Cluster Computing (2023) 26:2577–2589
https://doi.org/10.1007/s10586-021-03299-z

Interference-aware execution framework with Co-scheML on GPU clusters

Sejin Kim[1] · Yoonhee Kim[1]

Received: 4 January 2021 / Revised: 7 March 2021 / Accepted: 3 May 2021 / Published online: 18 May 2021

Abstract
Recently, improving the overall resource utilization through efficient scheduling of applications on graphic processing unit (GPU) clusters has been a concern. Traditional cluster-orchestration platforms providing GPUs exclusively for applications constrain high resource utilization. Co-execution of GPU applications is suggested to utilize limited resources. However, the co-execution of GPU applications without considering their diverse characteristics can lead to their unpredictable performances owing to interference resulting from contention and unbalanced usage of resources among applications. This paper proposes an interference-aware execution framework with Co-scheML for various GPU applications such as high performance computing (HPC), deep learning (DL) training, and DL inference. Various resource-usage characteristics of GPU applications are analyzed and profiled to identify various degrees of their application interference. As interference prediction is challenging owing to the complexity of GPU systems, an interference model is generated by applying defined GPU metrics to machine learning (ML) models. A Co-scheML scheduler deploys applications to minimize the interference using the predicted interference from the constructed model. Experimental results of our framework demonstrated that the resource utilization improved by 24%, the average job completion time (JCT) improved by 23%, and the makespan shortened by 22% on average, compared to baseline schedulers.

Keywords GPU applications · Interference · Co-execution · Co-ScheML scheduler · Resource contention · GPU utilization

1 Introduction

With the recent increase in popularity of graphic processing units (GPUs), platforms and architectures using GPUs face challenges related to their optimization, application performance, and system throughput. The existing schedulers of cluster-orchestration platforms, such as YARN [34] and Kubernetes [16], might not fully utilize GPU resources when a single application is executed on a GPU. Co-execution of multiple applications that have dynamic resource-usage patterns is proposed to solve this limited use of GPU.

To achieve general purpose GPU (GPGPU) sharing, NVI DIA proposed the multiple process service (MPS) for concurrent execution of multiple kernels [22]. Performance of simple leftover strategy in MPS may vary according to co-executed kernel. Previous studies on GPU sharing include concurrent deployment of several applications using user requirements, monitor information [3, 9, 27], and profile information [11]. However, these studies do not consider interference during resources sharing, leading to performance degradation. Moreover, interference minimization using resource-usage profiles is challenging because of the complexity of GPU systems.

Recent considerations of interference problems addressed resource contention among applications caused by resource sharing [8, 30, 33]. Mystic [30] defined certain

Both authors have contributed equally to the work.

A preliminary version [15] of this article was presented at the 1st IEEE International Conference on Autonomic Computing and Self-Organizing Systems, Washington, DC, USA, August 2020.

✉ Yoonhee Kim
yulan@sookmyung.ac.kr

Sejin Kim
wonder960702@gmail.com

[1] Department of Computer Science, Sookmyung Women's University, Seoul, South Korea

resource performance metrics that can lead to contention and proposed an interference avoidance scheduler with accumulated profiling of the metrics. Nevertheless, an execution failure owing to out of memory (OOM) may occur as the profiling metrics are limited. Xu et al. [33] selected features for modeling using ML techniques leveraging observations of a co-located virtual machine (VM). As Jiang et al. focuses on the intrinsic characteristics of the NVIDIA vGPU, it cannot be applied to a cluster or bare-metal environment. Geng et al. [8] defined different factors that affect interference on clusters at node levels and scheduled applications by applying different ML models at each level. However, these studies included experiments on only DL workloads with uniform resource-usage patterns.

This paper introduces an interference-aware execution framework using Co-scheML [14] to reduce the completion time of workloads and maximize resource utilization for applications running on GPUs. To overcome interference problems caused by GPU sharing, we analyze the resource-usage characteristics of applications in various domains. Based on that observation, we define appropriate metrics representing the characteristics of applications and propose a framework integrating Co-scheML [14] with an interference-prediction model.

In summary, we make following contributions :

- The differences in resource-usage characteristics (GPU, GPU memory, and PCIe) of GPU applications (HPC, DL training, and DL inference) in a GPU cluster are identified. These may cause diverse interference effects depending on the pairing applications. The necessity of profiling is explained.
- The interference among GPU applications is produced by ML modeling using accumulated GPU application profiling data. The interference-prediction modeling uses random-forest regression to predict the degree of interference.
- The interference-prediction model and Co-scheML constitute our scheduling framework. Co-scheML is an interference aware scheduler that decides application deployment depending on the degree of interference. We implemented our framework in Kubernetes and conducted experiments with various workloads.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation of the study. In Sect. 3, resource-usage characteristics of multiple GPU applications are analyzed, and an interference problem that can possibly occur during GPU sharing is identified. Interference modeling is described in Sect. 4. The execution framework using Co-scheML is explained in Sect. 5, and its experiments are described in Sect. 6. The related studies are provided in Sect. 7, and concluding remarks are presented in Sect. 8.

2 Background and motivation

2.1 GPU sharing on cluster orchestration platform

GPU sharing is implemented using the virtualization technology. The NVIDIA vGPU technology enables the use of a GPU in a VM [23]. The NVIDIA vGPU software, installed with a hypervisor layer, produces vGPUs in each VM to allow multiple VMs to share a physical GPU. OpenStack [24] is an open cluster-orchestration platform for a VM environment. It offers vGPU scheduling, and its default filter-based scheduler allocates vGPU instances according to user requirements.

Meanwhile, the use of software containers that reduce the isolated kernel overhead by utilizing full VMs and host system kernel calls has been increasing. As the number of container applications using GPUs has increased, container-orchestration frameworks, such as YARN and Kubernetes, have begun offering GPUs [16, 34]. Both YARN and Kubernetes consider a GPU as a simple extended resource in a scheduling procedure. They enable containers to exclusively use a GPU, thus resulting in low GPU-resource utilization.

