2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications

GaiaGPU: Sharing GPUs in Container Clouds

Jing Gu
School of Software and Microelectronics
Peking University
Beijing, China
gu.jing@pku.edu.cn

Shengbo Song
Data Platform
Tencent Inc.
Beijing, China
thomassong@tencent.com

Ying Li
National Engineering Center of Software Engineering
Peking University
Beijing, China
li.ying@pku.edu.cn

Hanmei Luo
Data Platform
Tencent Inc.
Beijing, China
mavisluo@tencent.com

***Abstract*— Containers are widely used in clouds due to their lightweight and scalability. GPUs have powerful parallel processing capabilities that are adopted to accelerate the execution of applications. In a cloud environment, containers may require one or more GPUs to fulfill the resource requirement of application execution, while on the other hand exclusive GPU resource of a container usually results in underutilized resource. Therefore, how to share GPUs among containers becomes an attractive problem to cloud providers. In this paper, we propose an approach, called GaiaGPU, to sharing GPU memory and computing resources among containers. GaiaGPU partitions physical GPUs into multiple virtual GPUs and assigns the virtual GPUs to containers as request. Elastic resource allocation and dynamic resource allocation are adopted to improve resource utilization. The experimental results show that GaiaGPU only causes 1.015% of overhead by average and it effectively allocates and isolates GPU resources among containers.**

Keywords—GPU virtualization, Kubernetes, Container.

I. INTRODUCTION

Operating System (OS) virtualization, also known as containerization, is a server virtualization technology that involves tailoring a standard operating system so that it can run different applications handled by multiple users on a physical server [1]. Each isolated user space instance is called a container. Unlike Virtual Machine (VM) which emulates the underlying hardware, a container emulates an operating system. Compared with VMs, containers have the advantages of lightweight, scalability and ease of deployment so that they have become the de facto standard for packaging and publishing applications and microservices in the cloud [2]. Several cloud providers (e.g., Amazon, Google, Tencent, Alibaba) integrate container orchestration frameworks such as Kubernetes [3] in their infrastructure to support container clouds.

Graphics Processing Units (GPUs) have strong parallel processing capabilities because they integrate thousands of compute cores on a chip. Therefore, GPUs are widely used in compute-intensive tasks such as image recognition, speech recognition, and natural language processing to speed up computation. CUDA is one of the most popular platforms for general-purpose GPU which provides APIs for easy to use. The significant performance benefits have attracted cloud providers to deploy GPUs in their cloud environments. Following the technology scaling trend, modern GPUs integrate more and more computing resources [4]. In a cloud environment, one application deployed in a container may require one or more GPUs for execution, while on the other hand exclusive GPU resources of an application results in resource underutilization. So how to share GPUs among different containers is of great interest to most cloud providers.

This work was supported by PKU-Tencent Joint Innovation Lab.

GPU virtualization techniques are used to share GPUs among isolated virtual environments such as VMs and containers. Most of existing GPU virtualization techniques are applied to VMs. GPU virtualization based on containers is still at an initial stage [5]. Current implementation of GPU virtualization based on containers have the following limitations: a) require specific hardware devices [6]; b) assign an entire GPU to a single container without sharing [7]; c) only share memory among containers [8]; d) only support a single GPU [8].

We propose an approach called GaiaGPU for sharing GPU memory and computing resources among multiple isolated containers transparently. Users do not need to modify container images for sharing underlying GPUs. In order to achieve this goal, we partitions physical GPUs into multiple virtual GPUs (vGPUs) based on the device plugin framework of Kubernetes. Each container can be assigned with one or more vGPUs as request. In addition, we provide two ways to change the container resources during runtime. One is elastic resource allocation which temporarily alters the container resources, and the other is dynamic resource allocation which permanently modifies the container resources.

In this paper, sharing GPU memory means that the entire GPU memory is divided for multiple containers and each container has a fraction of the GPU memory. Sharing computing resources means that each container owns a portion of the GPU's threads to perform computation in parallel. A vGPU is composed of GPU memory and computing resource. The memory of a vGPU is the allocated physical memory. The computing resources of a vGPU is measured by the GPU's utilization which is defined as the percent of time over the past sample period during which container was executed on the GPU [9].

In summary, this paper makes the following contributions:

- We propose GaiaGPU, an approach for sharing GPU memory and computing resources among different containers transparently.
- We adopt elastic resource allocation and dynamic resource allocation to improve the utilization of resources.
- We conduct four experiments to evaluate the performance of GaiaGPU. The experimental results reveal that our approach has low overhead (1.015%) and it effectively allocates and isolates the resources of vGPUs.

The rest of this paper is organized as follows: Section II presents up-to-date related work on GPU virtualization and the device plugin framework of Kubernetes. Section III describes the design and implementation of GaiaGPU in detail. Section IV evaluates the performance of GaiaGPU

978-1-7281-1141-4/18/$31.00 ©2018 IEEE
DOI 10.1109/BDCloud.2018.00077

and explains the experimental results. Finally, we conclude this paper in Section V.

