Co-scheML: Interference-aware Container Co-scheduling Scheme using Machine Learning Application Profiles for GPU Clusters

Sejin Kim
Department of Computer Science
Sookmyung Women's University
Seoul, Korea
wonder960702@gmail.com

Yoonhee Kim
Department of Computer Science
Sookmyung Women's University
Seoul, Korea
yulan@sookmyung.ac.kr

***Abstract*—Recently, efficient execution of applications on Graphic Processing Unit(GPU) has emerged as a research topic to increase overall system throughput in cluster environment. As a current cluster orchestration platform using GPUs only supports an exclusive execution of an application on a GPU, the platform may not utilize resource of GPUs fully relying on application characteristics. Nonetheless, co-execution of GPU applications leads to interference coming from resource contention among applications. If diverse resource usage characteristics of GPU applications are not deliberated, unbalanced usage of computing resources and performance degradation could be induced in a GPU cluster. This study introduces Co-scheML for co-execution of various GPU applications such as High Performance Computing (HPC), Deep Learning (DL) Training, and DL Inference. Interference model is constructed by applying Machine Learning (ML) model with GPU metrics since predicting interference has a difficulty. Predicted interference is utilized and deployment of an application is determined by Co-scheML scheduler. Experimental results of the Co-ScheML strategy show that average job completion time is improved by 23%, and the makespan is shortened by 22% in average, as compared to baseline schedulers.**

Keywords— GPU applications, interference, co-execution, Co-scheML scheduler, resource contention, GPU utilization (key words)

I. Introduction

Issues related to optimization, application performance, and system throughput of new and emerging general-purpose graphics processing architectures, programming environment, and platforms are varied as graphical processing unit(GPU) becomes popular in general. Current cluster schedulers of cluster orchestration platforms such as Yarn [1] and Kubernetes [2] might not fully exploit GPU computing resources because they execute an application exclusively on a GPU at a time. To overcome this bounded use of GPU resource, co-executing multiple and diverse applications, which have miscellaneous patterns of resource usage is suggested.

For General Purpose GPU (GPGPU) sharing, NVIDIA has introduced multiple process service (MPS) to execute multiple kernels in parallel [3]. However, performance of this technique may degrade due to interference from co-executed kernels accessing the same device at a time. Studies on GPU sharing cover co-deployment of applications with strategies coming from monitoring information, user requirements [9,10,11] or weights using GPU usage profiles [12]. Nevertheless, interference occurred at the time of co-execution, which leads to performance degradation is not considered in these studies. Furthermore, avoiding interference using resource usage records of various applications is challenging.

Recent studies have discussed interference issues caused by resource contention that occurs from its sharing [8,13,14]. Some performance metrics of resources related to interference are defined and accumulated profiling of the metrics are applied for interference avoidance scheduling by a previous study [14]. However, an execution failure such as out-of-memory (OOM) may occur because its profiling metrics are too limited to avoid the failure. Machine learning (ML) is applied to scheduling for predicting interference using resource metrics [13].It defines features of vGPU resources (GPU, GPU memory, PCIe, vCPU) for deep learning(DL) applications. [8] identifies various factors which affect interference according to node and cluster levels, and then implements scheduling with diverse ML model to each level relying on the factors. However, those studies analyze only DL workloads, which have static resource usage patterns.

This paper introduces Co-ScheML to improve performance and optimize resource usage for various applications running on GPUs. To solve the interference problem that occurs during GPU sharing, it proposes a scheduling method to avoid interference with ML on GPU using appropriate metrics for applications with various characteristics. Depending on the degree of interference, it decides whether a GPU application is possibly co-located with a currently running application. Execution results are profiled to improve the model accuracy for future experiments.

In experiments, Co-ScheML's performance is compared to Binpack, Loadbalance, and Mystics[14] methods. An evaluation includes job completion time (JCT), makespan, speedup, and GPU utilization depending on a variety of workload task density. Overall results show improving GPU utilization rate by 24% in average, and shortening average job completion time rate by 23% and makespan by 22% in average, respectively.

