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

Gaia Scheduler: A Kubernetes-based Scheduler Framework

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

Lelai Deng
Data Platform Department
Tencent Inc.
Beijing, China
jacobdeng@tencent.com

Jun Gong
Data Platform Department
Tencent Inc.
Beijing, China
jungong@tencent.com

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

***Abstract*—This paper proposed a topology-based GPU scheduling framework. The framework is based on the traditional kubernetes GPU scheduling algorithm. In existing algorithms, GPU can only be completely allocated (In other words, a GPU is the smallest unit of resource allocation), the GPU resources are not fully used and the load is uneven. In our algorithm, GPU cluster topology is restored in a GPU cluster resource access cost tree, and different GPU resource application scenarios are scheduled and dynamically adjusted based on the resource access cost tree to obtain an optimal scheduling effect. The kubernetes GPU cluster load is effectively improved. Experimental results show that performance on load balance and resource utilization are also improved in this way. GaiaGPU has been widely used in the production practice of Tencent, the application of GaiaGPU has increased the resource utilization of GPU cluster by about 10%.**

***Index Terms*—GPU cluster scheduling; Kubernetes; topology;**

I. Introduction

With the rise of cloud services and cloud computing, container cloud services are increasingly favored by researchers and technology companies due to their portability, scalability and resource isolation. Especially in today's deep learning boom, how to build a high-performance GPU parallel computing cluster has become a question that many researchers consider about.

In existing GPU parallel computing researches, most of them are designed for physical machines. The main consideration is to design high-performance point-to-point communication based on GPU Direct Technology. This kind of researches compensate for the scheduling problem of cluster GPU to some extent, while the space availability is improved.

Some researches on GPU-aware Message Passing Interface (MPI) parallel computing technology optimized the transmission efficiency of Microcontroller Unit (MCU) node's memory data, which to some extent compensates for the lack of basic multi-GPU parallel computing algorithm for multi-GPU scheduling and GPU network topology. Meanwhile, there are few researches on resource isolation and load balancing, which leads to resource competition between different GPU processes during GPU cluster operation and the performance of GPU clusters is not fully utilized.

Kubernetes is one of the most widely used open-source container orchestration systems. It has realized some researches on the combination of clustered GPU parallel computing and container cloud services. However, Kubernetes-based GPU clusters still have the problem of utilization for large-scale computing resources of GPU clusters.

Based on Kubernetes' GPU cluster scheduler, this paper designed a topology-based GPU scheduling framework, Gaia Scheduler, which focused on improving load balancing and GPU resource maximization. Experimental results show that Gaia Scheduler is Both GPU resource utilization and load balancing, which have better performance than existing algorithms.

The main innovations of this paper are as follows:

(1) Designed and implemented a container cloud scheduling framework based on GPU topology mechanism - Gaia Scheduler, which solve the problem of low resource utilization found in the existing GPU scheduling process to some extent.

(2) A perceptual-based GPU resource allocation algorithm is demonstrated in this paper. The algorithm makes full use of the GPU topology and can cope with multiple applications for applying for less than one GPU, applying for a GPU and applying for multiple GPU, with strong applicability.

可以少于1个gpu,也可以多个gpu

(3) Gaia Scheduler also optimizes some computational auxiliary resources such as CPU and Driver during the scheduling process, which further improves the computational efficiency.

II. Related Work

A. Research on Multi-GPU Parallel Computing Technology

As a kind of massive parallel computing tool, GPU has gradually developed into a mature high-performance processor for computing [1]. GPU acceleration technology is also deeply applied in various parallel computing scenarios [2], [3]. Most of the early massively parallel computing problems were achieved by fully invoking massively parallel processing power on a single GPU [4]. In 2007, Nvidia proposed a parallel computing framework for CUDA software and multi-GPU. Under the premise that the problem to be solved can be paralleled, multi-GPU parallel computing technology is used to obtain a larger computational speedup ratio [5].

978-1-7281-1141-4/18/$31.00 ©2018 IEEE
DOI 10.1109/BDCloud.2018.00048

Multi-GPU systems are somewhat similar to CPU multi-core systems, working together through multiple parallel processors, thereby expanding the number of parallel processing units involved in the computational process, achieving higher computational throughput rates. Thus we could reduce the overall calculation time and improve the computational efficiency [6], [7].

