CCF Transactions on High Performance Computing (2023) 5:304–321
https://doi.org/10.1007/s42514-023-00154-y

REGULAR PAPER

ArkGPU: enabling applications' high-goodput co-location execution on multitasking GPUs

Jie Lou[1,2] · Yiming Sun[1,2] · Jie Zhang[1,2] · Huawei Cao[1,3] · Yuan Zhang[1,2] · Ninghui Sun[1,2]

Received: 11 March 2023 / Accepted: 4 May 2023 / Published online: 24 May 2023

Abstract
With the development of deep learning, hardware accelerators represented by GPUs have been used to accelerate the execution of deep learning applications. A key problem in GPU cluster is how to schedule various deep learning applications, including training applications and latency-critical inference applications, to achieve optimal system performance. In cloud datacenters, inference applications often require fewer resources, and the exclusive GPU execution of one inference application can result in a significant waste of GPU resources. Existing work mainly focuses on the co-location execution of multiple inference applications in datacenters using MPS (Multi-Process Service). There are several problems with this execution pattern, datacenters may be in low-workload state for long periods of time due to the diurnal pattern of inference applications, MPS-based data sharing can lead to interaction errors between contexts, and resource contention may cause Quality of Service (QoS) violations. To solve above problems, we propose ArkGPU, a runtime system that dynamically allocates resources. ArkGPU can improve the resource utilization of the cluster, while guaranteeing the QoS of inference applications. ArkGPU is comprised of a performance predictor, a scheduler, a resource limiter, and an adjustment unit. We conduct extensive experiments on the NVIDIA V100 GPU to verify the effectiveness of ArkGPU. We achieve High-Goodput for latency-critical applications which have an average throughput increase of 584.27% compared to MPS. We deploy multiple applications simultaneously on ArkGPU, and in this case, goodput is improved by 94.98% compared to k8s-native and 38.65% compared to MPS.

Keywords Goodput · GPU sharing · Co-location · Latency critical jobs scheduling · QoS guarantee

Jie Lou and Yiming Sun contributed equally to this work.

✉ Huawei Cao
caohuawei@ict.ac.cn

Jie Lou
loujie18b@ict.ac.cn

Yiming Sun
sunyiming20g@ict.ac.cn

Jie Zhang
zhangjie20s@ict.ac.cn

Yuan Zhang
zhangyuan-ams@ict.ac.cn

Ninghui Sun
snh@ict.ac.cn

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China

[2] University of Chinese Academy of Sciences, Beijing 100049, China

[3] University of Chinese Academy of Sciences, Nanjing 211135, China

1 Introduction

With the rapid development of deep learning technology, recent applications often rely on (Artificial Intelligence) AI accelerators such as GPUs, to improve computational efficiency and accuracy. In addition to this, tens of thousands of information devices are generating a magnitude increase in concurrent tasks. A key problem in GPU cluster is how to schedule a large number of deep learning applications, including training applications and (Latency-Critical) LC inference applications, to achieve optimal system performance. The current practice is to deploy a single service on one GPU to meet the QoS target of the application (Burns et al. 2016). However, the model of exclusive GPU resources for one application can often result in wasted resources, reducing GPU utilization. As a result, how to share GPU resources among different applications is an issue worth investigating. After the sharing problem has been resolved, High-Goodput computing must be implemented to increase

the number of quality-assured tasks completed per unit time (Xu et al. 2022).

Operating system containerization in datacenters allows different applications from different users to run on physical servers. At present, cloud service providers such as Amazon, Google, Tencent, Baidu, and others are using Kubernetes (Burns et al. 2016) as a container orchestration platform to centrally manage various resources in the cluster. However, Kubernetes can only assign a GPU to a pod and doesn't support sharing GPU resources among multiple pods. To improve the utilization of GPU, co-location execution of applications needs to be implemented. However, executing multiple applications simultaneously on a single GPU increases the response time of queries, destroying the QoS target of the application. We need to guarantee the QoS target while improving resource utilization.

In this work, we present the ArkGPU, a runtime system that solves the following problems: allocating GPU resources accurately for applications to meet their demand while ensuring the QoS target of LC applications, co-locating LC applications with (Best-Effort) BE applications while maximizing BE application performance. ArkGPU consists of a resource limiter, a two-level performance predictor, a two-level scheduler, and a performance adjustment unit. The main contributions of ArkGPU are as follows:

- **Enable accurate predictions of applications resource utilization and execution time.** We determine the features of the deep learning application and train the predictive model. The model predicts the application's resource requirements and ensures that the application meets QoS targets when it executes within that resource quota.
- **Enable precise resource allocation.** We divide GPU computing resources and storage by percentage to avoid error sharing among contexts generated by MPS while improving the granularity of that resource division.
- **Provide a runtime system that guarantees high goodput.** We achieve increased cluster utilization and guaranteed task throughput by predicting and limiting resource usage. Furthermore, we provide performance adjustment unit to prevent QoS violations caused by resource contention and insufficient resource limits. Application aborts and migrations are achieved through a checkpointing mechanism provided by the deep learning framework.
- We conduct extensive experiments on four NVIDIA V100 GPUs to verify the effectiveness of ArkGPU. We achieve High-Goodput for LC inference applications which have an average throughput increase of 584.27% compared to MPS. We deploy multiple applications simultaneously on ArkGPU, and in this case, goodput is improved by 94.98% compared to Kubernetes-native (k8s-native) and 38.65% compared to MPS.

2 Related work

There has been some research that focuses on improving resource utilization while guaranteeing applications' QoS targets. Bubble-up (Mars et al. 2011) is a work implemented on the CPU. It improves resource utilization in CPU-based datacenters by predicting interference between co-located applications. Bubble-flux(Yang et al. 2013) uses dynamic interference measurements and enables safe co-locations. SMiTe (Zhang et al. 2014) implements interference prediction on simultaneous multithreading architectures, extending Bubble-up. Dirigent (Zhu and Erez 2016) analyzes the performance model of LC applications to accurately control the QoS of LC applications at a fine time scale. PARTIES (Chen et al. 2019) proposes a solution on the CPU to evaluate the performance by adding fewer resources each time for LC applications to meet QoS targets. CLITE (Patel and Tiwari 2020) uses a black-box model to maximize the throughput of the BE applications while meeting the QoS targets of the LC applications. These works are implemented on the CPU platforms and can't be directly migrated to the GPU, which has different interference factors.

In order to enable applications' co-location on GPUs, NVIDIA has proposed MPS (Multi xxx), which can share GPUs among multiple applications concurrently. MPS doesn't have the flexibility to adjust resource limits based on GPU state, and a fatal GPU exception generated by a kernel may affect all kernels that share GPU resources (Multi xxx). rCUDA (Duato et al. 2019) implements GPU pooling and sharing through API interception, but its main purpose is to make remote calls to GPUs without strict restrictions on resource usage. KubeShare (Yeh et al. 2020) and GaiaGPU (Gu et al. 2018) implement the division of GPUs through API interception, with the difference that KubeShare isolates applications through time-division multiplexing and GaiaGPU limits the resources percentage used by each pod through monitoring-adjustment. However, the co-location of LC and BE applications on the GPU can cause interference that affects the QoS of these applications. Baymax (Chen et al. 2016) implements applications' co-location using time-division of GPU, which predicts the duration to reorder the LC applications. Themis (Zhao et al. 2019) can predict the slowdown of co-located applications, but it is based on the work of the simulator and various statistics are not available on real GPUs. C-Laius (Zhang et al. 2021) guarantees the QoS target by dividing the task types and allocating sufficient resources to LC application. Unfortunately, previous works are based on MPS, and as previously stated, errors from different kernels can interfere with each other. What's worse, they can't strictly limit the GPU resources

available to the application. Fine-grained GPU allocation to different applications to guarantee the QoS target while improving GPU utilization is a new challenge.

In current implementations, inference jobs and training jobs are often deployed separately. Inference jobs are executed in a dedicated inference cluster. The inference job, as a user-oriented job, has a diurnal pattern. The inference cluster can only be kept at full workload during peak hours and is relatively idle during the rest of the day. This diurnal pattern for datacenters shows the possibility of co-locating processing of user-oriented applications and batch applications without QoS targets. Therefore, co-located execution of training and inference jobs has the potential to improve the resource utilization of the cluster. There has been already some works considering a hybrid of training and inference jobs. Kube-Knots (Thinakaran et al. 2019) performs co-scheduling of inference jobs and offline batch jobs. It constructs a scheduler by correlation prediction and peak prediction to solve the problem that inference jobs can't fully utilize GPU. Aryl (Li et al. 2022) enables the inference cluster to deploy training jobs on idle GPUs during low-traffic periods, enabling improved cluster resource utilization.

3 Background and motivation

3.1 GPU sharing solutions

GPU sharing refers to running multiple tasks concurrently on the same GPU. With GPU sharing, more tasks can be executed simultaneously, improving resource utilization and reducing total tasks end time. NVIDIA's latest GPUs now add physical support for multitasking shared resources (named MIG NVIDIA yyy). For previous GPUs, resource sharing can be achieved using MPS (Multi xxx) provided by NVIDIA. (MPS has some problems, a fatal GPU exception generated by an MPS client process will be contained within the subset of GPUs shared between all clients with the fatal exception-causing GPU.). Due to the numerous problems caused by MPS, API interception is commonly used to achieve resource isolation. Figure 1 shows the software stack of DNN application. The isolation layer generally sits on top of the CUDA driver API layer and replaces the CUDA library through the LD_LIBRARY_PATH mechanism. Using LD_LIBRARY_PATH, we can intercept the relevant APIs in the CUDA library to achieve resource limits. Currently, there are two main sharing schemes based on API interception: time-division multiplexing and monitoring-adjustment.

