2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)

GPU Scheduling for Short Tasks in Private Cloud

Jialun Shao, Junming Ma, Yan Li, Bo An, Donggang Cao
Institute of Software, School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing 100871, China
{shaojialun, mjm520, yan.l, anbo, caodg}@pku.edu.cn

***Abstract*—GPUs are usually very expensive and not easily affordable by individuals. Therefore, GPU sharing is necessary to lower cost and avoid GPU idling among a group of users. Unlike jobs in production environments, which often last for days or weeks, the running time of programs in development and testing environments tend to be much shorter. Assigning a separate GPU to a person for development always leads to idling of the GPU. Therefore, for economic reasons, researchers usually share a small number of GPUs for development, especially in some small teams or labs. Users hope to automatically lease and release GPUs and get job responses as soon as possible. Current GPU sharing approaches either do not have good support for multiple users, or not designed to work effectively for such cases. This paper proposes a GPU-sharing method among multiple users for short GPU tasks. We implement a container-based batch computing system, which accepts and executes users' jobs through container images and specified configurations. A shortest-job-first based scheduling policy is used to ensure the priority of the short tasks and to prevent long tasks from starving. Evaluation demonstrate that our proposed method is effective and the system has a low overhead.**

Keywords— GPU, container, batch computing, scheduling

I. INTRODUCTION

In recent years, with the public's increasing attention to artificial intelligence and the advancement of data computer hardware, the study on deep learning has become more and more popular. Deep learning workloads usually need to use GPUs to accelerate executing. At the same time, with the emergence of more and more deep learning frameworks such as TensorFlow [1], PyTorch [2], Keras [3], etc., the use of GPUs is no longer a cumbersome task, making more and more people begin to write GPU-based programs. However, GPU is a kind of relatively expensive resource. For example, on Amazon AWS [4], a *r5.2xlarge* on-demand EC2 instance in region US East (Ohio) with 8 vCPUs, 38 ECU and 64 GiB memory charges 0.504 dollars per hour, while a *p3.2xlarge* on-demand instance in the same region with a similar configuration except an extra Tesla V100 GPU charges 3.06 dollars per hour, nearly six times more expensive than the former. Because of the high price, it is very important to use the GPU efficiently.

Typically, the development of a GPU program such as a deep learning program in a production environment requires the following two steps: 1) programming and debugging on a local development machine, and testing on a test cluster; 2) deploying and running on a production cluster such as a Kubernetes [5] cluster. Our focus is on the first stage. It usually includes several iterations of modifying the code and tuning its performance on a small dataset by running in a local testing machine. The program runtime is usually short because of the small data set. Therefore, the GPU is likely to idle when the programmer is programming or resting if it is monopolized by one person, which may cause waste of resources. Therefore, for economic reasons, it is necessary to share GPUs to avoid GPU idling and effectively improve the GPU utilization ratio.

Unlike CPUs that can use cgroups [6] to limit resource usage, the management of GPU resources is more difficult. There are many technical problems if giving GPU access to multiple users at the same time. For example, a TensorFlow program will automatically allocate all the GPUs and unlimited memory by default if the program does not specify it. At this time, if other users want to access the GPU, they will receive an error and can't do anything but wait. Besides, if an error occurs within the first user's program, e.g., causing a suspended animation, the GPU will not actively release the resources it holds unless the program exits. Therefore, this kind of GPU-sharing method is often accompanied by off-line manual negotiation among users, which is cumbersome and inefficient. Furthermore, it also cannot prevent users from erroneously running long training tasks on the shared GPU, which will block other users' access for a long time. Therefore, an upper-level scheduling mechanism is needed to manage these short tasks in development and debugging environments to prevent resource conflicts.

