Job Generator for Evaluating and Comparing Scheduling Algorithms for Modern GPU Clouds

Michal Konopa[a], Jan Fesl[b] and Ladislav Beránek[c]
Department of Data Science and Computing Systems, Faculty of Agriculture and Technology, University of South Bohemia, České Budějovice, Czech Republic

Keywords: Generator, Job, Scheduling Algorithm, MIG, Cluster.

Abstract: The steep technological and performance advances in GPU cards have led to their increasing use in data centers in the recent years, especially in machine learning jobs. However, high hardware performance alone does not guarantee (sub) optimal utilization of computing resources, especially when the cost associated with power consumption also needs to be increasingly considered. As consequence of these realities, various job scheduling algorithms have been and are being developed to optimize the power consumption in data centers with respect to defined constraints. Unfortunately, there is still no known, widely used, parametrizable dataset that serves as a de facto standard for simulating scheduling algorithms and the resulting ability to compare their performance against each other. The goal of this paper is to describe a simple job set generator designed to run on modern GPU architectures and to introduce the newly created data set suitable for evaluation the scheduling algorithms.

1 INTRODUCTION

Today's data centers are overwhelmingly built on GPU clusters. In recent years, the performance of GPU cards has increased rapidly, but unfortunately the utilization rate of GPU clusters is often not very high - e.g. (Narayanan, 2020), (Li, 2023), (Hu, 2021), (Weng, 2022) report utilization rates of only between 25 and 50%. This leads to a significant waste of electrical energy, which is needed not only to power the GPU clusters themselves, but also to cool them efficiently. To address this problem, GPU sharing techniques have been developed that allow securely running (with guaranteed isolation) multiple computational jobs on a single GPU, where each job is allocated partial resources, either through virtualization (Gu, 2018), (Shi, 2012), (Han, 2022), (Yeh, 2020), (Xiao, 2018) or through the Multi-Instance-GPU (MIG) feature supported by new Nvidia GPU architectures.

MIG technology, developed by NVidia for GPU with architectures Ampere, Hopper and Blackwell, allows a single physical GPU to be partitioned into multiple virtual GPU instances. Each virtual GPU instance is allocated a portion of the computational resources of the original GPU, specifically memory, cache, and compute cores. The computational resources of each instance are fully isolated from each other, allowing multiple jobs to run in parallel, fully isolated on each instance. Compared to the traditional allocation of the entire GPU to one specific job, MIG allows to reduce the waste of GPU computational resources while increasing the number of completed jobs per unit time. However, the allocation of GPU computational resources between instances cannot be done completely arbitrarily, but only by selecting from a fixed set of variants, specific to one concrete model of GPU. GPU can also be dynamically repartitioned (or reconfigured) according to expected computing resource requirements of incoming jobs.

However, sharing GPUs across multiple jobs does not in itself guarantee high GPU utilization, but often is the source of another problem - high fragmentation, which does not allow to allocate enough GPU computing resources to the input job when the aggregate capacity of the cluster is otherwise high

[a] https://orcid.org/0000-0003-1694-1529
[b] https://orcid.org/0000-0001-7192-4460
[c] https://orcid.org/0000-0001-5004-0164

Konopa, M., Fesl, J. and Beránek, L.
Job Generator for Evaluating and Comparing Scheduling Algorithms for Modern GPU Clouds.
DOI: 10.5220/0013227500003950
In *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025)*, pages 136-143
ISBN: 978-989-758-747-4; ISSN: 2184-5042

enough. In practice, it is quite common that only 85-90% of the maximum GPU capacity of a cluster can be allocated. A natural solution to the fragmentation problem is to proceed with the allocation of computational resources in such a way that as few GPU and nodes as possible are continuously reserved. This is an inherently complex combinatorial problem, compounded in practice by the fact that today's high-end data center (Google, Alibaba) typically contains thousands of GPU nodes. For this reason, various heuristics have been applied to this problem – e.g. (Weng, 2023), or heuristic and mixed integer programming solvers (Turkkan, 2024).

