Resource Aware GPU Scheduling in Kubernetes Infrastructure

Aggelos Ferikoglou
Microprocessors and Digital Systems Laboratory, ECE,
National Technical University of Athens, Greece

Dimosthenis Masouros
Microprocessors and Digital Systems Laboratory, ECE,
National Technical University of Athens, Greece

Achilleas Tzenetopoulos
Microprocessors and Digital Systems Laboratory, ECE
, National Technical University of Athens, Greece

Sotirios Xydis
Department of Informatics and Telematics, DIT, Harokopio University of Athens, Greece

Dimitrios Soudris
Microprocessors and Digital Systems Laboratory, ECE,
National Technical University of Athens, Greece

Abstract

Nowadays, there is an ever-increasing number of artificial intelligence inference workloads pushed and executed on the cloud. To effectively serve and manage the computational demands, data center operators have provisioned their infrastructures with accelerators. Specifically for GPUs, support for efficient management lacks, as state-of-the-art schedulers and orchestrators, threat GPUs only as typical compute resources ignoring their unique characteristics and application properties. This phenomenon combined with the GPU over-provisioning problem leads to severe resource under-utilization. Even though prior work has addressed this problem by colocating applications into a single accelerator device, its resource agnostic nature does not manage to face the resource under-utilization and quality of service violations especially for latency critical applications.

In this paper, we design a resource aware GPU scheduling framework, able to efficiently colocate applications on the same GPU accelerator card. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration frameworks. We show that our scheduler can achieve **58.8%** lower end-to-end job execution time 99%-ile, while delivering **52.5%** higher GPU memory usage, **105.9%** higher GPU utilization percentage on average and **44.4%** lower energy consumption on average, compared to the state-of-the-art schedulers, for a variety of ML representative workloads.

2012 ACM Subject Classification Computing methodologies; Computer systems organization → Cloud computing; Computer systems organization → Heterogeneous (hybrid) systems; Hardware → Emerging architectures

Keywords and phrases cloud computing, GPU scheduling, kubernetes, heterogeneity

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.4

Funding This work has been partially funded by EU Horizon 2020 program under grant agreement No 825061 EVOLVE (`https://www.evolve-h2020.eu`).

1 Introduction

In recent years, the adoption of artificial intelligence (AI) and machine learning (ML) applications is increasing rapidly. Several major Internet service companies including Google, Microsoft, Apple and Baidu have observed this trend and released their own intelligent personal assistant (IPA) services, e.g. Siri, Cortana etc., providing a wide range of features.

12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021).
Editors: João Bispo, Stefano Cherubin, and José Flich; Article No. 4; pp. 4:1–4:12
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Compared to traditional cloud applications such as web-search, IPA applications are significantly more computationally demanding [13]. Accelerators, such as GPUs, FPGAs, TPUs and ASICs, have been shown to be particularly suitable for these applications from both performance and total cost of ownership (TCO) perspectives [13]. With the increase in ML training and inference workloads [18, 13], cloud providers begin to leverage accelerators in their infrastructures, to catch up with the workload performance demands. This trend is also evident as Amazon AWS and Microsoft Azure have started offering GPU and FPGA based infrastructure solutions.

In particular, for the case of ML inference oriented tasks, public clouds have provisioned GPU resources at the scale of thousands of nodes in data-centers [25]. Since GPUs are relatively new to the cloud stack, support for efficient management lacks. State-of-the-art cluster resource orchestrators, like Kubernetes [9], treat GPUs only as a typical compute resource, thus ignoring their unique characteristics and application properties. In addition, it is observed that users tend to request more GPU resources than needed [3]. This tendency is also evident in state-of-the-art frameworks like Tensorflow which by default binds the whole card memory to an application. This problem, also known as *over-provisioning*, combined with the resource agnostic scheduling frameworks lead to under-utilization of the GPU-acceleration infrastructure and, thus, quality of service (QoS) violations for latency critical applications such as ML inference engines. To overcome the aforementioned issues, real-time monitoring, dynamic resource provisioning and prediction of the future status of the system is required, to enable the efficient utilization of the underlying hardware infrastructure by guiding the GPU scheduling mechanisms.

In this paper, we propose a novel GPU resource orchestration framework that utilizes real-time GPU metrics monitoring to assess the real GPU resource needs of applications at runtime and based on the current state of a specified card decide whether two or more application can be colocated. We analyze the inherent inefficiencies of state-of-the-art Kubernetes GPU schedulers concerning the QoS and resource utilization. The proposed framework estimates the real memory usage of a specified card and predicts the future memory usage, enabling better inference engine colocation decisions. We show that our scheduler can achieve **58.8%** lower end-to-end job execution time 99%-ile for the majority of used inference engine workloads, while also providing **52.5%** higher GPU memory usage, **105.9%** GPU utilization percentage average and **44.4%** lower energy consumption compared with the Alibaba GPU sharing scheduler extension.