GPU Direct [8] is a technology installed between multiple Nvidia GPUs in a single machine for data P2P Direct Access [2] and P2P Direct Transfer for communication dependent PCIe bus hardware topology [9]. In a single computing process, when performing multi-GPU collaborative computing communication, by perceiving the GPU topology, the performance of local point-to-point communication can be improved, and the overall computing occupancy rate is improved [10].

The existing basic GPU parallel computing technology has satisfactory performance in point-to-point communication and coordination. However, simple point-to-point communication is not well adapted to the complex network structure in multi-GPU clusters, due to the basic GPU parallelism. The calculation does not fully establish the connection with the GPU topology. Also, simple application of the basic GPU parallel computing technology can not solve the cluster GPU parallel computing problem well.

B. Research on GPU Aware MPI Parallel Computing Technology

Message Passing Interface (MPI) parallel computing technology [11] is the main solution in high-performance computing at present [12], [13]. MPI provides a rich communication interface [14] to meet the needs of multi-process and cluster multi-process data communication scenarios in a single machine. In a MPI-based high-performance heterogeneous computing cluster, one or more host nodes are generally set up, and one or more computing hardware units (CPU/GPU) are provided in each node [15]. Cluster applications need to integrate the computing power of each node and rely on MPI to provide peer-to-peer communication and aggregate communication to complete data exchange.

GPU Direct RDMA is a GPU direct data exchange data path technology between Nvidia Kepler and its back-end GPU architecture [8], with GPU Direct RDMA support, cross-node CPU memory - CPU memory GPU memory - GPU memory between the transmission and reception bandwidth of the network can reach more than 90% of the theoretical bandwidth of the network card [16]. When the model parameters are exchanged, the 100-megabit data exchange is compressed in a hundred milliseconds, thereby reducing the proportion of communication in the overall training of the model training and improving the computing performance.

MVAPICH [17] is an open source implementation of the GPU Aware MPI library, optimized for GPU memory data between multiple MPI nodes through InfiniBand [18], 10GigE/iWARP [19], RoCE 40GE network awareness [20] and GPUDirect awareness with high transmission efficiency.

There are also some improved GPU Aware-based MPI parallel computing technologies similar to MVAPICH, which compensate for the lack of basic GPU multi-GPU parallel computing algorithm for multi-GPU scheduling and GPU network topology. However, from the scheduling of GPU clusters, the existing GPU Aware MPI parallel computing algorithms are limited in algorithms such as BF (Best Fit) [21], FCFS (First Come First Service) [22], and The basic scheduling algorithms such as time slice rotation [23] are not satisfactory in complex scenes. In addition, the GPU Aware MPI parallel computing technology also has room for improvement in the adaptability of the networking structure. In addition, most of the existing GPU Aware MPI parallel computing algorithms have less research on resource isolation and load balancing, which leads to resource competition between different GPU processes during GPU cluster operation. The performance of GPU clusters is not fully realized. This will also bring about data skewness [24].

C. Kubernetes support for GPU parallel computing

Kubernetes [25] is an open source container orchestration system for automating the deployment, expansion, and management of containerized applications [26]. Originally designed by Google, it was designed to provide a platform for ”automatic deployment, expansion, and operation of application containers across clusters of hosts.” [27] Since resources on kubernetes are resource-isolated and support for load balancing It is better, since it was open source, it has been widely used in production scenarios with the above requirements [28]–[30].

Support for Nvidia brand GPUs is provided in kubernetes 1.3 [31], and an alpha feature for Nvidia brand GPUs is added to each node in the cluster managed by kubernetes. When starting the kubelet, add the node with GPU to kubernetes for management by adding the parameter –experimental-nvidia-gpu.

More support for Nvidia brand GPUs is provided in kubernetes 1.6 [32]. Only one Nvidia GPU on the node can be utilized in kubernetes 1.3, but all Nvidia GPUs on the node are automatically identified in kubernetes 1.6, and Schedule. All Nvidia GPU devices in the node can be obtained in 1.6.

The Kubernetes version number used in this article is 1.9. When using GPU in 1.9, different containers can't share GPU [29], which means that each docker will monopolize the entire GPU, and also need all nodes in the kubernetes cluster. The Nvidia GPU types above are all the same. If there are different Nvidia GPU types on different nodes in a cluster, you need to configure the scheduler with node labels and node selectors to distinguish nodes of different Nvidia GPU types.