For time-division multiplexing, the runtime of GPU is usually divided into several fixed-length time slices, and tasks request services from the GPU by requesting a share of the time slice. Within one time slice, only the task that occupy the slice can emit instructions. A typical time-division multiplexing pattern is shown in Fig. 2(a). KubeShare (Yeh et al. 2020) uses a token-based approach for time slice allocation. The token is circulated among all tasks and only the task holding the token currently can use the GPU.

For monitoring-adjustment, a fixed monitoring period is set to monitor the status of tasks and make adjustments. The monitored objects can be states of GPUs, such as GPU utilization and memory utilization. It can also be DNN applications, such as the execution time of each iteration. The speed of task delivery is adjusted by slow delivery according to the change of task status. A typical monitoring-adjustment pattern is shown in Fig. 2(b). Tencent's GaiaGPU (Gu et al. 2018) uses this model to share GPU resources, monitor GPU utilization and memory utilization, and adjust the resource usage of each task. Our implementation is similar to GaiaGPU and also based on a monitor-tune approach for GPU sharing.

Resource sharing based on API interception requires consideration of the mechanism of memory sharing. Some works (Yeh et al. 2020, Gu et al. 2018) allow all tasks on the same GPU to share memory, while other work (Xiao et al. 2018) keeps swapping memory in and out and optimizing it for task characteristics.

3.2 Real system setup

We use NVIDIA V100 as our experimental platform. In this work, we use both user-oriented LC applications (inference tasks) and throughput-oriented BE applications (training tasks). Our work doesn't rely on any specific features of

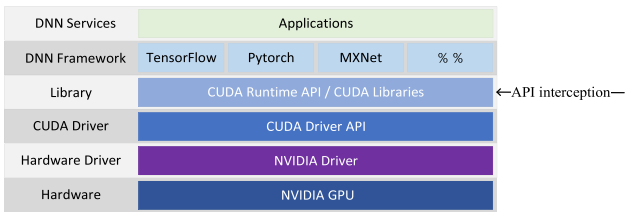


Fig. 1 DNN application layering structure

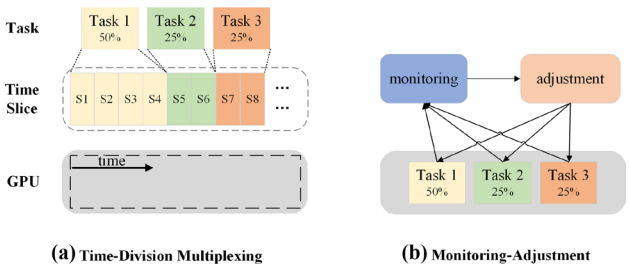


Fig. 2 Time-division multiplexing and monitoring-adjustment

Table 1 The benchmarks used as the LC applications

Scenario	Neural model	Dataset
Classification and detection	resnet50-v1.5	imagenet2021
Classification and detection	ssd-mobilenet 300×300	coco resized to 300×300
Classification and detection	ssd-resnet34 1200x1200	coco resized to 1200x1200
Language	bert	squad−1.1
Recommendation	dlrm	Criteo Terabyte

Table 2 The models used as the BE applications

Models
Resnet50, Resnet101, Resnet152, VGG16, VGG19, Inception_v4

V100 and is therefore applicable to other GPUs as well. In our experiments, we execute the LC application and BE application together to obtain their execution performance. We gather LC workloads using MLperf (Reddi et al. 2020), which is designed for the evaluation of inference applications. We use single-stream mode. This mode is a stream of inference queries of query size 1, which measures the response performance of the LC application. We select five inference tasks in MLperf as LC workloads, including three CV models and two NLP models.

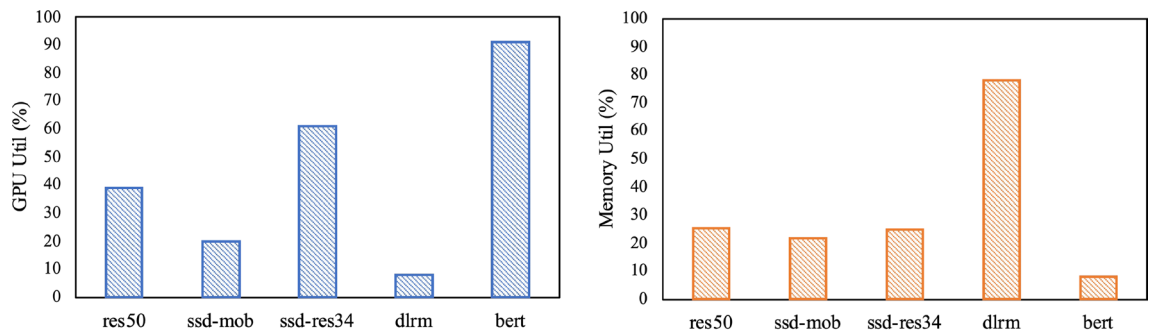
Table 1 shows the detailed model information. For the BE workloads, we selected the training tasks of the corresponding models. The specific network types are listed in Table 2.

3.3 Execute inference application on real systems

In this subsection, we explore the execution characteristics of inference tasks by executing a single inference task on GPU. In this experiment, we executed the inference application in MLperf on NVIDIA V100 and report its resource usage in Fig. 3. As can be seen from the figure, when a GPU is assigned exclusively to an inference application, the GPU utilization is relatively low, wasting a large amount of computational and storage resources. Moreover, for different applications, they have different occupancy of computational and storage resources. For example, the dlrm-based implementation of the recommended application has a fairly high demand for memory resources, while consuming almost no computational resources. In this work, we implement co-location of applications to improve resource utilization. In the meantime, we analyze the resource requirements of the applications and allocate appropriate resources for different applications.

3.4 MPS-based co-location execution

To explore the QoS violation due to co-location, we implement co-location execution of inference and training tasks using MPS. Similar to the previous work (Shen et al. 2019), the QoS target for the inference application is set to twice the latency of this application execute exclusively on GPU. Figure 4 shows the results of the experiment. In Fig. 4, the x-axis shows the co-located application pairs and the y-axis shows the 99%-ile latency of the inference application, normalized to the QoS target. Among these 30 co-located execution pairs, 12 sets of applications showed QoS violations. For example, When the resnet50-based classification and detection application and the vgg16-based neural style application are co-located, QoS violation will occur for the LC application. The main reason for QoS violation is the competition for resources between two applications. When



(a) GPU Utilization **(b)** Memory Utilization

Fig. 3 Time-division multiplexing and monitoring-adjustment

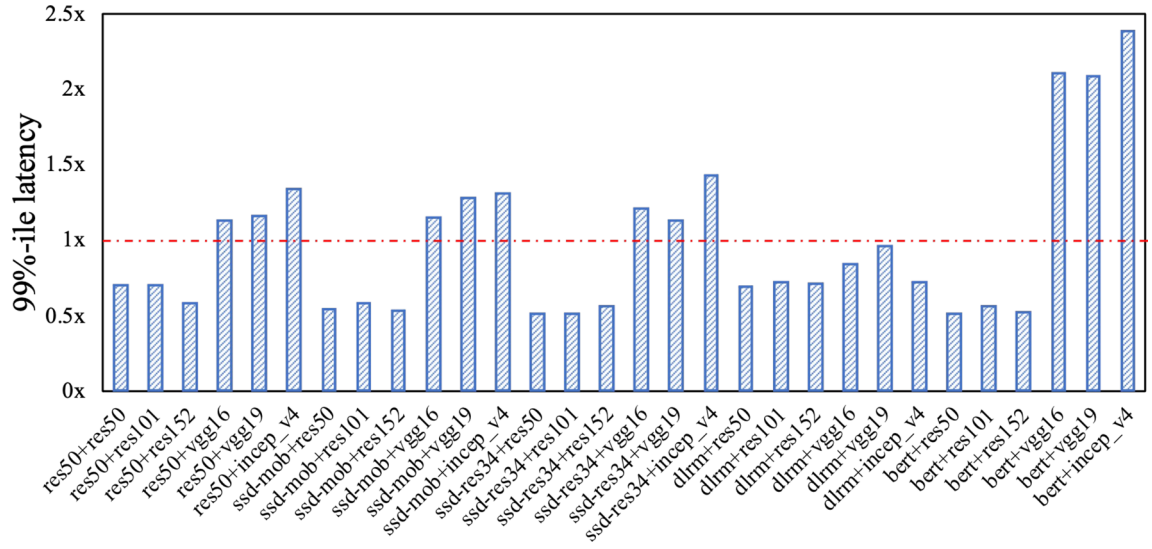


Fig. 4 The 99%-ile latency of LC applications normalized to the QoS target with MPS

the BE application and the LC application are co-located, the BE application competes for the computing resource and storage of the LC application, leading to QoS violation. As a result, we need to predict the resource requirements of the LC execution in advance to ensure that the LC application receives enough resources to guarantee its QoS target.

3.5 Challenges for applications co-location execution

The above experiments show that there exists severe wasting of resources when the application executes exclusively on the GPU. However, when using MPS-based GPU sharing mechanism to implement co-located execution of applications, it can cause serious QoS violations. These facts show that there are several key challenges in achieving QoS goals for applications while sharing GPU resources.

(1) **Applications have different network models and different resource requirements.** Different applications usually consist of different network models, and the operators that make up the models are therefore quite different. It is difficult to determine the resource requirements for each LC application.