It is a common idea to cluster GPUs for sharing, even in joint-cloud computing environment [7, 8]. In order to increase the GPU utilization of clusters, there are many solutions of managing and scheduling GPUs. Many existing researches are based on a large number of distributed GPU servers in production environment, in which the tasks' running time typically varies from hours to weeks. There are also many researches optimized the schedule policy for specific types of GPU tasks with distinctive features, such as deep learning tasks, whose characteristics are iterativeness and convergence, as described by Optimus [9]. However, not every GPU task is a deep learning task. GPUs are also used for scientific computing, image processing, movie rendering, etc. In short, there is less focus on GPU scheduling for short tasks with general purpose in the development environment.

This paper proposes a GPU-sharing method among multiple users for general purpose and implements it in a container-based batch computing system. We use containers to encapsulate user programs and dependencies, build application environments, and propose a modified shortest-job-first

978-1-7281-1442-2/19/$31.00 ©2019 IEEE
DOI 10.1109/SOSE.2019.00037

scheduling policy. The user needs to set the task expiration time when submitting it to the batch computing system. The shorter the expiration time, the higher its scheduling priority, and the earlier it executes. In order to increase the priority of their tasks and prevent the task from being killed due to timeout, the user will provide the task execution time as accurately as possible. So we don't have to worry about the prediction of the task runtime. Besides, the task waiting time is considered in the scheduler to ensure that long tasks will not starve while keeping the average latency of the tasks low.

Our work has the following contributions:

1) We propose a scheduling policy for short GPU tasks, taking the user-specified timeout and waiting time into account to determine the priority of a task. The policy effectively prevents tasks from starving while ensuring scheduling delay.

2) We implemented the policy in a batch computing system and evaluated the feasibility and effectiveness of the policy.

The rest of paper is organized as follows. The related work is discussed in section 2. The design and implementation of our system are presented in section 3. The evaluation of our work is demonstrated in section 4. Finally, section 5 presents the conclusion and future work.

II. Background and Related Work

GPU virtualization. There are three fundamental types of GPU virtualization in general: Pass-Through [10], API Forward [11] and Virtualized GPU [12]. Pass-Through connects the physical GPU directly to the virtual machines or containers to get the full features of GPU and has the highest performance compared to the previous two schemes. API Forward intercepts GPU commands, sent them to the GPU and get the results back to users. It has good support for multi-person sharing while lacks some software capabilities because the functionality of the GPU depends on whether the API supports it. The performance of Virtualized GPU is typically better than API Forward, but they are usually not a universal solution. Intel GVT-g [12] offers a full GPU virtualization approach with mediated pass-through support for Intel Processor Graphics, which are usually not used by data centers. NVIDIA GRID vGPU [13] is a formal commercial solution for GPU virtualization but is not open-sourced and is only supported by some special professional GPU models with limited numbers of vGPUs. Our system runs on the upper level of the GPU driver and manages GPU resources, whether it is a physical GPU or a virtual one.

Production GPU cluster scheduling. Existing mainstream cluster systems support GPUs. Mesos [14] and Yarn [15] use Dominant Resource Fairness (DRF) [16] to allocate resources while we put the GPU resources to the first place and focus on the low latency of jobs. Kubernetes has also started to support GPUs since version 1.8. Kubernetes is a popular open-source platform for managing containerized micro-services in production environments, while we focus on effectively improving the GPU utilization for short tasks in development environments. There is also a lot of work for scheduling optimization of existing GPU cluster systems. For example, Optimus [9] makes deep optimizations for the scheduling of deep learning jobs on the parameter server architecture. Prophet [17] proposes a model for GPU performance prediction. Gandiva [18] use intra-job predictability to time-slice GPUs efficiently across multiple jobs and dynamically migrating jobs to better-fit GPUs. They are all focused on deep learning tasks and require users to modify their program to provide information of each iteration, while we are committed to providing a common GPU sharing solution that eliminates the need for users to modify the code.

Batch processing. Batch processing is the processing of transactions in a group or batch, which requires no user interaction. batch processing is an important form of sharing GPUs, but it does not work the same way as traditional batch jobs. In traditional batch solutions, resources can be shared through cgroups, and scheduling schemes can be more flexible. Since we can only control whether the application can access the GPU and cannot limit its specific resource usage, in order to prevent conflicts between tasks, the basic unit for GPU scheduling can only be one and cannot be split. In addition, CPU tasks can be suspended/resumed at any time, so more preemption strategies can be taken during scheduling. However, suspending and resuming GPU tasks is not as simple as CPU tasks, so our scheduling policy is non-preemptive.

