Exploiting Interference-aware GPU Container Concurrency Learning from Resource Usage of Application Execution

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*— The advent of GPGPU (General-Purpose Graphic Processing Unit) containers enlarges opportunities of acceleration and easy-to-use in clouds. However, there is still lack of research on utilizing efficiently GPU resource and managing multiple applications at the same time. Co-execution of applications without understanding applications' execution characteristics may result in low performance caused by their interference problems. To solve the problem, this paper defines resource metrics that causes performance degradation when sharing resource. We calculate the degree of interference during concurrent execution of multi applications using a ML (Machine Learning) method with the metrics. The experiments show that the execution of interference aware groups improves 7% in execution time compared to non-interference aware group in overall. For a workload consisting of several applications, the overall performance was improved by 18% and 25%, respectively, when compared to SJF and random.**

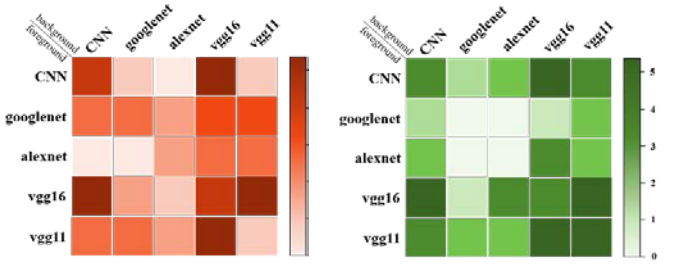
Keywords*— *Interference, Resource Metrics, Profiling, Machine Learning, Interference-aware Scheduling, GPU Virtualization, Container

I. INTRODUCTION

General Purpose Graphics Processing Unit (GPGPU) recently plays an essential role in high-performance computing to achieving high parallelism. The concept of Container as a Service (CaaS) [1] has appeared due to its easy-to-use packaging, scalability, and probability in cloud. The advent of GPU containers emphasizes importance of GPU resource management in clouds.

Co-execution of applications helps to make full advantage of GPU resource. For GPU sharing, NVIDIA recently provides MPS (Multi-Process Service) [2], which runs kernels of multiple applications as a single process. However, MPS technique helps to improve performance only if applications' kernel patterns are known in advance. There are existing studies of scheduling methods for co-execution of applications in GPU resources. The prior works either focus on the methods of co-locating applications based on [3-5], a method of deploying according to GPU usage profiling information in applications [6], or methods of predicting interference among DL (Deep Learning) applications with resource contention features [7, 8]. Interference prediction for various applications has not been studied yet.

Applications co-located on servers cause contentions when they require same resources at the same time and their interference results in performance unpredictable. It comes from that an application shares basic resources such as cache, streaming multi-processor (SM), and I/O, as well as resources



(a) Slowdown of execution time (b) utilization of GPU memory and OOM

Fig. 1. The co-execution of application pairs

(CPU, GPU, memory) generally considered by the scheduler [8]. A method of co-execution with low levels of interference increases performance and resource utilization. However, it is difficult to predict their interference of diverse applications with a variety of resource usage characteristics.

This paper proposes a method to avoid from execution interference using ML techniques. For the prediction of interference, we define resource metrics shared during co-execution of applications. Metrics are gathered from ML benchmarks in Tensorflow [9] and HPC applications provided in NGC (NVIDIA GPU Cloud) [10]. Profiling metrics are utilized in measuring effect of interference with a LR (Linear Regression) method. The contribution of this paper is as follows:

- We evaluate performance degradation coming from co-execution of applications. Detailed resource metrics affected by co-executions are identified from application execution profiling.
- We propose prevention scheme using a ML method for concurrent execution based on the metrics. We confirm that overall performance is improved by 7% with LR.

The rest of the paper is organized as follows. Section II describes a motivation of the paper. Section III described the related work. Section IV describes the metric definitions and a ML method and describes overall interference-aware architecture. Section V presents the results of the experiment and evaluation. The conclusion and future work is followed in Section VI.

II. MOTIVATION

Interference, which comes from resource contention, is decided by co-locating applications sharing GPU concurrently.

