ConVGPU: GPU Management Middleware in Container Based Virtualized Environment

Daeyoun Kang, Tae Joon Jun, Dohyeun Kim, Jaewook Kim, Daeyoung Kim
School of Computing
Korea Advanced Institute of Science and Technology
Email: {ikasty, taejoon89, stalker7, kjwook, kimd}@kaist.ac.kr

***Abstract*—Nowadays, Graphics Processing Unit (GPU) is essential for general-purpose high-performance computing, because of its dominant performance in parallel computing compare to that of CPU. There have been many successful trials on the use of GPU in virtualized environment. Especially, NVIDIA Docker obtained a most practical way to bring GPU into the container-based virtualized environment. However, most of these trials did not consider sharing GPU among multiple containers. Without the above consideration, a system will experience a program failure or a deadlock situation in the worst case. In this paper, we propose ConVGPU, a solution to share the GPU in multiple containers. With ConVGPU, the system can guarantee the required GPU memory which the container needs to execute. To achieve it, we introduce four scheduling algorithms that manage the GPU memory to be taken by the containers. These algorithms can prevent the system from falling into deadlock situations between containers during execution.**

I. Introduction

The most important thing to support the current cloud computing system is the existence of Virtual Machine (VM). VM has been widely used to simulate devices on the Operating System (OS) virtually. VM virtualizes a single computer as virtualized environment and is essential for cloud computing service providers who need to support multiple users. However, container-based virtualization has emerged as a new technology to replace VMs due to its lightness and ease of use. The container has a benefit of faster build time and execution time compared to those of VM. The fast processing speed of the container comes from its size. Unlike a VM that requires virtualization of the entire OS, the container virtualizes only a small fraction of the OS. Because of its lightness, a machine using the container-based virtualization solution can provide increased number of virtualized environments with reduced amount of resource.

Since the Graphics Processing Unit (GPU) is originally designed as a component for faster graphics rendering, it has a distinct architecture for parallel computing to apply the same algorithm to each pixel. In recent years, not only graphics rendering but also the powerful parallel computing capabilities of the GPU have begun to use as general-purpose high-performance computing in both industry and research areas. Several platforms, such as CUDA (Compute Unified Device Architecture) [1] and OpenCL (Open Computing Language) [2], have been released for general-purpose use of GPUs. For example, CUDA is a parallel computing platform that includes a programming model and application programming interface (API) for NVIDIA GPU. GPU is applied in various fields such as machine learning and is frequently used in the cloud environment for performance enhancement. Therefore, there has been an effort to use the GPU more efficiently in a virtualized environment.

A virtualization system controls the host's hardware computing resources to provide it for multiple virtualized environments. The virtualization system provides various hardware components required for the virtualized environment. For example, the Docker [3], one of the popular container-based virtualization solution, uses *cgroups* provided by the Linux kernel to control CPU time, memory, and network bandwidth. It also manages disk storage using the Union File System (UnionFS). However, GPU had not been available in any virtualization system for a long time.

There have been several attempts such as GViM [4], gVirtuS [5] and vCUDA [6] to provide virtualized GPU (vGPU) in virtualization environments. They created a copy of CUDA API to virtualize the GPU and provide it to the virtual machine. rCUDA [7], a solution for using remotely located GPUs, had a similar trial and therefore have tried to use the GPU in the virtualization system. Nonetheless, these methods have their weaknesses since it degrades the performance of the GPU during the process of virtualizing the GPU itself. Moreover, those methods have remained a subset copy of original CUDA API because the CUDA is not publically documented. There were some other attempts exist such as NVIDIA GRID [8] to virtualize the GPU at the hardware level. However, it requires to use only a specific device and does not support a container-based virtualized environment. Unlike the concept of container-based virtualization, providing a fully virtualized GPU in the container has never been successful.

Recently, NVIDIA Corporation proposes the NVIDIA Docker [9] for the Docker environment, which is entirely different approach from the other we introduced above. The NVIDIA Docker simply assigns the entire GPU to a single container. This method can solve most of the problems discussed above. In the Docker container, the allocated GPU is freely available, and there is almost no performance degradation. However, a fatal problem occurred when trying to use the same GPU by different containers at the same time. The NVIDIA Docker only assigns GPU and does not care how the user program inside the container uses GPU. In this case, concurrent access including memory allocation to the GPU

2168-9253/17 $31.00 © 2017 IEEE
DOI 10.1109/CLUSTER.2017.17

memory may happen by multiple containers. However, the total amount of GPU memory is limited, and swapping GPU memory is currently not supported. Therefore, accessing the same GPU at the same time by different containers may cause a program failure. In the worst case, a deadlock situation can occur which is shown in more detail in our previous study [10].

Other attempts exist for the better resource sharing among the container based virtualized environment. Monsalve et al. provide a solution to allocate CPU resource to the container with optimizing for cache access [11]. McDaniel et al. proposed a solution for the load balancing of input/output quality of service (QoS) [12]. Dusia et al. present an extension to configure the network priority to the container in the matter of QoS [13]. However, there is no prior research to share GPU resource among multiple containers.