II. Related Work

A. GPU Virtualization

GPU virtualization technology is applied to share GPUs across multiple virtual environments, which greatly improved performance of applications.

The majority of existing GPU virtualization techniques are based on VMs. In general, there are three fundamental types of virtualized GPUs: API remoting, para and full virtualization, and hardware-supported virtualization [5]. vCUDA [10] and rCUDA [11] use API remoting technology to implement GPU virtualization. They build a wrapper CUDA library in VMs to intercept GPU calls and redirect those calls to the host for execution. GPUvm [13] implements both para and full virtualization in Xen hypervisor. To isolate multiple VMs running on a physical GPU, GPUvm divides the physical GPU memory and MMIO regions into partitions and assigns each partition to a single VM. NVIDIA GRID [6] implements GPU virtualization at the hardware level. It creates virtual GPUs and assigns them to containers.

Compared to VMs, containers share the host OS instead of having an independent guest OS. Therefore, containers directly use the GPU driver on the host and achieve performance that is close to the performance of native environments. There is some research devoted to supporting GPUs in containers. NVIDIA GRID also can be used in container environment. However, it requires specific hardware devices and the same resource configuration of virtual GPUs. Each container only can be assigned with one virtual GPU. GaiaGPU virtualizes GPUs at software level without hardware limitation. Meanwhile, the resources of each virtual GPU can be different and a container can be assigned with more than one virtual GPUs. NVIDIA Docker [7] enables Docker images to leverage GPUs. It allows GPU driver agnostic CUDA images and provides a Docker command line wrapper that mounted the user mode components of the driver and the GPU device files into the container at launch. However, there is no sharing in NVIDIA Docker since it assigned an entire GPU to a container. ConvGPU [8] is a solution to share GPU memory with containers. It intercepts the CUDA library to manage the memory allocation/deallocation of each container. However, ConvGPU only supports sharing of memory resources and only virtualizes a single GPU. GaiaGPU can share both memory and computing resources of GPUs in a container cluster.

B. Device plugin

The device plugin framework is provided by Kubernetes for cloud vendors since in version 1.8.0 , which aims to advertise computing resources (e.g., GPUs, High-performance NICs, FPGAs) to the cluster without changing Kubernetes core code. The workflow of device plugin is divided into two phases, as shown in Fig. 1.

1) ***Resource Discovery***. First, each type of extended resource implements a Device plugin. A device plugin register itself with Kubelet [12] (i.e., node agent) through gRPC service. Following a successful registration, the device plugin sends the Kubelet the list of devices it manages. Finally, the Kubelet is in charge of advertising those extended resources to the Kubernetes Master.

2) Resource allocation. When a user request devices in a container specification, the scheduler in the master chooses a specific Kubernetes node to launch the container according to the required resources including the extended resources. Then the Kubelet in the chosen Kubernetes Node sends the device request to the device plugin. At last, the device plugin assigns the corresponding devices to the container.

Vaucher [2] implements SGX device virtualization by leveraging the device plugin framework, which modifies the Kubelet and SGX code to limit the usage of device memory of virtual SGX devices. However, this method only handles memory resources and imposes a hard limit on memory. GaiaGPU also adopts the device plugin framework to share resources among containers. However, we leverage elastic limit rather than hard limit on resources in order to improve utilization.

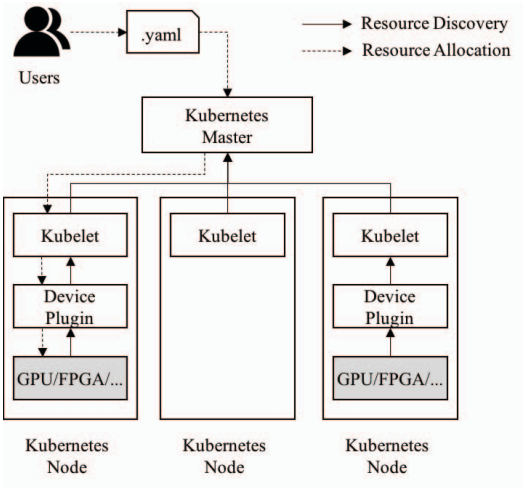

Fig. 1. The discovery and allocation of computing resources.

III. Design and Implementation

A. Design and Implementation

The target of GaiaGPU is to share both memory and computing resources among containers with minimum cost and maximum performance enhancement. There are some challenges in achieving this goal.

- **Transparency**. GaiaGPU should not modify the Kubernetes code or container images for sharing GPUs. An application is executed with sharing GPUs should be as if it was executed on physical GPUs.
- **Low overhead**. The performance of applications with shared GPUs should be as close as possible to the performance with physical ones.
- **Isolation.** GaiaGPU should manage GPU resource allocation/deallocation of each container and make the container is completely isolated from each other while sharing GPUs.

The overall architecture of GaiaGPU is shown in Fig. 2. The GaiaGPU consists of four components: GPU Manager, GPU Scheduler, vGPU Manager, and vGPU Library. In order to manage resources effectively, we leverage two-layer resource management. At the host level, the GPU Manager is in charge of creating vGPUs, and the GPU Scheduler is responsible for allocating physical GPU resources to vGPUs. At the container level, the vGPU Library component takes charge of managing GPU resources of a specific container.