2 Related Work

The continuous increase in the amount of containerized workloads uploaded and executed on the cloud, has revealed challenges concerning the container orchestration. Workload co-location and multi-tenancy exposed the interference agnostic nature of the state-of-the-art schedulers [26] while the integration of accelerator resources for ML applications revealed their resource unawareness [25]. To enable better scheduling decisions, real-time [8] or even predictive [19] monitoring is required to drive the orchestration mechanisms. Extending to the case of GPU accelerators, real-time GPU monitoring can allow the colocation of containers on the accelerator in a conservative manner to avoid out-of-memory issues [25].

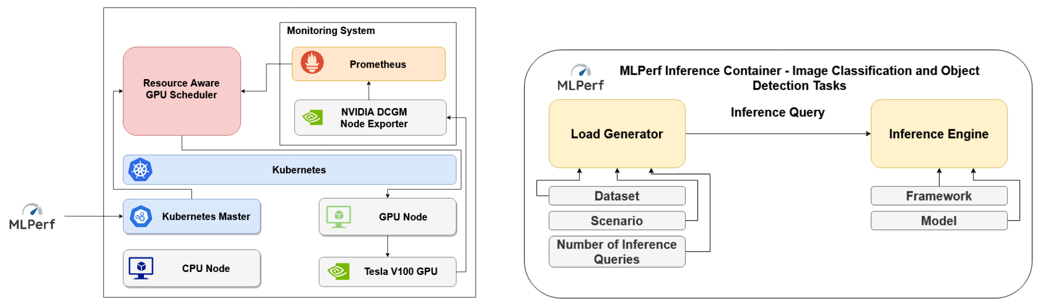
Container orchestration on GPU resources has been in the center of attention of both academia and industry. Throughout the years, various GPU scheduling approaches have been proposed. Ukidave et al. [27] and Chen et al. [10] have proposed GPU runtime mechanisms to enable better scheduling of GPU tasks either by predicting task behavior or reordering queued tasks. More recent works [17, 11] have introduced docker-level container sharing solutions by allowing multiple containers to fit in the same GPU, as long as the active working set size of all the containers is within the GPU physical memory capacity. As distributed deep neural network (DNN) training based applications have started taking advantage of multiple GPUs in a cluster, the research community proposed application specific schedulers [20] that focus on prioritizing the GPU tasks that are critical for the DNN model accuracy. Hardware support for GPU virtualization and preemption were also introduced. Gupta et al. [12] implemented a task queue in the hypervisor to allow virtualization and preemption of GPU tasks while Tanasic et al. [24] proposed a technique that improves the performance of high priority processes by enabling GPU preemptive scheduling. The integration of GPU sharing schemes on GPU provisioned cloud infrastructures managed by Kubernetes is a trend that is also observed. Yeh et al. proposed KubeShare [29], a framework that extends Kubernetes to enable GPU sharing with fine-grained allocation, while Wang et al. [28] introduced a scheduling scheme that leverages training job progress information to determine the most efficient allocation and reallocation of GPUs for incoming and running jobs at any time.

Regarding container orchestration within GPU environments, Kubernetes itself includes experimental support for managing AMD and Nvidia GPUs across several nodes. Kubernetes GPU scheduler extension [4] exposes a card as a whole meaning that a container can request one or more GPUs. Even though this implementation does not provide fractional GPU usage, it allows better isolation and ensures that applications using a GPU are not affected by others. To overcome this problem, the authors in [1] proposed a GPU sharing scheduling solution which relies on the existing working mechanism of Kubernetes. Alibaba GPU sharing extension aims to improve the utilization of GPU resources by exposing the memory of a card as a custom Kubernetes resource, thus, allowing containers to specify their required amount of memory. Even though this approach allows the concurrent execution of multiple containers, its resource agnostic nature makes it dependable on the credibility of the memory requests. Kube-Knots [25] overcomes this limitation by providing a GPU-aware resource orchestration layer that addresses the GPU orchestration problem. Kube-Knots dynamically harvests spare compute cycles by enabling the co-location of latency-critical and batch workloads, thus, improving the overall resource utilization. This way, it manages to reduce QoS violations of latency critical workloads, while also improving the energy consumption of the cluster. However, its predictive nature fails to face the problem of container failures due to incorrect memory usage predictions and thus GPU memory starvation.