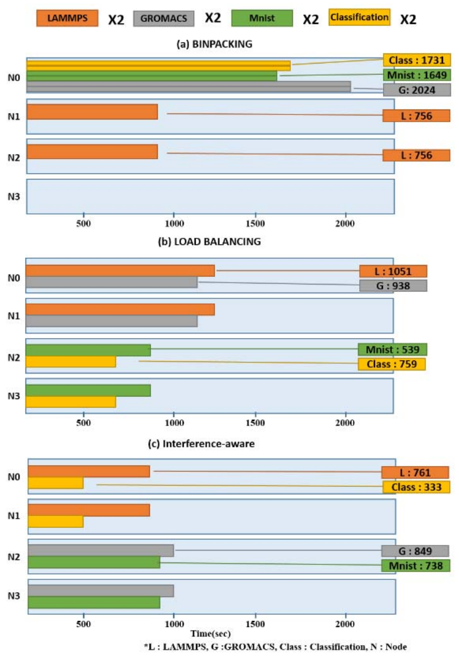

Figure 1. Job placement according to different scheduling policies

The rest of this paper is organized as follows. Section 2 introduces motivation. The architecture of Co-ScheML and an interference-aware scheduling algorithm are explained in Section 3. Its experiments are described in Section 4. The related studies are provided in Section 5 and concluding remarks are given in Section 6.

II. MOTIVATION

A. Necessity of scheduling considering interference

Fig. 1 shows a sample experiment that compares scheduling according to three policies, assuming that there are two of each LAMMPS, GROMACS, Mnist, and Classification applications in the pool. Fig. 1-(a) illustrates a binpacking policy scheduling layout considering the max memory. This is a policy that minimizes the number of nodes by placing tasks on the node with sufficient available resources, but with the highest resource usage. Fig. 1-(b) shows a method that distributes the load by simply considering the average GPU utilization. It co-locates the application with the maximum average GPU utilization and the application with the minimum average GPU utilization. The bottom of the figure is a scheduling layout that considers interference. To calculate interference values, each application pair is co-executed and compared with the time which running them individually. The pair with the minimum interference value based on the obtained interference values is selected and executed.

When comparing the scheduling considering interference with the binpacking policy and the loadbalancing policy, makespan was decreased by 2.38- and 1.23-fold, respectively. When each application performance was compared, the LAMMPS, GROMACS, Mnist, and Classification performances were improved by up to 38%, 138%, 123%, and 419%, respectively; however, LAMMPS was approximately 0.6% worse than the binpacking policy, and Mnist was approximately 36% worse than the loadbalancing policy. As a result, when the interference aware policy was

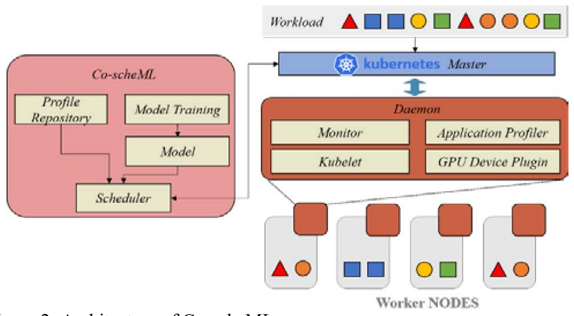

Figure 2. Architecture of Co-scheML

applied, the average job completion time of the application was improved by 74% and 22% compared to the binpacking policy and the loadbalancing policy (670.25 s versus 1168.75 and 821.75 s, respectively). This result shows that the binpacking method recognizing only the max memory does not use all of the available resources. Furthermore, it does not reflect interference that may be appeared among the applications, thereby leading to performance degradation. Although the method used to reduce the interference by naively balancing the loads among nodes with the average GPU utilization shows a better performance compared to the binpacking method, there is a limitation in the improvement achieved. Our motivation is to design a scheduler that minimizes interference while maximizing the use of available resources from these results.

III. CO-SCHEML DESIGN

This section describes the overall architecture design of Co-scheML and scheduling method for dynamically arrived applications of each node in a GPU cluster.

A. Architecture

The overall system design is shown in Fig. 2. Kubernetes manages containers with modified device-plugin for sharing of GPUs. If an application is first submitted, it is executed alone and profiled to collect metrics. Its profiling information is labeled as the application name and input data and stored in the Profile repository. Metrics affecting co-execution of applications are stored in a time-series-based database, influxDB[5] and a profiling step is carried out offline. Scheduler requests interference value from the Profile repository and Model, and schedules accordingly. Model construct random forest regression model [17] to predict interference with offline profiling information. 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. The decision of Scheduler is sent to each worker node's Kubelet. Kubelet launches applications using the modified GPU Device Plugin for sharing resources. While the application is being executed, the progress of the application is continuously monitored and the profiling information is updated to improve the accuracy in Monitor.