1.1 Job Scheduling in GPU Clusters

A GPU cluster is essentially a set of servers (nodes) where each server contains one or more GPUs. In the GPU Cluster, all servers are interconnected via an ultrafast computer network. The individual GPUs should all be of the same manufacturer and model - then we are talking about a homogeneous GPU cluster, otherwise it is a heterogeneous GPU cluster. Depending on the size of the GPU cluster, tens to thousands of nodes can be connected to the GPU cluster. An important part of a GPU cluster is the input job queue, where jobs scheduled to run are placed. The scheduler sequentially selects individual jobs from the queue and allocates free GPU instances to them. The selection of individual jobs from the queue can also be controlled by their priority, if the job has a priority assigned. The selection of a target GPU instance for a particular job may be conditioned on information about the computational resource requirements, e.g., maximum memory consumption, if this information is available to the scheduler.

The scheduler plans the running of individual jobs according to defined optimization criteria. One of the most common criteria is minimizing the computational resources used, which is typically minimizing the number of concurrently running GPUs or minimizing the waste of computational GPU resources. Subject to defined constraints. Given the ever-increasing emphasis on "green energy", maximising the use of "green energy" can also be an optimisation criterion.

1.2 Problem Statement

To test the performance of their scheduling algorithms and heuristics, their authors most often create their own (micro)benchmarks. These can be based on operational data from data centers to which the authors have access, e.g., (Weng, 2023). If it is an ML job, authors sometimes specify the ML model used (Cui, 2021) along with the publicly available dataset that this model processed during testing, e.g. (Xio,2018), (Choi, 2021). Some authors, e.g., (Hu, 2021) and (Li, 2022) use publicly available datasets containing specific job types - here deep learning, from a production environment. Other authors, e.g., (Turkkan, 2024) test on randomly generated jobs.

Despite our best efforts, we have not been able to find any dataset that is a de facto standard for comparing the performance of MIG-enabled GPU scheduling algorithms. One possible reason for this could be the heterogeneity of different types of jobs designed for GPU processing, where different types of jobs exhibit different parameters such as memory consumption as a function of job runtime. For this reason, it makes sense to have different datasets for evaluating and comparing algorithms, especially for scheduling specific types of jobs.

There are certainly countless ways in which individual jobs can be categorised. Rather than attempting to design some single, standardised system of job categories, which would be a very difficult job to accomplish in practice, we have proposed a system that allows us to generate a set of jobs whose properties are specified through pre-selected parameters. We called this system a job generator.

The very idea of a system for generating jobs to evaluate and compare scheduling algorithms is not new. (De Bock, 2018) proposed a generator for RT multicore systems that can also generate for each job executable code with a declared worst-case runtime.

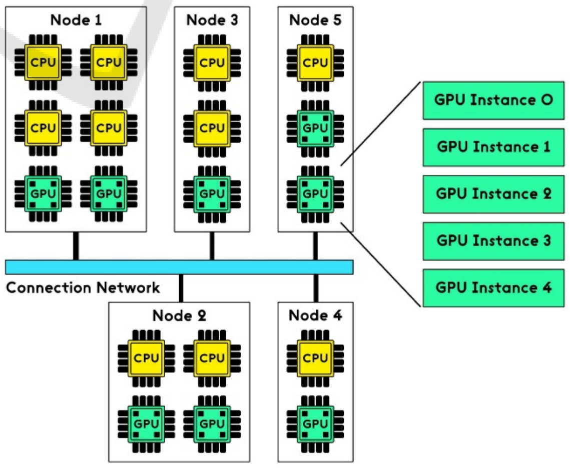

Figure 1: GPU-cluster scheme.

1.3 Generator Configuration Requirements and Runtime Prepositions

The job generator was designed to enable the evaluation and inter-comparison of the performance of job scheduling algorithms on MIG-enabled GPUs. For the generator to work successfully in practice, the following requirements need to be met:

1. Support for MIG technology including dynamic reconfiguration.

   It makes sense to perform dynamic reconfiguration if there is a high chance that performing the reconfiguration will result in more efficient use of computing resources in the future. To predict such a state more easily, it is necessary to have at least a rough idea of the future usage of specific types of computing resources (e.g. RAM) by a specific job. The generator allows the user to specify the expected use of computational resources by jobs during their execution.

2. Reproducibility of generated job sets.

   This requirement simply means that the same set of jobs will always be generated for the same values of all generator input parameters.