3 Experimental Setup & Specifications

We target high-end server systems equipped with GPU acceleration capabilities found under today's data-center environments. Specifically, our work targets an ML-inference cluster, where a GPU-equipped node is responsible for serving the computational demands of inference queries effectively. In the proposed framework, whenever an inference engine arrives on the cluster, the Kubernetes master redirects it to our custom resource aware GPU scheduler. By leveraging real-time GPU monitoring and prediction, our scheduler decides whether to schedule it on the GPU, or enqueue the task on a priority queue and delay the execution until there are enough GPU resources available. Figure 1 shows an overview of our experimental setup.

Hardware Infrastructure Characterization. All of our experiments have been performed on a dual-socketed Intel® Xeon® Gold 6138 server equipped with an NVIDIA V100 GPU accelerator card, the specifications of which are shown in Table 1. On top of the physical



(a) Proposed Scheduling Framework. **(b)** MLPerf Inference Engine Architecture.

Figure 1 Proposed scheduling framework and MLPerf inference engine architecture.

machine we have deployed three virtual machines, which serve as the nodes of our cluster, using KVM as our hypervisor. The V100 accelerator is exposed on the inference-server VM (24 vCPUs, 32GB RAM) using the IOMMU kernel configuration, while the rest of the VMs (8 vCPUs, 8GB RAM each) are utilized to deploy critical components of our system, such as the master of our Kubernetes cluster and our monitoring infrastructure.

Software & Monitoring Infrastructure Characterization. On top of the VMs, we deploy Kubernetes container orchestrator (v1.18) combined with Docker (v19.03) which is nowadays the most common way of deploying cloud clusters at scale [15]. Our monitoring system consists of two major components, NVIDIA's Data-Center GPU Manager exporter (DCGM) [5] along with Prometheus [6] monitoring toolkit. DCGM exports GPU metrics related to the frame buffer (FB) memory usage (in MiB), the GPU utilization (%) and the power draw (in Watts). In particular, a DCGM exporter container is deployed on top of each node of the cluster through Kubernetes. This container is responsible for capturing and storing the aforementioned metrics into our Prometheus time-series database every specified interval. We set the monitoring interval equal to 1 second to be able to capture the state of our underlying system at run-time. Finally, metrics stored in the Prometheus time-series are accessed from our custom Kubernetes scheduler by performing Prometheus-specific PromQL queries, as described in section 5.

Inference Engine Workloads. For the rest of the paper, we utilize MLPerf Inference [21] benchmark suite for all of our experiments, which is a set of deep learning workloads performing object detection and image classification tasks. As shown in Figure 1b, each MLPerf Inference container instance consists of two main components, *i)* the Inference Engine and *ii)* the Load Generator. The Inference Engine component is responsible for performing

Table 1 CPU & GPU Specifications.

Intel® Xeon® Gold 6138	
Cores/Threads	20/40
Sockets	2
Base Frequency	2.0 GHz
Memory (MHz)	132 GB (2666)
Hard Drive	1 TB SSD
OS (kernel)	Ubuntu 18 (4.15)

NVIDIA V100	
Architecture	Volta
Comp. Cap.	7.0
CUDA Cores	5120
Memory Size	32 GB HBM2
Interface	PCIe 3.0 x16
Sched. Policy	Preemptive

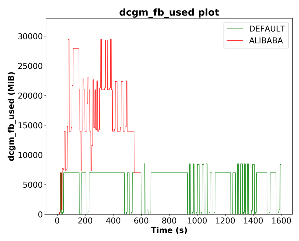
the detection and classification tasks. It receives as input the pre-trained DNN model used during inference (e.g. ResNet, Mobilenet etc.) as well as the corresponding backend framework (e.g. PyTorch, Tensorflow etc.). The Load Generator module is responsible for producing traffic on the Inference Engine and measure its performance. It receives as input the validation dataset (e.g. Imagenet, Coco) as well as the examined scenario and the number of inference queries to be performed. The scenario can be either Single stream (Load Generator sends the next query as soon as the previous is completed), Multiple stream (Load Generator sends a new query after a specified amount of time if the prior query has been completed, otherwise the new query is dropped and is counted as an overtime query), Server (Load Generator sends new queries according to a Poisson distribution) and Offline (Load Generator sends all the queries at start). Considering the above inputs, the Load Generator performs streaming queries to the Inference Engine and waits for the results. For the rest of the paper, we utilize the Single Stream scenario and evaluate our inference engine through the 99%-ile of the measured latency.

