) also show that the default condition results in the longest average makespan time among three conditions, mostly because of the waiting time between jobs caused by the random placements of the different applications to the identical container.

For both fair-share and adaptive fair-share conditions, the results present huge improvements, especially on the proposed method. The improvement in the proposed method mostly seems to be possible because of the grouping in similar jobs which induces rapid recycling of the containers and the evaluation step from by the adaptive update module.

4.2.2 GPU memory utilization

Figure 16 shows the impact of the different conditions on the GPU memory utilization. We can observe that the default condition has the inferior results for both workloads. Because the different kinds of jobs generate diverse kinds of containers, thus the average of utilization in heterogeneous seems to be lower than the one with homogeneous. We can also see that the fair-share condition led to lower memory utilization results, while it achieved relatively quite good execution time. Since multiple containers need to share GPU memory among groups and its amount is not really even regarding jobs' scale within each container, it results in the waste of resources leading to low utilization.

The adaptive fair-share condition shows higher GPU memory utilization than the default conditions in both two workload groups, and shows 67.43% and 22.54% higher memory utilization when compared with two conditions in homogeneous workload environment, 77% and 24.7% in heterogeneous environment.

To sum up, the overall experimental results show that the adaptive fair-share condition proposed in this paper is superior to the other two conditions (default, fair-share) in terms of execution time and GPU memory utilization in the homogeneous group and the heterogeneous group.

4.3 Pause & restart based mechanism

Figure 17 shows the example that our container scheduling method is applied. We run 12 CNN-MNIST, 1 LAMMPS, 2 QMCPACK and 1 GROMMAC containers on the same time and keep tracking the GPU resource usage of each container. From the Fig. 17, we can see that after executing 1400 s, the scheduler noticed that some container will require a large amount of GPU memory that may exceed the capacity, then it pause the containers that with lowest priority and keep those with higher priority running. After 1600 s, the previous unpaused containers finished their jobs and the scheduler then restart the previous paused containers to keep them running. Though our proposed method, multi containers can run simultaneously on a single GPU without OOM occurs, and the GPU resource utilization is substantially increased.

4.4 Checkpoint based mechanism

As mentioned above, one of our designing goals in solving the GPU memory over subscription problem for DL workloads running with TensorFlow in container based cluster. To see how our proposed checkpoint based mechanism to prevent OOM error occurs during the time when multiple containers launched on one single GPU, we execute the containers that running CNN training workload by using Alexnet model at 5 s interval to simulate the real situation in a cluster environment. We choose the Alexnet as the training model because it uses relatively fewer resources, which can maximize the effect of concurrent execution. We compared the average execution time on each case where the number of containers differs, by using our proposed mechanism to the cases that running with a Vanilla solution. As demonstrated in Fig. 18, the average execution time increased due to the contention of the GPU cores, since a single container will consume almost 50% of

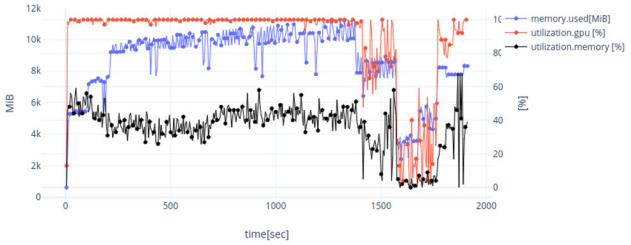


Fig. 17 Execution 12 CNN, 1 LAMMPS, 2 QMCPACKs and 1 GROMAC containers with pause/unpause

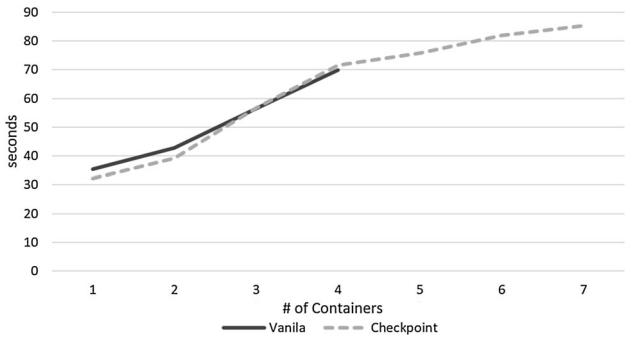


Fig. 18 Verification for preventing OOM error

the total GPU cores. We should notice that the Vanilla solution can only enable 4 containers running on the GPU concurrently when the 5th container tries to allocate GPU memory, an OOM error occurs due to the exceed of GPU memory capacity.

On the other hand, our proposed checkpoint based mechanism can handle the GPU memory over subscription problem very well. From the Fig. 18, we can tell the fact that even there are 7 concurrently running containers, it can still ensure all of them running safely.

5 Related work

There also exists various studies that handled GPU contention issues in Cloud or multiprocessor environment. Steran [21] suggested a high-level GPU distribution mechanism rather than a distribution of GPU system through low-level such as CUDA through SkelCL method. It shows a mechanism for automatic data redistribution by implicit movement between CPU and GPU memory through a container that can be accessed by both CPU and GPU. However, in applications such as machine learning, redistribution between CPU memory and GPU memory requires overhead to be resolved by proper distribution of GPU memory per container. Kämäräinen [22] explained the advantages of container by comparing the performance of virtual machine configuration and container configuration in GPU cloud gaming system. Contrary to a virtual machine, a container does not require pre-allocated memory, but it can use resources efficiently because it requires resources at a specific time of an application's runtime. Therefore, we propose efficient allocation of GPU memory resources using the advantages of containers.