B. Scheduler

The Scheduler does not waste resources from an idle GPU and execute an application exclusively when no task is in a waiting queue. It distributes tasks to as many nodes as possible to maximize its performance. When an application is in the queue, the application is the queue, the application

Algorithm 1 Co-scheML Scheduler

```
Input : App, gpu_n, pending_app
1:  selected_n ← ∅
2:  idle_n ← find_idle_nodes(g     pu_n)
3:  if idle_n ≠ ∅ then
4:    selected_n ← idle_n[0]
5:  else
6:    single_apps ← find_exclusive_app(gpu_n)
7:    if single_apps ≠ ∅ then
8:      app_p ← cal_interf(single_app, pending_app)
9:      sorted_p ← sort_by_interf_val(app_p)
10:     selected_p ← find_pairs(sorted_p)
11:     if App ∈ selected_p then
12:       selected_n ← find_n(App, selected_p)
13: return selected_n

14: procedure calculate_interf(single_app, pending_app)
15: for s_app in single_app
16:   for p_app in pending_apps
17:     s_metrics ← Q_profile_reposit(s_app)
18:     p_metrics ← Q_profile_reposit(p_app)
19:     interf_val ← Q_ML_model(s_metrics, p_metrics)
20:     app_p← append(s_app, p_app, interf_val)
21: return app_ps

22: procedure find_pairs(app_pairs)
23: for p in app_pairs
24:   if sel(p.s_app) ≠ true and sel(pair.p_app) ≠ true then
25:     if can_co-sched(p.s_app, p.p_app) then
26:       selected_pairs ← append(p)
27: return selected_pairs
```

is executed as a pair to utilize the resources that are wasted when an application exclusively uses the GPU. Scheduler considers different interference values that are generated from each pair of applications. We
choose the pair with the minimum interference value according to the greedy algorithm.

Kubernetes default scheduler requests Co-scheML to filter out nodes of the application in the queue to be scheduled either when the user submits an application that arrives in the queue or when the running application is terminated. Therefore, our scheduling algorithm is called when a new application is submitted or when a running application is terminated. The scheduling operation of Co-scheML is the same as in Algorithm 1.

When Co-scheML scheduler receives an application to be scheduled, a list of GPU nodes present in a cluster, and a list of applications in a waiting queue from the Kubernetes default scheduler, it returns a node allocation result. The *find_idle_nodes* function returns nodes without any running applications in *gpu_n* (line 2). If there is no idle node, the *find_exclusive_app* function is called to find an application that is being executed exclusively in *gpu_n*. If such application presents, interference values of all application pairs between *single_app* and *pending_app* are calculated through the *cal_interf* function. To obtain the interference value, metrics of each application are queried from the Profile repository (lines 17 and 18). The interference values are returned from the interference model by inputting metrics obtained through the query, as described in Section 4 (line 19). The returned values are sorted in ascending order and the *find_pairs* function is called to select the pairs to be co-executed according to the greedy algorithm (lines 9 and 10). The *find_pairs* function confirms whether co-scheduling is possible if the applications of the pair are not yet selected. This leads to the prediction of OOM with the profiling and monitoring information (lines 25–26). The target application is subject to verification regarding whether it is included in the list of pairs selected from the minimum interference values and can also be co-located (line 11). If it is included, the node that *s_app* of the pair is executing is found and returned as a *selected_n* through the *find_n* function (lines 12).

IV. Evaluation

A. Evaluation methodology

Experiment environment: The Kubernetes-based private GPU cluster is used for the evaluation. The cluster is comprised of one master node with Intel® Core™ i7-5820K, 32GB RAM and three computing nodes with Nvidia GeForce Titan Xp D5x GPU which has 12GB memory and CUDA 10.0 API. The Monitor is an NVML-based resource monitoring component and records the metrics in the influx DB every 5 s. In the master node, the Kubernetes default scheduler, Co-scheML, and Model are used. The Co-scheML Scheduler uses the scheduler extension mechanism of Kubernetes.