Kubernetes-based GPU clusters have made some progress in resource isolation and load balancing. However, Kubernetes-based GPU clusters still have strong coupling between Container and image, and the computing resources of GPU clusters have not been maximized. Based on Kubernetes' GPU cluster scheduler, this paper designs a topology-based GPU scheduling framework, Gaia Scheduler, which focuses on improving

独占

耦合

load balancing and GPU resource maximization. Experimental results show that Gaia Scheduler is Both GPU resource utilization and load balancing have better performance than existing algorithms.

III. Method

A. *Kubernetes cluster preprocessing.*

GPU and CPU core are automatically bound. On a GPU cluster, since the GPU needs to communicate frequently with memory and CPU, we first bind the GPU to the closest CPU on the Kubernetes cluster. In kubernetes, the link information of the CPU and the GPU is stored in the master node, and by accessing the link network information, the distance between the CPU and the GPU in the physical space can be obtained.

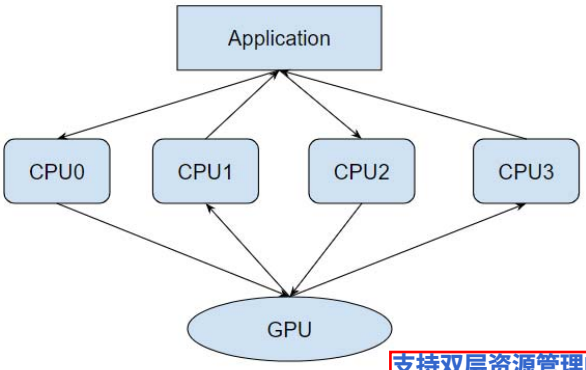


Fig. 1. GPU and CPU core are not bound.

In the existing Kubernetes cluster scheduling algorithm, when the GPU needs to communicate with the CPU core, it will randomly select an idle CPU core to communicate. As can be seen from Figure 1, the four communications applied to the GPU access four different CPU cores respectively. It may be assumed that the distance between the CPU and the GPU in the topology is $CPU0 > CPU1 > CPU2 > CPU3$, and it can be found that the second communication generates redundant time overhead. In order to reduce the unnecessary communication cost, Gaia Scheduler uses the topology stored in master node to sense the GPU. Bind GPU to the nearest n CPU cores, as shown in the Figure 2 (assuming $n = 2$, the GPU will be bound to CPU0 and CPU1), the communication overhead is reduced, and the problem occurred during processing will be easier to located.

Heterogeneous cluster quota. In this paper, quota refers to the cluster computing resources allocated to the container, including but not limited to GPU resources, CPU resources, storage resources and so on. In a common GPU cluster, which is a heterogeneous cluster because the GPU used in the machine expansion are different. Gaia Scheduler classifies the entire heterogeneous cluster by GPU model. When assigning the GPU quota, it uses a more refined allocation strategy to ensure that each kubernetes service gets the expected GPU

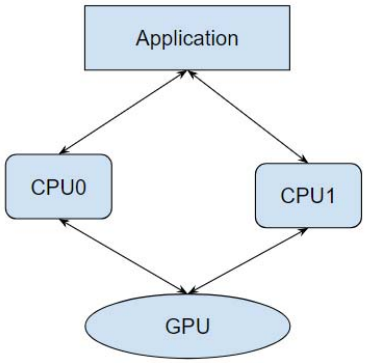


Fig. 2. GPU and CPU core are automatically bound.

resources and avoids resource pool pollution (in the same service allocation, there are different kinds of GPUs, thereby the cluster does not work properly).

GPU resource virtualization. On existing Kubernetes GPU clusters, GPUs are the smallest unit of resource allocation. We use Kubernetes' device plugin framework to divide a physical GPU into multiple virtual GPUs and assign these virtual GPUs to the container. To achieve effective management of GPU resources, we have adopted a two-tier resource management architecture. The physical machine is responsible for allocating GPU resources according to the resource request of the container. At the container level, we manage the resource usage of the container by intercepting the memory and computing resource related APIs in the CUDA library.

B. *Kubernetes GPU Cluster Resources - Access Cost Tree*

In the process of GPU parallel computing, the purpose of scheduling is to reduce the communication overhead between the GPUs, so that the GPU resources are utilized as much as possible in the training process of the model. The communication overhead of the GPU is related to the network topology of the GPU cluster. As shown in the Figure 3, Nvidia implements the classification of the GPU cluster communication mode based on the network structure.

