Toward An Adaptive Fair GPU Sharing Scheme in Container-based Clusters

Jisun OH
Dept. of Computer Science
Sookmyung Women's University
Seoul, Republic of Korea
jsoh8088@gmail.com

Seoyoung KIM
Dept. of Computer Science
Sookmyung Women's University
Seoul, Republic of Korea
sssyyy77@gmail.com

Yoonhee KIM
Dept. of Computer Science
Sookmyung Women's University
Seoul, Republic of Korea
yulan@sookmyung.ac.kr

***Abstract*—Virtualization is an innovative technology that accelerates software development by providing portability and maintainability of applications. However, it often leads underperformance especially caused by overheads from managing virtual machines. To address the limitation of virtual machines, container technology has emerged to deploy and operate distributed applications without launching entire virtual machines. Unfortunately, resources contention issues in container-based clusters, bringing substantial performance loss are still challenging. This paper proposes an adaptive fair-share method to share effectively in container-based virtualization environment. In particular, we focus on enabling GPU sharing between multiple concurrent containers without lack of GPU memory. 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.**

***Index Terms*—GPU scheduling, GPU memory, container-based virtualization, Docker, Mesos, heterogeneous, adaptive**

I. Introduction

A Graphics Processing Unit (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 efforts to utilize GPUs effectively have increased rapidly since the great success of deep learning and the introduction of several frameworks such as CUDA (Compute Unified Device Architecture) [2], OpenCL (Open Computing Language) [3]. This has led to their deployment in a wide range of platforms such as cloud computing, large-scale distributed computing environments.

Virtual Machine (VM) has then emerged, offering various options for users to choose, which helped to maximize their applications' performance. This, however, led to underperformances especially caused by overheads from their management (e.g., deploying & destroying) and the rise of running costs on the cloud.

To overcome such performance overheads, Docker [4] which is an open-source virtualization technology of container is adopted as an alternative. It is similar to a virtual machine, but it is very light weight and takes seconds to build. In addition, it isolates each independent container running on the same instance of operating system by making use of Linux kernel features like control groups and namespaces. Each docker container encapsulates an application and can be run on different machines on top of a docker engine. Their images can also be easily shared and distributed once they have been built. This in turn reduces the time for testing, development and deployment.

The above mentioned shift, however, require effective methods to improve the utilization of GPUs and to share them properly in the virtualized environment.

NVIDIA [5] has presented a way to share GPU drivers from host to containers, without having them installed on each container individually. However, the way that NVIDIA Docker assigns the physical resources is generally limited to a single container. That is, only one container is able to occupy the whole GPU memory without sharing it.

This paper aims to investigate an effective method for sharing GPUs and GPU memory without contention among containers. Specifically, this paper's contributions are as follows :

- GPU & GPU memory sharing's drawbacks analysis in the current container-based environment
- An algorithm to share GPU memory effectively for GPU containers
- A framework that enables effective & scalable on-demand container-based computing environment

The organization of this paper is as follows.

In Section II, we discuss the current system's drawbacks by analyzing the problems and some related works. Section III introduces our algorithm in detail. The succeeding section shows the overall architecture that we have designed and how the proposed algorithm can be applied to the architecture. After that, we present the algorithm's demonstration by evaluations. Finally, we conclude the paper in Section V.

978-1-5386-5175-9/18/$31.00 ©2018 IEEE
DOI 10.1109/FAS-W.2018.00029

II. Problem Statement & Related Works

We start this section by discussing the problem that may happen in container-based virtualization environment for GPUs. The main problem is depicted on Figure 1. Suppose

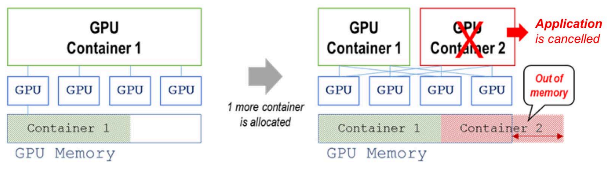

Fig. 1: GPU memory contention problem

that only one GPU container exists and it uses certain amounts of GPU and memory. Once the second container is added, it starts to assign GPUs and memory to container 2. However, as shown on the figure 1, it returns "Out of memory" since the remaining memory space's size cannot cover what the second container needs. In the end, the system cancels the execution of container 2 instead of controlling memory size or sharing the memory. It is challenging to make the system share the memory to multiple containers within appropriate proportions. Despite the fact that the container technology supports better isolation and performance to utilize the limited resources, it still needs a proper strategy which can assign and share the GPUs & their memory to containers depending on their real-needs. The benefits and challenges of containerized systems have been studied in many aspects. Related with the container-based environment, there exist several researches which emphasize usually managing CPU, Network and I/O contention [8], [9], [10], [11], but it is very rare for the GPUs.