2.2 Resource over-commitment

Figure 1 shows GPU resource-utilization patterns of applications over NVIDIA Titan Xp GPU and i7-5820K CPU. Two HPC applications, namely LAMMPS [17] and QMCPACK [26], and three CNN models, namely MNIST, AlexNet, and VGG11, are executed with standard input sets from the NVIDIA GPU Cloud (NGC) [21] and TensorFlow CNN benchmark [28]. The GPU memory is over-

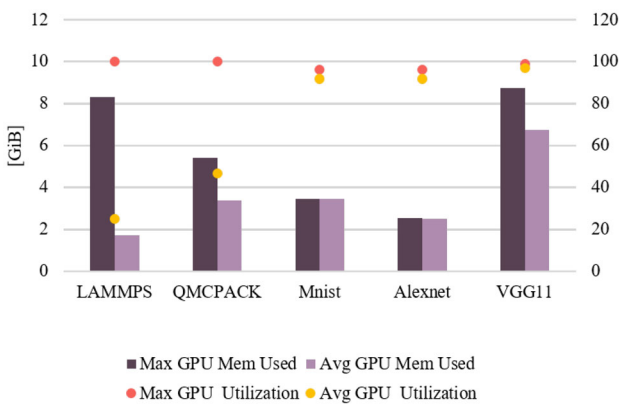

Fig. 1 Resource over-commitment of GPU applications

committed by approximately 54% on average. The GPU-utilization gap is approximately 51% from average to maximum. HPC applications have having relatively large deviations from mean values compared to that for DL applications. Consequently, overstated resource requirement for a job scheduler results in waste of resources. If a scheduler sticks to allocating the average number of resources to HPC applications, an OOM failure and a performance degradation may occur owing to the overlapping peak-resource-usage times for a GPU application. This experimental result indicates that it is necessary to prevent resource over-commitment by utilizing profiling and modeling to avoid an OOM failure or a severe performance degradation.

2.3 Necessity of scheduling considering interference

We assume that two of LAMMPS, GROMACS, MNIST, and classification applications wait in a queue. Figure 2 shows the makespan and average JCT according to three policies, namely the Binpacking, load-balancing, and interference-aware policy. The Binpacking policy only considers the maximum memory usage of applications. It places a task on a node with the highest resource usage but with sufficient available resources to minimize the number of nodes. The load-balancing policy divides loads based on their average GPU utilization. It places an application with the largest average GPU utilization together with an application with the lowest average GPU utilization. The interference-aware policy selects a pair of applications with the minimum interference value and executes them concurrently. The interference value is given by equation (1).

$$I(app_1, app_2) = \frac{Time_{colo}(app_1)}{Time_{solo}(app_1)} + \frac{Time_{colo}(app_2)}{Time_{solo}(app_2)} \quad (1)$$

$Time_{colo}$ denotes the co-execution time, and $Time_{solo}$ denotes the time required to execute them exclusively.

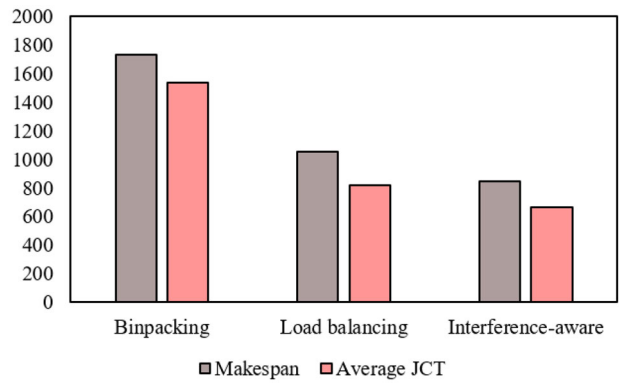

Fig. 2 Performance using three different policies

When applications are co-executed, sharing GPU resources causes a slow-down. The interference-aware policy intends to minimize this slow-down by comparing the co-execution and solo-run times.

The interference-aware policy achieves improvements of 138% and 23% in makespan over the Binpacking and load-balancing policies, respectively; LAMMPS, GROMACS, M- NIST, and classification are improved by nearly 1.38, 2.38, 2.23, and 5.19 times, respectively. The average JCT of the interference-aware policy is 74% and 22% lower than those of the Binpacking and load-balancing policies, respectively. The makespan and average JCT are the longest for the Binpacking policy because this policy only recognizes the maximum memory and does not consider all the available resources. The other two policies aim to avoid the interference caused by resource sharing. Avoiding this interference can reduce both makespan and average JCT as the degrees of performance degradation caused by the interference are diverse depending on the co-executed applications. The load-balancing policy avoids the interference by balancing the loads of each node, but its performance improvement is limited. Our policy achieves the best performance by minimizing the interference while maximizing the use of available resources.

3 Characteristics of GPU applications

The characteristics of applications on GPU are analyzed to develop an interference-aware co-scheduling method. Applications are executed in the environment as mentioned in Sect. 2. Four HPC applications (LAMMPS, GROMACS, QMCPACK, and HOOMD) from NVIDIA GPU Cloud (NGC) [21] were used with standard input sets. For DL training tasks, five CNN models, namely, mnist, googlenet, alexnet, vgg16, and vgg11 [28] were chosen. For DL inference tasks, classification, regression, and multiout of DJINN workload suite [7] were selected.

3.1 GPU utilization

Figure 3 shows GPU utilization over time based on a application category. As shown in Fig. 3a, all HPC applications show a dynamic GPU utilization except for HOOMD. HOOMD has particular utilization due to its massive parallelism. Unlike static utilization of HOOMD, there is a rapid increase in the GPU utilization in the last part of the application execution for LAMMPS and QMCPACK. They transfer data between host-devices frequently, resulting in a low utilization prior to the dramatic increase in utilization. The overall utilization is high and relatively constant for GROMACS, although a decline of utilization occurs at times owing to CPU tasks, and a