III. The Design and Implementation

In order to prevent GPUs from idling and to make full use of the GPU resources, this paper proposes and implements a batch computing system, which provides a GPU-sharing method among multiple users for general purpose. Considering that each task depends on a different execution environment, all batch tasks will run in separate Linux Containers for security and efficiency reasons. In order to manage the global user data and the image market, we use GlusterFS as a distributed storage system, which can also be replaced by other file storage and object storage systems. Tasks submitted by users are required to be organized into DAG (Directed Acyclic Graphs) to meet different resource requirements at different stages of a complex mission. For example, data preprocessing does not require a GPU but model training requires one. Since the system is running at the resource level, the flow of data between tasks needs to be implemented by the users themselves through the distributed file system.

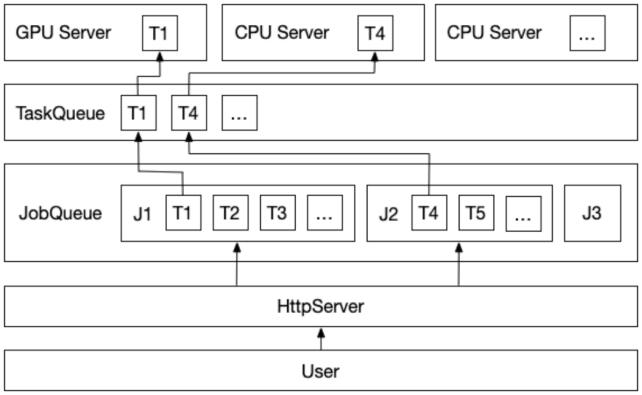

Fig. 1. The overview of the schedule policy of the batch system

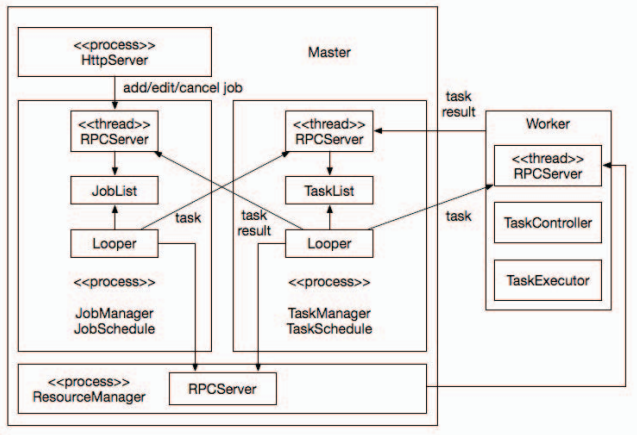

Fig. 2. The system architure

In order to run jobs using our system, the following steps are needed: 1. Make some personalized modifications based on the base image we provide. 2. Upload the modified image. 3. Upload the data to the NFS (Network File System) we provide. 4. Submit the task and upload the code to their personal space in NFS. An image market and a data market are provided to simplify the first three steps, which can be completed by other users.

As shown in Fig. 1, the user jobs will be put into a job queue and all tasks in a job will be sorted topologically according to their dependency. All runnable tasks, which means that the tasks they depend on are all finished, are managed by the task queue and finally run on the different machines in turn according to the schedule policy.

The system architecture is shown in Fig. 2. The system is running on a distributed environment and divided into five parts: *HttpServer*, *ResourceManager*, *JobManager*, *TaskManager* and *Worker* (*TaskController* and *TaskExecutor*). Each part can run independently on a physical machine, and they communicate with each other via remote procedure calls.