There also exists various studies that handled GPU contention issues in Cloud or multiprocessor environment. Steran [12] 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. Kmrinen [13] 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.

III. Proposed GPU container scheduling Strategy

To minimize the weakness that was mentioned in the previous section and to make better use of the container system, this paper propose an *Adaptive Fair-Share* (after denoted as ***Adaptive F-S***) algorithm for GPU-memory.

A. 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.

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.

TABLE I: 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

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. After grouping, it calculates memory distributions according to the properties such as the number of jobs per group or input size, etc., by

Algorithm 1 Adaptive Fair-Share Algorithm

```
1: function ADAPTIVEFS( $\mathbb{J}$ )
2:     $\mathbb{G}$ = GROUPING($\mathbb{J}$);
3:     $\mathbb{SET}$ = CALCULDISTS($\mathbb{G}$);
4:     $GPU$ = MONITORAVAILGPUMEM();
5:     for each $Set_i$ in $Set_{1...n}$ of $\mathbb{SET}$ do
6:         ASSIGNJOBSTOCONTAINER($Set_i$, $GPU$);
7:     end for
8:     MONITORSYSTEM();                          ▷ Evaluation
9:     SLEEP($interval$);
10: end function
```

the function CALCULDISTS($array$)(line 3), where it applies the following Equation 1 for each defined group i.

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

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 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 Figure 2. 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.

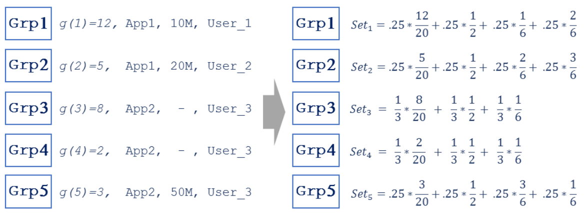
Grp1 g(1)=12, App1, 10M, User_1
Grp2 g(2)=5, App1, 20M, User_2
Grp3 g(3)=8, App2, - , User_3
Grp4 g(4)=2, App2, - , User_3
Grp5 g(5)=3, App2, 50M, User_3
Grp1 $Set_1 = .25 * \frac{12}{20} + .25 * \frac{1}{2} + .25 * \frac{1}{6} + .25 * \frac{2}{6}$
Grp2 $Set_2 = .25 * \frac{5}{20} + .25 * \frac{1}{2} + .25 * \frac{2}{6} + .25 * \frac{3}{6}$
Grp3 $Set_3 = \frac{1}{3} * \frac{8}{20} + \frac{1}{3} * \frac{1}{2} + \frac{1}{3} * \frac{1}{6}$
Grp4 $Set_4 = \frac{1}{3} * \frac{2}{20} + \frac{1}{3} * \frac{1}{2} + \frac{1}{3} * \frac{1}{6}$
Grp5 $Set_5 = .25 * \frac{3}{20} + .25 * \frac{1}{2} + .25 * \frac{3}{6} + .25 * \frac{1}{6}$

Fig. 2: An example of grouping and scoring, the properties of each group(left), the calculated dist. ratio(right)

Each group's memory distribution ratio Set_i (after several executions) can be defined as shown on the right part of Figure 2. 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 monitors the overall system status for both local and global aspects and evaluates the executed jobs (line 8), which will be discussed further details in Algorithm 2. A cycle of the whole processes in ADAPTIVEFS procedure repeats itself within regular *interval* (line 8).