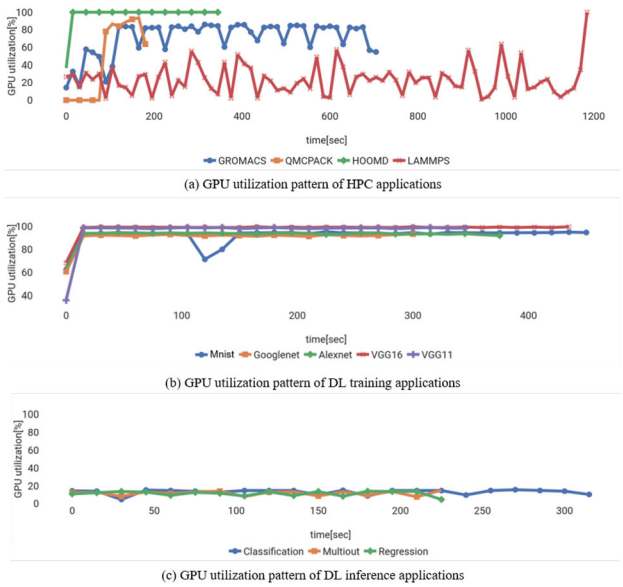

Fig. 3 GPU utilization pattern of applications on GPU

Fig. 4 GPU memory usage pattern of applications on GPU

memory transfer. A DL training task shows a static and high GPU utilization, as shown in Fig. 3b. Hyper-parameters which are set before a learning process begins including the learning rate, number of steps, and batch sizes and types of deep learning models affect the model convergence and accuracy. They are thus considered important in deep learning related studies. However, in this study, fixed hyper-parameters were used in an effort to focus on the interference among applications. When conducting experiments by modifying hyper-parameters, the GPU utilization values change but the static and high utilization characteristics are maintained. Meanwhile, Fig. 3c shows a graph of the GPU utilization for a DL interference task. A static characteristic can be observed with a maximum change of approximately 10%, compared to an HPC application with a maximum change of approximately 90%. In particular, the maximum GPU utilization is below 20%, displaying a low utilization. This result indicates that a DL inference application is not computationally intensive and there are idle GPU cores owing to an under-utilization of computational resources. As with DL training applications, this experiment demonstrated that DL inference applications also have static and low utilization characteristics, even when hyper-parameters are modified. Considering these experimental results, there is a limit in characterizing GPU applications with the average or maximum GPU utilization.

3.2 GPU memory used

The memory usage patterns of each GPU application are displayed as a graph in Fig. 4. Each application category shows a similar trend as GPU utilization. HPC applications have a high rate of change in memory usage, resulting in a difficulty predicting such usage. QMCPACK shows a cascading pattern of memory usage and LAMMPS first shows an increasing pattern with a high peak during the last part of the application execution. A static memory usage pattern was observed without a further release or request in deep learning tasks. The amount of memory used for each DL model differs, but the static pattern was maintained, even though hyper-parameters are modified. For a deep learning inference task as well, a static memory usage pattern was observed, although the number of memory resources used was extremely small compared to that of other applications. In general, the amount of GPU memory usage and GPU utilization are proportional. GROMACS showed a low GPU memory use of approximately 0.5 GB even with a high GPU utilization exceptionally. Prior studies have predicted memory usage either adjusting hyper-parameters or profiling in short term for DL tasks. However, as with the results of this experiment, the prediction of memory usage when using methods from prior studies is difficult to achieve for the cascading or high peak patterns of dynamic HPC jobs. Therefore, it is necessary to predict with profiling and monitoring.

3.3 PCIe throughput

Global memory throughput and PCIe throughput are generated by a kernel on a GPU. Global memory throughput is related to a kernel load or store requests, cache hit, and miss rates. If a data hit occurs on the on-chip cache, i.e., L1 cache, the throughput increases. PCIe throughput is a metric related to the amount of data movement in a PCI

express bus to approach a host memory, DRAM. Therefore, the higher the global memory throughput and the lower the PCIe throughput, the better performance and the lower latency.

The throughput of each application is shown in Fig. 5. It was shown that a deep learning inference task not only has a low global memory throughput, but also a very low PCIe throughput. HPC applications display various global/dram memory throughputs according to the type of application. Meanwhile, all deep learning training applications demonstrate a similar throughput, whereas PCIe throughput shows different aspects for each model. Mnist and Googlenet models show high in host-device PCIe throughput owing to frequent data movement, whereas Alexnet, VGG11, and VGG16 have low. Even if an application runs memory operations at a time, it may be overlapped to another one using different resources, such as cache and PCIe. That leads to co-execution of applications beneficially with low resource contention. A high PCIe throughput means that latency occurs since the host memory is accessed through the bus. However, overlapping between GPU kernel and host-device memory operation can be advantageous for a co-execution.

4 Interference modeling

We define interference value as co-executed time of an application normalized by solo-run time. If interference values are obtained directly after executing all application pairs, the most accurate and optimal results can be achieved. However, it is not realistic to calculate interference values with all pairs with large number of applications with long execution time. For N applications, interference values of $(N \times (N-1))/2$ pairs are needed. Interference is predicted through offline profiling information without executing all application pairs for this reason. Profiling information includes hardware characteristics, which affect the co-execution of applications on a GPU, and metrics deriving from the prior observation.

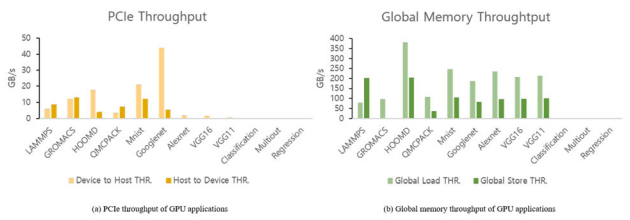

Fig. 5 PCIe and global memory throughput of applications on GPU

4.1 Metrics used for profiling resource usage

We define resource metrics that influence performance at the time of actual co-execution of applications. Each metric was collected using the NVIDIA profiler tool, nvprof, for predicting interference prevention during the co-execution. Table 1 shows detailed information regarding the metrics obtained during the profiling of each related resource.

GPU utilization average is the average execution time of one or more kernels on the GPU. SM efficiency average is the average time at which one or more warps are active in a particular multiprocessor on the GPU in percentage. Warp efficiency average is the average number of active threads for each warp in the SM. IPC is the number of commands executed per cycle. Occupancy average is the average number of active wraps per active cycle supported to the SM. GPU memory used max is the maximum amount of GPU memory used during the application program execution. GPU memory used average is the average GPU memory used during the application program execution. GPU memory utilization average is the average time for reading or writing GPU memory over a specific period of time during the application execution in percentage. Device to host throughput is the data throughput moving from global memory to CPU memory. Host to device throughput is the data throughput from CPU memory to GPU memory. The Cache: GLD (Global memory Load) throughput metric includes transactions served by the L1 and L2 caches. This metric is the amount of cache hit when loading into global memory. The GST (global memory store) throughput is also related to the L1 and L2 cache, but indicates a cache hit when storing in global memory. The execution time of each application and input during an application execution are recorded because the profiling information can vary depending on the input and parameters used even with the same applications. If an application with the same configuration is submitted, previously collected profiling information can be used.