4 Motivational Observations and Analysis

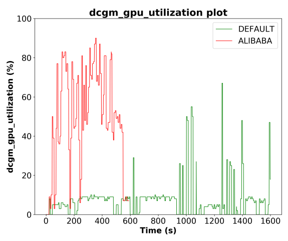
Latest advancements in the micro-architecture of NVIDIA's GPUs allow the transparent, cooperative execution of CUDA applications on the underlying accelerator, either through CUDA's streams [2] or through CUDA's Multi-Process Service (MPS) [22] capabilities. These functionalities increase the utilization of GPU accelerators, thus, offering increased computing capacity, yet, state-of-the-art frameworks, such as Kubernetes do not provide mechanisms that expose them to end-users. In fact, Kubernetes default GPU scheduler [4] mechanism provides exclusive access to applications requesting GPU accelerators. Even though, this approach allows isolation and ensures that applications using a GPU do not interfere with each other, it can cause high resource under-utilization or QoS violations, especially in deep-learning inference scenarios on high-end GPUs, which have low requirements in terms of CUDA cores and memory. In order to allow more prediction services to share the same GPU and, thus, improve their QoS and the utilization of the card, partitioning of the GPU memory resource is required. Towards this direction, Alibaba offers a GPU sharing extension [1], which allows the partitioning of the GPU memory. This scheduler allows end-users to define the requirements of their workloads in terms of GPU memory and combines this information with the total available memory of the GPU to decide whether two or more inference engines can be colocated or not.

To demonstrate the inefficiency of the Kubernetes GPU scheduler extension compared with Alibaba GPU sharing extension, we perform a straight comparison between them for the scheduling of a workload that consists of 6 inference engines from the MLPerf suite.

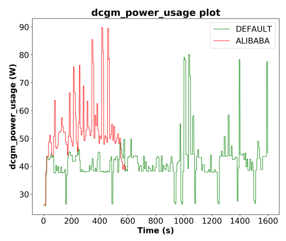
Figure 2 shows the GPU memory utilization (MB), the CUDA cores utilization (%) and the power usage signals of the inference engine workload for the above-mentioned schedulers. As shown, the Kubernetes GPU scheduler extension has an average memory utilization of 5GB, which can be considered relatively low compared to the available 32GB memory of the underlying GPU card. The same observation can be made for the GPU utilization signal (7.22% on average) and the power consumption (41.5 Watts on average), as the GPU binding per inference engine leads to resource under-utilization. On the other hand, the average GPU memory usage for the Alibaba GPU sharing extension is 16GB, which is x3.24 higher. Similarly, we see an average of 49% utilization improvement (x6.8 increment) and an average of 52.9 Watts higher power consumption (x1.28 increase). It also leads to a 52.8% decrease of the average energy consumption as Kubernetes GPU scheduler extension consumption is



(a) GPU Memory Usage Signal

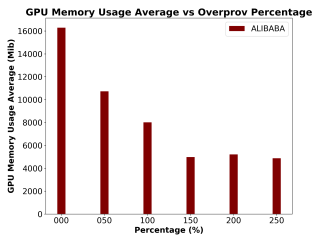


(b) GPU Utilization Percentage Signal

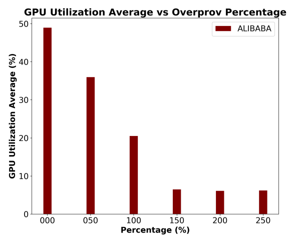


(c) GPU Power Usage Signal

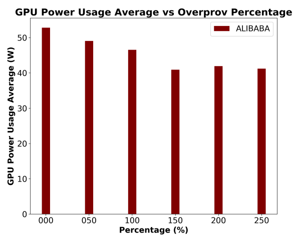
Figure 2 GPU memory usage, utilization percentage and power usage signals for Kubernetes GPU scheduler extension and Alibaba GPU sharing extension.



(a) Memory Usage Average



(b) GPU Utilization Average



(c) Power Usage Average

Figure 3 Memory usage, GPU utilization and power consumption averages vs over-provisioning percentage for Alibaba GPU sharing scheduler extension.

66.4 kJ and Alibaba GPU sharing extension consumption is 31.3 kJ. Finally, we observe that the overall inference engine workload duration using the Alibaba GPU sharing extension is x2.67 faster than the Kubernetes GPU scheduler extension, meaning that the card sharing improves the overall workload duration.

Even though Alibaba's scheduler outperforms the default one, it highly depends on the provisioning degree of the inference engine memory request. For example, if an inference engine requests more memory than it actually needs, this may affect future GPU requests of other inference engines, which will not be colocated, even though their memory request can be satisfied. To better understand the impact of the resource over-provisioning problem within Alibaba's scheduler, we perform 6 different experiments, where we schedule the same inference-engine task, each time with a different memory over-provisioning percentage, ranging from 0% to 250%. Figure 3 depicts the memory usage, the utilization percentage and the power usage averages. For low over-provisioning percentages, Alibaba GPU sharing extension leads to high resource utilization due to the inference engine colocation. However, as shown, it is not able to efficiently sense and handle user-guided over-provisioning scenarios.