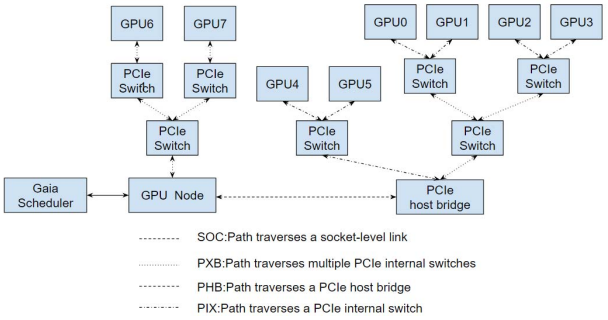
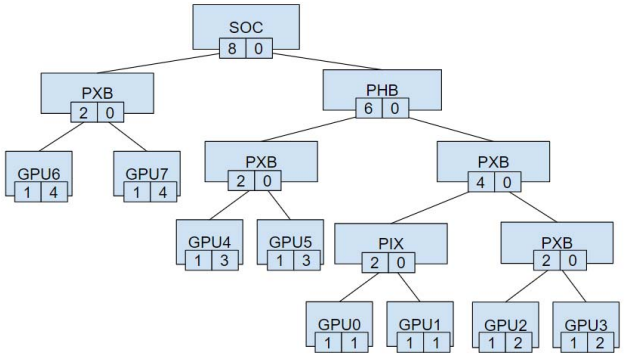


Fig. 3. The communication structure of GPU cluster.

Among the four types of communication methods, the communication overhead is SOC, followed by PXB, and then PHB. The communication overhead between GPUs in PIX communication mode is the smallest. Based on the NVidia research, Gaia Scheduler implements the GPU cluster resource-access cost tree algorithm and performs Kubernetes GPU resource scheduling based on the results of the algorithm.

Algorithm framework. The structure of the access cost tree is shown in Figure 4:



4种通道的通信代价：soc>phb>pxb>pib

of access cost tree.

The following information is stored in the tree:

- Is there a fragment node in the tree (the node resource is partially allocated);
- Number of fragment nodes.

The following information is stored in the topology node:

- GPU communication mode of the child node (SOC, PXB, PHB or PIX);
- The number of available GPU resources (the number on the left side of the node in the Figure 4, if the subordinate GPUs is n);
- Node communication overhead (the number on the right side of the node in the Figure 4, the non-GPU node is 0).

The GPU nodes store the following information:

- GPU id;
- The number of available GPU resources (the number on the left side of the node in the figure, the GPU node is 1);
- Node communication overhead (the number on the right side of the node in the figure, the smaller the number, the lower the access cost)

When an application applies for GPU resources, based on the resource-access cost tree, the Gaia Scheduler will allocate GPU resources according to the following algorithm.

It should be noted that, because the GPU resources are previously virtualized, when the applied GPU resource is less than 1, it may be a non-integer block GPU (such as 0.5). However, when the applied GPU resource is greater than 1, considering that the heterogeneous GPU resources cannot implement parallel computing, the GPU resource request with a requirement greater than 1 can only apply for an integer block GPU.

异构gpu不能进行并行计算

Based on the cluster resource-access cost tree, we designed the following algorithm to allocate the GPU resource application respectly (As in Algorithm 1).

Algorithm 1 Main Algorithm: Gaia Scheduler.

Input: The resource-access cost tree, T; The number of required GPUs, $m(m > 0)$;
Output: Allocated GPU nodes set, S;

```
Candidate nodes set S_c ← empty;
if 0 < m < 1 then
    S = Fragment(m, S_c, T);
else if m == 1 then
    S = Singular(m, S_c, T);
else
    S = Link(m, S_c, T);
end if return S;
```

Under the main algorithm, there are also Fragment (for the case where the number of GPUs applied is less than 1), Singular (for the case of applying for 1 GPU) and Link (for the case of applying an integer GPU larger than 1), we will explain these three sub-algorithms separately.

Fragment. For the case where the number of applied GPUs is less than one, set to m ($0 < m < 1$), the following two cases are handled:

遍历所有的满足条件 的节点 >m,优先分配给最接近m的节点。因为碎片浪费才最小。

碎片节点

(1) There is a fragment node in the resource-access cost tree (there are already resources allocated on the GPU node, but the allocated resources are less than 1 card). At this time, all the fragment nodes are added to the candidate set, and the candidate set is traversed, and the candidate is selected. A node that satisfies the remaining resources greater than m and closest to m is allocated to the applicant.