In this paper, we propose ConVGPU, a solution to sharing GPU with containers in container-based virtualized environment. ConVGPU can containerize and restrict GPU resources for multiple containers. The concept of the ConVGPU is captures the GPU resource allocation/deallocation data from each containers and manages the containers using these information. ConVGPU is fully compatible with existing CUDA API, so the user does not need to take care of it. We implement this solution on NVIDIA Docker we described above. Moreover, ConVGPU schedules containers in the manner of fair use of GPU resources, which enables to execute additional GPU dependent containers in the machine more than its capacity. With our experiment, the ConVGPU has a negligible performance degradation. We implement four basic scheduling algorithms to compare their performance. Our experiment shows that the Best-Fit algorithm is the fastest algorithm in the overall execution time for all given containers. The overall system architecture is depicted in Figure 1.

The rest of this paper is organized as follows: In Section II we provide background information about GPU, CUDA, and Docker. Section III explains details of the ConVGPU with its design concept and example of the scheduling algorithm. In Section IV, we evaluate the efficiency of our proposed solution. Finally, we conclude this paper in Section V.

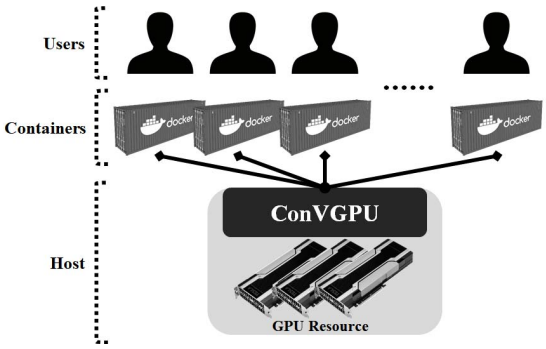


Fig. 1. Overall System architecture

II. Backgrounds

A. GPU and CUDA

GPU is computing device component for faster graphics rendering. Since rendering needs to compute same independent algorithms for every pixel, GPU is optimized for parallel execution. Nowadays, GPU has also been used for general-purpose computing because of its characteristic. Because the data consumed by GPU should be copied from CPU, many organizations introduce several computing platforms to use GPU easier for general-purpose.

Since CUDA [1] was released in February 2007, it has become one of the most widely used platforms for general-purpose GPU (GPGPU) [14]. CUDA extends ANSI C language and provides APIs for an easy way to use. CUDA API serves two programming interfaces which are Driver API and Runtime API. Runtime API is high-level API which is implemented on top of low-level Driver API. Runtime API is easier to use than Driver API because it provides implicit initialization and management. However, Driver API can perform fine-grained context control and module loading. The user program which uses GPU, some part of it is written to be executed on the GPU. This piece of the program is called a *kernel* and runs on the GPU after it is sent to the GPU. Any program using CUDA has to be compiled with *nvcc*, a compiler which is part of the CUDA platform [14].

B. GPU Virtualization

Many frameworks are introduced to make the GPU usable in the virtualized environment. These frameworks use Remote-API technique to serve virtualized GPU in the virtual environment. The concept of Remote-API technique is capturing API calls from the virtualized environment and redirecting it to the host system. Thus, API calls for using GPU can be executed even the GPU does not exist in the virtualized environment. As a result, the method communicating between virtualized environment and the host matters. Table I describes and compares the frameworks. The main disadvantage of these methods is a difficulty of capturing APIs. If there is a change in API, the extra effort requires to follow-up the new version of it.

Another method to virtualize GPU is PCI pass-through technique. This technique directly assigns GPU to the virtualized environment. In conventional VM, this technique requires IOMMU support from CPU. However, in container based virtualized environment, it can easily be acquired by linking the device mount to the container. OpenStack [15] supports this method by recording the device ID of GPUs in the Nova configuration file [16]. NVIDIA Docker allows the docker container to use the GPU by using `--device` option.

TABLE I
Comparing the Remote-API frameworks

Frameworks	GViM [4]	gVirtuS [5]	vCUDA [6]	rCUDA [7]
Network method	XenStore	TCP/IP (VMSocket)	VMRPC	Sockets API

The advantage of PCI pass-through technique is the user program in virtualized environment can access GPU directly. This yields performance without degradation. However, GPUs which is connected to the virtualized environment are not able to be shared with others, because there is no controller or solution exist.

Compared to the techniques we mentioned above, NVIDIA GRID [8] virtualizes GPU at the hardware level. It creates vGPU and directly assigns to each VM. For example, NVIDIA GRID K1 can serve up to four simultaneous VMs. However, NVIDIA GRID requires a specific type of GPU like NVIDIA GRID K1 and K2. Moreover, GRID only supports Citrix XenApp (including some other Xen series) and VMware Horizon; it is not compatible with any container based virtualization solutions.

C. Docker

Docker is an open source application container engine [17] that provides container-based virtualization. After its release in 2013, docker has become the most widely used container-based virtualization system and is now a de-facto choice. Docker currently supports Linux, FreeBSD, and Windows x64. In this paper, we only considered the Linux environment, where docker is mostly used.

Docker uses libcontainer [18] for process sandboxing. The docker is built on a variety of previously available kernel technologies. It uses *cgroups* (abbreviated from control groups) to separate processes belonging to each container and to handle their CPU time or memory limit using subsystems. It also uses Linux namespaces to have a separate process ID (pid), mount point (mnt), hostname, and domain name. Using these kernel features, the docker can effectively isolate particular processes from the host system. However, there is no scheme to manage GPU resources, which is covered in this paper.