5 Resource-aware GPU Sharing Kubernetes scheduler

Figure 4 shows an overview of our proposed resource-aware GPU-sharing scheduler. Whenever an inference engine is scheduled from the custom scheduler, the corresponding workload enters a priority queue which defines their scheduling order. The inference engine assigned priority is proportional to the corresponding GPU memory request. As a result the scheduler always tries to schedule the inference engines with the bigger memory requests. If a workload is chosen to be scheduled, the following three co-location mechanisms are successively executed:

Resource Agnostic GPU Sharing. Our custom scheduler holds a variable that is used as an indicator of the available GPU memory. This variable is initialized to the maximum available memory of the used card in the GPU node. If the inference engine memory request is smaller than the value of this variable, the request can be satisfied and the workload can be scheduled. Whenever an inference engine is scheduled, the value of the indicator variable is decreased by the amount of the memory request. Resource agnostic GPU sharing does not face the memory over-provisioning problem as it is not possible to know a priory that the amount of requested memory is actually the amount that the workload needs to run properly. In our proposed scheduler, we overcome this problem by using real-time memory usage data by our GPU monitoring sub-system. The monitoring system data are collected by performing range queries to Prometheus time series database.

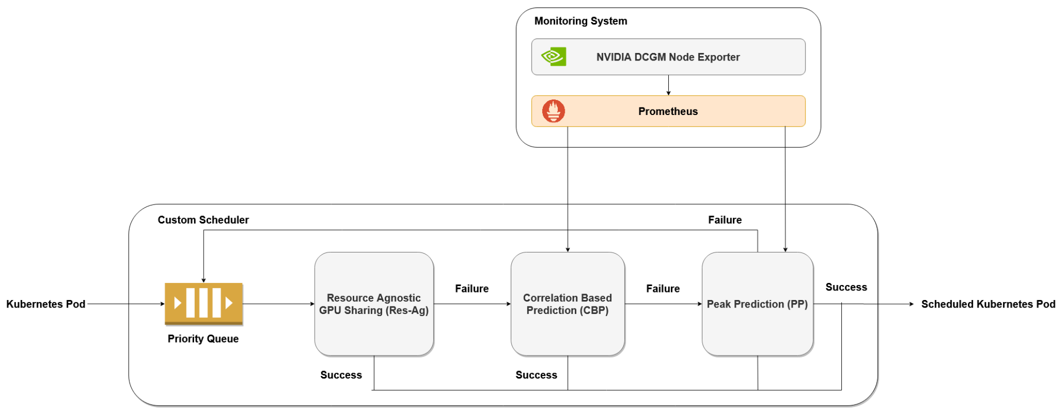
Correlation Based Prediction. Correlation Based Prediction (CBP) [25] provides an estimation for the real memory consumption on a GPU node. The estimation is defined from the 80%-ile of the GPU memory usage rather than the maximum usage. The basic idea of this algorithm is that GPU applications, on an average, have stable resource usage for most of their execution, except for the times when the resource demand surges. In addition, the whole allocated capacity is used for a small portion of the execution time while the applications are provisioned for the peak utilization. CBP scheduler virtually resizes the running workloads for a common case, letting more pending inference engines to be colocated.

In order to have an accurate estimation, low signal variability is required. The signal variability is calculated using the coefficient of variation (CV) metric [7]. If CV is lower than a defined threshold, the memory usage is defined by calculating the 80%-ile of the signal. The free GPU memory estimation is equal to the difference of the maximum available GPU memory and the memory usage estimation. Finally, if the memory request can be satisfied the workload is scheduled. Otherwise the Peak Prediction algorithm is used.

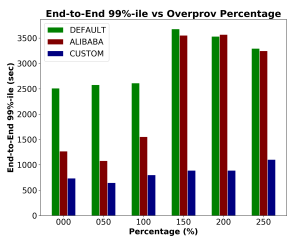
Peak Prediction. Peak Prediction (PP) [25] relies on the temporal nature of peak resource consumption within an application. For example, a workload that requires GPU resources will not allocate all the memory it needs at once. So, although the GPU memory request cannot be satisfied at the scheduling time, it may be satisfied in the near future. The memory usage prediction is based on an auto regressive model (AR) [23]. For an accurate prediction the auto correlation value of order k is calculated. If the auto correlation [16] value is larger than a defined threshold, auto regression of order 1 is performed using linear regression (LR) [14]. If the predicted GPU memory request can be satisfied from PP, then the workload is scheduled. Otherwise, the workload is put into the priority queue and our algorithm attempts to schedule the next available workload from the queue. As we see, PP scheduling decisions depend on the accuracy of the used auto-regressive model and thus linear regression. Even though linear regression is a simplistic approach for predicting the unoccupied memory of the GPU, it can accurately follow the memory utilization pattern (as we further analyze in section 6). In addition, its low computing and memory requirements, allows the PP mechanism to provide fast predictions at runtime with minimal resource interference.