6 Conclusion

This paper proposes a method to share GPU memories effectively in GPU-container clusters. The proposed adaptive fair-sharing strategy helps to overcome the limitation of sharing GPU memories among the containers which causes fatal performance degradation. We analyzed the problems that might happen in the GPU container clusters and conducted several experiments to show its performance degradation.

Our approach is compared with baseline and static fair share method by the evaluations. According to their results, it is able to improve the overall performances in terms of execution time, both GPU and GPU memory utilization. We also proposed a checkpoint based mechanism for DL workloads running with TensorFlow in container based environment to solve the GPU memory over-subscription problem. our evaluation result shows our proposed mechanism can ensure multiple containers that running TensorFlow jobs to share one GPU safely.

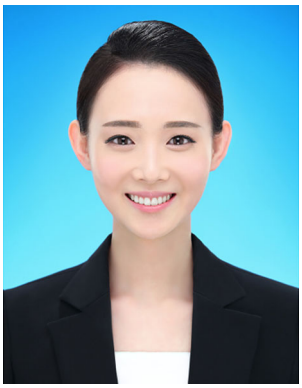
Acknowledgements The authors would like to thank all students who contributed to this study. We are grateful to Sejin Kim, who assisted with evaluation. This work has supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. NRF-2017R1A2B4005681).

References

1. Graphics Processing Unit. https://en.wikipedia.org/wiki/Graphics_processing_unit
2. Docker container. https://www.docker.com/
3. Calmels, J.: "nvidia docker". https://github.com/NVIDIA/nvidia-docker/wiki/nvidia-docker
4. Tensorflow. https://www.tensorflow.org/
5. Radchenko, G.I., Alaasam, A.B.A., Tchernykh, A.N.: Comparative analysis of virtualization methods in big data processing. Supercomput. Front. Innov. **6**(1), 48–79 (2019)
6. Naik, N.: Migrating from virtualization to dockerization in the cloud: simulation and evaluation of distributed systems. In: 2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA). pp. 1–8. IEEE (2016). https://doi.org/10.1109/MESOCA.2016.9
7. 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: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles—SOSP '03. pp. 164–177. ACM Press, New York (2003). https://doi.org/10.1145/945445.945462
8. Gupta, Vishakha, et al.: GViM: GPU-accelerated virtual machines. In: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing. ACM (2009)
9. Giunta, G., et al. A GPGPU transparent virtualization component for high performance computing clouds. European Conference on Parallel Processing. Springer, Berlin, Heidelberg (2010)
10. Shi, Lin, et al.: vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans. Comput. **61.6**, 804–816 (2011)

11. Duato, José, et al.: rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. 2010 International Conference on High Performance Computing and Simulation. IEEE (2010)
12. Kim, J., Jun, T.J., Kang, D., Kim, D., Kim, D.: GPU Enabled Serverless Computing Framework. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 533–540. IEEE (2018). https://doi.org/10.1109/PDP2018.2018.00090
13. Herrera, A.: NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. Tech. Rep. (2014)
14. Herrera, Al.: NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. Nvidia Corp 1–18 (2014)
15. Boettiger, C.: An introduction to Docker for reproducible research. ACM SIGOPS Oper. Syst. Rev. **49**(1), 71–79 (2015). https://doi.org/10.1145/2723872.2723882
16. Jisun O., et al.: Toward an Adaptive Fair GPU Sharing Scheme in Container-based Clusters, Foundations and Applications of Self* Systems (FAS*) (2018)
17. Convolution Neural Network. https://cs231n.github.io/convolutional-networks/
18. MNIST. http://yann.lecun.com/exdb/mnist/
19. Salomon-Ferrer, R., Case, D.A., Walker, R.C.: An overview of the Amber biomolecular simulation package. WIREs Comput. Mol. Sci. **3**, 198–210 (2013)
20. AMBER. http://ambermd.org/AmberMD.php
21. Breuer, Stefan, et al.: Extending the SkelCL skeleton library for stencil computations on multi-GPU systems. In: Proceedings of the 1st International Workshop on High-Performance Stencil Computations (2014)
22. Kämäräinen, Teemu, et al.: Virtual machines vs. containers in cloud gaming systems. In: 2015 International Workshop on Network and Systems Support for Games (NetGames). IEEE (2015)

Qichen Chen received his B.S. degree in Department of Computer Science and Infomation from NanJing XiaoZhuang University in 2011. Currently, he is an Ph.D. Candidate in Department of Computer Science and Engineering of Seoul National University. His research interests are operating, distributed, and database systems.



Jisun Oh is currently a Master's student in the Department of Computer Science, Sookmyung Women's University. She received her B.S. degree from Sookmyung Women's University in 2017. She is also researcher at the Distributed & Cloud Computing Lab of Sookmyung Women's University. Her research interests include cloud computing and management in distributed computing systems.

Seoyoung Kim received her Master degree from Sookmyung women's University in 2012 and She received her B.S. degree from Sookmyung Women's University in 2010. She is also researcher at the Distributed & Cloud Computing Lab of Sookmyung Women's University. Her research interests include cloud computing and management in distributed computing systems.

Yoonhee Kim she is the professor of Computer Science Department at Sookmyung Women's University. She received her Bachelors degree from Sookmyung Women's University in 1991, her Master degree and Ph.D. from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was the faculty of Computer Engineering dept. at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed computing systems. She is a member of IEEE and OGF, and she has served on variety of program committees, advisory boards, and editorial boards.