(2) If there is no fragment node in the resource-access cost tree, or if there is no qualified node in step (1), the entire GPU with the lowest cost is assigned to the applicant.

A more systematic algorithm is described in Algorithm 2:

Algorithm 2 Sub Algorithm: Fragment Scheduler.

Input: The resource-access cost tree, T; The number of required GPUs, $m(0 < m < 1)$; Candidate nodes set S_c;
Output: Allocated GPU nodes set, S;

if T.FragmentNode > 0 **then**
  for Node $n_i \in T$ **do**
    if $m <= n_i$.Resources<1 **then**
      $S_c \leftarrow n_i$;
    end if
  end for
  BestNode = candidate $\in S_c$ with lowest Resources;
  $S \leftarrow$ BestNode;
end if
if S is empty **then**
  $S \leftarrow$ node $\in T$ with lowest AccessCost and Resources is 1;
end if return S;

singular. For the case where the number of applied GPUs is equal to 1 (set to m, m=1), for computational performance considerations, we hope that in the resource-access cost tree, we want to be a topology node (PXB, PHB in the figure below. One of the child nodes of the SOC, PIX) is assigned, and in the next allocation, a limited allocation of another idle child node is assigned. (Here we call this strategy Singular strategy)

As shown in the Figure 5, the red node represents the resource has been allocated, and the blue is the idle node. If a container applies for a GPU at this time, according to Kubernetes' traditional scheduling algorithm, GPU0 or GPU1 will be assigned to the container (with the lowest access cost), but Gaia Scheduler will assign GPU5 to the container (in accordance with Singular Strategy, with the lowest cost of access).

这里为了更少的交流损失，分配给gpu5 而不是满足条件的gpu0 puch1

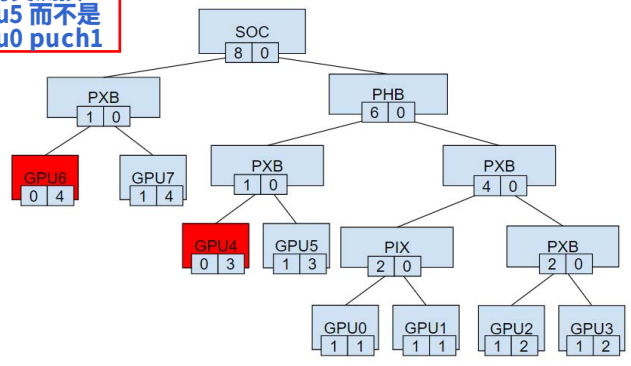


Fig. 5. An example of the access cost tree in use.

When the container applies for a single card, each application will preferentially execute the Singular policy until the current resource-access cost tree does not need to execute the Singular policy. At this time, the idle minimum access cost node in the search resource-access cost tree is assigned to container.

A more systematic algorithm is described in Algorithm 3.

Algorithm 3 Sub Algorithm: Singular Scheduler.

Input: The resource-access cost tree, T; The number of required GPUs, $m(m = 1)$; Candidate nodes set S_c;
Output: Allocated GPU nodes set, S;

for Node $n_i \in T$ **do**
  if n_i.Resources is 1 and n_i.Cousin.Resources is 0 **then**
    $S_c \leftarrow n_i$;
  end if
end for
BestNode = candidate $\in S_c$ with lowest AccessCost;
$S \leftarrow$ BestNode;
if S is empty **then**
  $S \leftarrow$ node $\in T$ with lowest AccessCost and Resources is 1;
end if return S;

深度优先搜索

可行的解决方案加入到候选树

筛选

Link. When container applies for multiple GPUs, the Gaia Scheduler uses the depth-first search, traverses the resource-access cost tree, adds the parent node of the feasible solution to the candidate set, and performs another screening in the candidate set to ensure the access of the finally assigned node to ensure the cost is minimal. As shown in the Figure 4, when applying for two cards, there are multiple possible root nodes, but considering the minimum access cost, the GPUs finally assigned to the applicants will be GPU0 and GPU1.

A more systematic expression is as shown in Algorithm 4:

Algorithm 4 Sub Algorithm: Link Scheduler.