The typical image processing applications on GPU such as DL(Deep Learning) tasks (CNN, alexnet, googlenet, vgg16, vgg11) that train ML models on a Tensorflow benchmark are chosen to identify application-specific interference in a pair of co-execution. The applications that execute are image classification using imagenet [9] as dataset. Figure 1 shows difference levels of interference when different tasks are placed together. The degree of interference and GPU memory usage expressed in dark colors.

Figure 1 (a) shows the evaluation of interference, which comes from the execution slowdown, generated from the execution combination of applications. For example, in the case of the CNN-vgg11 and vgg11-CNN pairs, darkness of colors is different. It can be seen that the degree of interference between CNN and vgg11 is different from that of vgg11 to CNN. Figure 1 (b) shows the utilization of GPU memory usage. The application pair in memory contention in (b) does not match to the one in execution slowdown in (a). That means that some interference caused not only by GPU memory but also by multiple shared resources.

According to the paper [7], the interference among applications may be affected by kernel length, memory, data transfer, and so on, but it may not be sufficient to predict interference. The actual resource usage patterns of various applications are different from the ML applications used in the above experiments. For example, ML applications utilize GPU and GPU memory consistently during their execution as shown in the resource usage patterns in Figure2 (a) CNN and (b) vgg16. However, Lammps, an HPC application in Figure 2 (c), shows dynamic usage of memory and GPU, especially, the increase of the usage just before the end of the execution. Qmcpack, a HPC application, has a significant increase in memory usage and GPU utilization, approximately 95 seconds after application is initiated (shown in Figure (d)). Due to these different resource usage patterns, it is difficult to predict application-specific interference for various ML and HPC applications.

III. Related works

Related work on GPU resource sharing in container-based framework and GPU resource management scheduler is following.

A. GPU resource sharing scheduling

A lot of GPU sharing technologies are being introduced to increase resource utilization of GPU clusters and cloud servers. For Diab [11], several users share GPU resource and propose a system that can co-execute tasks. It intercepts the CUDA API and provides two kernels for execution. However, this scheduling method is only possible for application targets with repetitive or distinct resource usage characteristics. [4, 5] share GPU resource in a manner that is divided into a certain size and allocated to containers that run through a cost tree to place and assign GPUs. Nevertheless, these only take into account minimum resource requirements of running jobs which lead to performance degradation when co-execute. Targets of co-execution applications should share GPUs, including not only ML applications, but also HPC applications with inconsistent resource usage patterns. We avoid resource contention by collecting and using detailed resource usage information of various application.

B. GPU interference-aware scheduling with ML

With the emergence of technologies that implement multiple applications on GPUs, many scheduling techniques have been introduced to avoid resource contention that occurs in co-execution. Xu [7] implemented ML-based interference-aware scheduler by defining features of applications running on GPUs. In simple applications, performance is greatly affected by kernel length, but interference may vary for ML applications. However, further feature definitions of affected resources are needed because actual evaluation was performed with resource usage patterns as a repeated ML application. The paper of [12] proposes a work placement framework using the DRL model in a cluster environment with GPU servers. The training conducts and arranges tasks with low level of interference when multiple applications coexist by inputting worker id, CPU, and GPU. To reduce the impact of interference with the usage value of CPU and GPU, further details of resources to be considered are needed.

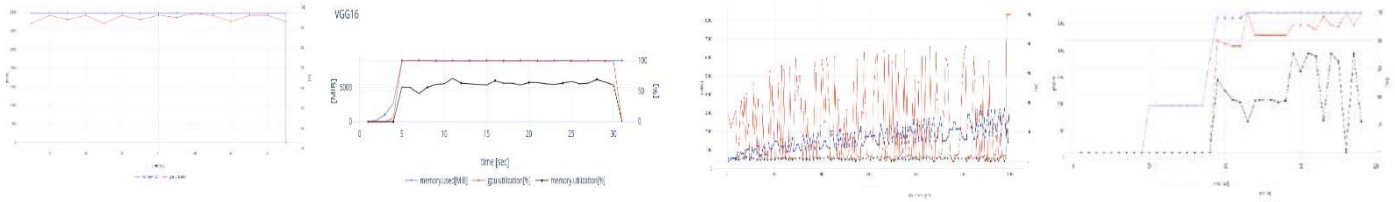
IV. Interference Prediction

Interference prediction process includes identifying resources, which affects performance during their sharing, collecting their metrics, and then applying the metrics to linear regression model for the prediction.