Table 1 Resource metrics for interference modeling

Metric	
GPU utilization avg.	GPU memory utilization avg.
SM efficiency avg.	Device to host throughput
Warp efficiency avg.	Host to device throughput
IPC	GLD throughput
Occupancy avg.	GST throughput
GPU memory used max	Execution time
GPU memory used avg.	

avg average

4.2 Model construction

For interference modeling, we established three machine learning models- linear regression, random forest regression, and decision tree regression. We used total of 12 applications, described in Sect. 3. To establish the models, a total of 144 datasets were used as their pairs were modeled. We used a combination of metrics for each application as an input of the model. Metric values were normalized because the units and scales used for each metric differ. The model output is interference value, which is represented as a ratio between the time of co-execution and the time when the application is executed alone. We executed all application pairs and obtained the execution time to calculate the interference for constructing the model. The interference value applied the average value from three experimental results for accurate measurements. In addition, 5-fold validation was used to improve the model accuracy and reliability of the performance evaluation. The entire dataset is divided into five subsets, four of which are designated as the training data, with the remainder designated as the validation data. This process is repeated five times.

Table 2 shows the mean squared error (MSE) values and R2 scores for three types of machine learning models. The MSE represents the difference between the predicted and actual values, and the closer it is to zero the higher the accuracy. R-Square is a validation measure of a regression model, the explanatory power of which is higher as it reaches closer to 1. The random forest regression model showed the best performance in a container environment. The random forest model was used in this experiment for this reason.

5 Design and implementation

This section describes the overall architecture design of the interference-aware policy and Co-scheML scheduler that dynamically schedules the arrived applications for each node in a GPU cluster in a greedy manner.

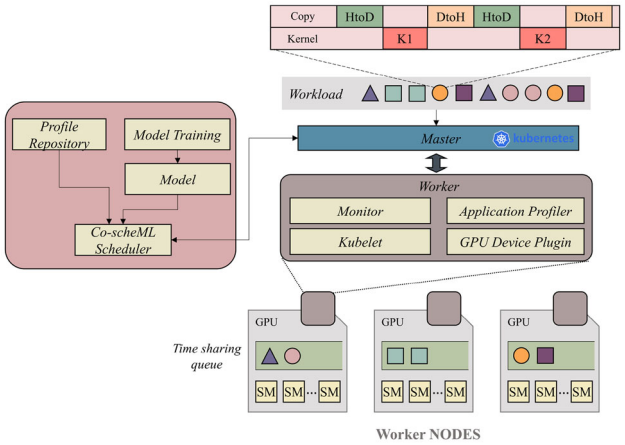

Fig. 6 Architecture of interference-aware execution framework

5.1 Architecture

Figure 6 presents the overall system design. We implemented device plugins for GPU sharing on Kubernetes. The profile repository stores profiled metrics, which are collected offline for scheduling or interference prediction. The metrics are stored in a time series-based database, InfluxDB [12], and labeled with the application's name and input data. The Co-scheML scheduler requests interference values from the constructed model described in Sect. 4. The scheduler sends its scheduling decision to the kubelet of each worker node. The kubelet dispatches applications to designated GPUs. The monitor constantly monitors the progress of applications and updates the profiling information to improve the accuracy of the constructed model.

5.2 Scheduler

The Co-scheML scheduler is called when users submit an application or when an application run is completed. The scheduling operation of the Co-scheML is the same as that in Algorithm 1 [14]

Table 2 Mean squared error and R-square of regression models

	Linear regression	Decision tree regression	Random forest regression
MSE	0.0546	0.0269	0.0222
R-Square	0.6946	0.8500	0.8758

Algorithm 1 Co-scheML Scheduler [14]

Require: App, gpu_n, pending_app
selected_n ← ∅
idle_n ← $find\_idle\_nodes(gpu\_n)$
if idle_n ≠ ∅ **then**
  selected_n ← idle_n[0]
else
  single_app ← $find\_exclusive\_app(gpu\_n)$
  if single_app ≠ ∅ **then**
    app_p ← $cal\_interf(single\_app, pending\_app)$
    sorted_p ← sort_by_interf_val(app_p)
    selected_p ← $find\_pairs(sorted\_p)$
    if App ∈ selected_p **then**
      selected_n ← find_n(App, selected_p)
    end if
  end if
end if
return selected_n
function $calc\_interf(single\_app, pending\_app)$
  for s_app ∈ single_app **do**
    s_metrics ← Q_profile_reposit(s_app)
    p_metrics ← Q_profile_reposit(p_app)
    interf_val ← Q_model(p_metrics, s_metrics)
    app_p ← append(s_app, p_app, interf_val)
  end for
  return app_p
end function
function $find\_pairs(app\_pairs)$
  for p ∈ app_pairs **do**
    if sel(p.s_app) ≠ T and sel(p.p_app) ≠ T **then**
      if can_co-sched(p.s_app,p.p_app) **then**
        selected_pairs ← append(p)
      end if
    end if
  end for
  return selected_pairs
end function

The applications submitted by users arrive in a queue. The Co-scheML scheduler does not waste resources of an idle GPU. It executes an application exclusively only when no task is present in the waiting queue. It distributes tasks to as many nodes as possible for performance maximization. Thus, if there are four nodes and four tasks in the queue, the scheduler allocates one task to each node. When there is no idle node, it co-executes pairs of applications to maximize the GPU-resource utilization. To avoid interference, the scheduler chooses an application pair with the minimum interference value based on the greedy algorithm. The scheduler compares the interference values between a waiting task and the running tasks in nodes of all the application pairs. Subsequently, it requests the interference values from the profile repository and the constructed interference model. Further, it exploits the monitored and profiled information to confirm whether co-scheduling is possible. This allows the OOM-failure prediction. The detailed information of the scheduler is presented in Co-scheML [14].

6 Evaluation

6.1 Evaluation methodology

6.1.1 Experiment environment

GPU cluster based on Kubernetes is used for experiment. The cluster is comprised of one master node and three GPU nodes, the work node as shown is Tables 3 and 4. NVML-based Monitor records metrics in influxDB every 5 s. There is Kubernetes default scheduler, Co-scheML, and Model in the master node.

