Fault-tolerant Scheduler for Shareable Virtualized GPU Resource

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

Abstract—Recently container-based virtualization is variously used to maximize utilization of computer resource, along with traditional Virtual Machine. However, different from traditional resources, GPU was hard to be shared by multiple containers. Lately, a GPU can be shared by multiple containers using volume share feature. In addition, high-end GPU like NVIDIA K20 supports Hyper-Q which allows multiple CPU processes to access a single GPU. Although, there still are problems exist because of GPU’s distinctive characteristics. Unlike system memory, GPU memory cannot be swappable. Also, GPU kernels in single Streaming Microprocessor cannot be switched during its running. These restrictions make hard to share GPU by multiple containers, and may result in deadlock situation. In this paper, we propose an interface for new fault-tolerant scheduler considering GPU memory usage. We have implemented this interface to restrict the usage of GPU memory for each container to prevent deadlock situation.

I. INTRODUCTION

Recently, container-based virtualization [1] is used as an alternative of Virtual Machine (VM). Container-based virtualization method like Docker makes container as a minimum execution unit. Container brings benefit for faster execution of applications compared to VM because it doesn’t virtualize operating system itself. The container shares kernel with host system which makes the system lighter. With container-based virtualization products, one machine can contain increased number of virtualized environment than VM.

Typically, container-based virtualization solutions use cgroups to control its computing resources that is contained in Linux kernel, although there are some exceptions such as disk usage in Docker. Docker uses Union file systems (UFS) [2] to control its file system which contains changed history of the file. But there is no GPU controller which results in container-based virtualization cannot manage container’s GPU memory usage. Currently, only NVIDIA GRID supports multiple VMs to share a single GPU. However, it lacks of diversity of virtualization solutions since it only supports Citrix XenServer [3] and certain type of GPUs. Therefore, general solution for sharing a single GPU by multiple containers or VMs does not exists.

Some GPU allows to run multiple kernels simultaneously by different CPU processes. For this kind of GPU, it is possible sharing a GPU by multiple applications concurrently. In addition, sharing host’s volume of GPU device and driver makes multiple containers to share a single GPU without driver installation in every containers. However, there is no GPU resource controller exists in Linux kernel. This may occur several problems, including GPU memory deadlock situation.

In this paper, we propose an interface to restrict GPU memory allocation of each container. To make a fault-tolerant container scheduler which considering GPU memory usage, we implemented custom CUDA API interface which modifies original runtime API as a primary research.

II. DEADLOCK SITUATION WHEN SHARING GPU

Neglecting a GPU memory from access of several processes can results to serious problems including GPU memory deadlock problem. Imagine that there are two containers sharing a single GPU which has 5GiB of GPU memory. First, each container requests for 2GiB GPU memory. After running GPU kernel several times, each container requests additional 2GiB GPU memory. However, at this time GPU has only 1GiB of memory left, therefore these requests will failed. In each user’s view, total amount of GPU memory was 5GiB, assuming to use the GPU memory up to 5GiB. If these two containers wait for each other, no container can allocate additional GPU memory. Since GPU memory cannot be swapped, this raises deadlock problem.

Moreover, this deadlock situation can be occurred by malicious container. If some containers allocate all of 5GiB GPU memory and never release it, no other container could allocate GPU memory. One of solution to solve this problem is detection and deletion this greedy containers, although this
cannot fundamentally prevents deadlock. Therefore, restriction of each container’s allocable GPU memory is mandatory.

III. CONSTRAIN MAXIMUM VALUE OF GPU USAGE

Generally, using GPU as general purpose calls the API functions provided by the library to control GPU. For example, CUDA, which is widely used in both research and industry, provides CUDA Runtime API. When application tries to use GPU as general purpose, they require to use this runtime API to allocate GPU memory. Our interface can be applied when using CUDA and hooks the application APIs that tries to use CUDA Runtime API. Currently, we considered to implement based on dynamic library since predefined Linux environment variable LD_PRELOAD forces dynamic linker to load custom API prior to the original one.

Our custom interface which relies on CUDA captures information from the container and checks if current application could allocate requested amount of GPU memory. If the request is validated, the interface passes the CUDA Runtime API. In case of invalid request, the interface returns erroneous value to the caller so that the application can make decisions for further actions.

IV. EVALUATION

We implemented our custom interface in Linux (Ubuntu 14.04) environment with NVIDIA Kepler K20m GPU which supports Hyper-Q with 5GiB of GPU Memory. For the evaluation, we made a test application which requests 128MiB of GPU memory on every 10 seconds, until the memory allocation failed. We implemented this application into each container and executed simultaneously.

Without our custom interface, GPU memory allocation succeeded until 2GiB for each container. The test application can request additional GPU memory since there is no boundary for the memory allocation. In contrast, using our interface with 1GiB of GPU memory boundary, test application can allocate less than 1GiB of GPU memory. This results permit to allocate two additional containers in the future, without deadlock.

V. DISCUSSION

Management of GPU memory should be different from that of system memory. Unlike system memory, GPU memory cannot be swapped since swapping GPU memory is totally entrusted to the developer for faster functionality of GPU. Once deadlock occurs, the only solution is killing one of the containers. Result of the killing these containers is loss of data, and its cost is countless since deadlock could occur at anytime, even after a month from the container starts. Our definitive goal is to prevent deadlock situation and provides sharing GPU memory. Since our custom interface recognizes each container’s GPU memory usage, we can schedule the containers based on this information.

Our next step to implement scheduler on Docker based cloud solution such as Kubernetes. Main purpose of scheduling will be assigning containers to single GPU and schedule them to prevent the deadlock situation. Furthermore, our interface will hide un-allocated GPUs from the container and only shows assigned GPUs.

VI. CONCLUSION

In this paper, we proposed the possible problems when sharing a GPU as a general purpose to multiple containers. Because GPU memory can not be swapped, allocation of GPU memory for the containers should be strictly controlled, otherwise it will result deadlock situation. As result, we concluded that a new scheduler needs to orchestrate container’s GPU memory usage. Therefore, we implemented the custom interface to control allocation GPU memory as primary work for further research.

ACKNOWLEDGMENT

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2016-H8601-16-1007) supervised by the IITP (Institute for Information & Communications Technology Promotion).

REFERENCES