(2) **Using MPS to implement co-location enforcement can cause serious QoS violations.** The main reason for the poor performance is the contention for resources, and the QoS of LC applications can be ensured by limiting the maximum resource request of the application. MPS can't manage computing resources and storage at a granular level based on the execution characteristics of the application, and a new approach is needed to place limits on the application's resource usage.

(3) **The QoS violation of LC application still occurs, due to the overhead of resource constraints.** To prevent QoS violations, the resources allocated to LC applications need to be dynamically adjusted to satisfy the resource requests of LC applications at different moments.

4 Design of ArkGPU

In this section, we present the basics of ArkGPU, which guarantees the QoS targets of LC applications while improving the resource utilization of GPU. To address above challenges, we adopt the following scheme.

- ArkGPU should include a performance predictor to predict the resource requirements of LC applications, based on the network model structure and workload, thus guaranteeing the QoS targets of the applications.
- ArkGPU should implement a new resource sharing scheme to achieve precise resource allocation and limits to prevent QoS violations.
- ArkGPU should be able to sense the interference of LC applications during execution, and if QoS targets cannot be guaranteed, they should be handled accordingly.

Figure 5 shows the overview of ArkGPU. It consists of a *two-level performance predictor*, a *two-level scheduler* that makes decisions based on the predicting results, a *resource limiter*, and a *performance adjustment unit*. The performance predictor consists of two levels, one in the control plane and the other in the work plane, which accurately predicts the maximum resource request and the corresponding

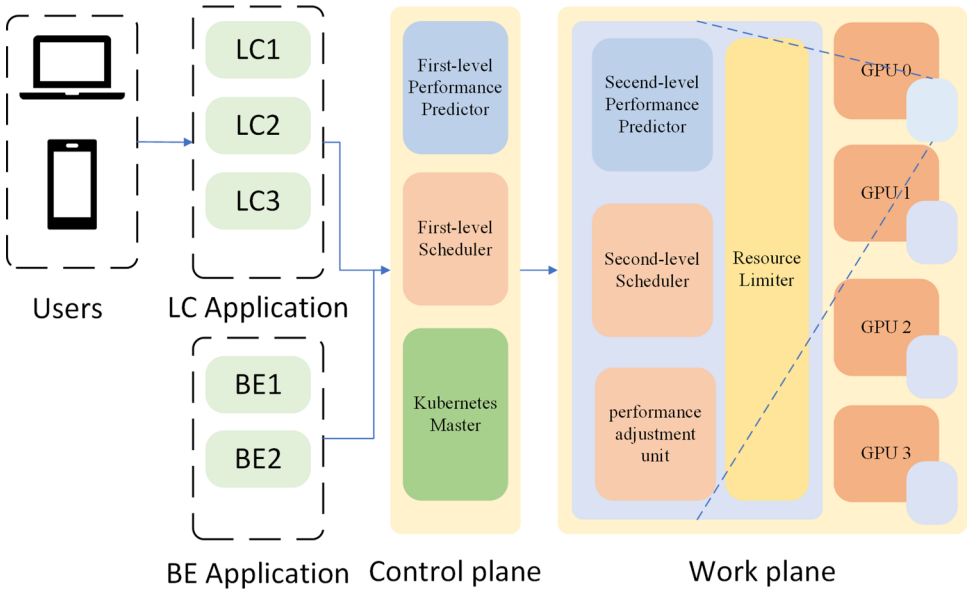


Fig. 5 Design overview of ArkGPU

processing time during task execution. The two-level scheduler receives the prediction results from the two-level predictor and assigns the appropriate host and GPU to the application. The resource limiter monitors the resource usage of each task and defers the task's API when resource requests exceed the limit. The performance adjustment unit monitors the execution progress of tasks and adjusts resource allocation in case of possible QoS violations.

As shown in Fig. 5, ArkGPU handles inference applications (LC applications) and training applications (BE applications), separately. Specifically, when ArkGPU receives a scheduling request from LC application, it performs the scheduling process in the following steps.

(1) ArkGPU predicts the maximum resource request for task q by receiving the network model structure and input size. Based on the predicting results, the GPU resources which are sufficient to meet the QoS goals are determined.

(2) The scheduler allocates GPUs for task q based on the predicting results. When performing the allocation, the first-level scheduler selects the host for q and the second-level scheduler selects the GPU for q based on the specifics of the GPU. When allocating resources, ArkGPU performs the allocation according to the maximum resource requirements based on the results of the predictor.

(3) The scheduler allocates the remaining resources for the BE application. During the allocation process, a second-level predictor predicts the execution time of the BE application on each GPU, aiming at maximizing the performance of the BE application. The resource limiter monitors both LC applications and BE applications running on the same GPU and limits the over-used applications.

(4) Due to the incomplete restriction, the BE application may overuse the resources and cause the slowdown of task q. ArkGPU checks the execution of q regularly, and if it is judged that it cannot meet the QoS, the performance adjustment unit will reallocate resources for the LC application. The reallocation of resources involves the suspension and restart of tasks, which can be achieved through the checkpoint mechanism of the deep learning framework.

4.1 Performance prediction

In order to obtain the execution of a deep learning application in advance, we train a performance prediction model. The model can predict the resource requirements and duration of the application.

Figure 6 shows a network model of a DNN application, which network structure can be described using a data flow graph (DFG). Each node in the DFG represents an operator that is processed in certain topological order. Rather than calling large functions, DNN applications are processed by executing operators one by one in sequence. During the execution, the two operators connected by the directed edge need to be executed in order. The basic set of operators is very small, and some operator libraries (e.g., cuBLAS (cuB zzz), OpenAI (Open aaaaa), etc.) implement GPU Kernels for common operators. Frameworks such as TensorFlow parse models written in Python and build DFGs from them, mapping the DFG nodes to kernel implementations in the operator libraries.

We use performance predictors to predict the maximum resource requirements of LC application and thus allocate sufficient resources to them. In order to build the

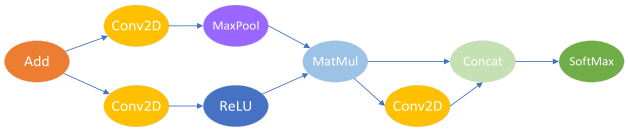


Fig. 6 Network model for a DNN application

performance prediction model, we need to gather the performance parameters of the application.

4.1.1 Impact of input size on resource utilization

Each operation in a deep learning application has to run on specific input data, so the size of the input data affects the resource utilization of the application. Figure 7 shows the variation of resource utilization for different input sizes. The input data is a square with side length n. The information of GPU utilization is obtained using the NVML API(Nvml bbbb). We implemented the Conv2D operator using TensorFlow, with a special setting since TensorFlow requests as much memory as possible by default. Obviously, the GPU resource usage depends on the size of the input data, and with the increase of input data size, the GPU utilization also increases.

We performed a similar analysis for other operations (e.g., ReLU, MaxPool, etc.), and the resource utilization all behaved as related to the input size. Except the size of the input data, resource utilization may also be related to other factors.

4.1.2 Impact of batch size on resource utilization

Usually, deep learning inference applications use batch processing to obtain a higher degree of parallelism. In a single execution, multiple queries are organized into the same batch and packaged for execution on the network. Figure 8 shows the Conv2D operator executed at different batch sizes. It can be seen that the batch size also affects the GPU resource utilization. As the batch size increases, more data needs to be processed during one execution of the neural network, which requires extra memory footprint and therefore increases the requested storage. The increase of batch size also requires extra computing resources, thus increasing GPU utilization as well.

4.1.3 Impact of operator types

We further analyze the way that the DNN model is processed to understand the impact of operators on application execution. The DNN network, as shown in Fig. 6, is executed by

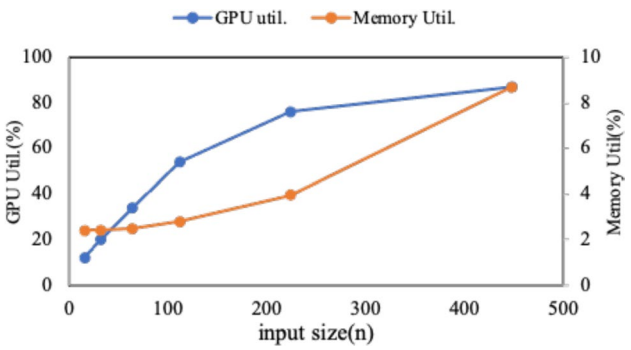


Fig. 7 The relationship between utilization and input size (Conv2D)

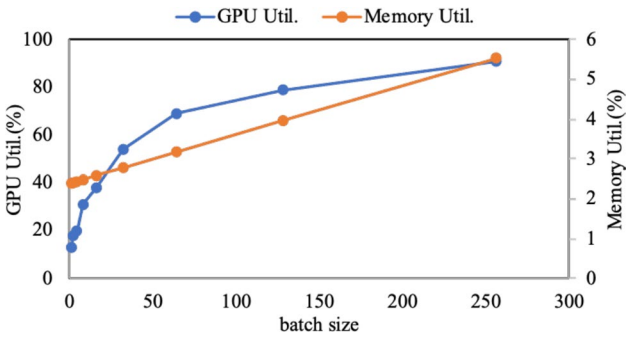


Fig. 8 The relationship between utilization and batch size (Conv2D)

executing each operator in topological order. The specific processing is shown in Fig. 9.