TABLE I. CHARACTERISTICS OF EACH WORKLOAD SEQEUNCE

Workload sequence	***Characteristics of workload***
0	High GPU utilization(HOOMD, Mnist, Googlenet, VGG11, VGG16, GROMACS)
1	High memory utilization(Mnist, Googlenet, VGG16, VGG11)
2	Hight PCIe bandwidth(LAMMPS, GROMACS, HOOMD, QMCPACK, Mnist, Googlenet)
3	DL Training applications
4	DL Training applications
5	HPC applications
6	DL applications
7, 8, 9	Random applications

Workloads: Twelve real-world applications were selected. Four HPC applications (LAMMPS, GROMACS, QMCPACK, HOOMD) [6], five DL training jobs (mnist, googlenet, alexnet, vgg16, vgg11) [15], and three DL inference jobs (classifiaciton, regression, multiout) were used [7]. All DL tasks used Tensorflow, executed in the GPU and containerized as a Docker container. A total of ten workloads were selected, seven distinctive workloads and three random workloads. The characteristics of the workloads applied are shown in Table 5. The sensitivity of the scheduler is evaluated by varying the arrival interval [14,16]. 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.

Baseline schedulers: A max-memory-based Binpack scheduler and interference aware schedulers, such as the Loadbalance and Mystic [14] 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.

B. *Scheduling Performance*

The effects of various workloads on the performance of each scheduling method are analyzed. The task density was designated as the medium. Fig. 7-(a) shows the average JCT for a total of ten workloads. In all workloads except for three, Co-scheML shows the shortest JCT, and the average JCT for all workloads was 844.09 s for Co-scheML, 1089.32 s for Binpack, 943.05 s for Loadbalance, and 967.94 s for Mystic. This results showing the performance of Co-scheML improved approximately 30%, 11%, and 15%, respectively. The makespan for all workloads is shown in Fig. 7-(b). In the makespan aspect, Co-scheML showed the best performance for all workloads except for three, and the average makespan for all workloads for Co-scheML, Binpack, Loadbalance, and Mystic were 1705.1 s, 2156.1 s, 2027.3 s, and 2031.8 s, respectively. It shows a decreased makespan in comparison to Co-scheML of 26%, 18%, and 19%. For workloads 1, 3, 5, and 7, there was a trade-off between the makespan and average JCT. For example, for workload 7, the average JCT of Co-scheML was 1,060 s, a performance decrease of approximately 9% compared to Mystic (970 s), which showed the best performance. However, makespan showed a performance of 2,411 and 2,931 s, an improvement over Co-scheML of approximately 18%. Workload 4 is only composed of a DL inference task, which showed a decreased performance in the average JCT and makespan of 2% and 7%, respectively. The DL inference task is an application that uses fewer resources, which is advantageous in a co-execution owing to less interference. Workloads 2, 8, and 9, in which Co-scheML showed a good performance had an average improvement in the JCT of approximately 22%, 69%, and 12%, and an improved makespan of approximately 24%, 44% and 48% on average, respectively. As the workloads are comprised of resource-consuming applications, which often result in interference between them, Co-scheML showed the greatest advantage in terms of their scheduling. The performance of Co-scheML was also improved by avoiding from OOM using profiling information, compared to one of Loadbalance and Mystic without predicting OOM were.

Fig. 8 shows a comparison of the speedup for each workload. Co-scheML, Binpack, Loadbalance, Mystic demonstrated a speedup of 78%, 56%, 67%, and 75%, respectively. Although the Loadbalance scheduler displayed a good speedup because it shares computing resources fairly well considering the average GPU utilization, it did not show a sufficient performance regarding the makespan and average JCT by not considering the overall resource usage. In particular, it showed poor performance in workload 5, which consists only of HPC applications. Average GPU utilization is not sufficient to characterize HPC applications which have dynamic GPU utilization. Workload 3 suffered the highest performance degradation, which was most affected by interference. It consists of DL training applications with high GPU utilization among workloads. Although applications are scheduled by interference-aware policy, there is relatively

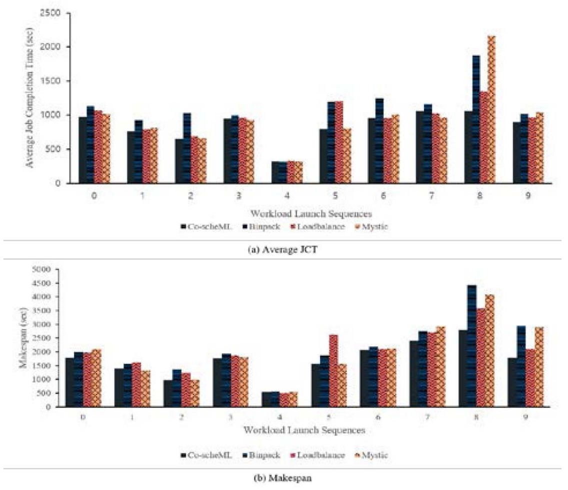

Figure 3. Performance of Co-scheML and others in (a) Average JCT, and (b) Makespan for workload launch sequences