A. Metrics used for profiling resource usage

Co-execution of applications causes interference due to contention between resources in GPUs, leading to performance degradation problems. Applications compete in resources such as SM (Stream Multi-processors), GPU memory, DRAM and cache [8, 13, and 14]. However, A GPU architecture has been recently integrated into the L1 cache with SHEM (Shared Memory) and texture cache [15]. L1 and texture cache throughput defined in the above papers are also integrated or divided, so correct information on correct metrics are required.

Table 1 shows the detailed metrics of resources that affect performance in co-execution, obtained during profiling of each related resource. Each metric is collected using NVIDIA profiled tools nvprof, nsight and nvidia-smi.



(a) CNN (b) vgg16 (c) Lammps [22] (d) Qmcpack [22]

Fig. 2. Resource usage pattern in ML and HPC applications

TABLE III. RESOURCE METRICS

Metric	Resource	Metric	Resource
GPU utilization average	SM	*GPU memory utilization average*	Global Memory
SM efficiency average	SM	*Device to host throughput*	DRAM
Warp efficiency average	SM	*Host to Device throughput*	DRAM
IPC	SM	*GLD throughput*	Cache
Occupancy average	SM	*GST throughput*	Cache
GPU memory used max	Global Memory	*Execution time*	•
GPU memory used average	Global Memory		

Resource metrics are following:

SM: Performance metrics for SM include GPU utilization average, SM efficiency average, Warp efficiency average, IPC, and Occupancy average. GPU utilization average is the average of the times one or more kernels of an application have been running on the GPU. SM efficiency average represents an average of the time that one or more warp is enabled on a particular multiprocessor in the GPU as a percentage. Warp efficiency average means the average number of active threads per warp in a multiprocessor. IPC is the number of instructions executed per cycle. Occupancy average refers to the average number of active warps per active cycle supported by multi processors.

Global Memory: Global memory means GPU memory. GPU memory used max is a metric that represents the maximum amount of GPU memory that an application uses when running. GPU memory used average represents the average value of GPU memory used during application execution. GPU memory utilization average is the average of the percentage of time that the GPU memory is read or written over period during the application execute.

DRAM: Device to host throughput represents the throughput of data moving from global memory to CPU memory. Host to device throughput represents the throughput of data moving from CPU memory to GPU memory.

Cache: GLD (Global memory Load) throughput metric includes transactions served by L1 and L2 caches. This metric represents the amount of cache hit when loading into global memory. GST (Global memory Store) throughput is also related to L1 and L2 caches, and conversely, cache hits when storing to global memory. And the execution time of each application was collected.

B. Linear regression modeling

To predict potential interferences in HPC and DL applications, the metrics defined above are collected by each application and used in the LR to infer the interference values between applications.

From given dataset $\{y_i, x_{i1}, \dots, x_{ip}\}_{i=1}^{n}$, a LR is modeled between the predictable (dependent) variable y_i and the collection x_i of p. Expression $\mathrm{y} = \mathrm{X}\beta + \varepsilon$ is a vector representation of the predictor y_i modelling.

Here β is a parameter vector and it is interpreted as a partial derivative of predictor variable. E includes all error factors for predictor y. In addition, MSE (Mean Square Error) was used to minimize the sum of squares of errors for parameter (β). Therefore, the prediction of interference y is obtained from X which is a set of resource metrics by applications to identify possible interference in co-execution. For example, the interference was calculated by applying LR using metrics collected from the five tensorflow benchmark applications used above. Table 2 below shows the interference result of the application pair. The results of interference is based on 1 and normalized by each application based on the overall metric value of resources.