Under the circumstances, the resource usage of the whole application depends on the resource usage of each operator. That is, the maximum resource usage of the application should be the resource usage of the operator with the maximum resource requirements. Obviously, some operators that occupy fewer resources can be ignored. With reference to prior work (Hafeez and Gandhi 2020), we select operators with relatively higher resource requirements. We select following operators: Pad, AddV2, ConcatV2, Slice, AddN, BiasAdd, Mul, Relu, FusedBatchNormV3, L2Loss, Conv2D, MaxPool, AvgPool. If the same operator exists in the network model, the one with higher operations is kept. For the Bert-like model, we also used sequence length of execution as a parameter feature.

4.1.4 Predictive model implementation

For one application, we need to predict its GPU utilization (GPU util.) and memory utilization (Memory util.), where GPU utilization is a value between 0 and 100 using NVML (Nvml bbbb) queries.

We only have access to a small amount of information before the task is officially executed. For user-oriented inference applications, according to above analysis, their resource utilization is related to the input data size, batch size, and operator type. These parameters can be obtained before the model is executed. For each inference application to be executed, we use its input data size, batch size, and the operators mentioned above as parameters. If there is no operator of the corresponding kind in the network model, the value of this item is 0, otherwise, it is the input size of the operator. We implemented the prediction model using multilayer perceptron (MLP, which has 6 hidden

Fig. 9 Linear order of execution of network operators

layers with sizes 32, 64, 128, 128, 128, 128, 128 respectively) (Gardner and Dorling 1998), linear regression (LR) (Seber and Lee 2012) and decision tree (DT) (Myles et al. 2004). To obtain training and test data, we test a large number of neural networks and use NVML to collect and record the resource utilization of the networks. We collected 1000 samples using different inputs, batch sizes, and operators mentioned in Section 4.1.3, and randomly selected 800 for training and the rest 200 for evaluation and testing. We evaluate the prediction results using the test set on each of the three models, as shown in Fig. 10. This figure shows the prediction errors for the 13 networks (N1 to N13). The prediction error is calculated by Eq. 1.

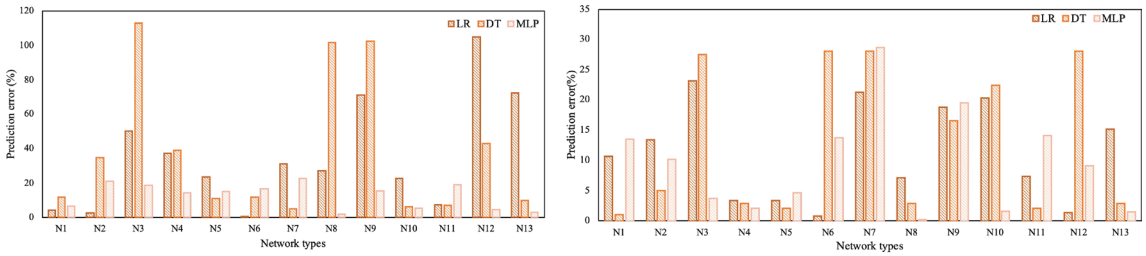
$$Error = \frac{\|Predicted\ value - Measured\ value\|}{Measured\ value}. \quad (1)$$

For the prediction of computing resources, we can find that the prediction accuracy of LR and DT is poor, while MLP has better results. For the prediction of storage, the prediction accuracy of the three is maintained at the same level. To further evaluate the differences among the models, we also tested the execution times of the LR, DT, and MLP models, respectively, as shown in Fig. 11. the time taken for LR and DT is around 1 ms, while MLP is longer than 3 ms. Considering above results, we use the **MLP** model to implement the prediction of computing resources, use the **LR** model to implement the prediction of storage.

When scheduling a BE application, the secondary scheduler needs to predict its execution time with a specified share of resources in order to select the most suitable GPU for the application. We used a similar approach as above to construct the prediction model. We collect the execution time of the network under different resource shares to extend the training set. Figure 12 shows the results of our predictions for the execution time. Since the accuracy of LR and DT is very poor, we only list the prediction results implemented using MLP. Based on our measurements, we chose the MLP model to construct our secondary predictor.

4.2 Scheduler design

The scheduler uses the results of the predictor to allocate execution resources for different kinds of applications. It can assign appropriate resources to the inference and training applications separately to improve resource utilization, guarantee the QoS of LC applications, and increase the



(a) Compute resource of the networks **(b)** Memory resource of the networks

Fig. 10 Errors of predicting the computing resources and storage

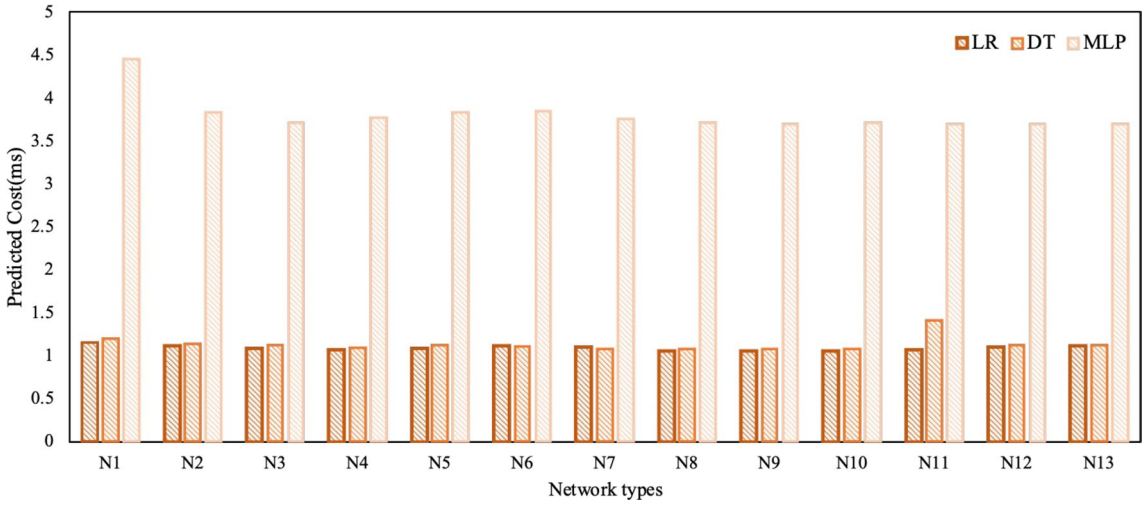


Fig. 11 Prediction overhead for predicting using LR, DT, and MLP

execution efficiency of BE applications as much as possible. The scheduler is a two-level architecture.

4.2.1 Scheduling of LC application

As shown in Fig. 5, when a user submits a task for an LC application (in this case, an inference application), ArkGPU accepts the prediction result from the first-level scheduler and allocates resources to the task according to the predicted resource request amount. The specific scheduling process is shown in Algorithm 1.

When the predictor predicts the maximum resources that a task may occupy, the first-level scheduler (line 1–8) selects a suitable host for it. Compute_{pre} and Memory_{pre} indicate the predicted computing resources occupancy and storage occupancy. The algorithm traverses all hosts in the cluster, determines whether the remaining resources of the hosts satisfy the application execution, and filters out the nodes that do not satisfy the QoS (line 1–4). If all the nodes do not meet the resource requirements, select a suitable node and release the resources of the BE task on the node (line 6–8). In order for the LC application to get enough resources, all candidate nodes are sorted and the node with the most remaining resources is selected (line 9–10). Subsequently, the scheduling process is taken over by the second-level scheduler (line 11–15). The scheduler holds a copy on each host and it traverses all the GPUs on the host, scoring each GPU (line 13). If there are not enough resources to satisfy the task execution, the score of this node is set to -1 (line 19–24). Otherwise, the score is positively correlated with the remaining resources and inversely correlated with the number of LC tasks executing simultaneously on current GPU (line 26). The scheduling algorithm selects the GPU node with the highest score and binds the task to the selected GPU (line 16).

Algorithm 1 Scheduling of LC jobs

Require: Compute_{pre}, Memory_{pre}
Ensure: GPU_k
1: **for** host in cluster[N] **do**
2: **if** isAvailable(host, Compute_{pre}, Memory_{pre}) **then**
3: All_Available_Hosts.append(host)
4: **end if**
5: **end for**
6: **if** All_Available_Hosts.isNULL() **then**
7: All_Available_Hosts(Release_Resource(cluster[N]))
8: **end if**
9: Host_List ← Sort_by_Resource(All_Available_Hosts)
10: Selected_Host ← Head(Host_List)
11: gpu_nodes[M] ← getAllGPU(Selected_Host)
12: **for** gpu_node in gpu_nodes[M] **do**
13: score[m] ← getScore(gpu_node, Compute_{pre}, Memory_{pre})
14: **end for**
15: GPU_List ← Sort_by_Score(gpu_nodes)
16: **return** Head(gpu_nodes)
17:
18: **function** GETSCORE(gpu_node, Compute_{pre}, Memory_{pre})
19: **if** getCompute(gpu_node) <Compute_{pre} **then**
20: **return** -1
21: **end if**
22: **if** getMemory(gpu_node) <Memory_{pre} **then**
23: **return** -1
24: **end if**
25: Num_of_LC ← getLCNum(gpu_node)
26: **return** (getCompute(gpu_node) + getMemory(gpu_node)) / Num_of_LC
27: **end function**