D. NVIDIA Docker

NVIDIA Docker is a utility set for using NVIDIA GPU easily inside a docker container. NVIDIA Docker contains two executable programs which are `nvidia-docker` and `nvidia-docker-plugin`.

nvidia-docker is a thin wrapper on top of docker [19] which captures the command from the user so that the user can use nvidia-docker command instead of the original docker command. Because nvidia-docker only adds some options to the user input, it is compatible with docker. Its role is interpreting and editing user command, and sending it to the docker command interface. nvidia-docker only captures `run` and `create` command, and the other docker commands are passed through to the docker. It can check whether the image uses CUDA API or not, using `com.nvidia.volumes.needed` and `com.nvidia.cuda.version` docker label, which indicates required CUDA version. nvidia-docker uses these label information, detects number and location of installed GPU devices, and links it with a `--device` option. Also, it links proper version of GPU drivers using `--volume` option, which is directly connected to nvidia-docker-plugin. Our proposed solution customizes nvidia-docker to add a feature that controls GPU memory.

nvidia-docker-plugin is one of the volume plugins of the docker, which expands docker's functionality [20]. The intention of nvidia-docker-plugin is checking the existence of NVIDIA GPU and CUDA API, and serving a proper version of binaries and library files to the container. The CUDA API version which nvidia-docker inspects is delivered to the nvidia-docker-plugin using docker volume name. When the container stops running, the volume that mounted to the container will be unmounted.

III. Design and Implementation

ConVGPU is designed for sharing GPU resources among multiple containers by virtualizing amount of GPU resources, especially GPU memory. ConVGPU consists of two major components; *GPU memory scheduler* and *CUDA wrapper API module*. GPU memory scheduler checks the GPU memory limitation of each container. When the container uses GPU memory until it reaches its limitation, scheduler denies the allocation call or pauses its execution if needed. The CUDA wrapper API module is inserted inside into the container when the container is created. This module captures original CUDA API calls from the user program inside the container. These components (including NVIDIA Docker) are connected and communicating using UNIX Domain Socket (UNIX socket) with JSON (JavaScript Object Notation) format. In the rest of this section, we provide details of each component, and scheduling example is followed.

A. System Design

The overall system design is shown in Figure 2. When a user sends a command to the customized nvidia-docker, it redirects its command to the original docker with options for using NVIDIA GPU. At the same time, ConVGPU inserts the CUDA wrapper API module to the container. During container runtime, CUDA wrapper API module captures some APIs and send it to the GPU nemory scheduler. GPU memory scheduler schedules container and its GPU memory usage.

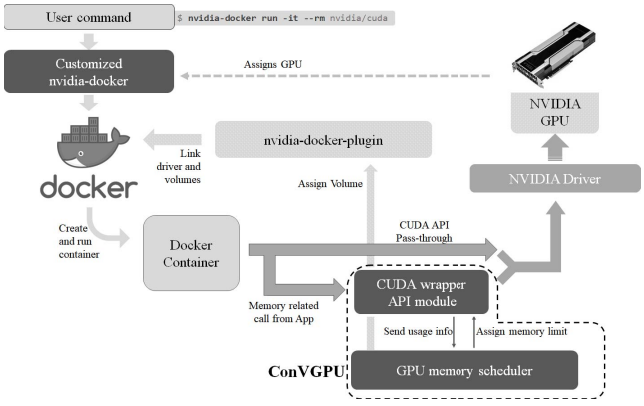


Fig. 2. System design of ConVGPU

We have considered the following factors in the ConVGPU:

- **Isolation**. Conventional GPU virtualization solutions did not fit into a containerized environment because it attempted to virtualize the entire GPU which is opposed to the concept of the container. Instead, we designed to ensure that each user program is completely isolated when using GPU.
- **Consistency**. Without proper GPU memory management system, containers using the GPU may collide or occur malfunction. ConVGPU should manage the GPU memory to prevent this problem so that failures in one container would not affect other containers. Also, the container should work well even if it is created more than the capacity of the GPU memory.
- **Shareability**. We adopted the original CUDA API with the host as much as possible, because the CUDA API includes features that are not publicly revealed. Also, if the compatibility with the existing API is ensured, then ConVGPU should be used without extra efforts.

We adopted UNIX socket to the ConVGPU to communicate between host and the program inside the container because the docker inherently blocks interprocess communications (IPCs) between host and containerized environment. There are other options we considered to communicate, like shared memory and plain file share. However, neither of them is safer than using UNIX socket, since a third party can intercept information when using other methods. We also consider conventional TCP/IP socket, but we did not choose it, because of its complexity and low performance compared to that of UNIX socket.