A. HttpServer

HttpServer is a user interface for receiving requests from users for submitting tasks. User input will be converted to a json format Job request and passed to *JobManager*. A user-friendly web page is provided to help users build job configurations such as resource requirements and DAG. Users can customize the number of retries, timeouts, commands executed, output paths, container images, CPU requirements, memory requirements, and GPU requirements for each Task. Input files can be placed in the NFS, which will be mounted to the /root/nfs directory inside the container. The progress of the job and each task is also displayed on this page. In addition, image management and storage management also have separate pages for users to upload and share data.

B. ResourceManager

ResourceManager monitors all physical resources and provides support for *JobManager* and *TaskManager*, including CPU, memory, disk and GPU usage, by periodically sending rpc requests to *Workers* to collect real-time resource status. The current design is relatively basic, but there can be more extensions here, such as automatic scaling of resources.

C. JobManager

JobManager receives formatted Job requests from *HttpServer*. Job is the scheduling unit of *JobManager*, which corresponds to a request from the user. It parses the DAG and topologically sorts all tasks and submits all executable tasks, with all tasks they depend have been completed, to the *TaskManager*. In addition, the *JobManager* monitors the execution status of tasks in the *TaskManager* and returns them to the user interface when needed.

D. TaskManager

TaskManager receives Task requests from *JobManager*. Task is the scheduling unit of *TaskManager* and it is an independent process in the user program, which has different requirements for different resources according to the needs of users. To avoid programs that do not require a GPU occupying the computing resources of the GPU server, the *TaskManager* assigns tasks to different servers based on resource requirements.

As we mentioned in the first and the second section, Our main optimization scenario is to lease limited GPU resources to a large number of users for development and testing. For tasks that require long-running tasks, a simple priority-based FIFO algorithm is sufficient. Our main client is beginners who often write bugs, students who might submit similar jobs because of the same course, and researchers who needs to test the correctness of the program in a development environment. Their common feature is that each job does not take long to execute, but may need to resubmit and view the results frequently.

So the system needs to meet the following requirements:

- Restrict users from submitting tasks that take days to run unless we have enough GPU resources
- Reduce the scheduling priority of tasks that take hours to run to ensure the priority of short tasks
- Need to respond in a short time for debugging tasks (can be judged by timeout).
- Avoid GPU idling if the task queue is not empty

Our scheduling algorithm of *TaskManager* is based on a non-preemptive, priority-based FIFO (First In First Out) algorithm. As long as there are free GPU resources (we assume that other physical resources are always sufficient, as they are much cheaper than the GPU), the scheduler will choose the task with the highest priority to be executed by *TaskController*. The priority here is defined by ourselves, so we can easily modify the scheduling policy by adjusting the definition of this priority. For example, if we set the priority equal to the submit timestamp, we can actually get a FILO (First In Last Out) algorithm.

Due to some characteristics of short tasks (debug tasks), such as high response time requirements, uncertain submission

time, high submission frequency, etc., we must ensure that at least one GPU is available at any time the user may submit a task, which means that long-term tasks have to be banned if we only have very few GPUs in total. The shortest task first algorithm is used to ensure the response time of short tasks, but we made some modifications to it.

$$P = \alpha p - \beta T - \gamma t \quad (1)$$

To prevent low-priority programs (programs with long timeouts) from starving, we associate priority with their waiting time (commit time) by increasing the priority of programs that are waiting too long. The priority formula is shown in (1), where P is the scheduling priority, p is the priority submitted by the *JobManager*, T is the timeouts set by users (in seconds), and t is the submission timestamp (in seconds). α can be set to be a large number to ensure the scheduling priority provided by *JobManager*. We can modify the scheduling policy by adjusting the ratio of β and γ, which is currently set to be 1. The scheduling algorithm is exposed to the user to encourage the user to accurately estimate the execution time of their program.

TaskManager also monitors the ruuning status of the tasks by monitoring the heartbeat request of the *Worker* and restarts tasks if the worker loses connection.