6.1.2 Workloads

Twelve real-world applications were selected. Four HPC applications (LAMMPS, GROMACS, QMCPACK, HOOMD), five DL training jobs (mnist, googlenet, alexnet, vgg16, vgg11), and three DL inference jobs were used for experiments as described in section 3. All DL tasks used Tensorflow, executed in the GPU and containerized as a Docker container. Ten workloads were used for experiments and the sensitivity of the scheduler is evaluated by varying the arrival interval [4, 30]. At this time, the arrival interval was arbitrarily designated as 15, 30, and 60 s each for light, medium, and heavy loads, respectively. The default task density was a medium load.

Evaluation metrics

- The average job completion time (JCT) is the average completion time from when each job is submitted.
- Makespan is the time when all jobs in the workload are completed.
- Speedup is the value of the execution time when the application is co-scheduled normalized to the time when the application is executed alone. The value is between 0 and 1, and the closer it is to 1, the smaller is

Table 3 Experimental settings

	CPU (master)	GPU (worker)
Architecture	Intel® *Core*™ i7-5820 K	NVIDIA GeForce Titan Xp D5x
Core Clock	3.30 GHz	1.58 GHz
No. of cores	6	3840
Mem. size	32 GB	12 GB
Threading API	–	NVIDIA CUDA 10.0
OS	Ubuntu 16.04.6 LTS	Ubuntu 16.04.6 LTS

Table 4 Static environment features

GPU memory	11.91 GB	Warps per SM	64
GPU speed	1582 MHz	Thread blocks per SM	32
GPU architecture	TITAN Xp	Shared Memory per SM	96 KB
PCIe bandwidth	32 GB/s	Threads per SM	2048

the performance difference compared to when it is executed alone.
- GPU utilization is the percentage of time over the past sample period during which one or more kernels were executing on the GPU.

6.1.3 Baseline schedulers

A max-memory-based Binpack scheduler and interference aware schedulers, such as the Loadbalance and Mystic [30] schedulers, are used. The Loadbalance scheduler is based on the average GPU utilization and selects the pairs to co-execute such that it has the minimum GPU utilization. The Mystic scheduler calculates the similarity among the application metrics and schedules in order of low similarity. Every baseline scheduler is implemented and integrated to Kubernetes for comparing.

6.2 Scheduling performance

Figure 7 compares the performances of four approaches over 10 workloads. Our framework outperformed the Binpacking policy by 26%, 29%, and 28% in terms of makespan, average JCT, and speed up, respectively. The Binpacking policy was the most affected by interference. As the scheduler attempted to minimize the number of used nodes, it was not possible to use the available resources fully. The number of submitted jobs varied depending on the time in the cloud environment. Users tended to rarely request their tasks at certain times such as night. Hence, considering only the memory usage and not resource contention can waste resources and lead to severe performance degradation.

The makespan and average JCT of our framework were 18% and 11% lower than those of LoadBalance; the speed was also improved by 0.2%. LoadBalance exhibited high speed because it shared computing resources fairly well, considering the average GPU utilization. However, it did not deliver a satisfactory performance improvement in terms of makespan and average JCT because it did not consider the overall resource usage. In particular, it exhibited poor performance for workloads including only HPC applications. An average GPU utilization is not sufficient to characterize the HPC applications, which have dynamic GPU utilization.

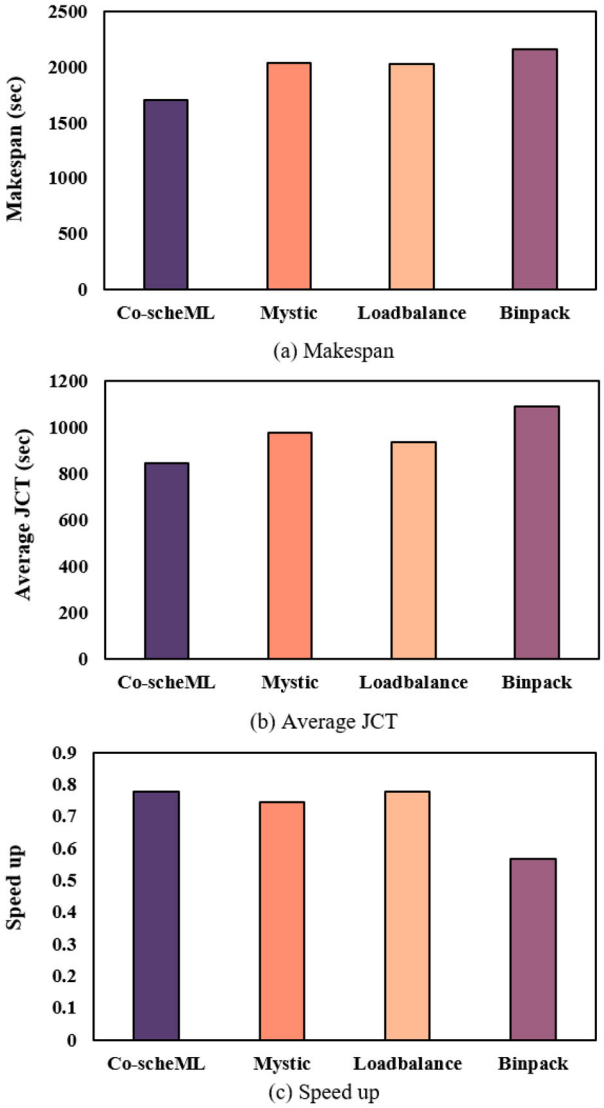

Fig. 7 Performance comparison of Binpack, Loadbalance, Mystic, and Our framework

Our framework achieved performance improvements of 19%, 16%, and 4% in terms of makespan, average JCT, and speedup, respectively, over Mystic. The effect of each metric on interference was different. Mystic did not reflect the weights of metrics and calculated interference based on similarity. Although it offered the largest improvement among those of other baseline schedulers, its performance improvement was limited. Furthermore, it could not detect OOM failure.