B. Customized NVIDIA Docker

In many cases, the container does not use whole GPU memory at a time. GPU memory can be shared if each container uses under some limitation. We customized NVIDIA Docker to set maximum GPU memory that the container can use. To handle this, we included `--nvndia-memory=<size>` custom option to the nvidia-docker. We designed the system to use `com.nvidia.memory.limit:<size>` label in docker image when the option input is absent, or to set 1 GiB as a default if both the option and the label are absent. This limitation is sent to the scheduler via the UNIX socket before the container is created. Next, we customized nvidia-docker to get the directory path from the scheduler. nvidia-docker links the container and the directory that contains CUDA wrapper API module and UNIX socket which is prepared for the container using `--volume` option. Furthermore, to make the CUDA wrapper API module to work on the container, we inserted an `--env` option to set the `LD_PRELOAD` Linux environment variable inside the nvidia-docker. This environment variable contains a list of libraries, which is loaded by the dynamic linker before any other libraries [21]. Our system set this environment variable to make the CUDA wrapper API module to be loaded before CUDA API module. To detect the stop state of the container, we made the nvidia-docker to add a dummy volume into the container which is connected to nvidia-docker-plugin. When the container exits its execution by any reasons, docker unmounts the volume; therefore, nvidia-docker-plugin can identify the container is exited. Subsequently, nvidia-docker-plugin can send a *close* signal to the scheduler for that container.

C. CUDA wrapper API module

A user program uses CUDA API to communicate and control NVIDIA GPU. These APIs perform multiple jobs including running kernel program and copying data from CPU to GPU. To manage and schedule containers and its GPU memory usage, API capturing module should be needed. We built CUDA wrapper API module (wrapper module) which is a Linux shared library [22]. This wrapper module whose filename is `libgpushare.so` covers a sub-part of original APIs, which mainly includes memory allocation or deallocation APIs. By adding the path to this shared library in LD_PRELOAD Linux environment variable, we can override original CUDA API and intercept API calls [21]. In comparison with the trials of previous works, we did not implement entire copies of CUDA API because wrapper module only overrides the function symbol name of some CUDA APIs and it leaves other CUDA API available. Using LD_PRELOAD also has the advantage to use internal CUDA APIs. Moreover, wrapper module can be adopted in other custom CUDA APIs such as rCUDA, because it can use the existing API without any effort. In addition, our wrapper module can cover both CUDA Driver API and Runtime API. Notice that the *nvcc* compiler links CUDA Runtime API statically inside the user program by default. In this case, overriding function symbol name using LD_PRELOAD does not work since the shared library is already inserted into the user program. To prevent wrapper module from the failure of intercepting Runtime API calls, user program should be compiled with `-cudart=shared` option.

The CUDA API which is captured by the wrapper module can be roughly categorized into two types: allocation APIs and deallocation API. Allocation APIs are used when user program tries to allocate GPU memory. At the moment when user program tries allocation, wrapper module only knows the requested memory size. This memory size information is sent to the scheduler and checked whether the size is available or not. Scheduler returns positive response when the required memory size is available and rejects if the memory is already exceeded. Wrapper module allocates memory using original CUDA API, only if the requested size of the memory is available. After the actual allocation, wrapper module sends the allocated memory address, current pid, and the size information to the scheduler to track the memory usage. Some allocation APIs which is used as a texture memory like `cudaMallocArray` are not captured, since they are not used in GPGPU.

Some memory allocation APIs may allocate increased size of memory which is different user program's first memory request. For example, `cudaMallocPitch` API allocates pitched memory to boost the access speed by multiple GPU threads. This pitched size varies among the GPU model. Consequently,

the wrapper module retrieves the pitched size of current GPU using `cudaGetDeviceProperties` API on the first call. Also, `cudaMallocManaged` API allocates memory size which is multiple of 128MiB since it uses mapped memory. For these cases, wrapper module calculates adjusted allocate size before checking available memory size.

When user program tries to deallocate GPU memory, wrapper module knows the address of the memory. In this case, wrapper module deallocates the memory using the original CUDA API and sends the address to the GPU memory scheduler. User program will get the result of deallocation from the wrapper module. In addition, when the user program is finished, `__cudaUnregisterFatBinary` CUDA API is invoked. This API unregisters CUDA fat binary from the GPU when exited. Wrapper module captures this API and sends the information to the GPU memory scheduler to deallocate the GPU memory used by the current process. The list of allocation APIs and deallocation API is summarized in Table II.

D. GPU memory scheduler

The container should be scheduled based on its maximum usable GPU memory. GPU memory scheduler is a standalone program written in Go programming language [23]. It runs on the host machine similar to nvidia-docker-plugin. GPU memory scheduler determines to accept, pause, or reject every GPU memory allocation from the containers. It creates UNIX socket for each container and communicates with the wrapper module which is inserted in the container. Before executing the container by the user, nvidia-docker requests the unique directory path for this container to the scheduler. Scheduler creates a directory to share the volume with the container, builds a UNIX socket inside the directory, and copies the wrapper module to the directory.

When the user program calls the memory allocation API, wrapper module sends the memory size information to the scheduler via the UNIX socket which is prepared for the container. Scheduler tracks every memory allocation call from the container; thus the scheduler knows how much free memory is allowed to this container. If memory size is sufficient to allocate, the scheduler sends a message to the wrapper module. After the actual allocation invokes with CUDA API by the wrapper module, it sends the allocated address with its memory size to the scheduler. Scheduler tracks this information using hash structure and calculates total memory usage. Each step is protected by a mutex lock to prevent the race condition.