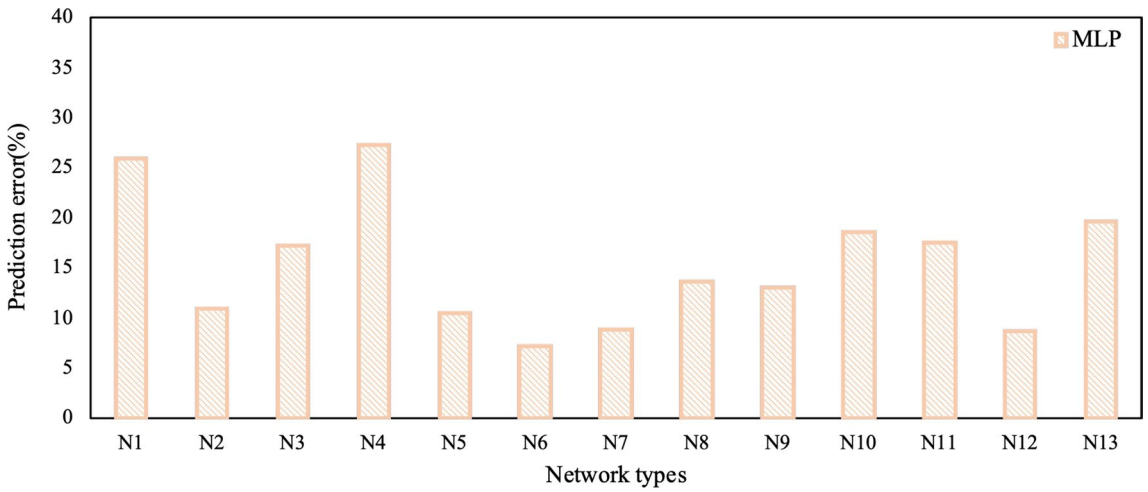


Fig. 12 Errors of predicting the execution time

4.2.2 Scheduling of LC applications

After completing the resource allocation for the LC application, we then consider allocating the remaining resources for the BE application. When allocating resources for BE applications, ArkGPU wants to achieve optimal execution performance for the BE tasks, while avoiding QoS violations for LC tasks. To achieve this goal, we also use the prediction results of the predictor. When a BE application is submitted, it first enters the waiting queue and waits until there are no LC applications to process and the system has free resources before executing the scheduling process. The scheduling flow of the BE application is shown in Algorithm 2. Compute_{pre} and Memory_{pre} request are the resources that need to be used by the BE application, and Q is the characteristic parameter of the BE application. In the first-level scheduler, when the available resources are not enough, the scheduling is directly exited and the task returns to the waiting queue (line 7). To improve resource utilization, the node with the least resources that meets the resource requirements is selected (line 9–10). In the second-level scheduler, the getPredict() function calls the second-level predictor to predict the execution time of the application under this resource configuration (line 13). To maximize the performance of BE application, ArkGPU will choose the GPU with the least amount of execution time (line 15–16).

Algorithm 2 Scheduling of BE jobs

Require: Compute_{pre}, Memory_{pre}, Q
Ensure: GPU_k
1: **for** host in cluster[N] **do**
2: **if** isAvailable(host, Compute_{pre}, Memory_{pre}) **then**
3: All_Available_Hosts.append(host)
4: **end if**
5: **end for**
6: **if** All_Available_Hosts.isNULL() **then**
7: **return**
8: **end if**
9: Host_List ← Sort_by_Resource(All_Available_Hosts)
10: Selected_Host ← Head(Host_List)
11: gpu_nodes[M] ← getAllGPU(Selected_Host)
12: **for** gpu_node in gpu_nodes[M] **do**
13: score[m] ← getPredict(Q, gpu_node)
14: **end for**
15: GPU_List ← Sort_by_Score(gpu_nodes)
16: **return** Head(gpu_nodes)

4.3 Limits and adjustments of GPU resources

4.3.1 Resource limiter

After allocating resources to tasks, it is necessary to limit their resource usage, so as to realize the sharing of GPU resources among multiple tasks. However, on the cloud environment, NVIDIA does not provide a shared interface other than MPS, which has a series of problems. We implement GPU sharing using a monitoring-adjustment scheme. The resource limiter uses the API provided by the NVML library to query the state of GPU and obtain the processes executing on the GPU and their resource utilization.

We choose the API interception solution to implement resource limiter, which is a GPU vendor-independent solution. The API isolation layer generally sits on top of the CUDA driver API layer and replaces the CUDA library through the LD_LIBRARY_PATH mechanism. Using LD_LIBRARY_PATH, the relevant CUDA APIs can be intercepted to achieve resource restriction. When the GPU program calls the relevant API, it will load our own API implementation. Before calling the API, the resource limiter will check whether current program is using more computing resources than the limit. If the limit is not exceeded, it will continue to execute the API call. If not, *nonosleep()* is called to suspend the execution of the API. The role of *nanosleep()* is to make the current task suspend for a period of time, during which its usage of GPU will be reduced, thus achieving the purpose of limiting resource usage. After that, the limiter polls the GPU for computing resource usage, and once the computing resource usage is below the limit, the API call will be executed. The specific restriction method is shown in Fig. 13 (a).

When an application consumes much more resources than the limit, no extra resources will be allocated for it. To further improve task execution efficiency, current application is allowed to overuse resources when there is no other application on the same GPU. When another application is scheduled to that GPU, the over-allocated resources will be reclaimed. A formal description of this process is given in Fig. 13(b).

4.3.2 Resource adjustment

Since co-located applications may compete for resources, the LC application needs to be dynamically adjusted to prevent QoS violations. When the LC application is executed, the current execution time is recorded for each query executed. Since the neural network structure and inputs are the same for each query, it can be approximated that the time is the same for each query. Suppose there are N tasks in total, n tasks are currently executed, and the execution time is t. Then the predicted execution time t' is $\frac{N}{n}t$. If t' is larger than

```
cuMyAPI():
JUDGE:
  if GPUused > GPUconfig:
    nanosleep(5)
    goto JUDGE
  cuAPI();
```

(a)

```
DynamicAllocation():
  While (1) :
    if numofthread(GPU) == 1:
      GPUcanbeused = 100
    else
      GPUcanbeused = GPUconfig
```

(b)

Fig. 13 Method of limiting GPU resource usage

the QoS target, it means that the first n queries are slower than expected and may generate a QoS violation. Once this happens, the resources of the BE application that is co-located and executed with the LC application are released.

Since the deep learning framework has a checkpointing mechanism, it is possible to save the current states when BE application exits. When the system resources are sufficient, the released BE application is restarted and its execution continues from the checkpoint. While if there is only one LC application on the GPU after the BE application exits, the LC application will be automatically expanded. If there are other LC applications present, the resource share of the process where the QoS violation is going to occur is modified in the resource limiter (That is, GPU_{config} in Fig. 13 and additional resources are allocated for it. In this way, we can guarantee that LC applications will not have QoS violations.

5 Evaluation

5.1 Experimental setup

We have implemented ArkGPU on native Kubernetes. Our experiments were conducted on a machine equipped with 4 NVIDIA V100, and the specific configurations are listed in Table 3. As shown in Table 1, we choose 5 applications from MLperf inference as LC applications. We selected 6 DNN training models, which are widely used in the image field and natural language processing field, as BE applications, shown in Table 2. In our experiments, we describe the quality of service using a 99%-ile delay and define the QoS for co-located execution twice as high as when executed alone.

The evaluations are performed using the single-stream of MLPerf inference, which is a stream of inferring queries with query size 1, reflecting the response timeliness of multi-clients applications. It gives various performance metrics, including 99%-ile delay, which is the upper bound of delay satisfying 99% of queries. The multi-stream contains a series of queries, but each query contains multiple inferences. The multi-stream is primarily used to measure the maximum number of streams supported by the system for a given QoS target and is therefore not used. Offline represents a batch application where all data is immediately available, and latency is not limited. Offline is mainly used to test throughput and is not a suitable benchmark for LC queries. Given this, we choose single-stream.

5.2 QoS and throughput

We evaluated the effectiveness of ArkGPU in improving GPU utilization while meeting the QoS for LC applications. Figure 14 shows the 99%-ile latency when the LC application and the BE application are co-located for execution. We used a total of $5 \times 6 = 30$ co-location pairs to compare the normalized 99%-ile latency using ArkGPU and MPS (Non-ArkGPU) as comparison. As can be seen in Fig. 14, the co-location pairs using ArkGPU all ensure user-oriented QoS requirements. We also measure the average latency of user-oriented queries and find that they all meet the QoS target when using ArkGPU. The results are shown in Fig. 15. Here, 99%-ile latency is the latency at the 99th percentile of a set of queries, i.e., 99% of all queries have latency less than this value, and it measures how well the system performs at high latencies that are rarely seen. The average latency is the arithmetic average of all query latencies and indicates the general condition of the system.

However, resource sharing using MPS can result in a 2.5x QoS violation in some cases. When the resource request of LC applications or BE applications increases, the scheduling using MPS can't be aware of their resource usage, which causes resource contention and leads to severe QoS violations. By limiting the resource occupation of the BE application, we can ensure the QoS of the LC application, and the above experiments prove this conclusion. As shown in Figs. 14 and 15, QoS violations do not occur using ArkGPU. Nevertheless, some LC applications in ArkGPU have slightly higher latency than applications using MPS (e.g., ssd-mob+res101). This is because the process of monitoring and adjusting causes some performance loss.