Algorithm 2 states the monitoring procedure in charge of monitoring and evaluating, which consists of two main functions. One part, referred as LOCAL monitoring, is to observe and compare the performances of jobs running over containers to the previous records, thereby it is able to examine if there exists the performance degradation on the jobs and containers and to update the coefficients value in the end (line 2-7). Second part, GLOBAL monitoring, is to read global-scale status and to apply it to the scheduling by adjusting the coefficients and the number of groups. If the number of waiting jobs in the queue is over the threshold ($\varepsilon_{workload}$), it reflects the state to the scheduling system so that it can update the interval and window size. In addition, it supplements additional containers to avoid waiting jobs.(line 8-14).

In the case of that *ER* counts over the threshold (ε_{error}), the system updates properties' scores as well as their coefficients (line 15-18). The implementation issue for the aforementioned algorithms will be treated further in Section IV-A.

Algorithm 2 *MonitorSystem ()*

```
1: function MONITORSYSTEM
2:     procedure LOCAL( )
3:         $diff \leftarrow$ PROFILESAVECHECK();
4:         if $diff \ge \varepsilon_{perf}$ then
5:             ER++
6:         end if
7:     end procedure
8:     procedure GLOBAL( )
9:         $queue \leftarrow$ CHECKWORKLOAD();
10:        if $queue \ge \varepsilon_{workload}$ then
11:            $\mathbb{G}.increase()$;
12:            interval.$decrease()$; window.$increase()$;
13:        end if
14:    end procedure
15:    if ER $\ge \varepsilon_{error}$ then
16:        prop.$update()$;
17:        ADAPTIVEUPDATE();
18:        ER.$initial()$;
19:    end if
20: end function
```

IV. Evaluation

In this section, we introduce the framework that we implemented and utilized for the experiments.

A. Experimental Architecture

Figure 3 presents the abstract system architecture that we have designed. Overall system (Figure3) is classified in three layers, which are framework, kernel and infrastructure parts. The topmost layer, *Framework*, is containing the wrapper, application profiler, resource manager, scheduler, and monitoring module. Our adaptive F-S algorithm is incorporated in the scheduling module of this layer.

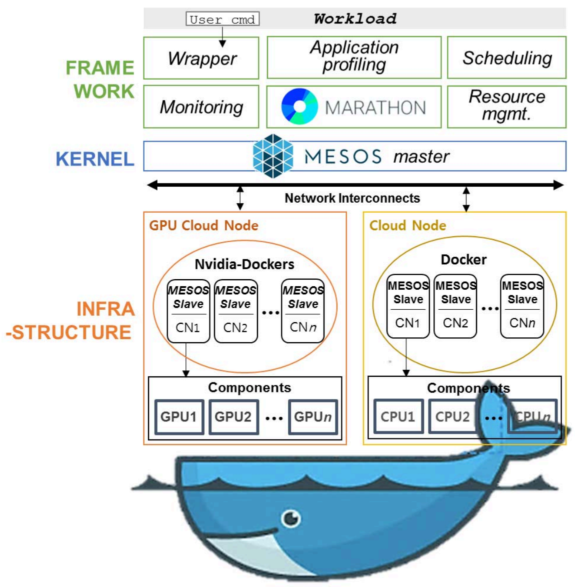

Fig. 3: System Architecture

The resource scheduler generates resource-specific tasks and manages the orchestration of heterogeneous resources. Tasks can be created by setting the amount of resources, number of containers, network setting, environment variable setting. The scheduler is generally performed after confirming the profiled property information of each application. One of the main roles of this step is to create a scheduling plan that predicts resource usage of applications and appropriately places tasks on GPUs and CPUs container nodes. Application profiling module's main role is to provide and analyze the characteristics that applications have. The main technique used for this module is based on the previous work [15].

The second and third layers perform the actual task, and consist of the master node, resource pool, respectively. The master node is responsible for identifying, allocating, and managing different types of heterogeneous resources. It is based on Mesos mater. A system can dynamically allocate resources by controlling the resources of multiple nodes on a single computer and recognizing available resource information. It perceives and controls specific resources (CPU, GPU, memory, disk, etc) of each node's container according to the schedule created by the job scheduler. Nodes in the infrastructure layer are composed of nvidia-dockers and docker-based containers, respectively, and containers are controlled by the master node. It is based on the Mesos slave of the container and connects directly to the master node to inform the resources.