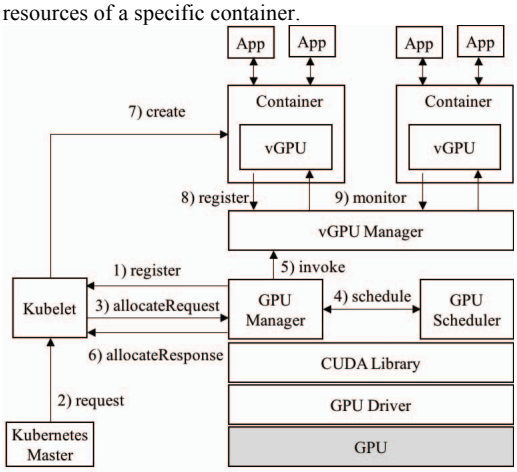

Fig. 2. The architecture of GaiaGPU

a) ***GPU Manager.*** We implement a device plugin called GPU Manager to advertise GPUs to the Kubelet. GPU Manager runs on the host and is responsible for creating vGPUs and communicating with the Kubelet through gRPC service. Fig. 3 shows the communication between GPU Manager and the Kubelet.

Register. GPU Manager registers itself with the Kubelet in order to inform the Kubelet of its existence.

ListAndWatch. After a successful registration, the Kubelet sends a *ListAndWatch* request to the GPU Manager for inquiring device information. GPU Manager returns a list of devices it manages to the Kubelet. Instead of physical GPUs, GaiaGPU sends a list of vGPUs to the Kubelet. GPUs are virtualized in two resource dimensions: memory and computing resources.

- Memory. We divide 256M memory as a unit and each memory unit is called a *vmemory* device.
- Computing resources. Since the device plugin framework does not support to allocate a fraction of GPU, we partition a physical GPU into 100 *vprocessor* devices and each *vprocessor* owns one percent of GPU utilization.

GPU Manager sends a list consisting of all *vmemory* and *vprocessor* devices to the Kubelet.

Allocate. When a user requires GPU devices in a container specification, the Kubelet arbitrarily selects the corresponding amount of devices from the device list sent by GPU Manager. Since there are virtual devices in the list, some extra steps are taken to map the virtual devices to physical ones.

An example is given to show the procedure of mapping. A container requires 50 *vprocessor* devices and 10 *vmemory* devices. First, the Kubelet sends the randomly selected device IDs to the GPU Manager. Second, the GPU Manager calculates the required physical GPU resources according to the received device IDs and sends a request to the GPU Scheduler. Finally, GPU Scheduler returns the physical GPUs assigned to the container.

After mapping, the GPU Manager returns an *allocateResponse* that contains configurations for accessing the allocated devices. These configurations are classified into three categories: 1) environment variables of the container, 2) the directories or files mounted into the container (e.g., NVIDIA Driver, CUDA libraries), 3) the assigned devices. The Kubelet passes these configurations to container runtime.

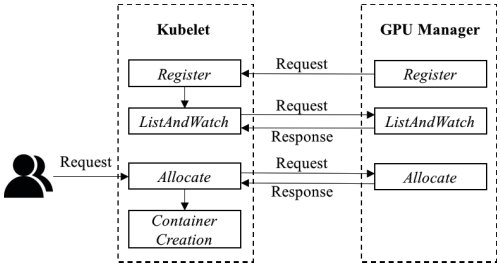

Fig. 3. The communication between GPU Manager and Kubelet

b) ***GPU Scheduler.*** The GPU Scheduler is responsible for serving scheduling requests sent by the GPU Manager. If the scheduling succeeds, the GPU Scheduler returns a response that contains allocated GPU information. The GPU Scheduler allocates GPUs based on topology. The GPU topology is a tree topology. The root node of the tree is a physical host and the leaf nodes are physical GPUs. When the required resouces are less than one physical GPU, the allocation strategy aims to minimize resource fragements in the tree. When the required resources equal one physical GPU, the allocation strategy of minimizing single leaf nodes (i.e., a leaf node without sibling nodes) is adopted. When the required resources are more than one physical GPU, the allocation strategy targets on minimizing the communication cost across GPUs. The communication cost is determined by the connection mode of two GPUs.

c) ***vGPU Manager***. The vGPU Manager running on the host delivers container configuration and monitors containers assigned with vGPUs. When a container applies for GPU resources, the GPU Manager sends the container's configuration such as the required GPU resources, the name of the container to the vGPU Manager. Following receiving the container's configuration, the vGPU Manager creates an unique directory for the container on the host. The directory is named after the container name and is included in the *allocateResponse*. The configuration of the container is also in this directory so that it can be delivered to the container by the Kubelet.