Figure 8 depicts a graph representing the GPU utilization based on the scheduler for each node. Each of the 12

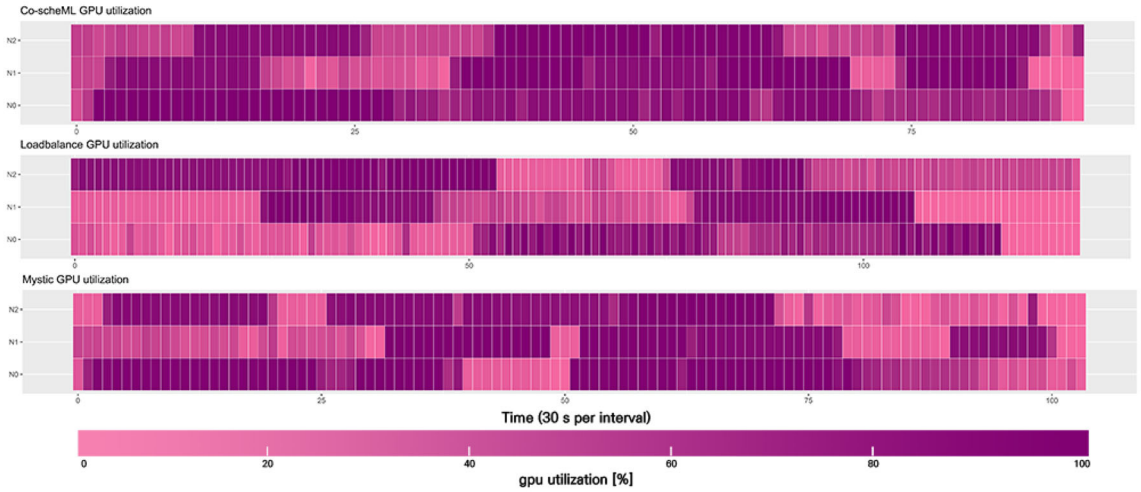

Fig. 8 Comparison of GPU utilization of schedulers

applications used in this experiment was executed twice, and the workload had a total of 24 jobs whose launch sequence was randomly generated. During the execution of all the workloads, the average GPU utilization was 78% for Co-scheML, 59% for LoadBalance, and 67% for Mystic, indicating Co-scheML's higher GPU utilization of 32% and 16%, respectively, compared to those of LoadBalance and Mystic. Co-scheML allowed each application to use complementary resources by considering interference, resulting in improved GPU utilization.

6.3 Sensitivity analysis

The behaviors of each scheduling method when different loads are applied to the server are evaluated for the same workload. Figure 9 shows the average JCT according to each task density. The overall applications of the workload were sorted by the JCT. Figure 9a–c show heavy, medium, and light loads, respectively. The average JCT of Co-scheML was lower than that of Loadbalance and Mystic by 12% and 32%, respectively for a heavy load. It was 43% and 3% lower, respectively for a medium load, and it was 96% and 18% lower for a light load.

Figure 10 shows the makespan depending on each server load. For the overall task density, Co-scheML showed a 1.53-, 1.32-, and 1.12-fold better makespan than that of Binpack, Loadbalance, and Mystic, respectively. As the system load increased, the performances of all schedulers showed a decreasing trend. Meanwhile, the Binpack scheduler was the least influenced by the task density, but the worst performance. It was confirmed that Co-scheML achieved the best average JCT and makespan at all task densities.

6.4 Overhead

The Binpack scheduler has a scheduler overload of approximately 1.001 s regardless of the number of pods. The Binpack scheduler compares the amount of available memory of all nodes and the required memory by the pod to be executed, and thus the overload increases proportionally to the number of nodes. In this experiment, three nodes were used with approximately 1 s of overhead.

The scheduler established in this study has a linearly increasing overhead owing to the increasing number of subjects to which the inference values are to be compared based on the number of pods, as shown in the blue line of Fig. 11. However, even with 50 pods, a scheduling time of 0.003 s or less is required. The orange line in Fig. 11 shows the overload compared to the overall execution time, which is within the range of 0.0003%-0.0035%. This demonstrates that the runtime overhead is trivial and that the overhead is lower than that of the Binpack scheduler.

In addition, a communication overhead from an http request occurs, which takes approximately 6 s on average. This is an overhead commonly included in the Kubernetes scheduler and exists in all schedulers, i.e., the Kubernetes default scheduler and Binpack scheduler, proposed in this study.

7 Related studies

7.1 GPU resource sharing scheduling

Many GPU sharing technologies have been introduced in an effort to improve the resource utilization of GPU clusters and cloud servers. Diab et al. [6] proposed a system in which many users can share GPU resources while co-