To achieve a stable architecture, we have employed several well-known researches and techniques. Here, we are going to explain them in details. *Mesos* [6] is a distributed system kernel, a resource management system designed to manage resources of cloud infrastructure and computing systems in an integrated manner. The resources of various computing systems bundled in a network are grouped into a pool, managed by grouping together CPU, GPU, memory, and disk. Second important framework that is used for our system is Marathon. *Marathon* [7] is a part of Mesos framework that supports long-running applications such as web-applications. If Mesos is considered as a kernel, Marathon acts as an *init* or *daemon*. The most important one among technologies we employed is *Nvidia docker* [5], which is a GPU-enabled docker container. GPU-accelerated applications can be containerized and deployed on a GPU-supported infrastructure. Container runtime as well as extensible Docker container technology throughout the orchestration system.

B. Evaluation Scenario & Setup

To 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. 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 [16] 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 x 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 macromolecules, 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 NVIDIAs Compute Unified Device Architecture (CUDA) language. In this experiment, we employed its python module to carry it out with Tensorflow. Our experiments are conducted on the framework (described in Sec. IV-A) consisting of two container-based clusters. Table II provides the details of the clusters. In particular, the experiments are conducted on the GPU containers.

TABLE II: 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

C. Evaluation & Results

With the conditions that are addressed above, Figure 4 to 6 depict the comparative results in terms of *execution time*, *GPU utilization*, and *GPU memory utilization*, respectively. The experimental environment is shown in Table II.

*1) **Execution Time**:* Figure 4 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.4) 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 3,376 seconds), 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 2,901 seconds. Overall, the proposed method could improve the execution time by 16.37% compared to the static fair-share condition in the homogeneous workload group.

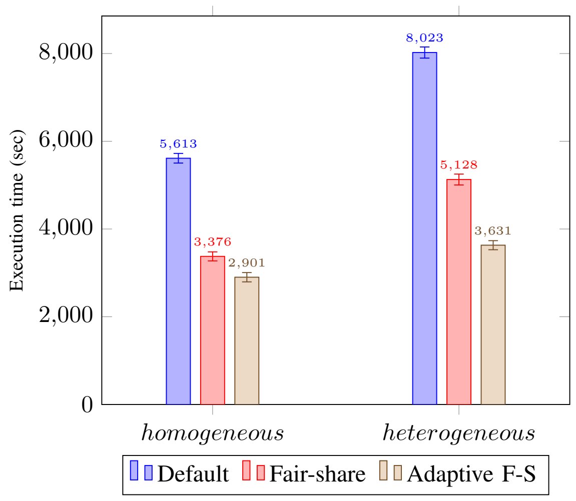

Fig. 4: Comparison of Container Execution time

The experimental results with *heterogeneous* workloads (right group in the Fig.4) 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.

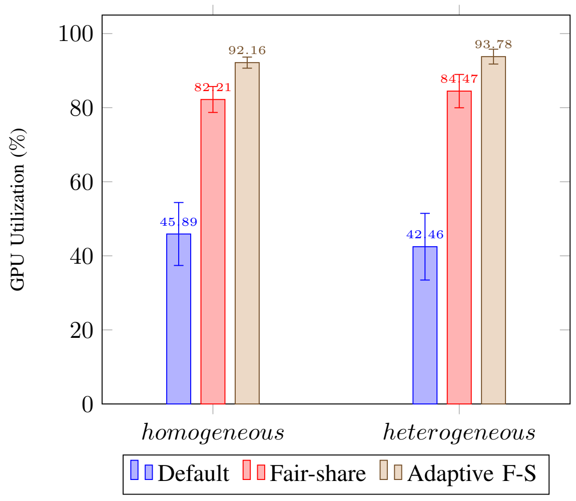

Fig. 5: Comparison of GPU Utilization

*2) **GPU Utilization**:* In the GPU utilization aspect, the results of default conditions show that it leads to poor GPU utilization for both workloads, because it cannot assign the whole GPUs to all containers due to the problem as mentioned in Section II. By sharing GPUs among multiple containers, the fair-share and adaptive fair-share could achieve better utilization than the default's one. However, in fair-share condition, all containers can only get the fixed amount ratio of memory regardless of the characteristics of jobs within a group. It has a bad effect on utilizing GPUs at their best. The adaptive fair-share method results in the highest GPU utilization as depicted on Figure 5, since it can adapt to the system status and control it depending on the status in order to achieve better resource utilization. Overall, it could achieve 50.2 %, 10.7 % improvements than the other conditions (default, F-S, respectively) in the homogeneous, and 54.72 %, 9.9 % with heterogeneous workloads.