TABLE II
LIST OF APIS COVERED BY WRAPPER MODULE

API name	Description
cudaMalloc	Memory allocation API in CUDA Runtime API, used for general purpose
cudaMallocManaged	Memory allocation with same address in CPU memory
cudaMallocPitch	Allocate pitched memory to fast access of multi-dimension array
cudaMalloc3D	Similar with cudaMallocPitch, but specialized in 3D array
cudaFree	Memory deallocation API in CUDA Runtime API
cudaMemGetInfo	API that retrieves current memory usage information
cudaGetDeviceProperties	Retrieves various informations of selected GPU device
__cudaUnregisterFatBinary	Unregister CUDA FAT binary (implicit API)

CUDA uses 64MiB of memory to store data related to current process and 2MiB to store CUDA context when the user program uses the CUDA API to allocate memory for the first time. Although detailed information about these memory allocations is not publicly available, and size varies among GPU driver version or type of GPU devices. Scheduler tracks the allocation information with its caller pid. Thus, our scheduler checks whether it is a first allocation from the given pid and calculates additional 66MiB of GPU memory of overall usage.

After the wrapper module detects the user program finishes its execution, it sends the information to the scheduler. Then, scheduler removes the memory allocation information from the pid. This feature is required because some program may not free its allocated GPU memory. Similarly, scheduler removes complete information from the container when it stops running, which is detected by nvidia-docker-plugin.

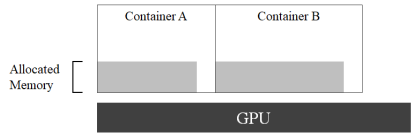
If the running container requires the valid size of GPU memory but the system has insufficient memory size, the response from the scheduler will be suspended until the required size of memory is available. There can exist multiple paused containers, and every memory allocation requested by these containers is suspended until the scheduler assigns more GPU memory to the container. When some container finishes its execution, the scheduler can assign its occupied GPU memory to other containers. In this paper, we deployed four scheduling algorithms to determine which containers should acquire additional memory. These algorithms are introduced as follows, and the performance evaluation of these algorithms are presented in Section IV.

- First-in, first-out (**FIFO**): The algorithm selects the oldest created container among paused containers. Then, it assigns available memory to the container until the assigned memory reaches the required memory size which the selected container requested at the creation time.
- Best-Fit (**BF**): The algorithm selects the container whose insufficient memory is closest, but not exceed to the remaining memory. If there is no such container, it chooses the container which has the least insufficient memory.
- Recent use (**RU**): The algorithm selects the most recently suspended containers. Then, it assigns available memory to the container until the assigned memory reaches the required memory size which the selected container requested at the creation time.

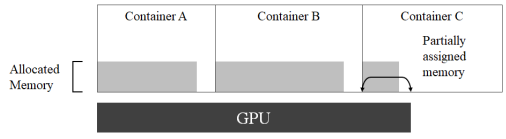
- Random **(Rand)**: The algorithm selects randomly among paused containers. Then, it assigns available memory to the container until the assigned memory reaches the required memory size which the selected container requested at the creation time.

E. Container scheduling example

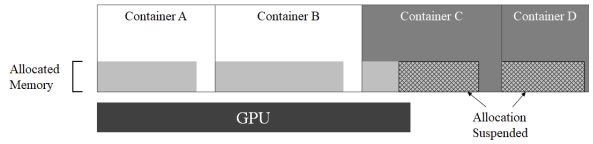
We provide a detailed scenario with multiple containers which use the same GPU within ConVGPU in Figure 3. As illustrated in Figure 3a, we assume that there are two containers (Container A and B) already running on the single GPU. When Container C starts execution, GPU memory is partially assigned to Container C which is requested at creation time. However, Container C is not suspended since it uses GPU memory within the assigned memory (see Figure 3b). As depicted in Figure 3c, Container C is suspended when it tries to allocate more GPU memory than physically assigned to itself. It is still a valid request since it is within the requested memory size from the container creation time. Container D, however, is immediately suspended because it does not have any GPU memory assigned to itself. After Container B terminates and returns its GPU memory (see Figure 3d), the scheduler selects Container C and guarantees all GPU memory which the container firstly requested. Now the Container C can resume its execution. Since GPU memory remains after the scheduler assigns to Container C, the scheduler allocates remaining memory to Container D. However, because it is insufficient for the Container D, the container remains suspended.



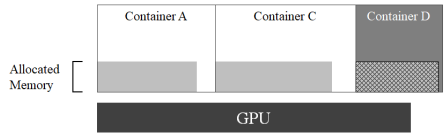
(a) Container A and B running on GPU



(b) Container C is assigned partial GPU memory which it requires



(c) Allocation request from container C and D are suspended



(d) Container C resumes, but not container D

Fig. 3. GPU memory assigns to multiple containers

IV. Experimental Evaluation

In this section, we present a set of experiments to evaluate the performance of the ConVGPU. First, we evaluated the performance of single container with our solution compared to that of without the solution, including the container creation time, API response time, and program running time. Second, we analyzed the performance of scheduling algorithms with multiple containers.