To further determine the impact of ArkGPU on application performance, we also tested the Queries-per-second(QPS) during tasks execution, and the results are shown in Fig. 16. We conduct separate experiments on k8s-native, on system using MPS, and on system using ArkGPU. It can be found that in some cases (e.g., when the LC

Table 3 Hardware and software specifications

	Specification
Hardware	Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz
	NVIDIA Tesla V100
Software	CentOS Linux release 7.9.2009 (Core)
	CUDA Driver Version: 460.106.00
	CUDA Version: 11.1
	CUDNN 7.6.5

application co-locates with VGG16 or VGG19), the QPS of the LC application increases significantly, which is because those BE training applications consume a large amount of GPU resources and competes with LC applications for resources. In ArkGPU, we limit BE applications' resource usage, which can improve the QPS of LC applications. The QPS of the LC application is more than doubled on average compared to the scheme using MPS. In some other cases, ArkGPU has some performance loss compared to the solution using MPS. For example, when the BE application (such as dlrm) requests fewer GPU computing resources, the LC application can get enough resources. At this point, the reduction in QPS caused by the resource limiter is typically around 5%. We consider this loss to be acceptable.

As shown in Fig. 17, the co-location execution of applications increased GPU utilization. The GPU utilization with MPS increased by an average of 36.1%, and ArkGPU was building on this by a further increase of 11.5%. Due to the reduction in resource contention, the application can make full use of the GPU's resources.

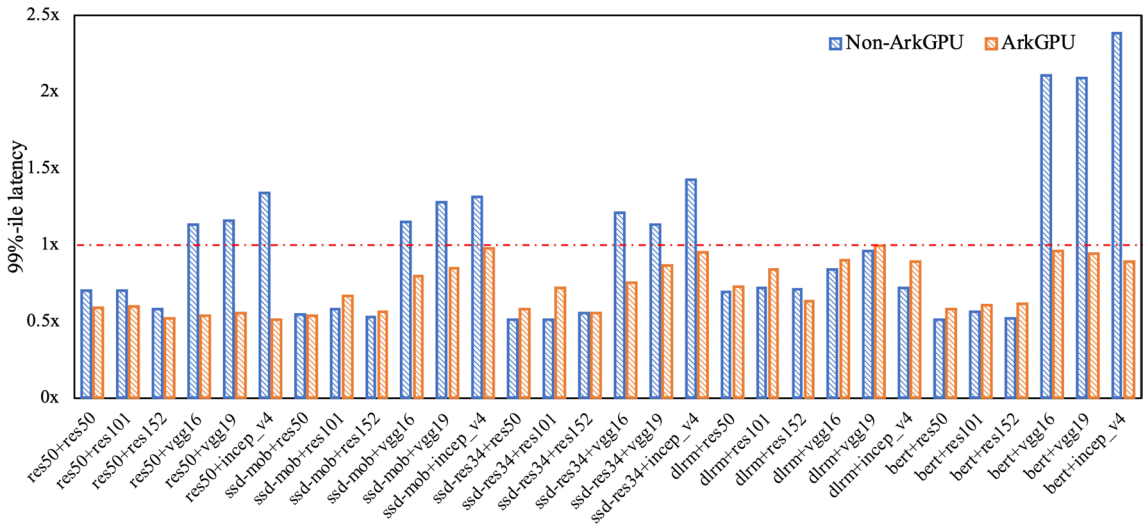


Fig. 14 99%-ile latency for each co-location pair with Non-ArkGPU and ArkGPU

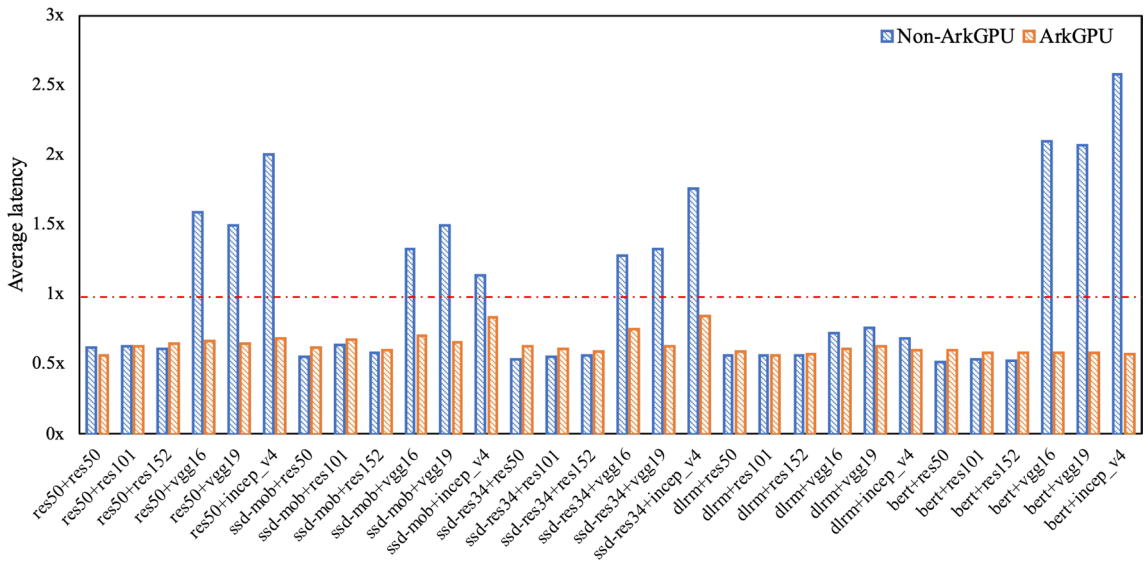


Fig. 15 Average latency for each co-location pair with Non-ArkGPU and ArkGPU

5.3 High-goodput

Our ArkGPU implements High-Goodput computing for the evaluation of throughput and QoS, proposed by Qiao et al. [29]. They use goodput to describe the quality of tasks completed per unit of time. The definition of goodput is described using Eq. 2.

$$Goodput = throughput \times yield. \tag{2}$$

We use the QPS of the query to describe the throughput, and the yield is expressed with the proportion of queries that meet the QoS target. We counted the proportion of queries that met the QoS target in the total query and showed the results in Fig. 18. Then we measured the goodput during co-located applications execution, and the results were shown in Fig. 19.

In most cases, High-Goodput can be achieved using ArkGPU, which improves the query execution yield while maintaining almost the same QPS as the k8s-native, thus achieving better goodput compared to the MPS case. Our experiments demonstrate that ArkGPU can effectively improve the goodput of the cluster. Classification and detection applications based on the resnet50, ssd-mobilenet, and ssd-resnet34 networks are improved by an average of 111.21%, 200.86%, and 759.96% over MPS using ArkGPU, respectively. ArkGPU achieves High-Goodput for these networks by improving the QPS and yield of queries. For the recommendation application, the use of ArkGPU improves only 9.62% over the use of MPS. This is because the recommendation application requests fewer GPU computing resources, which doesn't cause serious resource competition even when using MPS for resource sharing. For the NLP application using Bert model, using ArkGPU improves 1839.70% goodput over using MPS. Such a huge

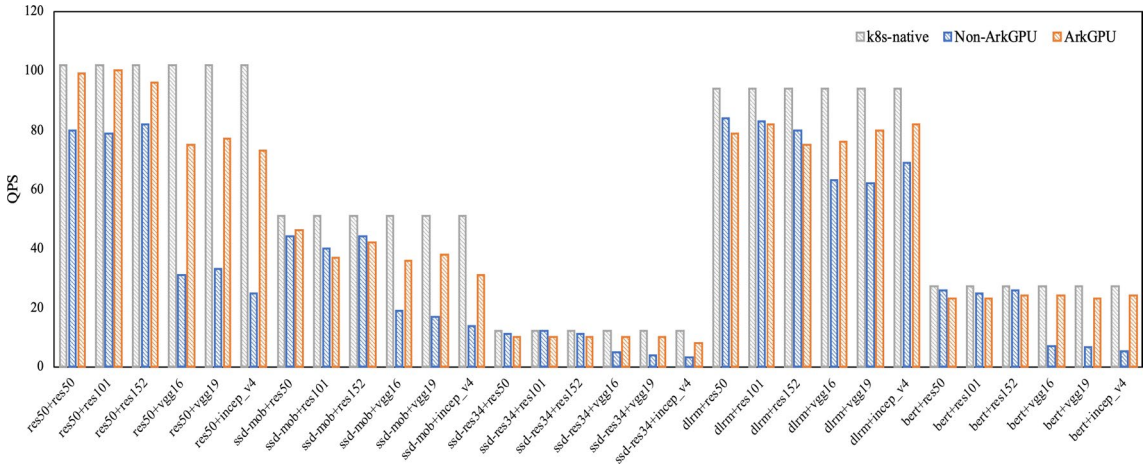


Fig. 16 QPS for each co-location pair with Non-ArkGPU and ArkGPU

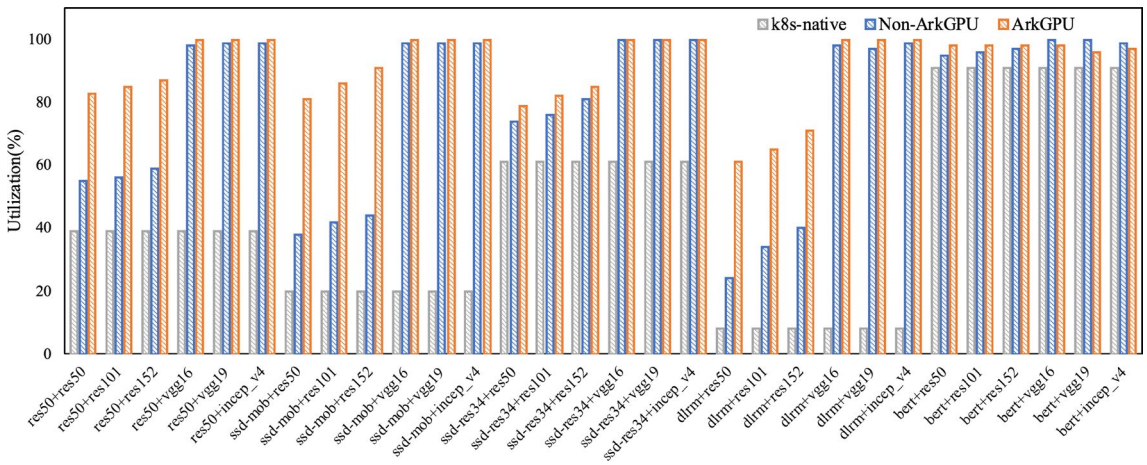


Fig. 17 GPU Utilization for each co-location pair with K8s-native, Non-ArkGPU and ArkGPU

improvement is due to the fact that Bert requires the use of a large amount of computing resources. Without the combined effect of ArkGPU's predictor and resource limiter, a rather severe resource competition would occur, resulting in a query yield close to 0. Since ArkGPU achieves a better resource limitation, we improve the query yield and achieve a balance between QPS and yield.