Input: The resource-access cost tree, T; The number of required GPUs, $m(m > 1)$; Candidate nodes set S_c;
Output: Allocated GPU nodes set, S;

for Node $n_i \in T$ **do**
  while Node $n_i! = root$ **do**
    if $n_i.Resources < m$ **then**
      $n_i \leftarrow n_i.Parent$;
      continue;
    end if
  end while
end for
for Node $n_i \in S_c$ **do**
  $TotalCost_i = m \times n_i.Children.AccessCost$;
end for
$BestNode = candidate \in S_c$ *with lowest* $TotalCost$;
$S \leftarrow m$ *different children nodes of* $BestNode$;
if S *is not empty* **then return** S;
end if

C. Flowchart of A Complete Scheduling Process

As is shown in Figure 6, the whole scheduling process includes seven steps.

(1) Containers apply GPU resources (GPU vcores) from Gaia Scheduler;

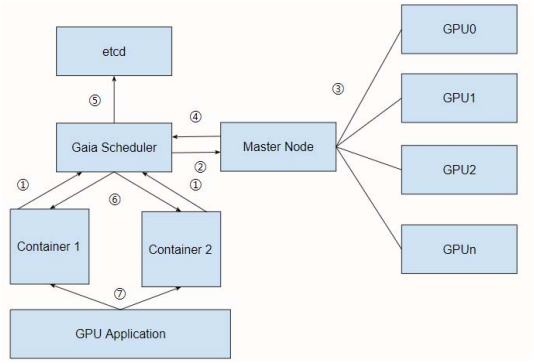


Fig. 6. The flowchart of a complete scheduling process.

(2) Gaia Scheduler submits resource requests to the master node;
(3) Master node select appropriate GPU vcores through the corresponding algorithm in Algorithm 1;
(4) The master node gives the GPU vcore information assigned to the container;
(5) Gaia Scheduler writes the allocated GPU vcore information in etcd (etcd is an open-source distributed key value store that provides shared configuration and service discovery for Container Linux clusters);
(6) Gaia Scheduler assigns GPU vcores to the corresponding container;
(7) GPU Application start;

IV. Results

In this part we carried out several experiments to test the effectiveness and performance of Gaia Scheduler.

The experimental environment is as shown in the Figure 7. The experiment was deployed on Tencent's container cloud GaiaStack. We built a Kubernetes cluster of five servers, three of which act as master nodes and two servers as Node nodes. The configuration of the Master node and the Node node is shown below.

Master

CPU: 12 Intel(R) Xeon(R) CPU E5-2680 v3 RAM: 24G

Node

CPU : 56 Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz RAM: 128G GPU: Nvidia Tesla P4 CUDA 9.0.0 CUDNN 7.0.4

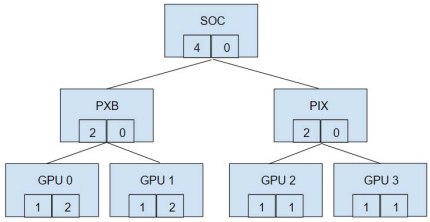


Fig. 7. The access cost tree for experiment.

Experiment 1: Simple Scheduling. In Experiment 1, we designed the case where the container applied for one GPU and the case where the container applied for two GPUs to test whether the Gaia Scheduler can allocate the GPU with the lowest access cost to the container. For the case of applying for one card, it is expected that the container will be assigned to any one of gpu2 or gpu3 (as shown in Figure 8(a)). For the case of applying for 2 GPUs, the container can only be assigned to gpu2 and gpu3 (as shown in Figure8(b)).

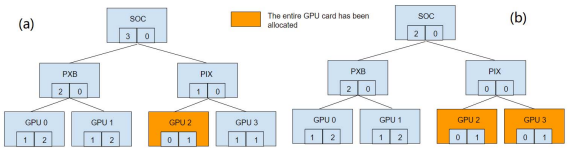


Fig. 8. Expected results for simple scheduling.

We repeat 500 times in the above two cases, and the results are shown in the following Table I.

TABLE I
Result of Experiment 1.

Cases	Container applied for 1 GPU			Container applied for 2 GPUs	
	gpu 2	gpu 3	others	gpu 2&gpu 3	others
Frequency	227	273	0	500	0

Experiment 2: Fragment Scheduling. In Experiment 2, we first assign 0.5 gpu2 to a certain container (denoted as container 1, and the GPU resource allocated to container 1 is the red part in the Figure 9). At this point, another container (container 2) requests 0.4 GPU resources from the cluster. The expected result is that container 2 gets 0.4 gpu2 (yellow part in the Figure 9). At this time, the third container (container 3) requests 0.1 GPU resources from the cluster, and it is expected that container 3 will get 0.1 gpu2 (green portion in the Figure 9). After the whole experiment, the cluster GPU resource status will be as shown in Figure 9.