E. Worker

A *Worker* consists of a *TaskController* and a *TaskExecutor*. The Task that the *TaskManager* decides to execute will be sent to the corresponding *Worker*, which runs on a separate physical machine, and the *TaskController* will accept the request and call *TaskExecutor* to start the container and run the task. The *TaskController* is also responsible for monitoring the running status of the *TaskExecutor*, processing timeouts, and reporting the results to the *TaskManager*.

We use LXC (Linux Containers) [19] to achieve resource isolation between different task processes. To maximize GPU's performance, we mount the physical GPU directly to the container by Pass-Through, which requires the graphics driver in the container must be the same as the host. It means that the driver version that the user can use has to be fixed. Fortunately, most of the requirements for using GPUs are irrelevant to the graphics driver's version, so we have prepared an image with the latest graphics driver and CUDA installed for the user. Users can use this image by default or customize their own images based on it.

The input and output file can be placed in the NFS folder, which is mounted from the user's global file path in host, which is organized by a distributed file system GlusterFS. The remote object storage provided by the public clouds can also be used as the input and output path.

After the preparation of the container startup is completed, we write the user command to a specific file in the container and start it in container. The standard output stream and the standard error output stream are redirected to another specific file in NFS folder, which can be shown or downloaded from the web page.

IV. Evaluation

We tested our system on a physical machine with 2 processors of Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz (with 6 cores each), 15M of L3 cache, 32 GiB of RAM and 2 NVIDIA GeForce 1080Ti GPUs. Since our design only requires the GPU, regardless of the specific model of it, to be accessible by the Linux Containers, we did not purchase additional GPUs for evaluation. In theory, other types of GPUs can be applied to our system with a few simple adaptations. A series of experiments were designed to verify the feasibility and effectiveness of our system. Different benchmarks were run to evaluate the performance loss of GPU, system overhead and the scheduling performance.

A. System overhead

The process of task execution is divided into five phases: scheduling, sending rpc, creating and starting containers, preparing for execution, and performing tasks. By submitting a TensorFlow example task three times, which trains a simple model for handwriting digital number recogination on the mnist dataset, we found that the average running time of the above five phases are shown in Table I. As the scheduling performance will be evaluated later, each task here is submitted when the task queue is empty. We can see that the main overhead of the system is the task scheduling and container startup, which are both far below the running time of the user programs.

In addition, we recorded the cpu and memory overhead of the system during the above progress. Fig. 3 records the cpu usage of the master process and the worker process in every ten seconds and Fig. 4 records the total memory usage of them. Note that the job is submitted at about 20 seconds and finished at about 180 seconds. It can be seen that the overall overhead of our system is quite small.

TABLE I. Execution Time of Five Phases

Phases	scheduling	RPC	startup	preparing	running
Time(s)	0.939	0.004	0.692	0.016	158.084