To further test the representation of High-Goodput on ArkGPU, we deploy multiple LC and BE applications simultaneously and test their performance under different execution environments. Based on the definition of goodput mentioned above, the overall system goodput can be defined as the sum of the system's throughput within a time slice multiplied by the proportion of queries that satisfy the latency requirement. The goodput of the system is calculated as shown in Equation 3, where n is the number of LC applications in this time slice.

$$Goodput = \sum_{i}^{n} throughput_i \times yield_i. \tag{3}$$

We use the number of queries within a time slice to represent the throughput of the application, and the yield is expressed with the proportion of queries in the task that meet the QoS target. We submit four BE applications and four LC applications to the cluster with four NVIDIA V100s and calculate their goodputs. We combine the applications in Table 1 into five groups of app-mixes, and the execution results are shown in Fig. 20, where the horizontal coordinates are the five groups of app-mixes. The specific configuration is shown in Table 4. The figure plots goodput for all app-mixes of LC and BE applications for the three configurations.

It can be noticed that the goodput of the system is improved by allowing resource sharing. There are still

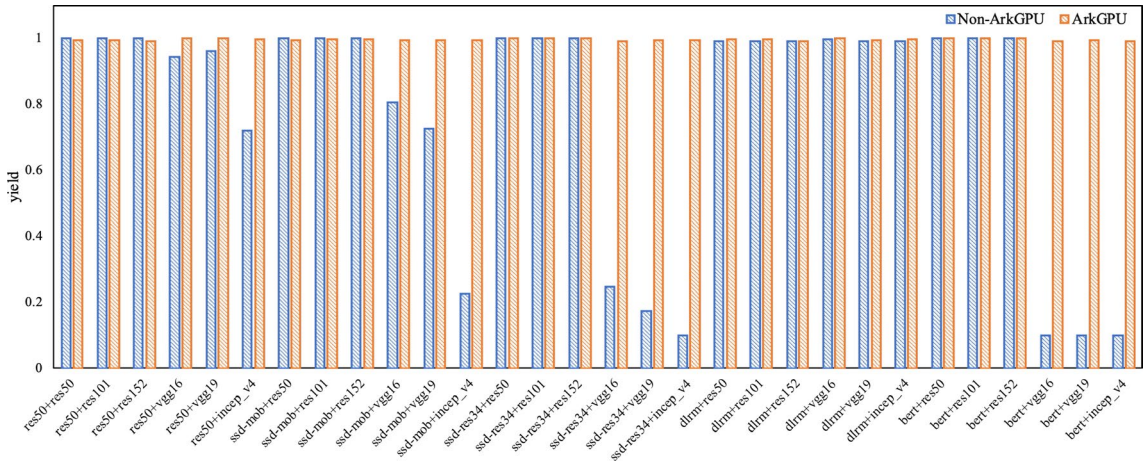


Fig. 18 Yield for each co-location pair with Non-ArkGPU and ArkGPU

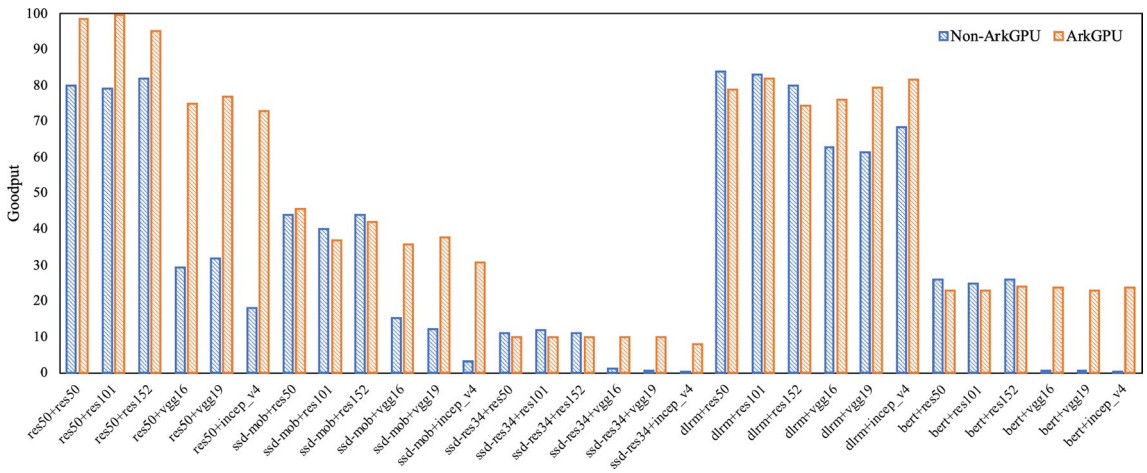


Fig. 19 Goodput for each co-location pair with Non-ArkGPU and ArkGPU

special cases, for example, the goodput using MPS sharing in app-mix 1 is lower than that of k8s-native. K8s-native brings low yield because some LC applications can't execute on time due to the lack of support for resource sharing. MPS can support resource sharing, but frequent resource contention may reduce the throughput and yield of queries thus bring poor goodput. As can be seen from the figure, the goodput of MPS is lower in app-mix 1. Considering all the cases, the goodput of ArkGPU is improved by 94.98% on average compared to k8s-native and 38.65% on average compared to MPS.

5.4 Effectiveness of ArkGPU

The core of the monitoring and adjusting scheme is a moratorium when the requested resources of a task exceed the limit. This limiting approach can cause a lag in tuning and thus affect the efficiency of task execution. Given these things, we have designed the performance adjustment unit. In this section, we paused the work on the adjustment unit and reran the test as shown in Fig. 21. We conduct experiments on system using MPS, GaiaGPU, and ArkGPU, respectively.

In GaiaGPU, QoS violation occurs for some co-location pairs. Among the 30 co-location pairs in GaiaGPU, 9 pairs are affected by QoS violation. For example, when ssd-mobilenet and resnet50 co-location is executed, a QoS violation of 1.76x is reached. Using MPS does not cause intolerable performance impairment when co-located applications take up fewer resources and the accelerator has sufficient remaining resources. In our example, the normalized 99%-ile latency using MPS (Non-ArkGPU) is only 0.54x. At this point, the dynamic adjustment of the resource limiter becomes the bottleneck of the whole system, and different applications are frequently tuned during execution, which leads to serious QoS conflicts. Under the circumstances, the resource limiter will not improve the execution performance of the application, and it could actually cause serious performance disruptions. Therefore, the optimization implemented by ArkGPU is effective. Compared to GaiaGPU, the standardized 99%-ile delay is dropped about 0.55x, meeting QoS targets. As can be seen from the Fig. 21, no more QoS violations occur.

Table 4 Configuration of 5 app-mixes

app-mix	Configuration
app-mix1	res50+ssd-mob+ssd-res34+bert
	res50+res101+vgg16+vgg19
app-mix2	res50+ssd-mob+dlrm+bert
	res50+res152+vgg16+incep$_{v4}$
app-mix3	res50+ssd-mob+ssd-res34+dlrm
	res50+res152+vgg16+vgg19
app-mix4	ssd-mob+res34+dlrm+bert
	res50+res101+vgg16+vgg19
app-mix5	res50+ssd-res34+dlrm+bert
	res101+res152+vgg16+vgg19

5.5 Overhead analysis

The overhead of ArkGPU mainly includes **online prediction** and **resource limitation**. During scheduling, each LC application needs to predict its resource usage. We have measured that this delay is usually within 5ms, which is