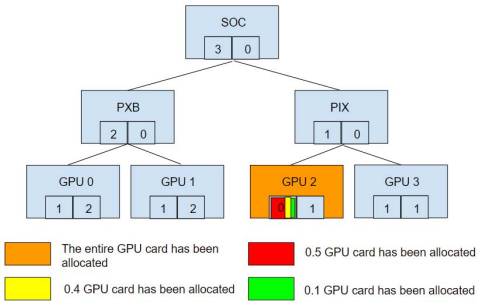


Fig. 9. The status of access cost tree in fragment scheduling.

We repeat 500 times the application process of container 2 and container 3, and the results are shown in the following Table II.

Experiment 3: Singular Scheduling. In Experiment 3, the initial state of the cluster is shown in Figure 8(a). We repeatly

TABLE II
RESULT OF EXPERIMENT 2.

Cases	Allocated GPU for Container 2		Allocated GPU for Container 3	
	0.4 gpu 2	others	0.1 gpu 2	others
Frequency	500	0	500	0

require 1 GPU from cluster, the experimental results are shown in the following Table III.

TABLE III
RESULT OF EXPERIMENT 3.

Cases	Allocated GPU			
	gpu 0	gpu 1	gpu 2	gpu 3
Frequency	0	0	0	500

Experiment 4: Link Scheduling. In Experiment 4, the initial state of the cluster is shown in Figure 8(a). We repeatly require 2 GPUs from cluster, the experimental results are shown in the following Table IV.

TABLE IV
RESULT OF EXPERIMENT 4.

Cases	Allocated GPUs	
	gpu 0 & gpu 1	others
Frequency	500	0

Experiment 5: Performance of Scheduling. In Experiment 5, we used the Kubernetes default GPU scheduling strategy and Gaia Scheduler for scheduling in the previous experiments 1-4. The experimental results are shown in the following Figure 10(Since the scheduler of kubernetes does not contain the fragment scheduling strategy, the scheduling time of kubernetes in experiment 2 is 0.).

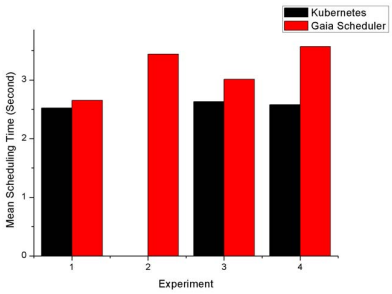


Fig. 10. Comparison of Gaia Scheduler and Kubernetes scheduling performances.

Experiment 6: Performance of Executing (Link mode). In the scheduler that comes with kubernetes, there is no link policy, so the situation shown in the Figure 11(a) may occur. However, after scheduling by Gaia Scheduler's optimized link scheduling strategy, only a state similar to that of Figure 11(b) will appear.

We performed the official sample training 10 times on Caffe [33], Pytorch [34] and Tensorflow [35] with MNIST dataset [36] in this experiment. The performance results are shown in the Figure 12.

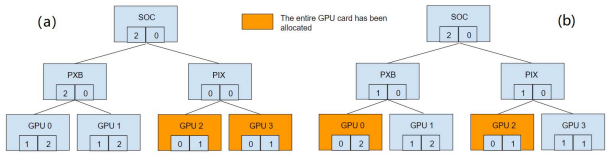


Fig. 11. Comparison of Gaia Scheduler and Kubernetes executing status.

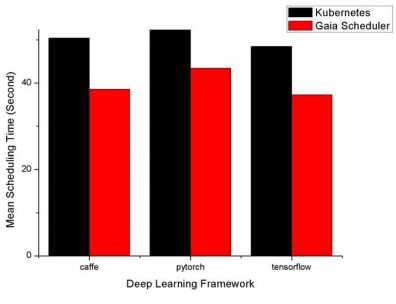


Fig. 12. Comparison of Gaia Scheduler and Kubernetes executing performances.

V. CONCLUSIONS

This paper demonstrated a topology-based GPU scheduling framework. The framework is based on the traditional kubernetes GPU scheduling algorithm. In existing schedulers, the GPU can only be allocated completely, the GPU resources are not in maximized usage and the load is uneven. In Gaia Scheduler, the GPU cluster topology is transformed into a GPU cluster resource-access cost tree, and different GPU resource application scenarios are scheduled and dynamically adjusted based on the resource-access cost tree to obtain an optimal scheduling effect. Thereby effectively improving the kubernetes GPU cluster load. GaiaGPU has been widely used in the production practice of Tencent, the application of GaiaGPU has increased the resource utilization of GPU cluster by about 10%.