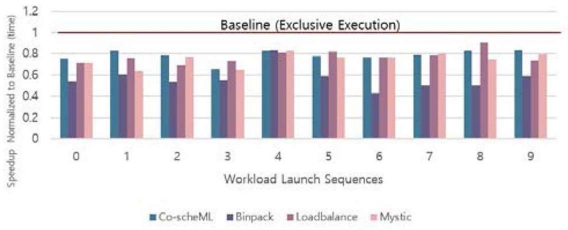

Figure 4. Speed up of schedulers

high performance decline for applications with high GPU utilization. Binpack was the most affected scheduler. As Mystic didn't count on the weight of metrics enforcing performance and calculate the interference of co-execution based on the similarity of metrics, its speedup is low compared to Co-scheML's. Its average JCT and makespan are poor as it is not able to detect OOM.

Fig. 9 shows a graph representing the GPU utilization based on the scheduler for each node. The 12 applications used in this experiment were each executed twice, and the workload consisted of a total of 24 jobs whose launch sequence was randomly generated. During the execution of all workloads, the average GPU utilization was 78% for Co-scheML, 59% for Loadbalance, and 67% for Mystic, showing a higher GPU utilization for Co-scheML by 32% and 16%, respectively. Co-scheML allows each application to use complementary resources by considering interference, resulting in improved GPU utilization.

V. Conclusion

This paper proposes Co-scheML, an interference-aware scheduler, which provides a ML model that predicts the interference values using application profiling data and minimizes interference among GPU applications in a GPU cluster. The experiment showed that the average JCT was improved by up to 30% and that makespan was by 26% for various workloads as compared to conventional schedulers. The resource utilization of the cluster was enhanced by 24% and the performance under various task densities was achieved 23% for JCT and 22% for makespan in average.

Future studies include identifying characteristics in various GPU execution environments and an extension of the scheduling method on multiple GPUs.

References

[1] Yarn, https://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/UsingGpus.html

[2] Kubernetes, https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

[3] NVIDIA Multi Process Service, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[4] Carvalho P, Cruz R, Drummond LM, Bentes C, Clua E, Cataldo E, Marzulo LA. Kernel concurrency opportunities based on GPU benchmarks characterization. Cluster Computing. 2020 Mar 1;23(1):177-88.

[5] InfuxDB, https://www.influxdata.com/

[6] NGC, https://ngc.nvidia.com/

[7] DJINN, https://github.com/LLNL/DJINN

[8] Geng X, Zhang H, Zhao Z, Ma H. Interference-aware parallelization for deep learning workload in GPU cluster. Cluster Computing. 2020 Jan 2:1-4.

[9] Chang CC, Yang SR, Yeh EH, Lin P, Jeng JY. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. in GLOBECOM 2017-2017 IEEE Global Communications Conference 2017 Dec 4 (pp. 1-6). IEEE.

[10] Gu, Jing, et al. "GaiaGPU: Sharing GPUs in Container Clouds." 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). IEEE, 2018.

[11] Song, Shengbo, et al. "Gaia Scheduler: A Kubernetes-Based Scheduler Framework." 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). IEEE, 2018.

[12] Hong, Cheol-Ho, Ivor Spence, and Dimitrios S. Nikolopoulos. "FairGV: fair and fast GPU virtualization." IEEE Transactions on Parallel and Distributed Systems 28.12 (2017): 3472-3485.

[13] Xu, Xin, et al. "Characterization and prediction of performance interference on mediated passthrough GPUs for interference-aware scheduler." 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19). 2019.

[14] Ukidave, Yash, Xiangyu Li, and David Kaeli. "Mystic: Predictive scheduling for gpu based cloud servers using machine learning." 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2016.

[15] Tensorflow CNN benchmarks, https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks

[16] Chen Z, Quan W, Wen M, Fang J, Yu J, Zhang C, Luo L. Deep Learning Research and Development Platform: Characterizing and Scheduling with QoS Guarantees on GPU Clusters. IEEE Transactions on Parallel and Distributed Systems. 2019 Jul 29;31(1):34-50.

[17] Random forest regression, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html