TABLE IV. INTERFERENCE VALUES

Interference values					
foregound / background	CNN	googlenet	alexnet	Vgg16	Vgg11
CNN	1.25	1.140	0.936	-1	1.047
googlenet	1.326	1.219	1.014	1.184	1.123
alexnet	1.285	1.176	0.973	1.145	1.084
Vgg16	-1	1.223	1.018	1.191	-1
Vgg11	1.299	1.188	0.984	-1	1.096

The results show that CNN and alexnet (0.936), alexnet and alexnet (0.973) have interference values lower than 1. These results can be inferred as low interferences when the alexnet application characteristics are low metric values associated with SM resources, and only relatively high GPU utilization average metric values are implemented with CNN. For Vgg16, vgg11, the memory-related metric value is very high, requiring more than the physical GPU memory resulting in OOM. In such a case, the interference value is marked as -1. More applications are investigated and applied to the LR model to predict the interference value.

The result of the interference would be used to schedule the application execution order of the application workload. Since the degree of influence of the applications is different from each other, the smaller the deviation of interference value of the same application pair, the smaller the influence of each other. Therefore, when several applications exist, the execution order can be determined by arranging application pairs having a small deviation value at the same time according to the range of interference values. For example, googlenet – CNN should be executed first because deviation of interference value is 0.039.

V. EXPERIMENT

A. Experiment setup

In the experimental environment of Kubernetes based private GPU cluster, the performance and resource usage efficiency were evaluated by applying the interference prediction method. The cluster system consists of a master node with two 1 GPUs and a computing node with two GPUs. Details of the cluster configuration are shown in Table III.

TABLE III. EXPERIMENTAL SETTINGS

	CPU	GPU
Architecture	Intel(R) Core(TM) i7- 5820K	Nvidia GeForce Titan Xp D5x
Core clock	3.30GHz	1.58GHz
Num of Cores	6 cores	3840 cores
Memory size	32GB	12GB
Threading API	-	Nvidia CUDA 10.1
Compiler	Gcc 5.4.0	Nvidia C Compiler (NVCC8.0)
OS	Ubuntu 16.04.3 LTS	Ubuntu 16.04.3 LTS

Scheduling on Kubernetes: The kubernetes device plug-in was modified so that two applications could be performed simultaneously on single GPU card. In addition, the scheduler of the kubernetes was revised to ensure that applications are paired according to each deployment strategy. After the two applications were paired in the order resulting from each arrangement strategy, the next pair of applications was allowed to be launched at the same time after the execution of both applications. OOM failure occurs if the sum of the application's maximum memory usage is greater than the amount of memory in this experimental environment. Interference results using LR predicted OOM failure with collected metric and excluded pairs of OOM in preprocessing stage. However, other deployment strategies did not have a pre-processing process, so users were required to specify the maximum memory usage for each application. If the sum of the maximum memory usage of the two applications exceeds the memory of the experimental environment, one application is placed in a pending state. Our method was to run when other applications were finished.

Workload: Workload is configured using 5 Tensoflow benchmark applications and 4 HPC applications (Lammps, Gromacs, Qmcpack, Hoomd). The profiling data of the applications was collected using NVIDA profiler according to defined metrics.

Interference values: After calculating the interference of a total of nine application pairs with LR, MSE is 0.168. The value of MSE is lower than that of minimum 0.207 and maximum 0.355 compared to the [7] paper. This shows that the metrics of this paper are more meaningful than the paper [7]. Application pairs with an interface value below Criterion 1 include Gromacs-Lammps (0.964), CNN-Lammps (0.926), and Gromacs-alexnet (0.979). Except in the case of OOM, applications with more than 2 value of interference have the most interference with the Qmcpack-Hoomd pair at approximately 2.160.

B. Comparison of performance using interference

Interference-aware experiments confirm that the performance of the overall work and the utilization of each resource are improved. The experiment evaluates the performance compared to the random placement without information on the interference. In this experiment, the two applications with the lower 10% interference value are composed of a pair of workloads of a total of six tasks, depending on the outcome of the interference prediction.