A. Experimental setup

The system we used to evaluate is composed of two Intel Xeon E5 CPUs, 64GB RAM, and one NVIDIA Tesla K20m GPU which has 5GB memory. For the GPU, we used the driver version 375.51 with CUDA version 8.0.44. Since our testing GPU supports Hyper-Q [24], it can run multiple GPU kernels concurrently up to 32 kernels. The system was operated with Ubuntu 14.04.5 LTS (kernel 4.2.0). We developed ConVGPU based on NVIDIA Docker version 1.0.0 RC 3 which the commit was released on GitHub [25] on December 17, 2016. Also, we used the docker version 1.12.3 for the container.

We wrote a test program to evaluate the performance of single container. The test program calls each CUDA API which we hooked with wrapper module. We omitted some APIs that operates the same function but different format with other APIs, like `cudaMalloc3D` and `cudaGetDeviceProperties`. The creation time of container and response time of CUDA API call is calculated using `clock_gettime` POSIX.1 function with `CLOCK_MONOTONIC` clock. A program running time is benchmarked using Convolutional Neural Network python script written with TensorFlow, which detects MNIST handwritten digit database. The actual code is obtained from TensorFlow Tutorial page [26]. All tests are repeated 10 times and the average value is used.

For the performance evaluation of scheduling algorithms, we classified the containers by the GPU memory size, similar to the T2 instance of Amazon Web Services (AWS) [27]. Table III shows the types of GPU memory size we used to evaluate with the corresponding T2 instance. We emulated the cloud usage by choosing the type of the containers randomly and running it every five seconds. Each container runs sample program, which allocates maximum GPU memory and the same size of CPU memory. This sample program copies dummy data from CPU memory to GPU, calculates the complement, and returns the result from GPU memory to CPU. The time consumed by the sample program varies by the size, from 5 seconds to 45 seconds. We changed the number of the containers from 4 to 38 and measured the finished time of

TABLE III
Evaluation Container types

Container type	nano	micro	small	medium	large	xlarge
Number of vCPU	1	1	1	2	2	4
Memory (GiB)	0.5	1	2	4	8	16
GPU memory (MiB)	128	256	512	1024	2048	4096

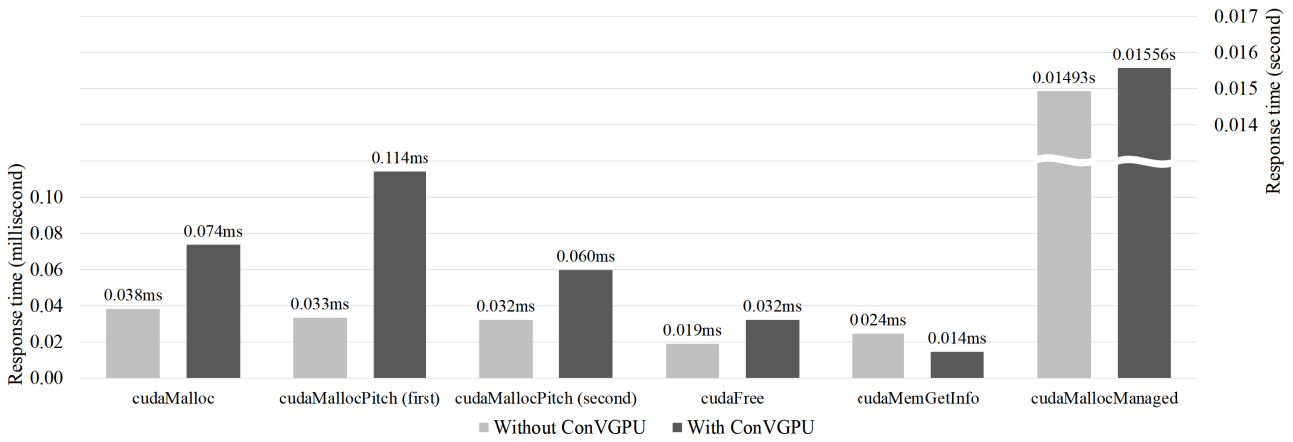


Fig. 4. Response time of the API call from the container

all containers and suspended time of each container. All tests are repeated 6 times and the average value is used.

B. Single container performance evaluation

Figure 4 shows the response time of six CUDA APIs which we mainly hooked using wrapper module. Response time to allocate APIs with the ConVGPU takes 0.082 milliseconds on average since the response time without the solution takes 0.035 milliseconds on average. In general, response time to allocate APIs with the solution takes twice more than it is without the solution. The main reason for this difference comes from the communication time between the scheduler and the wrapper module. However, `cudaMallocPitch` which called at the first time has around twice of a difference than that of other allocation APIs, as indicated in the figure. The reason of this is because wrapper module should retrieve the pitch size of current GPU at the first time. `cudaMallocManaged` API takes around 40 times more time than other APIs since this API allocates both CPU and GPU memory with the same memory address. CUDA use mapped memory to achieve this feature, which results in longer response time than the other APIs [28]. Contrary, response time of `cudaFree` API with ConVGPU takes only 0.032 milliseconds since the computation time in GPU is much less than the others. The response time of `cudaMemGetInfo` with ConVGPU is 0.01 millisecond faster than that of without ConVGPU. This is because the ConVGPU marks every memory information from the container, so the ConVGPU already knows the return value of the API without using the original CUDA API.

Container creation time is also evaluated as shown in Figure 5. According to the figure, creation time with the ConVGPU is around 15% (0.0618 seconds) longer than without the solution since the computation time which scheduler checks and assigns GPU memory to the container is considered.