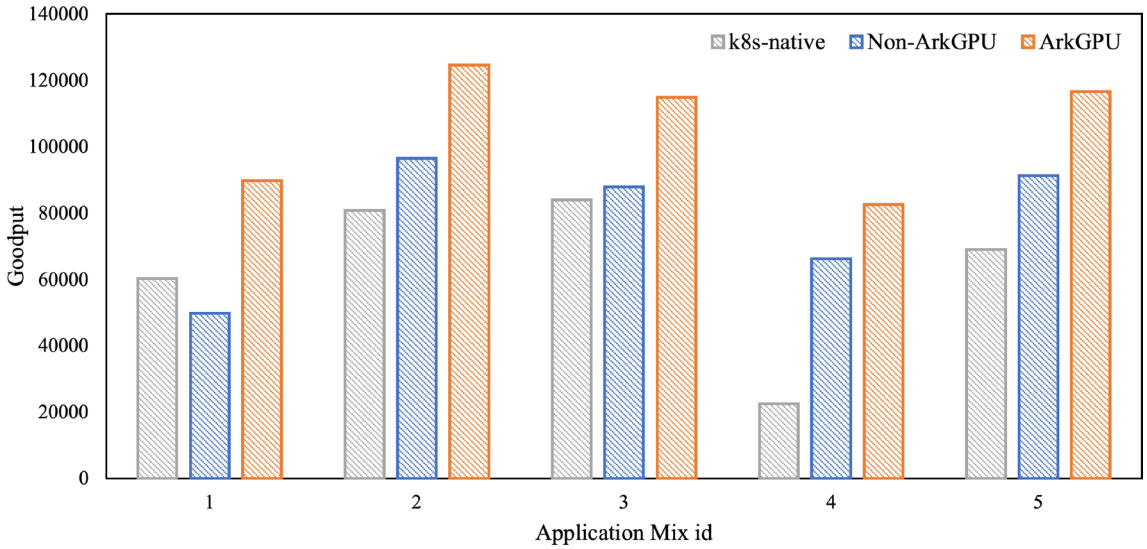


Fig. 20 Goodput for each app-mix with k8s-native, Non-ArkGPU and ArkGPU

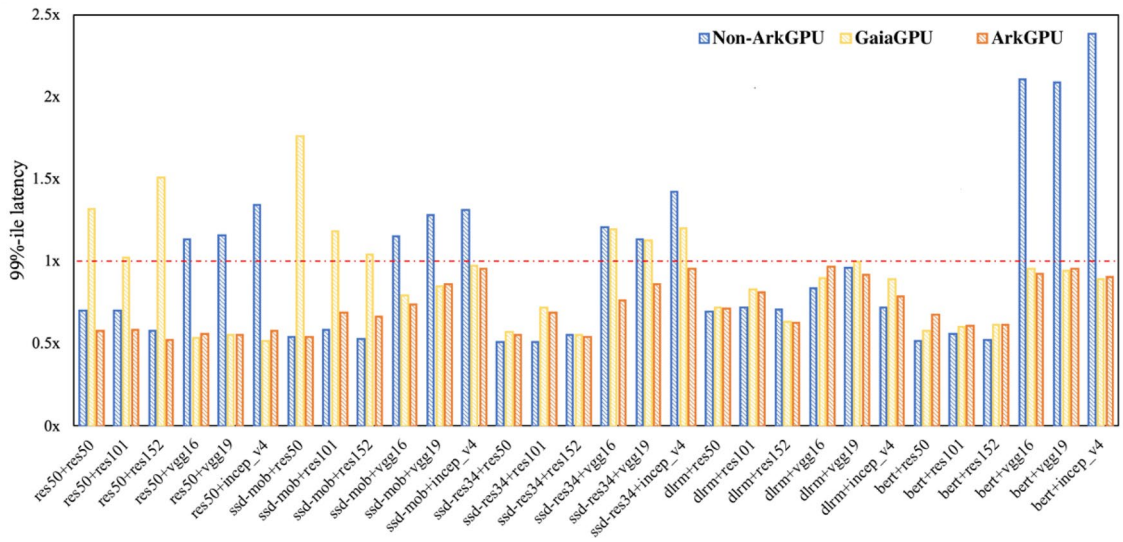


Fig. 21 99%-ile latency for each co-location pair with Non-ArkGPU, GaiaGPU and ArkGPU

acceptable. For BE applications, all GPUs on the host need to be traversed in the secondary scheduler. For a host with N GPUs, the complexity is O(N). Generally, there are not too many GPUs on one host. According to the analysis in Section 4.1.3, this delay is usually within 20 ms.

The overhead during application execution comes from the resource limiter. It continuously detects the application in execution and adjusts its resource usage. So we added tuning mechanisms to allocate more resources to applications when performance is affected.

5.6 Discussion and future work

We implement ArkGPU on Kubernetes, and the implementation relies on Kubernetes' scheduler extensions and device plugin mechanism. One possible effort is to make ArkGPU standalone so that it can run on universal cloud platforms. During the execution of the prediction, we predict the maximum resources used by the application, which somewhat causes a waste of resources. Since the resource usage of deep learning applications follows a certain pattern, we would like to implement a dynamic prediction of resource utilization in order to limit resources more accurately.

6 Conclusion

We propose ArkGPU, which improves the throughput of the cluster, ensures High-Goodput and guarantees that the services co-located on a single GPU can meet QoS requirements. To achieve this goal, ArkGPU predicts the resource utilization of LC applications and supports LC and BE applications share the resources of the same GPU in order to improve the system throughput while guaranteeing the goodput on the system. It is experimentally proved that ArkGPU achieves High-Goodput. Compared to the shared solution using MPS, the goodput of co-located pairs implemented using ArkGPU is improved by 584.27% on average. We deploy multiple applications simultaneously on ArkGPU, in this case, goodput is improved by 94.98% compared to k8s native and 38.65% compared to MPS. As a result, ArkGPU achieves High-Goodput.

Acknowledgements This work was supported by National Key Research and Development Program (Grant No. 2022YFB4501404), the Beijing Natural Science Foundation (4232036), CAS Project for Youth Innovation Promotion Association.

Availability of data and materials The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declarations

Conflict of interest No potential conflict of interest was reported by the authors

References

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (2016)

Chen, Q., Yang, H., Mars, J., Tang, L.: Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. ACM SIGPLAN Notices **51**(4), 681–696 (2016)

Chen, S., Delimitrou, C., Martínez, J.F.: Parties: Qos-aware resource partitioning for multiple interactive services. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 107–120 (2019)

cuBLAS. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed 25 Dec 2022

Duato, J., Igual, F.D., Mayo, R., Pena, A.J., Quintana-Ortí, E.S., Silla, F.: An efficient implementation of gpu virtualization in high performance clusters. In: European Conference on Parallel Processing, pp. 385–394 (2009). Springer

Gardner, M.W., Dorling, S.: Artificial neural networks (the multilayer perceptron)-a review of applications in the atmospheric sciences. Atmos. Environ. **32**(14–15), 2627–2636 (1998)

Gu, J., Song, S., Li, Y., Luo, H.: Gaiagpu: sharing gpus in container clouds. In: 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), pp. 469–476 (2018). IEEE

Hafeez, U.U., Gandhi, A.: Empirical analysis and modeling of compute times of cnn operations on aws cloud. In: 2020 IEEE International Symposium on Workload Characterization (IISWC), pp. 181–192 (2020). IEEE

Li, J., Xu, H., Zhu, Y., Liu, Z., Guo, C., Wang, C.: Aryl: an Elastic Cluster Scheduler for Deep Learning. arXiv (2022). https://arxiv.org/abs/2202.07896

Mars, J., Tang, L., Hundt, R., Skadron, K., Soffa, M.L.: Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 248–259 (2011)

Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed 25 Dec 2022

Myles, A.J., Feudale, R.N., Liu, Y., Woody, N.A., Brown, S.D.: An introduction to decision tree modeling. J. Chemometrics **18**(6), 275–285 (2004)

NVIDIA MIG. https://www.nvidia.cn/technologies/multi-instance-gpu/. Accessed 25 Dec 2022

Nvml-api. https://docs.nvidia.com/deploy/nvml-api/index.html. Accessed 25 Dec 2022

OpenAI. https://openai.com/. Accessed 25 Dec 2022

Patel, T., Tiwari, D.: Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 193–206 (2020). IEEE

Reddi, V.J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al.: Mlperf inference benchmark. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 446–459 (2020). IEEE

Seber, G.A., Lee, A.J.: Linear regression analysis. Wiley, Hoboken (2012)

Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., Sundaram, R.: Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 322–337 (2019)

Thinakaran, P., Gunasekaran, J.R., Sharma, B., Kandemir, M.T., Das, C.R.: Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER), pp. 1–13 (2019). 10.1109/CLUSTER.2019.8891040

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al.: Gandiva: Introspective cluster scheduling for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 595–610 (2018)

Xu Z W, L.G.J., H, S.N.: Superbahn: Towards new type of cyberinfrastructure. Bull. Chin. Acad. Sci. 37(1), 46–52 (2022)

Yang, H., Breslow, A., Mars, J., Tang, L.: Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. ACM SIGARCH Comput. Architecture News **41**(3), 607–618 (2013)

Yeh, T.-A., Chen, H.-H., Chou, J.: Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. In: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, pp. 173–184 (2020)

Zhang, Y., Laurenzano, M.A., Mars, J., Tang, L.: Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 406–418 (2014). IEEE

Zhang, W., Chen, Q., Zheng, N., Cui, W., Fu, K., Guo, M.: Towards qos-awareness and improved utilization of spatial multitasking gpus. IEEE Trans. Comput. **71**(4), 866–879 (2022)

Zhao, W., Chen, Q., Lin, H., Zhang, J., Leng, J., Li, C., Zheng, W., Li, L., Guo, M.: Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 653–663 (2019). IEEE

Zhu, H., Erez, M.: Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In: Proceedings of the Twenty-first International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 33–47 (2016)

Jie Lou born in 1975. PhD candidate. Member of CCF. His main research interests include high throughput computing architecture and cloud computing.

Yiming Sun born in 1997. PhD candidate. Member of CCF. His main research interests include cloud computing and high throughput computing architecture.

Jie Zhang born in 1999. She is currently pursuing the master's degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences. Her main research interests include high throughput computing architecture and paralleling computing.

Yuan Zhang born in 1990. PhD candidate. Member of CCF. Her main research interests include high throughput computing architecture and parallel computing.

Huawei Cao born in 1989. PhD. Member of CCF. His main research interests include parallel computing and high throughput computing architecture.

Ninghui Sun born in 1968. PhD. Member of CCF. He is the architect and main designer of Dawning2000, Dawning3000, Dawning4000, and Dawning5000 high-performance computers. His major research interests include computer architecture, operating system.