The vGPU Manager communicates with the vGPU Library in a server-client mode. The vGPU Manager maintains a list of containers which are alive and are assigned with GPUs and periodically checks if these containers are still alive. If a container is dead, the vGPU

Manage will remove the container from the list and delete the corresponding container directory.

d) ***vGPU Library.*** The vGPU Library running in the container is used to manage the GPU resources of its deployed container. The vGPU Library will be launched when the first GPU application is executed in the container. The vGPU Library registers itself with the vGPU Manager after booting. It intercepts the memory-related APIs and the computing-related APIs in the CUDA Library by the LD_LIBRARY_PATH mechanism. LD_LIBRARY_PATH is an environment variable for Linux systems that affects the runtime link of programs, which allows some directories to be loaded before the standard set of directories. TABLE I. shows the all intercepted CUDA APIs.

When a container consumes more GPU resources than its request, measures are taken to prevent exhausting the entire GPU. There two types of resource limits. One is hard limit which means resources will not be allocated when the container consumes resources exceeding its request. The other is elastic limit which refers that when the resources used by the container exceeds its request and there are free resources in the system, the container still can get resources.

For memory resources, the hard limit is adopted. There are some reasons to explain why impose the hard limit on memory. 1) The size of memory determines whether an application can run, while it has little impact on the performance of the application. 2) GPUs are computing devices and they adopt one-time resource allocation strategy. An application can only be executed after acquiring all memory resources and will not release the memory until it is completed. If using elastic memory allocation, applications with large memory requirement will be starved. 3) The only way to take back the over-allocated memory resources is to kill the process by throwing an out-of-memory exception. We also provide dynamic resource allocation which can add GPU memory of a container during runtime. Section B depicts the details.

The elastic limit is leveraged for computing resources because they have a great impact on application performance. Unlike memory, computing resources (threads) are released immediately after execution instead of waiting until the application is completed. Elastic resource allocation is a way to achieve elastic limit. The details of elastic resource allocation are shown in Section B.

To summarize, the workflow of GaiaGPU is as follows.

Step 1: GPU Manager registers itself with the Kubelet and advertise vGPUs.

Step 2: The Kubelet receives a container request sent by the master which requires GPU devices.

Step 3: The Kubelet sends an *allocateRequest* to the GPU Manager.

Step 4: The GPU Manager sends a scheduling request to the GPU scheduler. Then the GPU scheduler allocates physical GPUs according to the scheduling strategy. If the scheduling succeeds, it returns a response that contains information for the allocated GPUs.

Step 5: The GPU Manager sends the configuration of the container to the vGPU Manager.

Step 6: The GPU Manager returns the container's environment variables, mounting information, and device information to the Kubelet.

Step 7: The Kubelet creates and initializes a container according to the *allocateResponse*.

Step 8: The vGPU Library registers itself with the vGPU Manager and manages the GPU resources of its deployed container.

Step 9: The vGPU Manager monitors the running containers assigned with GPUs and cleans up these containers when they are dead.

TABLE I. THE INTERCEPTED CUDA DRIVER APIS

	CUDA Driver API	**Description**
Memory-related APIs	cuMemAlloc	Allocates device memory.
	cuMemAllocManaged	Allocates memory that will be automatically managed by the Unified Memory system.
	cuMemAllocPitch	Allocates pitched device memory.
	cuArrayCreate	Creates a 1D or 2D CUDA array.
	cuArray3DCreate	Creates a 3D CUDA array.
	cuMipmappedArrayCreate	Creates a CUDA mipmapped array.
Computing resources-related APIs	cuLaunch	Launches a CUDA function.
	cuLaunchKernel	Launches a CUDA function.
	cuLaunchCooperativeKernel	Launches a CUDA function where threads blocks can cooperate and synchronize as they execute.
	cuLaunchGrid	Launches a CUDA function.
Device info-related APIs	cuDeviceTotalMem	Returns the total amount of memory on the device.
	cuMemGetInfo	Gets free and total memory.

Algorithm 1: Elastic Resource Allocation

Input: The max utilization of GPU, GU_{max}; Used utilization of $vGPU_i$, $vGU_i(i = 1,2,\cdots,n)$; The container utilization in container config, CU_{config}; Used container utilization, CU_{used}.
Output: Updated container cores, CU_{cores}.

1. **while** True **do**
2. *nanosleep()*
3. $GU_{free} = GU_{max} - \sum_{i=1}^{n} vGU_i$
4. **if** $GU_{free} > 0$ ***or*** $CU_{used} < CU_{config}$ **then**
5. flag ← 1
6. **else**
7. flag ← −1
8. **end if**
9. CU_{cores} += flag × CU_{config}/100
10. **end while**

B. Optimization

The resources of a container not only affect performance of an application, but even determine the status of the application. However, users cannot accurately estimate the required resources when creating a container. Hence, we provide two ways to change the container's resources during runtime. Elastic resource allocation temporarily modifies the computing resource limit of the container, while dynamic resource allocation permanently alters the container's resources.

Elastic Resource Allocation. The purpose of elastic resource allocation is to make full use of free computing resources to improve resource utilization. As shown in Algorithm 1, *nanosleep()* is a Linux kernel function that suspends the execution of the calling thread until either at least a specified time has elapsed or the delivery of a signal invocates the calling thread (line 2).