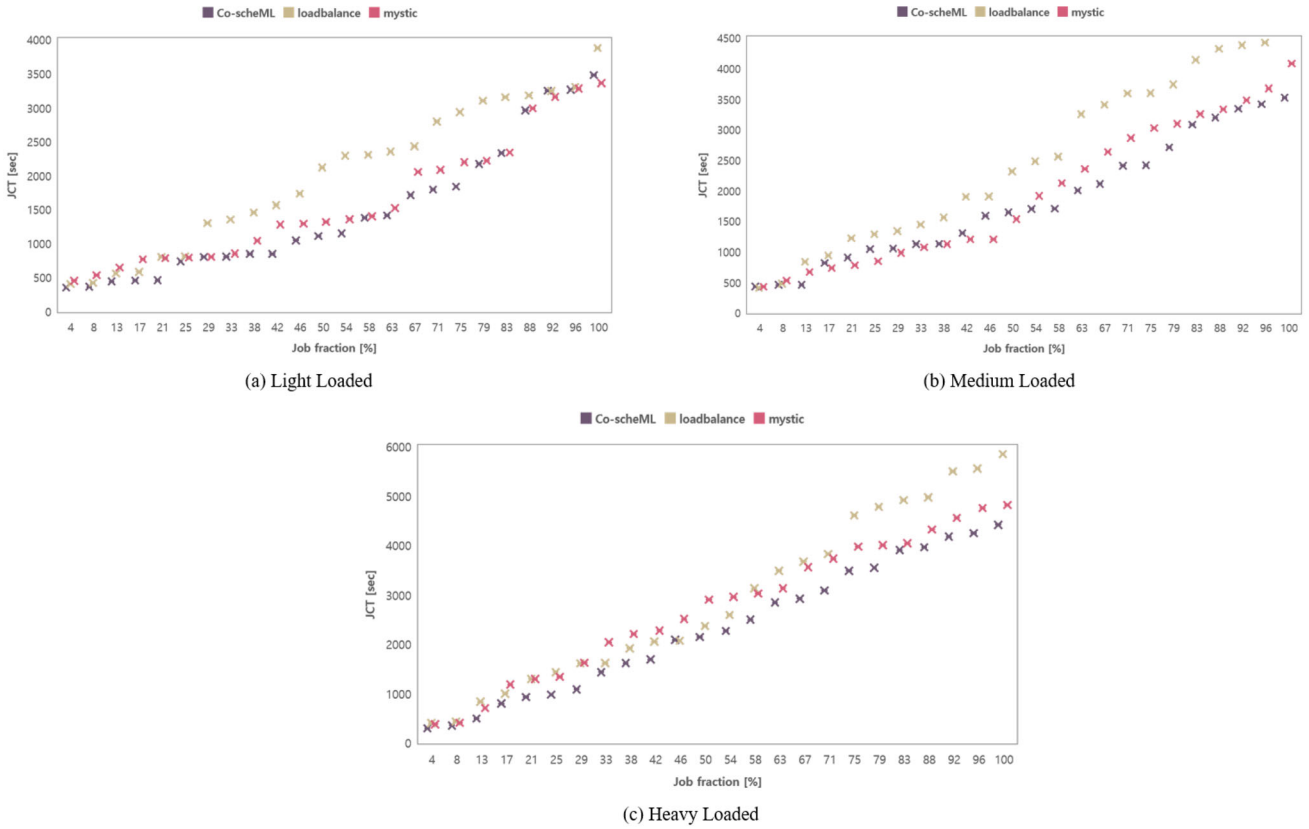

Fig. 9 Comparison of average JCT for various task densities

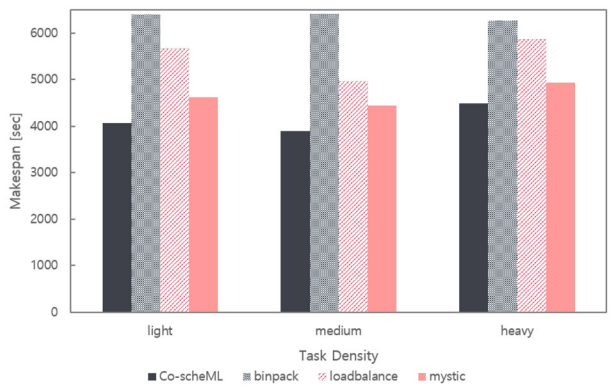

Fig. 10 Comparison of makespan for various task densities

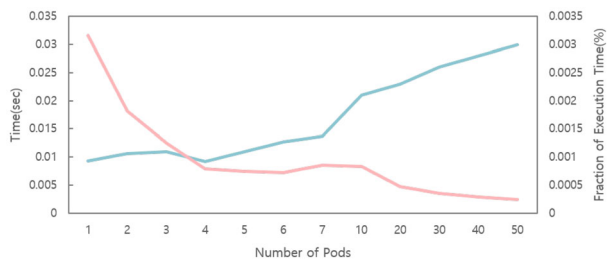

Fig. 11 Scheduling overhead of Co-scheML

executing tasks by intercepting a CUDA API to enable the execution of two kernels. However, this scheduling method can only be used for applications that have repetitive or distinct forms of resource usage. In Bao et al. and Song et al. [2, 27], the GPU resources are classified into a certain size and allocated to a container that executes a cost tree to allocate and provide the resources to the GPU. However, this only considers the minimum resource requirement of the running task, which can result in a degraded performance during a co-execution. There are cluster schedulers for DL workload [10, 18, 25, 29, 32]. Kube-knots [29] predicts the GPU utilization, PCIe bandwidth, and memory to guarantee the QoS, and minimize the energy efficiency. Although the DL workload is static and predictable, dynamic applications such as HPC applications are difficult to predict. Therefore, we conducted resource provisioning through profiling and monitoring in this study. In addition, the purpose of this study is to improve performance rather than achieve energy efficiency. Using GPU utilization, the PCIe bandwidth and memory usage alone are insufficient to improve the performance.

7.2 GPU interference-aware scheduling

With the introduction of technologies that can co-execute numerous applications on the GPU, scheduling methods to avoid resource contention that may occur during the co-execution in this environment have been proposed. Mystic [30] suggests a collaborate filtering (CF) based interference recognition scheduler for the co-execution of applications. It profiles interference metrics, and if a new application is applied, CF is used for prediction after lightweight profiling. The method used to obtain the interference values of each application pairs is based on the similarities in the vector of metrics, in which the lower the similarity, the lower the interference. Although the degree of influence of the interference for each metric is different and leads to different weights, this is not reflected. Rather, similarities are simply obtained, and the OOM failure issue that can occur during the co-execution of the GPU applications is not considered. Chen et al. [4] analyzes the characteristics of the DL and carries out a performance prediction modeling based on the results as a suggestion to the QoS-aware scheduler. This requires domain-specific knowledge of the DL, and thus cannot be applied to all applications, such as HPC applications. Xu et al. [33] defined features of the application characteristics executed on a GPU to implement an ML-based interference recognition scheduler. Although the performance of a simple application is significantly affected by the kernel length, a difference in interference for ML applications is shown. However, because the actual evaluation was conducted on ML applications that have repetitive resource usage patterns, additional feature definitions of the affected resources and other metric definitions for the container environment targeted to the VM are required. Moreover, a method for applying interference values to the cluster scheduling method is necessary because a simple round-robin scheduler was implemented. In Bao et al. [2], a learning placement framework using a DRL model in the cluster environment with GPU servers is proposed. The authors explained that learning is carried out by inputting the worker id, CPU, and GPU, and tasks are placed by a low level of interference when multiple applications coexist. However, detailed information on the resources is necessary to decrease the interference effect with the CPU and GPU usage values. Wen et al. [31] considers interference values and uses max-pair algorithms to decrease the overall workload execution time. However, the lengthy time required to execute all pairs each time a new application enters can be a disadvantage. In this study, we demonstrated that if the profiling information is available, a prediction of the interference is possible without executing all pairs, which cannot be applied to dynamic scheduling.

7.3 Interference aware co-scheduling on CPU

There are many approaches to co-scheduling for CPU-based servers. The key idea of these approaches is to co-run various applications rather than execute serially for increasing throughput of servers. Some applications need small portions of resources, or some others experience performance degradation. Muralidhara et al. [20] suggests application-aware memory channel partitioning (MCP) to map the data of applications that seem to interfere with each other to different memory channels. Lo et al. [19] dynamically manages hardware and software isolation mechanisms to guarantee latency-sensitive services. It increases the utilization of servers without latency violations in colocation scenarios. Aupy et al. [1] uses cache partitioning to optimize the performance of applications when they are co scheduled on the same node. Jiang et al. [13] constructs graph to represent co-run performance degradation. It optimizes scheduling with a min-weight perfect matching problem. Dauwe et al. [5] presents models that utilize various information on applications and predict the application's execution time performance degradation due to co-location. It characterizes applications on multicore processor architectures.