The performance of the balance and resource utilization shows that Gaia Scheduler has better performance than the existing algorithms in GPU resource utilization and load balancing.

In the follow-up study, we will further carry out the following work to improve the Gaia Scheduler:

(1) A more flexible quota distribution system that allows quota to use more than the requested value when there is only one GPU application on a single card, but strictly limits its quota usage when assigning new GPU applications.
(2) More fine-grained GPU virtualization scheduling, for non-integer GPU application resource requests that apply for more than one card.

REFERENCES

[1] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[2] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[3] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[4] D. Kirk *et al.*, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[5] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE micro*, no. 4, pp. 13–27, 2008.

[6] J. Enmyren and C. W. Kessler, "Skepu: a multi-backend skeleton programming library for multi-gpu systems," in *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM, 2010, pp. 5–14.

[7] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 1068–1079.

[8] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 80–89.

[9] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia gpudirect over infinibanda new model for gpu to gpu communications," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 267–273, 2011.

[10] M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, and M. Min, "An mpi/openacc implementation of a high-order electromagnetics solver with gpudirect communication," *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 320–334, 2016.

[11] H. J. Berendsen, D. van der Spoel, and R. van Drunen, "Gromacs: a message-passing parallel molecular dynamics implementation," *Computer physics communications*, vol. 91, no. 1-3, pp. 43–56, 1995.

[12] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "Mpi-2: Extending the message-passing interface," in *European Conference on Parallel Processing*. Springer, 1996, pp. 128–135.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[14] W. D. Gropp, W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[15] N. T. Karonis, B. Toonen, and I. Foster, "Mpich-g2: A grid-enabled implementation of the message passing interface," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551–563, 2003.

[16] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer, "Strong scaling of general-purpose molecular dynamics simulations on gpus," *Computer Physics Communications*, vol. 192, pp. 97–107, 2015.

[17] M. J. Koop, T. Jones, and D. K. Panda, "Mvapich-aptus: Scalable high-performance multi-transport mpi over infiniband," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[18] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[19] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 257, 2011.

[20] C. DeCusatis, "Optical interconnect networks for datacom and computercom," in *Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference (OFC/NFOEC), 2013*. IEEE, 2013, pp. 1–33.

[21] B. Sukhwani and M. C. Herbordt, "Gpu acceleration of a production molecular docking code," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 19–27.

[22] G. Teodoro, T. Pan, T. M. Kurc, J. Kong, L. A. Cooper, N. Podhorszki, S. Klasky, and J. H. Saltz, "High-throughput analysis of large microscopy image datasets on cpu-gpu cluster platforms," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 103–114.

[23] P.-A. Heng, Y. Xie, T.-T. Wong, and Y.-P. Chui, "Block-based fragment filtration with feasible multi-gpu acceleration for real-time volume rendering on conventional personal computer," Dec. 26 2006, uS Patent 7,154,500.

[24] J. Zhang, "Towards personal high-performance geospatial computing (hpc-g): perspectives and a case study," in *Proceedings of the ACM SIGSPATIAL international workshop on high performance and distributed geographic information systems*. ACM, 2010, pp. 3–10.

[25] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[26] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 167–167.

[27] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, p. 10, 2016.

[28] V. Medel, O. Rana, U. Arronategui *et al.*, "Modelling performance & resource management in kubernetes," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*. ACM, 2016, pp. 257–262.

[29] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. " O'Reilly Media, Inc.", 2017.

[30] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. S. de Souza, "State machine replication in containers managed by kubernetes," *Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017.

[31] S. R. Seelam and Y. Li, "Orchestrating deep learning workloads on distributed infrastructure," in *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*. ACM, 2017, pp. 9–10.

[32] D. Vohra, "Installing kubernetes using docker," in *Kubernetes Microservices with Docker*. Springer, 2016, pp. 3–38.

[33] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[34] A. Paszke, S. Chintala, R. Collobert, K. Kavukcuoglu, C. Farabet, S. Bengio, I. Melvin, J. Weston, and J. Mariethoz, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, may 2017."

[35] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[36] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.