We use the GPU utilization of a process to measure the amount of computing resources it uses. GU_{free} refers to the free resource of a physical GPU, which equals that the max utilization of GPU subtracts the total utilization of processes running on the same physical GPU (line 3). Since the vGPU Library monitors the GPU utilization periodically, it is necessary to reserve some free resources for the system in order to prevent exhausting GPU resources at next time interval. The max utilization of GPU (GU_{max}) is set as a parameter and the default value is 90%.

If a physical GPU has free resources ($GU_{free}>0$), even if GPU resources consumed by the container exceeds its requirement, the vGPU Library still allocates computing resources for the container (line 4-5). If the system has no remaining resources ($GU_{free}<=0$) and the container consumes more resources than its requirement, the vGPU Library will take back the over-allocated resources (line 6-7).

A non-preemptive strategy is adopted to recycle over-allocated resources, which means that containers occupy resources until the completion of the kernel. CU_{cores} is similar to tokens which is consumed when a container executes a kernel function and is produced when a kernel function is finished. The initial value of CU_{cores} equals the required computing resources of the container. When CU_{cores} is zero, the vGPU Library will not allocate any computing resources for the container until CU_{cores} is greater than 0.

We take Fig. 4 as an example to illustrate the process of elastic resource allocation. First, the Container A requests 0.3 GPUs and is scheduled to an idle physical GPU. Following successful scheduling, container consumes the resources of the GPU. Since only the Container A is running on the GPU, it can consume the GPU resources until the utilization reaches the max value. Then the Container B with 0.7 GPUs is scheduled to the same physical GPU. However, the GPU has no free resources. Therefore, it is necessary to take back the resources that are over-allocated to the Container A and to re-allocate these resources to the Container B. Repeat the process of recycling and allocating resources until the GPU utilization of each container is no more than its requirement.

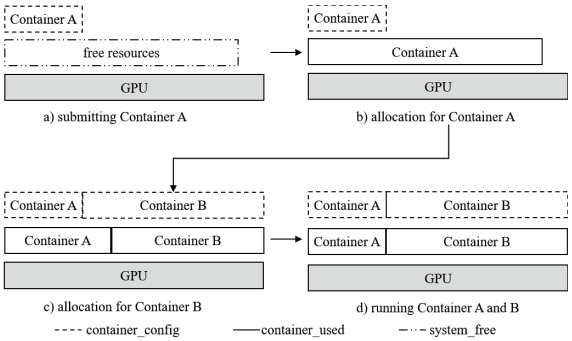

Fig. 4. An example for elastic resource allocation

Dynamic Resource Allocation. Dynamic resource allocation modifies the container resources, including memory and computing resources without stopping the container. Dynamic resource allocation is designed to solve two problems. The first problem is to update the memory and computing resources of the container under the hard limit. The second problem is to add memory resources to the container under the elastic limit. The vGPU Library component limits container resources by comparing the container's resource configuration with the container's actual utilization. To change the container's resource permanently, we only need to modify the resource configuration of the container and notify the GPU Scheduler to update the corresponding physical GPU allocation.

IV. EXPERIMENTS

To illustrate the effectiveness of GaiaGPU, we conduct four experiments to evaluate the performance of GaiaGPU in terms of overhead, GPU partitioning, resource isolation and elastic resource allocation. The first three experiments adopt the hard limit on resources while the elastic resource allocation experiment adopts the elastic limit. All experiments are deployed on Tencent container cloud GaiaStack [19] which uses Kubernetes version 1.9 as the container orchestrator.

TABLE II. HADRWARE CONFIGURATION OF THE CLUSER

Kubernetes Master	
CPU	12 Intel(R) Xeon(R) CPU E5-26xx v3
RAM	24 GB
Kubernetes Node	
CPU	56 Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
RAM	128 GB
GPU	Nvidia Tesla P4
GPU Memory	8 GB

We build a cluster which is composed of five hosts. Three hosts act as the Kubernetes masters and the other two hosts act as the Kubernetes nodes. Detailed information of the hardware can be found in TABLE II. Each Kubernetes node has two physical GPUs. For each GPU, we use the driver version 384.66, CUDA version 9.0.0, CUDNN version 7.0.4. Docker version 1.9.1 is leveraged to provide containers.