6 Experimental Evaluation

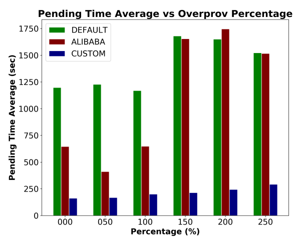
We evaluate our custom scheduler through a rich set of various comparative experiments. We consider inference engine workloads for differing intervals between consecutive inference engine arrivals. In each comparative analysis the exact same workload is fed to the Kubernetes GPU scheduler extension [4], the Alibaba GPU sharing extension [1] and the custom scheduler multiple times. Each time a different memory over-provisioning percentage is used.



■ **Figure 4** Resource Aware GPU Colocation Algorithm.



(a) End-to-End Job Execution 99%-ile.



(b) Pending Time Average.

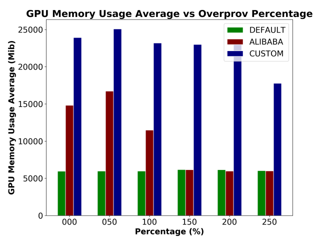
■ **Figure 5** End-to-end job execution 99%-ile and pending time average vs over-provisioning percentage homogeneous workload with MIN=5 and MAX=10.

We provide analysis for homogeneous, i.e., scaling out a single inference service, and heterogeneous workload scenarios. Each workload creates a different inference engine by using the MLPerf inference container we described in section 3. An inference engine is fully defined from the used backend (e.g. Tensorflow, PyTorch etc.), the pre-trained model, the dataset, the scenario, the GPU memory request and the number of inference queries that are going to be executed. The interval between two consecutive inference engine arrivals is defined by the values MIN and MAX (random number in [MIN, MAX] interval in seconds).

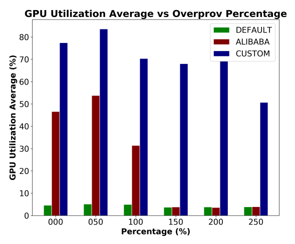
6.1 Homogeneous Workload Evaluation

For homogeneous workload, we consider the Tensorflow ssd-mobilenet engine which uses the Coco (resized 300x300) dataset while each inferences engine executes 1024 queries. Each inference engine requires approximately 7 GB of GPU memory meaning that in a card with 32 GB memory, only 4 can be safely colocated.

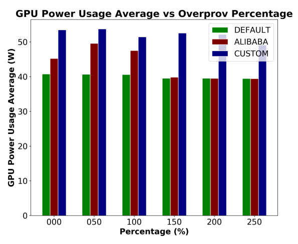
Figure 5 shows the end-to-end 99%-ile and the pending time average for all the available schedulers, for different over-provisioning percentages. Custom scheduler offers x6.6 (on average) lower pending time average and x3.6 (on average) lower end-to-end 99%-ile from Kubernetes default GPU scheduler extension. It also offers x5.2 (on average) lower pending time average and x2.8 (on average) lower end-to-end 99%-ile from Alibaba GPU sharing scheduler extension. However, due to the colocation of multiple inference engines, custom scheduler's decisions lead to higher inference engine 99%-ile average.



(a) Memory Usage Average.



(b) GPU Utilization Average.



(c) Power Usage Average.

Figure 6 Memory usage average, GPU utilization average and power consumption averages vs over-provisioning percentage for homogeneous workload with MIN=5 and MAX=10.

To understand the above mentioned results, the way each mechanism schedules workloads to the GPU should be analyzed. Kubernetes default GPU scheduler extension allocates the whole GPU resource for each inference engine, leading to severe increase of the pending time average (the average time an inference engine waits in the priority queue). The Alibaba GPU sharing scheduler extension uses a resource agnostic colocation mechanism to schedule workloads in the same card. In particular, for over-provisioning percentage equal to 0 % (7 GB memory request) 4 inference engines can be collocated, for 50 % (10 GB memory request) 3 inference engines can be colocated, for 100 % (14 GB memory request) 2 inference engines can be colocated and for 150 %, 200 % and 250 % each inference engine allocates the whole GPU resource. As a result, Alibaba GPU share scheduler extension has similar results with our custom scheduler for over-provisioning percentages equal to 0 % and 50 %. Custom scheduler handles the memory over-provisioning problem in a better way because of its resource aware nature. Figure 5 shows that the proposed scheduler has similar behavior concerning the quality of service metrics for all the different over-provisioning percentages.