Execution time: The execution time of the entire application is improved by up to 7% when running with a low interference pair. For the Lammps application and CNN application, it appears that the performance time is the shortest compared to the random. The GPU utilization average metric, which affects performance time is the least likely for CNN at

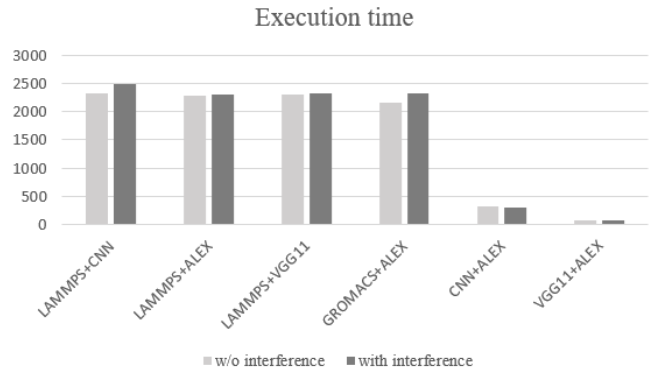


Fig. 3. Comparison of execution time with or without interference-aware

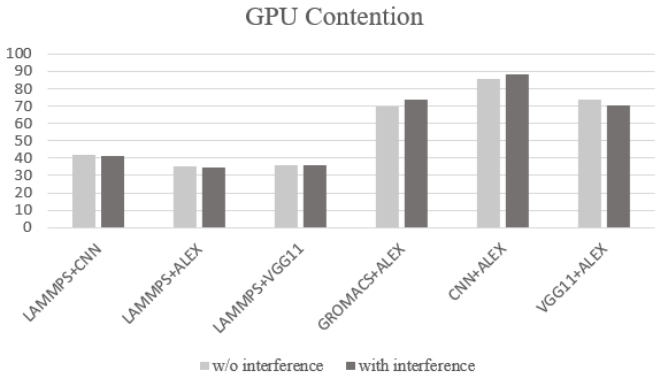


Fig. 4. Comparison of GPU contention with or without interference-aware

about 25, resulting in less resource competition. Moreover, it is expected that there would have been fewer GPU memory content with CNN because of the GPU memory utilization level of about 2% for the Lammps application. The overall performance of the five application pairs has improved, but for the CNN-alexnet application pair, the performance is similar to that of the random. This is because alexnet has a shorter execution time than CNN, which did not affect resource contention.

GPU contention: As GPU computing resource is not sufficiently utilized by GPU contention, GPU utilization which represents to percent of time over the sample period during which kernels were executing on GPU is decreased. As GPU contention via average GPU utilization is measured, it is less competitive than the random deployment for the rest of the application pair except for Gromacs-alexnet and CNN-alexnet applications has. This is because GPU resource usage in applications such as GPU utilization and SM utilization among input metric of LR was reflected. In the case of Gromacs-alexnet, Gromacs' SM efficiency is about 99.93, affected by high active warps. For CNN-alexnet, GPU content was generated by many active threads because CNN applications had the highest warp efficiency of 55.21 out of 9 applications.

GPU memory: Compared to randomly deployed non-aware execution, it has similar GPU memory usage (in Figure 7). However, a total of 5 OOMs occurred in the not interference-aware group. LAMMPS requires approximately 8.3GB of GPU memory and vgg11 requires approximately 8.745GB of GPU memory. In the case of workloads that include LAMMPS and vgg11 applications, the OOM occurred when the random physical execution exceeded the actual

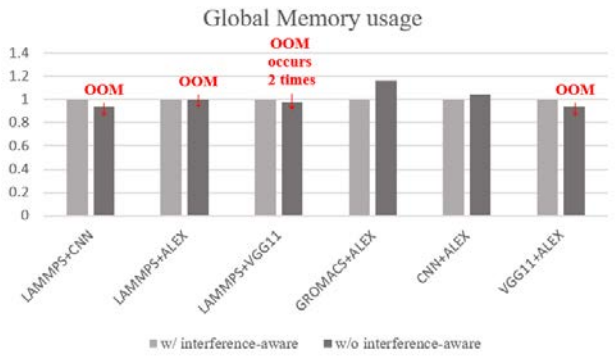


Fig. 5. Comparison of out-of-GPU memory with or without interference-aware

physical memory. Therefore, it is important to implement multi-application by considering GPU memory metric to maximize GPU memory that can be shared while preventing OOM from occurring.

C. *Workload execution sequence placement using interference deviations*

The performance in execution time of the overall tasks is various according to deployment strategies. The deployment strategies used in this experiment are as follows.