Although the response time of API call with the solution is around two times longer than without solution, it barely affects the overall runtime of the user program. In Figure 6, the runtime of the TensorFlow MNIST program with ConVGPU takes 404.93 seconds, which is only increased 0.7% more than that of without ConVGPU. It is because the user program most spends its time copying data from/to the CPU memory and running GPU kernel code. These jobs can take a minute, an hour, or even more depending on the program.

C. Scheduling algorithms performance evaluation

Figure 7 depicts the finished time of given containers among the four scheduling algorithms. As the number of the containers is doubled, finished time is also roughly increased to double, although the container size is chosen randomly. The four algorithms show similar performance when the number of containers is less than 16. However, the Best-Fit algorithm

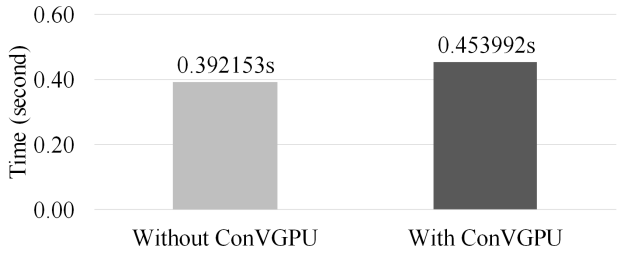


Fig. 5. Creation time of the container

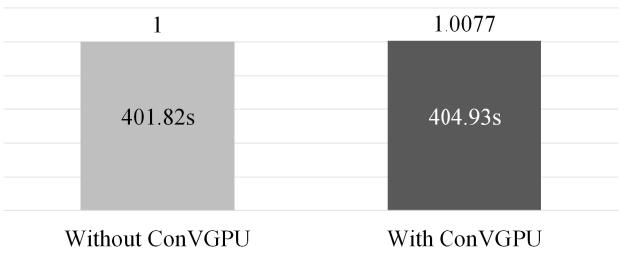


Fig. 6. Overall runtime of TensorFlow MNIST program

TABLE IV
FINISHED TIME OF GIVEN NUMBER OF CONTAINERS WITH THE FOUR ALGORITHMS

	Number of Containers																	
	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
FIFO (sec)	67.6	134.1	111.9	159.3	103.6	155.4	198.4	339.8	262.4	266.6	242.8	362.1	455.8	414.1	423.1	566.4	507.9	593.8
BF (sec)	68.2	134.0	110.9	158.8	104.0	159.5	195.9	318.4	281.1	257.2	241.8	345.9	426.0	368.0	405.2	537.5	501.0	588.7
RU (sec)	68.4	134.5	112.5	159.0	106.5	157.7	197.9	342.1	271.3	267.8	253.5	386.4	453.6	403.7	436.4	555.8	526.2	591.0
Rand (sec)	69.1	133.9	111.8	159.1	103.6	155.2	196.7	344.8	277.2	275.8	251.8	386.7	452.7	393.7	432.1	553.4	529.9	620.4

TABLE V
AVERAGE SUSPENDED TIME OF GIVEN NUMBER OF CONTAINERS WITH THE FOUR ALGORITHMS

	Number of Containers																	
	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
FIFO (sec)	2.4	33.7	35.7	27.2	19.2	38.1	45.6	90.4	72.2	77.6	58.1	72.6	135.8	132.5	115.3	162.5	157.1	182.7
BF (sec)	2.4	36.2	21.7	34.3	17.5	44.4	44.9	113.3	84.8	80.4	57.3	88.5	175.7	150.1	131.3	205.3	181.6	289.4
RU (sec)	2.4	35.6	31.2	28.1	20.0	37.7	51.3	88.9	76.6	75.2	56.7	80.9	117.4	122.9	95.3	170.9	154.6	182.6
Rand (sec)	2.4	32.0	23.6	27.3	19.4	37.5	44.0	76.5	75.5	78.4	60.5	67.9	129.4	127.5	118.1	165.3	175.8	174.2

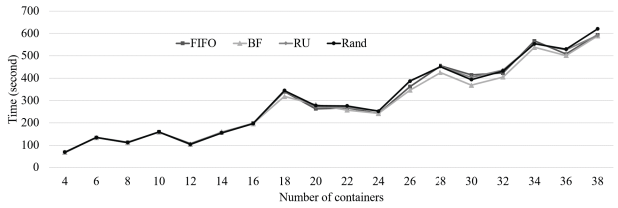


Fig. 7. Finished time comparison with the four algorithms

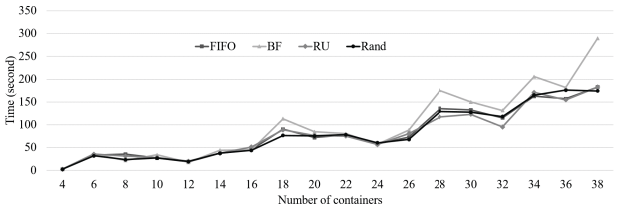


Fig. 8. Average suspended time comparison with the four algorithms

is average 30 seconds faster than other algorithms when the number of containers exceeds 18 (see Table IV). The reason for the higher performance of the Best-Fit algorithm is that it maximizes the GPU memory throughput. In most cases, the Random algorithm performs worst considering the error.