3. Support for specifying job arrival times to the GPU-cluster input queue.

   The ability to specify the arrival times of jobs in the input queue can be used for longer-term scheduling, including identifying rush-hours and idle hours. The generator allows to specify arrival times either via a fixed value or a random value generated from a predefined random distribution.

2 JOB GENERATOR IMPLEMENTATION

The Job generator is implemented in Java in the current version. The source code is publicly available through the repository https://github.com/mikon81/job-generator, where you can also find examples of input configuration files and their corresponding generated job sets. An example of one such file pair is also included in the appendix of this paper. Larger datasets generated by the generator can be freely downloaded here: https://github.com/mikon81/job-generator/tree/main/datasets

The generator produces a set of jobs whose properties are specified in the input configuration file. The generator works in steps as follows:

1. The generator settings are read and loaded from the configuration file.
2. Based on the loaded generator settings and the internally used random number generator, a set of jobs is generated.
3. The generated job set is written to the output file.

The configuration and output files are both in the widely used, programming language and platform-independent JSON format.

2.1 Configuration of the Generator

The job generator setting is performed via so-called *configuration parameters* (CP). They are divided into 2 types:

1. For setting the properties of individual jobs – job configuration parameters.
2. For setting the properties of the whole set of jobs – job set configuration parameters.

Each CP is assigned its own so-called *configuration type* (CT). Each CP is generally assigned a different CT, but each CT defines a set of options for how a particular property can be set.

The two main options for setting a specific property are naturally:

1. A fixed, specific value.
2. Random value.

The possibility of setting a random value is further specified within the CT by the choice of the probability distribution including the choice of values of the distribution parameters. The current implementation of the generator supports the following probability distributions:

1. Uniform, with lower and upper bound interval parameters.
2. Normal, with defined mean and standard deviation.

2.2 Job Configuration Parameters

The properties which can be set for each generated job are listed in the Table 1.

Table 1: Configuration types of job properties.

Property name	Configuration type	Probability distribution
Priority	Fixed value, Random value	Uniform, Normal
Maximum RAM consumption [in MB]	Fixed value, Independent random value, Random value dependent on previous minute's value	Uniform, Normal
Number of used CUDA cores	Fixed value, Random value	Uniform, Normal
Run length [in time units]	Fixed value, Random value	Uniform, Normal
Indication if job is interruptible	Fixed value, Random value	Uniform

To meet the requirement of MIG technology support and dynamic reconfiguration support, each generated job includes a sequence of natural numbers, where each number in the sequence specifies the maximum memory consumption in a particular time unit of a job's run. The *Maximum memory consumptio*n property is then applied to each element in this sequence. The length of the sequence equals the value of the *Run length* property.

To set the *Maximum memory consumption* property, the random value setting option is divided into the following categories:

1. The value is generated for each specific time unit of the job running completely independently of the values of other time units.
2. The value for each specific time unit, except the very first one, is generated based on the value from the previous time unit. If a random distribution is specified as normal, the mean of the distribution for generating the value in the current time unit is set to the value from the previous time unit, while the standard deviation remains the same for all time units. If a random distribution is specified as uniform, the value from the previous time unit is used as the midpoint of the interval, while its length is determined by the difference of the upper and lower bounds given in the uniform distribution specification for the property.

Category 2 is closer to the reality of the behaviour of most jobs, where the memory consumption of a job run somehow depends on the job run so far and its program code. The possibility that memory consumption would randomly "oscillate" is usually quite rare (unless the job's program code is written with that intention).

In case of setting the property *Indication if job is interruptible* by selecting a random value option, the probability of the job being interruptible must be specified.

2.3 Job Set Configuration Parameters

The properties which can be set for whole set of jobs are listed in the Table 2.

Table 2: Configuration types of job set properties.

Property name	Configuration type	Probability distribution
Number of generated jobs	Fixed value	
Number of time units between job arrivals in the input queue	Fixed value, Random value	Uniform, Poisson
Seed of internal random generator	Fixed value	

The *Number of time units between job arrivals in the input queue* property can be set in two different ways: by a fixed value or by a random value generated from a defined random probability distribution.

The preferred value for the *Seed of internal random numbe*r property is prime number. For the same seed value and otherwise the same all other configuration settings, the output job file is always the same. For a different seed value and all other configuration settings being otherwise the same, the output files are generally different.