- Interference-aware greedy: Select a pair with the smallest interference value and its variation in the current stage based on results of interference prediction using LR.
- Shortest Job Frist (SJF): Select applications with the shortest execution time.
- Random: Select a pair of applications in a random manner.
- Mystic[8]: Select a pair with the least similarity value generated by calculating similarity between pairs of applications

SJF and Random policy are default scheduler that schedule without profile information. Interference-aware greedy proposed in this paper and mystic policy are scheduler based on profile information.

Comparing policies that are not based on profiling to interference-aware greedy policy, the results show that performance improvement of interference-aware greedy strategy (3520s) is about 18% for SJF (4260s) and 25% for Random strategy (4646s) (shown in Figure 8). Applications in the same pair of interference-aware greedy strategy were less affected by each other because the pair was selected for small interference value and its variation, so its performance is better than the other strategies.

When comparing Mystic policy with interference-aware greedy, Mystic takes 4229 seconds and interference-aware greedy takes 3520 seconds. Interference-aware greedy reduced time by about 17% compared to Mystic. Mystic shows better performance than default scheduler by scheduling based on profiling. However, since it simply calculate similarities of metric values between pairs of application, it doesn't consider weight values for each resource. Our policy of modeling interference value to take into account weight values shows better performance for this reason.

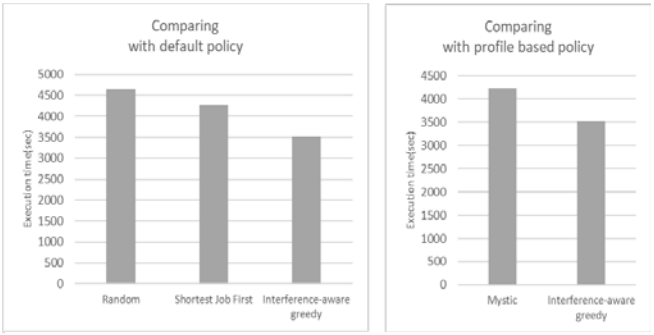


Fig. 6. Comparison of execution time with default policy and profile based policy

With LR modeling, it is possible to predict interference value without having to perform all pairs of applications. It leads to better performance than the other strategies. Arranging an order of execution using prediction of interference values helps improve overall performance. Based on each applications' resource metrics usage profiles, it performs (1) calculating interference values to reduce contention, (2) collocating the jobs' pair to improve utilization of resources and to conserve the performance.

VI. Conclusion

The advent of GPU containers emphasizes importance of GPU resource management in clouds. This paper identifies the interference that occurs when GPU resources are shared, based on the resource usage metrics of applications. The resource usage information of running applications defines metrics that affect performance during co-execution. The collected information is applied to LR model to calculate interference values for a pair of applications. The experiment results showed improvement by up to 7% in GPU utilization. In case of workload composed several applications, overall performance was about 18% better than SJF and 25% better than random deployment when arranged tasks using interference values.

In future work, we plan to extend this interference-aware scheduler on multi-node cluster server for co-locating several applications simultaneously.

Acknowledgment

This research was supported by the National Research Foundation of Korea(NRF) grants funded by the Korean government(MSIT) No. 2015M3C4A7065646 and No. 2020R1H1A2011685.

References

[1] Amazon EC2, https://aws.amazon.com/ec2/

[2] MPS, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[3] Chang, Chia-Chen, et al. "A kubernetes-based monitoring platform for dynamic cloud resource provisioning." GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, 2017.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Abadi, Martín, et al. "Tensorflow: A system for large-scale machine learning." 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). 2016.

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

[11] Diab, Khaled M., M. Mustafa Rafique, and Mohamed Hefeeda. "Dynamic sharing of GPUs in cloud systems." 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. IEEE, 2013.

[12] Bao, Yixin, Yanghua Peng, and Chuan Wu. "Deep Learning-based Job Placement in Distributed Machine Learning Clusters." IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE, 2019.

[13] Tanasic, Ivan, et al. "Enabling preemptive multiprogramming on GPUs." 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). IEEE, 2014.

[14] Ukidave, Yash, et al. "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus." 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing. IEEE, 2014.

[15] GPU architecure, https://www.hardwarezone.com.sg/feature-what-you-need-know-about-ray-tracing-and-nvidias-turing-architecture/introduction-turing