A. Overhead

In order to identify the overhead of sharing GPUs incurred by GaiaGPU, we evaluate the performance of GPU applications deployed in a container and the host. Most GPU applications are built with deep learning frameworks because they provide an easy way to construct deep networks. However, different deep learning frameworks have different libraries and resource allocation strategies, which leads to different overhead when sharing GPUs among containers. So we selected five popular deep learning frameworks for evaluation, including Tensorflow [14] , MXNet [15], Caffe [16], PyTorch [17], CNTK [18]. We run the MNIST application on five deep learning frameworks in both the host and the container, and measure its execution time. The MNIST application is a program that detects handwritten digits with database MNIST [20] using Convolutional Neural Network. We choose this application for fairness as all five frameworks provide it in official examples. The results of the experiments are shown in Table 2. *Native_time* in the table refers to the execution time of the MNIST application running on the host while the *GaiaGPU_time* refers to the execution time of the application running in a container with shared GPUs. *Difference* refers to the difference between the *Native_time* and the *GaiaGPU_time*, which is calculated by formula 1.

$$\text{Difference} = \frac{\text{GaiaGPU\_time - Native\_time}}{\text{Native\_time}} \times 100\% \quad (1)$$

All tests are repeated 10 times and the average value is used. As shown in the TABLE III. , the overhead to Tensorflow, Caffe, PyTorch and CNTK is less than 1%, and the average overhead to five frameworks is 1.015%. Experimental results show that the overhead of GaiaGPU is quite low and it can support performance to be achieved as native environment.

TABLE III. EXECUTION TIME OF NATIVE AND

	Native_time (seconds)	*GaiaGPU_time (seconds)*	*Difference (%)*
Tensorflow	47.82	47.88	0.13
MXNet	26.17	26.97	3.07
Caffe	22.47	22.50	0.15
PyTorch	69.33	69.64	0.44
CNTK	7.39	7.41	0.27

B. GPU partitioning

GaiaGPU partitions memory and computing resources of GPUs and assigns them to vGPUs. To evaluate the impact of GPU partitioning on application performance, we partitions a physical GPU into 1, 2, 3, 4, 5, and 8 vGPUs and assigns each vGPU to a container. Then run three GPU applications (MNIST [20], Cifar10 [21] , AlexNet [22]) in each container and measure their execution time. These applications are typical deep learning applications, both computing intensive and memory intensive, and the code can be obtained from the Github [23]. Tensorflow framework is used to run these applications. In order to evaluate the impact of partitioning of memory and computing resources on performance respectively, we conducted two experiments.

The first experiment evaluates the impact of partitioning of computing resources on application performance. The vGPUs configurations are shown in TABLE IV. , and the results are shown in Fig. 5. The x-axis is the number of vGPUs and the y-axis represents the execution time of the application. Although a physical GPU has 8GB memory, the actual memory is less than 8GB which leads that 8 vGPUs with 1GB memory cannot be launched at a time. As shown in the figure, the execution time is linear to the computing resources of the vGPU.

TABLE IV. CONFIGURATIONS OF VGPUS

Number of vGPUs	1	2	3	4	5
Computing resource per vGPU	1	0.5	0.3	0.25	0.2
Memory per vGPU (GB)	1	1	1	1	1

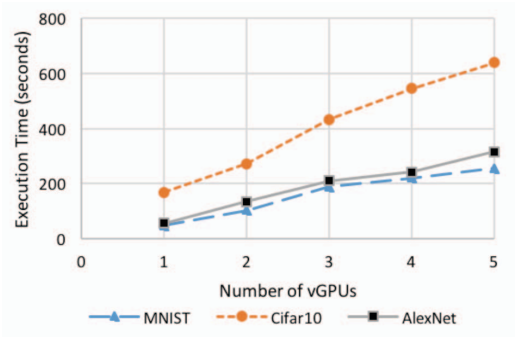

Fig. 5. The performance of applications under different partitioning of computing resources

The second experiment is devised to evaluate the impact of partitioning of memory on application performance. The vGPUs configurations are shown in TABLE V. , and the results are depicted in Fig. 6. It should be noted that the AlexNet application cannot be executed with 0.9GB memory so there is no data in the figure. As shown in the figure, the change of the vGPU's memory has little effect on the execution time of applications.

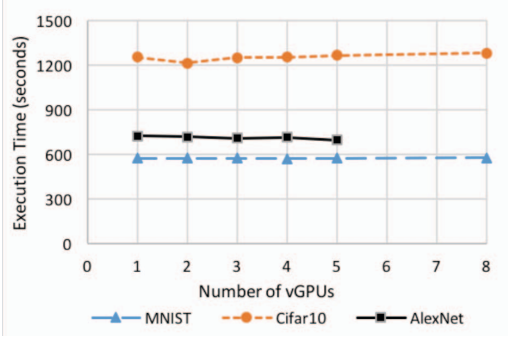

Fig. 6. The performance of applications under different partitioning of GPU memory

TABLE V. CONFIGURATIONS OF VGPUS

Number of vGPUs	1	2	3	4	5	8
Computing resource per vGPU	0.1	0.1	0.1	0.1	0.1	0.1
Memory per vGPU (GB)	7.2	3.6	2.4	1.8	1.44	0.9

C. Isolation

Resource isolation means that the allocation of resources to one job/container should not have any unintended impacts on other job. To demonstrate whether GaiaGPU achieve resource isolation among containers, we launch 2, 4, and 8 containers on a physical GPU and measure the memory utilization and GPU utilization of containers. Each container is assigned with a vGPU. The GPU workload is the MNIST application using Tensorflow framework. TABLE VI. shows the configurations of vGPUs and Fig. 7 illustrates the results.