Figure 6 shows the memory usage, the utilization percentage and the power consumption averages for all the available schedulers for different over-provisioning percentages. Custom scheduler leads to x3.7 higher memory usage, x16 higher GPU utilization and x1.3 higher power consumption from Kubernetes default GPU extension. It also leads to x2.2 higher memory usage, x2.9 higher GPU utilization and x1.2 higher power consumption from Alibaba GPU sharing scheduler extension. Although we observe an increase in the power usage average, it should be clear that due to the lower overall workload duration the average energy consumption is x2.6 lower from the Kubernetes GPU scheduler extension and x2.2 lower from the Alibaba GPU sharing extension.

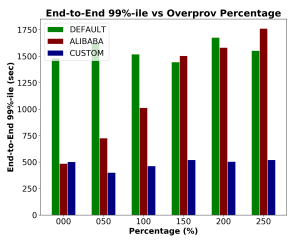
In particular, Kubernetes default GPU extension has the lower resource utilization because each inference engine allocates the whole GPU resource. Alibaba GPU share scheduler extension has similar results with our custom scheduler only for 0 % and 50 % over-provisioning percentages. This is expected, since for these over-provisioning percentages the scheduler can effectively colocate workloads. The higher the over-provisioning percentage is, the closer the resource utilization is to Kubernetes default GPU extension. Finally, we observe that our custom scheduler has similar behavior concerning the resource utilization for all the different over-provisioning percentages.

6.2 Heterogeneous Workload Evaluation

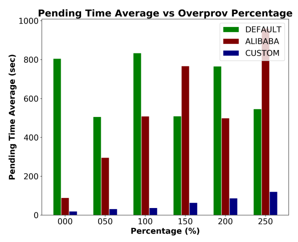
For heterogeneous workload, we consider different inference engines, where each one of them performs a different number of inference queries, as shown in Table 2. Figure 7 shows the quality of service metrics for the heterogeneous inference engine workload. Our proposed

Table 2 Inference engines used for heterogeneous workload evaluation.

Model	Dataset	Queries/Engine (#Engines)
mobilenet	Imagenet	1024 (2), 2048 (2)
mobilenet quantized	Imagenet	256 (2), 512 (2)
resnet50	Imagenet	4096 (2), 8192 (2)
sd-mobilenet	Coco (resized 300x300)	128 (3), 1024 (2)
ssd-mobilenet quantized finetuned	Coco (resized 300x300)	64 (2), 1024 (2)
ssd-mobilenet symmetrically quantized finetuned	Coco (resized 300x300)	512 (2), 4096 (2)



(a) End-to-End Job Execution 99%-ile.



(b) Pending Time Average.

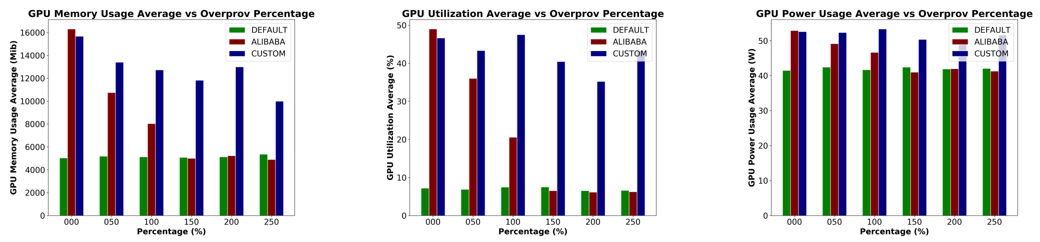
Figure 7 End-to-end job execution 99%-ile and pending time average vs over-provisioning percentage heterogeneous workload with MIN=5 and MAX=10.

scheduler offers x11 lower pending time average and x3.2 lower end-to-end 99%-ile and x8.6 lower pending time average and x2.4 lower end-to-end 99%-ile on average compared to the Kubernetes default and Alibaba's GPU schedulers respectively. Moreover, Figure 8 shows the respective GPU metrics. We see that, our scheduler leads to x2.5 higher memory usage, x6.1 higher GPU utilization and x1.2 higher power consumption compared to Kubernetes default GPU extension and x1.5 higher memory usage, x2.1 higher GPU utilization and x1.1 higher power consumption compared to Alibaba's GPU sharing scheduler extension.