3 NEW DATASET

As an output of the new generator, a new dataset was created. This dataset is publicly available here: https://github.com/mikon81/job-generator. Generator configuration parameters that were used to create the dataset are listed in the Table 3.

Table 3: Configuration parameters used for generating test dataset.

Property name	Value
Priority	Uniform(lowerBound=1, upperBound=5)
Maximum memory consumption [in MB]	Random dependent on previous, Normal(mean=200, sd=50)
Run length [in time units]	Normal(mean=5, sd=2)
Indication if job is interruptible	Fixed value = true
Number of used CUDA cores	Fixed value = 100
Number of generated jobs	1000
Number of time units between job arrivals in to the input queue	Poisson(lambda=10)
Seed of internal random generator	41

Some selected characteristics of the generated dataset are shown in the Figure 2 and in the Figure 3.

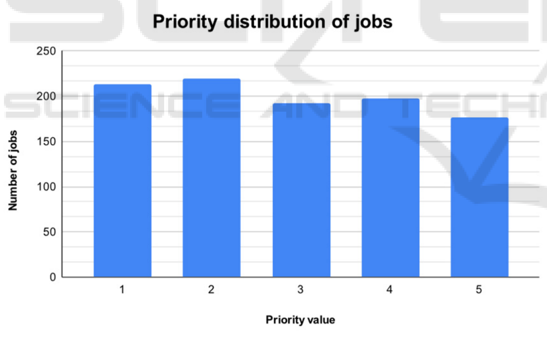

Figure 2: Distribution of priority property values.

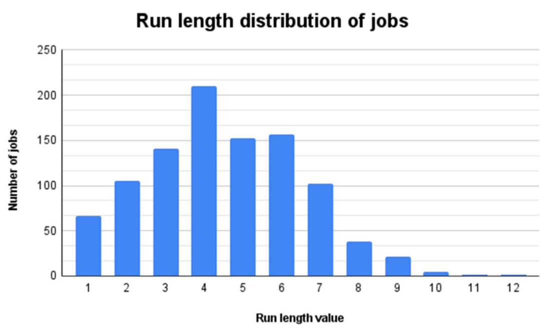

Figure 3: Distribution of run length property values.

4 EXPERIMENTAL RESULTS

The aim of the experiment was to use the generator to generate a set of datasets that would then be used in the process of experimentally comparing the performance of two contrasting approaches to scheduling jobs on a single MIG-enabled GPU, both in terms of the quality of the resulting solution and the amount of time required to find a particular solution. The first approach was to directly search for the optimum via the chosen SAT-solver (sat4j), while the second approach was to search for the suboptimal solution using the fast Best-Fit heuristic.

4.1 Requirements for Job Configuration Parameters

The performance testing of both scheduling algorithms is to be carried out by simulating these algorithms in the Java programming language. The algorithms will simulate the scheduling of an input sequence of jobs of defined properties on a fixed GPU configuration, divided into 3 GPU instances with the following RAM sizes: 2000 MB, 1000 MB, 1000 MB.

The following requirements were defined for the properties of the input tasks:

1. The maximum job length in minutes is generated randomly from a uniform distribution on the interval ⟨1,10⟩.
2. For each job, the maximum RAM consumption during its run must not exceed 1990 MB.
3. The maximum RAM consumption of a job *j* within minute *t* of its run, denoted *PeakRAM(j,t))*, is generated randomly from a normal distribution with mean equal to *PeakRAM(j,t-1))*, and standard deviation equal to 0.1 x *PeakRAM(j,t-1))*. The maximum for the first minute is randomly generated froma uniform distribution on the interval ⟨1,1990⟩ MB.
4. The completion deadline for all jobs is 250 minutes from the start of the experiment.
5. All jobs are interruptible.
6. All jobs have the same priority.
7. The total number of jobs to be scheduled in a single run of the algorithm: 50.
8. The arrival of each job to the queue is independent of the arrival of other jobs to the queue.
9. The average number of queued jobs arriving per minute: 20.

The Table 4 of job configuration parameters and job sets configuration parameters correspond to the above requirements for input job properties:

Table 4: Configuration parameters used for the experiment.