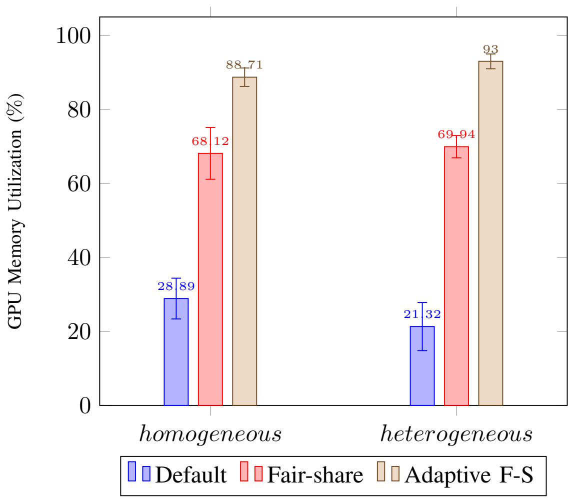

Fig. 6: Comparison of GPU Mem. Utilization

*3) **GPU Memory Utilization**:* Figure 6 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, GPU utilization, and GPU memory utilization in the homogeneous group and the heterogeneous group.

V. 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 are planning to extend our scheduling algorithm that is considering both CPU and GPU elements in the global framework. In addition, we are going to implement APIs to port and support general applications to apply the adaptive fair share scheduling method we researched.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2017R1A2B4005681)

References

[1] Graphics Processing Unit, https://en.wikipedia.org/wiki/Graphics_processing_unit
[2] CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit
[3] The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl/
[4] Docker container, https://www.docker.com/
[5] J. Calmels, "nvidia docker, https://github.com/NVIDIA/nvidia-docker/wiki/nvidia-docker
[6] Mesos, https://en.wikipedia.org/wiki/Apache_Mesos
[7] Marathon, https://mesosphere.github.io/marathon/
[8] S. Soltesz, H. Potzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, Container-based Operating System Virtualization: A Scalable, High performance Alternative to Hypervisors, in Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 07, 2007, pp. 275287.
[9] G. Calarco and M. Casoni, On the Effectiveness of Linux Containers for Network Virtualization, Simulation Modelling Practice and Theory vol. 31, 2013, pp. 169-185.
[10] McDaniel, Sean, Stephen Herbein, and Michela Taufer, A two-tiered approach to I/O quality of service in Docker containers, Cluster Computing (CLUSTER), 2015 IEEE International Conference on. IEEE, 2015, pp. 490-491.
[11] Bhimani, Janki, et al., Docker container scheduler for i/o intensive applications running on nvme ssds, IEEE Transactions on Multi-Scale Computing Systems, 2018.
[12] Breuer, Stefan, et al. "Extending the SkelCL skeleton library for stencil computations on multi-GPU systems." Proceedings of the 1st International Workshop on High-Performance Stencil Computations. 2014.
[13] Kmrinen, Teemu, et al. "Virtual machines vs. containers in cloud gaming systems." Network and Systems Support for Games (NetGames), 2015 International Workshop on. IEEE, 2015.
[14] Kay, J, P Lauder, C Maltby, The SHARE Charging and Scheduling System, Proc ACSC-6 (Aust Computer Science Conference), 1983, pp.14
[15] Seoyoung Kim, J-S. Kim, S. Hwang, and Y. Kim. An allocation and provisioning model of science cloud for high throughput computing applications. In Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC '13). ACM, New York, USA, , Article 27 , 8 page, 2013.

[16] Tensorflow, https://www.tensorflow.org/
[17] Convolution Neural Network, https://cs231n.github.io/convolutional-networks/
[18] MNIST, http://yann.lecun.com/exdb/mnist/
[19] R. Salomon-Ferrer, D.A. Case, R.C. Walker. An overview of the Amber biomolecular simulation package. WIREs Comput. Mol. Sci. 3, pp.198-210, 2013.
[20] AMBER, http://ambermd.org/AmberMD.php