Container Restarts Analysis. As it was mentioned in section 5, CBP involves the risk of incorrect scheduling decisions and thus inference engine failures. CBP's prediction accuracy depends on how representative is the free memory signal it receives as input. Since accelerated applications do not always request GPU resources at the beginning of their execution, it is possible that the used signal does not depict the real load of the node. Although several container restarts occured in the previous experiment, we observe that our proposed scheduler still offers better QoS and GPU resource utilization from the baseline state-of-the-art GPU schedulers.

7 Conclusion

In this paper, we design a resource aware GPU colocation framework for Kubernetes inference clusters. We evaluate the inference engine colocation algorithm using workloads that consist of inference engines using different scenarios. We identify and explain the disadvantages of the correlation based prediction (CBP) and peak prediction (PP) scheduling schemes. Finally, we show that our custom scheduling framework improves the defined quality of service metrics while also increases the GPU resource utilization.



(a) Memory Usage Average. **(b)** GPU Utilization Average. **(c)** Power Usage Average.

Figure 8 Memory usage average, GPU utilization average and power consumption averages vs over-provisioning percentage for heterogeneous workload with MIN=5 and MAX=10.

References

1 Alibaba GPU Sharing Scheduler Extension. URL: `https://www.alibabacloud.com/blog/594926`.

2 CUDA Streams. URL: `https://leimao.github.io/blog/CUDA-Stream/`.

3 GPU Memory Over-provisioning. URL: `https://www.logicalclocks.com/blog/optimizing-gpu-utilization-in-hops`.

4 Kubernetes GPU Scheduler Extension. URL: `https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/`.

5 NVIDIA Data Center GPU Manager. URL: `https://developer.nvidia.com/dcgm`.

6 Prometheus. URL: `https://prometheus.io/docs/introduction/overview/`.

7 B. S. Everitt A.Skrondal. *The Cambridge Dictionary of Statistics*. Cambridge University Press, 2554. URL: `http://library1.nida.ac.th/termpaper6/sd/2554/19755.pdf`.

8 Rolando Brondolin, Marco D Santambrogio, and Politecnico Milano. A Black-box Monitoring Approach to Measure Microservices Runtime Performance. *ACM Transactions on Architecture and Code Optimization*, 17(4), 2020.

9 Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, April 2016. `doi:10.1145/2890784`.

10 Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 02-06-Apri:681–696, 2016. `doi:10.1145/2872362.2872368`.

11 James Gleeson and Eyal de Lara. Heterogeneous GPU reallocation. *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, co-located with USENIX ATC 2017*, 2017.

12 Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *USENIXATC'11: Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011.

13 Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. *SIGARCH Comput. Archit. News*, 43(1):223–238, March 2015. `doi:10.1145/2786763.2694347`.

14 Howard J. Seltman. Experimental Design and Analysis. *Revista*, 20(2), 2016. `doi:10.35699/2316-770x.2013.2692`.

15 VMware Inc. Containers on virtual machines or bare metal ? *Deploying and Securely Managing Containerized Applications at Scale, White Paper*, December 2018.

16 John A. Gubner. *Probability and Random Processes for Electrical and Computer Engineers*. Cambridge University Press, 2554. URL: `http://library1.nida.ac.th/termpaper6/sd/2554/19755.pdf`.

17 D Kang, T J Jun, D Kim, J Kim, and D Kim. ConVGPU: GPU Management Middleware in Container Based Virtualized Environment. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 301–309, 2017. doi:10.1109/CLUSTER.2017.17.

18 Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. GrandSLAm. *EuroSys '19: Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019. doi:10.1145/3302424.3303958.

19 D Masouros, S Xydis, and D Soudris. Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):184–198, January 2021. doi:10.1109/TPDS.2020.3013948.

20 Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, 2018-Janua, 2018. doi:10.1145/3190508.3190517.

21 Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark, 2019. arXiv:1911.02549.

22 Multi-process Service. Multi-process service, 2020.

23 Robert Shumway and David Stoffer. *Time Series Analysis and Its Applications: With R Examples*. Springer, 2017. doi:10.1007/978-3-319-52452-8.

24 I Tanasic, I Gelado, J Cabezas, A Ramirez, N Navarro, and M Valero. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, 2014.

25 Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2019-Septe:1–13, 2019. doi:10.1109/CLUSTER.2019.8891040.

26 Achilleas Tzenetopoulos, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. Interference-Aware Orchestration in Kubernetes. In *International Conference on High Performance Computing*, pages 321–330. Springer, 2020.

27 Y Ukidave, X Li, and D Kaeli. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 353–362, 2016. doi:10.1109/IPDPS.2016.73.

28 Shaoqi Wang, Oscar J Gonzalez, Xiaobo Zhou, and Thomas Williams. An Efficient and Non-Intrusive GPU Scheduling Framework for Deep Learning Training Systems. *Sc*, 2020.

29 Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, pages 173–184, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3369583.3392679.