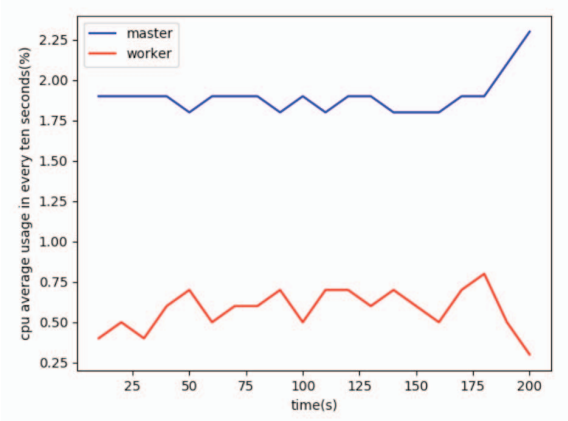

Fig. 3. The cpu usage of the system during the job submission

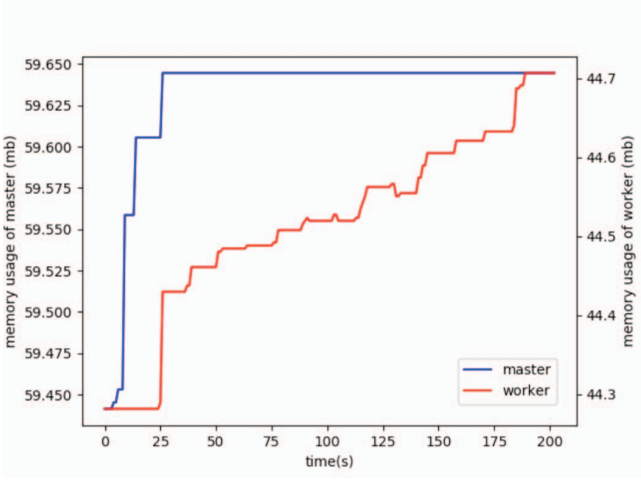

Fig. 4. The memory usage of the system during the job submission

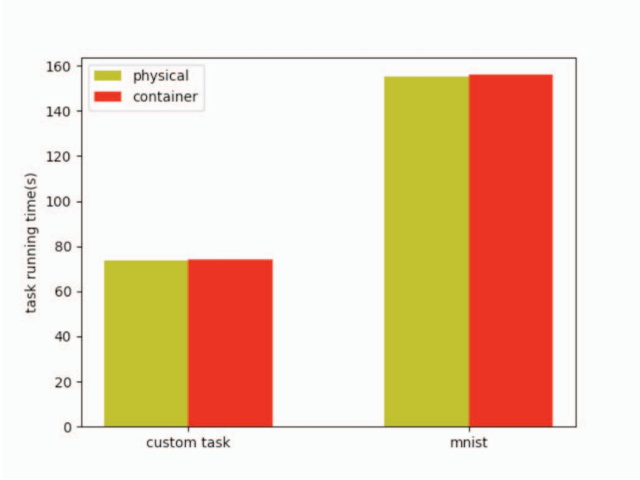

Fig. 5. GPU Performance Comparison

B. GPU Performance

In order to compare the GPU performance in container with native performance, we run a custom deep learning task and the TensorFlow mnist benchmark tasks three times both in the physical machine and in the batch container with two GPU. The result are shown in Fig. 5. As we expected, although the execution time of the task in the batch container includes some system overhead, there is no significant difference between the two results.

C. Scheduling

In order to compare the average scheduling delay of FIFO, SJF and our scheduling algorithm, we must simulate the submitted patterns by real users. Google trace [20] records tasks that run on a Google cluster with about 12k machines in a month. According to some analysis of Google trace [21, 22], we can find the rough rules of some tasks. About 90% of the tasks run for less than 1000 seconds, and almost no task runs for less than 10 seconds. The CDF (Cumulative Distribution Function) graph of the logarithm of the task running time and the total number of tasks shows that the task running time is roughly uniform distributed after taking the logarithm.

Since there is no data for the GPU task in development environment, we generated a task submission queue refer to the distribution of CPU tasks. The task arrival time interval follows the Poisson distribution, and the mean arrival interval is set to be 25 seconds. The logarithm of task runtime is evenly distributed from one to three, which means that the maximum running time is sixteen minutes and forty seconds while the minimum running time is ten seconds. All tasks are deep learning tasks which train an AlexNet on MNIST dataset, and the running time of which can be controlled by the number of the mini-batch iterations. The task's timeout is set to be the same as the task's expected runtime, because we assume that the user will minimize the task timeout if they understand the scheduling scheme.

We use the above rules to generate a task queue with 20 tasks, submit them to the system in turn, and count the response time and end time of each task. Each task requires two CPU, 8GB memory and one GPU. After switching the scheduling scheme of the system, we use the same task queue for repeated experiments. The entire experimental procedure is also repeated three times and the results are shown in Table II. Since the expected running time of the task in the experiment is greater than the expected arrival interval, it is inevitable that the task response time and the completion time become longer and longer. As expected, SJF and our modified algorithm can significantly reduce the average response time and finished time of all tasks compared to FIFO.

Further simulation experiment shows that our strategy can prevent long-term tasks from starving while maintaining similar effects with SJF. Assuming there are several short tasks and one long task in the task queue, and a new short task will be submitted every few seconds, then this long task will be starved and will never be scheduled under the SJF scheduling policy, while our strategy will prioritize the long task when appropriate.