Figure 8 illustrates the average suspended time of each container when using the four scheduling algorithms. There is almost no difference between four algorithms when the number of the container is less than 24. However, when more than 26 containers are required to run, Best-Fit algorithm spends average 15 seconds more time until to start the container than the other algorithms (see Table V). By the time when the container finishes its execution, the Best-Fit algorithm finds the container which has the closest size with currently finishing container. Hense, starving may occur if there is no same size matched among the running containers.

We can conclude from the observation of Figure 7 and 8 that the Best-Fit algorithm is fastest for the overall task but needs more waiting time for each container in average. This tendency is prominent when the workload is heavier. It can be observed when the number of the container is 18, which takes 1.5 times more time compared to the case when the number of it is 16 (see Table IV), and the Best-Fit algorithm needs 32% more average suspended time in the same cases (see Table V).

V. CONCLUSIONS AND FUTURE STUDY

Virtualizing GPU using a software method has not fit in the container based virtualized environment. The main reason of previous failures comes from the trials of virtualizing the whole GPU. In this paper, we proposed the ConVGPU, containerizing GPU resources for multiple containers. The ConVGPU isolates the GPU memory, allows the container to use only a fraction of GPU memory. Experimental measurements demonstrated that the ConVGPU has a competitive performance compare to the performance without the ConVGPU. Also, experimental proved the stability of the proposed system when multiple containers run simultaneously with four scheduling algorithms.

There are sufficient features to be improved in the ConVGPU. Our future work will extend the ConVGPU in a multiple GPU with an appropriate algorithm to achieve better performance. Our further step is to adopt the ConVGPU in the clustering system like Docker Swarm [29].

ACKNOWLEDGMENT

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. 2015-0-00215 , Development of agro-livestock cloud and application service for balanced production, transparent distribution and safe consumption based on GS1) and International Research & Development Program of the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT&Future Planning of Korea(2016K1A3A7A03952054).

REFERENCES

[1] "CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit, accessed: 2017-03-21.

[2] "The open standard for parallel programming of heterogeneous systems," https://www.khronos.org/opencl/, accessed: 2017-03-21.

[3] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[4] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. ACM, 2009, pp. 17–24.

[5] 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.

[6] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2012.

[7] 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.

[8] A. Herrera, "NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation," *Tech. Rep.*, 2014.

[9] "GPU-Enabled Docker Container," http://www.nvidia.com/object/docker-container.html, accessed: 2017-03-21.

[10] D. Kang, T. J. Jun, and D. Kim, "Fault-tolerant Scheduler for Shareable Virtualized GPU Resource," in *Poster at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*, 2016.

[11] J. Monsalve, A. Landwehr, and M. Taufer, "Dynamic CPU Resource Allocation in Containerized Cloud Environments," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 535–536.

[12] S. McDaniel, S. Herbein, and M. Taufer, "A two-tiered approach to I/O quality of service in Docker containers," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 490–491.

[13] A. Dusia, Y. Yang, and M. Taufer, "Network quality of service in docker containers," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 527–528.

[14] D. Kirk *et al.*, "NVIDIA Cuda Software and Gpu Parallel Computing Architecture," in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM '07, vol. 7. ACM, 2007, pp. 103–104.

[15] "What is OpenStack?" https://www.openstack.org/software/, accessed: 2017-05-17.

[16] T. J. Jun, M. H. Yoo, D. Kim, K. T. Cho, S. Y. Lee, and K. Yeun, "HPC Supported Mission-Critical Cloud Architecture," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 223–232.

[17] "What is Docker?" https://www.docker.com/what-docker, accessed: 2017-05-04.

[18] S. Hykes, "Docker 0.9: Introducing Execution Drivers and Libcontainer," http://blog.docker.io/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/, 2014, accessed: 2017-05-17.

[19] J. Calmels, "nvidia docker," https://github.com/NVIDIA/nvidia-docker/wiki/nvidia-docker, accessed: 2017-03-28.

[20] "Use Docker Engine plugins," https://docs.docker.com/engine/extend/legacy_plugins/, accessed: 2017-03-30.

[21] K. Pulo, "Fun with LD_PRELOAD," in *linux. conf. at*, 2009.

[22] "Shared Libraries," http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html, accessed: 2017-04-03.

[23] R. Pike, "The go programming language," *Talk given at Googles Tech Talks*, 2009.

[24] T. Bradley, "Hyper-Q example," *NVidia Corporation. Whitepaper v1. 1*, 2013.

[25] https://github.com/NVIDIA/nvidia-docker, Official GitHub repository of NVIDIA Docker. Accessed: 2017-05-10.

[26] "A Guide to TF Layers: Building a Convolutional Neural Network," https://www.tensorflow.org/tutorials/layers, accessed: 2017-05-12.

[27] "Amazon EC2 Instance Types," https://aws.amazon.com/ec2/instance-types/, accessed: 2017-05-10.

[28] D. Negrut, R. Serban, A. Li, and A. Seidl, "Unified Memory in CUDA 6.0. A Brief Overview of Related Data Access and Transfer Issues," *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09*, 2014.

[29] "Swarm: a Docker-native clustering system," https://github.com/docker/swarm, accessed: 2017-05-10.