In the figure, the x-axis represents the elapsed time and the y-axis is the GPU utilization over 120 seconds. Since Tensorflow adopts the one-time resource allocation strategy, the memory reaches the required value in a very short time (less than 2 seconds). As shown in the figure, the utilization of GPU are fluctuated over time because the GPU is a non-preemptive device. Fig 7.a illustrates that two containers are running on a physical GPU. The average GPU utilization of one container is 36.43% and the other is 36.08%. The maximum GPU utilization of a single container is 68% because the vGPU Library component monitors the resource utilization of each process running on the GPU at a given time interval, and the process can use more resources than its request between monitoring. As shown in Fig.7 b, there are four containers running on a physical GPU and their GPU utilization are 16.4%, 16.7%, 17.0%, and 17.2%, respectively. When a GPU is shared by 8 containers (Fig.7 c), the average GPU utilization of containers is 7.8% and the difference of GPU utilization among containers is less than 1%. The experimental results reveal that GaiaGPU effectively isolates the GPU resources of each container when sharing GPUs among containers.

TABLE VI. CONFIGURATIONS OF VGPUS

Number of vGPUs	2	4	8
Computing resource per vGPU	0.5	0.25	0.1
Memory per vGPU (GB)	3.6	1.8	0.9

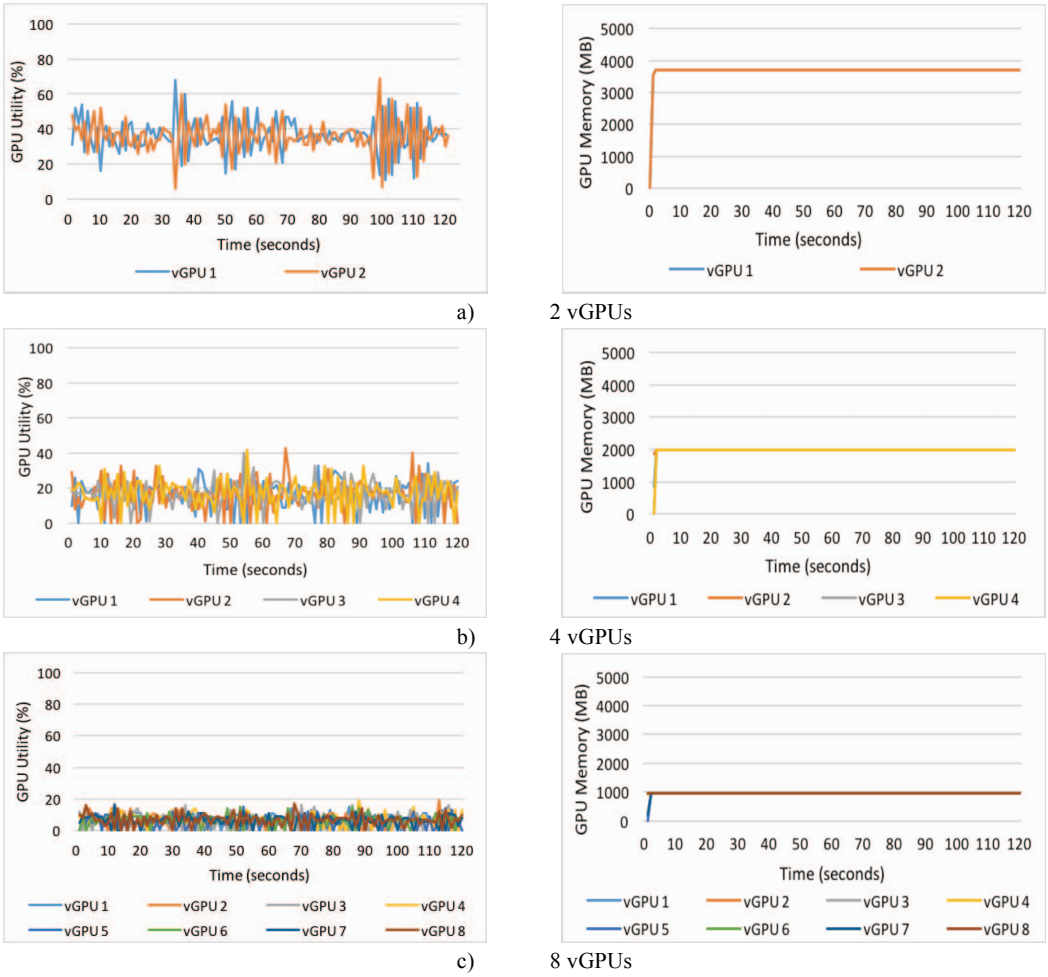

Fig. 7. GPU utilization (over 120 seconds) with different number of vGPUs

D. Elastic Resource allocation

To discern the impact of elastic resource allocation, we launch two containers on a physical GPU. One container requires 0.3 GPUs and 1GB memory and the other requires 0.7 GPUs and 1 GB memory. The MNIST application is executed with Tensorflow in two containers. At first, one container with 0.3 GPUs executes the MNIST application. After 10 seconds, the other container with 0.7 GPUs executes the MNIST application. The experimental results are shown in Fig. 8. Compared to the hard limit, the execution time of the MNIST with 0.3 GPUs is reduced by 46%, and the execution time with 0.7 GPUs is reduced by 13%. The average GPU utilization of the physical GPU is increased 24.6%. The experimental results illustrate that elastic resource allocation can significantly improve the application performance and the GPU utilization.