8 Conclusion

This paper proposes Co-scheML, an interference-aware co-execution framework, which provides an ML model that predicts interference values using application profiling data and minimizes interference among GPU applications in a GPU cluster. The experiment revealed that the average JCT and makespan improved by nearly 18% and 22%, respectively, for various workloads, compared to those of conventional schedulers. The resource utilization of the cluster was enhanced by 24%, the average JCT under various task densities improved by 23%, and the makespan shortened by 22% on average, compared to those of baseline schedulers. Future studies would include identification of characteristics in various GPU execution environments and an extension of the scheduling method for multiple GPUs.

Acknowledgements This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea Government (MSIT) (No. 2015M3C4A7065646, 2020R1H1A2 011685, NRF-2021R1A2C1003379).

Data availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

References

1. Aupy, G., Benoit, A., Goglin, B., Pottier, L., Robert, Y.: Co-scheduling HPC workloads on cache-partitioned CMP platforms. Int. J. High Perform. Comput. Appl. **33**(6), 1221–1239 (2019)
2. Bao, Y., et al.: Deep learning-based job placement in distributed machine learning clusters. In: IEEE INFOCOM 2019—IEEE Conference on Computer Communications (2019)
3. Chang, C.C., Yang, S.R., et al.: A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In: GLOBECOM 2017—2017 IEEE Global Communications Conference (2017)
4. Chen, Z., Quan, W., et al.: Deep learning research and development platform: characterizing and scheduling with GOS guarantees on GPU clusters. IEEE Trans. Parallel Distrib. Syst. **31**, 34–50 (2019)
5. Dauwe, D., Jonardi, E., Friese, R., Pasricha, S., Maciejewski, A.A., Bader, D.A., Siegel, H.J.: A methodology for co-location aware application performance modeling in multicore computing. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 434–443. IEEE (2015)
6. Diab, K.M., et al.: Dynamic sharing of GPUS in cloud systems. In: IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (2013)
7. DJINN: https://github.com/LLNL/DJINN
8. Geng, X., Zhang, H., et al.: Interference-aware parallelization for deep learning workload in GPU cluster. Clust. Comput. **23**, 2689–2702 (2020)
9. Gu, J., et al.: Gaiagpu: Sharing GPUS in container clouds. In: IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom) (2018)
10. Gu, J., Chowdhury, M., et al.: Tiresias: a GPU cluster manager for distributed deep learning. In: In16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19) (2019)
11. Hong, C.-H., et al.: FairGV: fair and fast GPU virtualization. IEEE Trans. Parallel Distrib. Syst. **28**(12), 3472–3485 (2017)
12. InfuxDB: https://www.influxdata.com/
13. Jiang, Y., Shen, X., Jie, C., Tripathi, R.: Analysis and approximation of optimal co-scheduling on chip multiprocessors. In: 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 220–229. IEEE (2008)
14. Kim, S., Kim, Y.: Co-scheml: interference-aware container co-scheduling scheme using machine learning application profiles for GPU clusters. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER), pp. 104–108. IEEE (2020)
15. Kim, S., Kim, Y.: Toward interference-aware GPU container co-scheduling learning from application profiles. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). IEEE, pp. 19–23 (2020)
16. Kubernetes: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/ (2020)
17. LAMMPS-Molecular-Dynamics-Simulator: https://lammps.sandia.gov/
18. Liaw, R., Bhardwaj, R., et al.: Hypersched: dynamic resource reallocation for model development on a deadline. In: Proceedings of the ACM Symposium on Cloud Computing (2019)
19. Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., Kozyrakis, C.: Improving resource efficiency at scale with heracles. ACM Trans. Comput. Syst. (TOCS) **34**(2), 1–33 (2016)
20. Muralidhara, S.P., Subramanian, L., Mutlu, O., Kandemir, M., Moscibroda, T.: Reducing memory interference in multicore systems via application-aware memory channel partitioning. In: 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 374–385. IEEE (2011)
21. NVIDIA-GPU-Container(NGC): https://ngc.nvidia.com/
22. NVIDIA-Multi-Process-Service: https://docs.nvidia.com/deploy/pdf/CUDA-Multi-Process-Service-Overview.pdf (2019)
23. NVIDIA-VGPU: https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/index.html
24. Openstack: https://specs.openstack.org/openstack/nova-specs/specs/queens/implemented/add-support-for-vgpu.html
25. Peng, Y., Bao, Y., et al.: Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: Proceedings of the Thirteenth EuroSys Conference (2018)
26. QMCPACK: https://qmcpack.org/
27. Song, S., et al.: Gaia scheduler: A kubernetes-based scheduler framework. In: IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom) (2018)
28. Tensorflow-CNN-benchmarks: https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_c- nn_benchmarks
29. Thinakaran, P., Gunasekaran, J.R., et al.: Kube-knots: resource harvesting through dynamic container orchestration in gpu-based datacenters. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER) (2019)
30. Ukidave, Y., et al.: Mystic: predictive scheduling for gpu based cloud servers using machine learning. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2016)
31. Wen, Y., O'Boyle, M.F., Fensch, C.: MaxPair: enhance OpenCL concurrent kernel execution by weighted maximum matching. In: Proceedings of the 11th Workshop on General Purpose GPUs (2018)
32. Xiao, W., et al.: Gandiva: Introspective cluster scheduling for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18) (2018)
33. Xu, X., et al.: Characterization and prediction of performance interference on mediated passthrough GPUs for interference-aware scheduler. In: 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19) (2019)
34. YARN: https://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/UsingGpus.html (2018)

Sejin Kim is currently a Master degree in the Department of Computer Science, Sookmyung Women's University. She received her B.S. degree from Sookmyung Women's University in 2017. She is also researcher at the Distributed & Cloud Computing Lab of Sookmyung Women's University. Her research interests include management in cloud computing and heterogeneous systems.

Yoonhee Kim she is the professor of Computer Science Department at Sookmyung Women's University. She received her Bachelors degree from Sookmyung Women's University in 1991, her Master degree and Ph.D. from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was the faculty of Computer Engineering dept. at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed computing systems. She is a member of IEEE and OGF, and she has served on variety of program committees, advisory boards, and editorial boards.