TABLE II. COMPARISON OF AVERAGE RESPONSE TIME AND FINISHED TIME FOR TASKS UNDER THREE DIFFERENT SCHEDULING STRATEGIES

Strategy	Average Response Time(s)	Average Finished Time(s)
FIFO	563.989	771.633
SJF	252.187	459.394
Our Strategy	288.516	496.586

V. CONCLUSION AND FUTURE WORK

In this paper, based on the situation where sharing a small number of GPUs to multiple users in debugging and testing environment is inconvenient and difficult to manage, we propose and implement a container-based batch computing service for GPU sharing, helping users execute programs that require GPUs without the user having to care about resource

leases and releases. In addition, A waiting-time-based modified shortest-job-first scheduling algorithm is used to ensure the priority of a debugging request. The evaluation verifies the feasibility and effectiveness of our system.

Our future work includes supporting multiple types and versions of GPU, adaptively scaling resources based on load, load balancing and further optimizing the scheduling algorithm.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China (2016YFB1000105), the Science Fund for Creative Research Groups of China under Grant, No. 61421091.

REFERENCES

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

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

[3] Chollet F. Keras: The python deep learning library[J]. Astrophysics Source Code Library, 2018.

[4] Amazon EC2 Pricing. https://aws.amazon.com/cn/ec2/pricing/on-demand/

[5] Kubernetes. https://kubernetes.io.

[6] Rosen R. Resource management: Linux kernel namespaces and cgroups[J]. Haifux, May, 2013, 186.

[7] Wang H, Shi P, Zhang Y. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization[C]//Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. IEEE, 2017: 1846-1855.

[8] SHI P, WANG H, ZHENG Z, et al. Collaboration environment for JointCloud computing[J]. SCIENTIA SINICA Informationis, 2017, 47(9): 1129-1148.

[9] Peng Y, Bao Y, Chen Y, et al. Optimus: an efficient dynamic resource scheduler for deep learning clusters[C]//Proceedings of the Thirteenth EuroSys Conference. ACM, 2018: 3.

[10] Walters J P, Younge A J, Kang D I, et al. GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications[C]//Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on. IEEE, 2014: 636-643.

[11] 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.

[12] Tian K, Dong Y, Cowperthwaite D. A Full GPU Virtualization Solution with Mediated Pass-Through[C]//USENIX Annual Technical Conference. 2014: 121-132.

[13] Herrera A. NVIDIA GRID vGPU: Delivering scalable graphics-rich virtual desktops[J]. Retrieved Aug, 2015, 10: 2015.

[14] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[C]//NSDI. 2011, 11(2011): 22-22.

[15] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache hadoop yarn: Yet another resource negotiator[C]//Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013: 5.

[16] Ghodsi A, Zaharia M, Hindman B, et al. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types[C]//Nsdi. 2011, 11(2011): 24-24.

[17] Chen Q, Yang H, Guo M, et al. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers[J]. ACM SIGARCH Computer Architecture News, 2017, 45(1): 17-32.

[18] Xiao W, Bhardwaj R, Ramjee R, et al. Gandiva: introspective cluster scheduling for deep learning[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). USENIX} Association}, 2018: 595-610.

[19] Helsley M. LXC: Linux container tools[J]. IBM devloperWorks Technical Library, 2009, 11.

[20] Reiss C, Wilkes J, Hellerstein J L. Google cluster-usage traces: format+ schema[J]. Google Inc., White Paper, 2011: 1-14.

[21] Reiss C, Tumanov A, Ganger G R, et al. Heterogeneity and dynamicity of clouds at scale: Google trace analysis[C]//Proceedings of the Third ACM Symposium on Cloud Computing. ACM, 2012: 7.

[22] Chung A, Park J W, Ganger G R. Stratus: cost-aware container scheduling in the public cloud[C]//Proceedings of the ACM Symposium on Cloud Computing. ACM, 2018: 121-134.