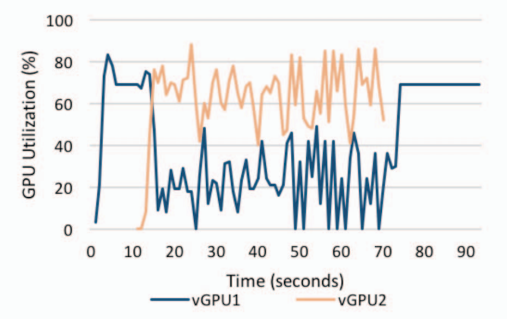

Fig. 8. GPU utilization on elastic resource allocation

V. Conclusion

In this paper, we propose an approach for sharing GPU memory and computing resources among containers. Based on the device plugin framework of the Kubernetes, we partitions a physical GPU into multiple vGPUs and assign vGPUs to containers. GaiaGPU adopts two-layer resource management, one for managing vGPU resources at the host layer and the other at the container layer for managing the resources consumed by containers. In order to improve resource utilization, we provide elastic resource allocation which temporarily modifies container resources and dynamic resource allocation which permanently changes container resources during runtime. Experiments are conducted to evaluate the performance of GaiaGPU. The results reveal that the overhead of GaiaGPU to five deep learning frameworks is 1.015% by average and the GPU resources of vGPUs can be effectively managed. In future work, we will optimize GPU resource allocation and deallocation to reduce fluctuation of vGPU resources.

References

[1] Soltesz S, Pötzl H, Fiuczynski M E, et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors[C]//ACM SIGOPS Operating Systems Review. ACM, 2007, 41(3): 275-287.

[2] Vaucher S, Pires R, Felber P, et al. SGX-Aware Container Orchestration for Heterogeneous Clusters[C]. 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2018, pp. 730-741.

[3] Kubernetes. https://kubernetes.io/, accessed 2018-08-23.

[4] Dai H, Lin Z, Li C, et al. Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls[C]//High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on. IEEE, 2018: 208-220.

[5] Hong C H, Spence I, Nikolopoulos D S. GPU virtualization and scheduling methods: A comprehensive survey[J]. ACM Computing Surveys (CSUR), 2017, 50(3): 35.

[6] Herrera A. NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation[J]. Nvidia Corp, 2014.

[7] NVIDIA Docker. https://www.nvidia.cn/object/docker-container-cn.html, accessed 2018-08-23.

[8] Kang D, Jun T J, Kim D, et al. ConVGPU: GPU Management Middleware in Container Based Virtualized Environment[C]//Cluster Computing (CLUSTER), 2017 IEEE International Conference on. IEEE, 2017: 301-309.

[9] nvidia-smi.pdf. https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, accessed: 2018-08-23.

[10] Shi L, Chen H, Sun J, et al. vCUDA: GPU-accelerated high-performance computing in virtual machines[J]. IEEE Transactions on Computers, 2012, 61(6): 804-816.

[11] Duato J, Pena A J, Silla F, et al. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters[C]//High Performance Computing and Simulation (HPCS), 2010 International Conference on. IEEE, 2010: 224-231.

[12] Kubelet. https://kubernetes.io/docs/concepts/overview/components/, accessed 2018-08-23.

[13] Suzuki Y, Kato S, Yamada H, et al. GPUvm: Why not virtualizing GPUs at the hypervisor?[C]//USENIX Annual Technical Conference. 2014: 109-120.

[14] Abadi M, Barham P, Chen J, et al. Tensorflow: a system for large-scale machine learning[C]//OSDI. 2016, 16: 265-283.

[15] Chen T, Li M, Li Y, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems[J]. arXiv preprint arXiv:1512.01274, 2015.

[16] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. ACM, 2014: 675-678.

[17] Paszke A, Gross S, Chintala S, et al. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration[J]. 2017.

[18] Seide F, Agarwal A. CNTK: Microsoft's open-source deep-learning toolkit[C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2016: 2135-2135.

[19] GaiaStack. https://market.cloud.tencent.com/products/3966?productId=3966#, accessed 2018-08-23.

[20] Kussul E, Baidyk T N. Improved method of handwritten digit recognition tested on MNIST database[J]. Image and Vision Computing, 2004, 22(12): 971-981.

[21] The CIFAR-10 Dataset. https://www.cs.toronto.edu/~kriz/cifar.html, accessed 2018-08-23.

[22] Krizhevsky A, Sutskever I, Hinton G E, et al. ImageNet Classification with Deep Convolutional Neural Networks[C]. neural information processing systems, 2012: 1097-1105.

[23] Tensorflow Models. https://github.com/tensorflow/models, accessed 2018-08-23.