Property name	Value
Priority	the same value for each job
Maximum memory consumption [in MB]	Maximum = 1990 MB First minute = Uniform(1, 1990) Random dependent on previous, Normal(mean=consumption from the previous minute, sd=0.1x consumption from the previous minute)
Run length [in minutes]	Uniform(1, 10)
Deadline [in minutes]	250
Indication if job is interruptible	Fixed value = true
Number of used CUDA cores	any value (not used in the experiment)
Number of generated jobs	50
The average number of job arrivals per minute	Poisson(lambda=20)
Seed of internal random generator	unique prime number for each dataset to be generated

An experiment with the same input job configuration parameters is run for each algorithm under test a total of 100 times, but each time with a different prime constant.

4.2 Implementation of the Experiment

The experiment was implemented through a Producer-Consumer design patter. The producer stored the jobs in a shared queue. The arrival of each job to the queue was independent of the arrival of other jobs to the queue; the total number of job arrivals to the queue was directly dependent on the length of the time interval being monitored. Each algorithm under test (consumer), tested the queue for the presence of jobs at fixed time intervals, then removed all the jobs from the queue and moved them to its local memory to update the schedule. The experiment ended with the inclusion of the last job from the queue in the schedule.

The implementation of the experiment required parsing and loading the generated test datasets. For this reason, a simple parser for the Java programming language was implemented to read the input dataset in JSON format and return the corresponding job queue. The producer was then given a reference to this job queue during its initialization and then moved individual jobs from this queue to the shared queue according to the arrival time specified in each job instance.

4.3 Discussion

In the course of experimenting with the task generator, it can be said that in cases where the generator contains functionality necessary for specifying user-selected properties of the generated dataset, its use is simple and fast. The generated dataset file has a simple structure and is easy to read.

However, in more complex cases of dataset property specification and looking to the future, the following issues need to be considered:

The first issue is the lack of support for bulk generation of datasets that have the same configuration parameter values, except for the seed of the internal random number generator. It is not possible to specify directly in the configuration file that e.g. 100 different datasets are to be generated in this way. However, the implementation of this functionality is not complex and will be added in the future.

Another issue is that the generator currently does not allow the user to specify and use probability distributions other than a few basic ones. Modeling the internal behavior of data centers and clouds in terms of the frequency of arrival of individual jobs and their demands on computing resources may require the use of a much wider variety of different probability distributions, including the ability to specify these distributions using, e.g., histograms. Next problem is the tendency for the frequency of job arrivals to change during the day (rush hour vs. idle hour), which will be reflected at least in the change of the parameters of the probabilistic distribution of job arrival times.

The third, and more general issue, which is more closely related to the previous one, is the support of (some) user-defined functionality without the need to change and rebuild the generator source code. This can be implemented on 2 levels: the dataset configuration file and/or the compiled user code. The question is how far to go in supporting these functionalities - given the ease of use in practice, good maintainability and reasonable complexity of the whole generator. What level of support for user-defined functionalities can JSON handle? Wouldn't it be more appropriate to add support and/or switch to a different format for specifying job properties over time? Support for user functionality at the compiled code level, typically in the form of user objects implementing a defined programming interface, is not currently implemented in the generator, nor was this form of support anticipated when the generator was designed. Implementation of this support would require major changes to the basic structure of the generator and would in principle bring disadvantages, such as the need to program user functionality directly in the form of Java code and the dependence of user code on programming interfaces defined by the generator. The question of possible programming support will therefore be the subject of future in-depth analyses.

5 CONCLUSIONS

This paper presents a new dataset generator designed to evaluate and compare the performance of job scheduling algorithms on modern GPU-clusters with MIG technology support. The properties of the generated datasets, including their sizes, can be easily set via the generator's configuration parameters. The generated datasets are reproducible and publicly available as well as the generator source code.

At the present time, the generator supports only a few basic probability distributions. In the future, it is considered to extend the generator's support by specifying more complex cases of the properties of the generated jobs, including the possibility of defining histograms, adding user-defined configuration items and functionalities, the possibility of specifying the dynamics of the frequency of incoming jobs during the monitored period. An open problem is how to implement such support given the user-friendliness, good maintainability and reasonable complexity of the generator.

ACKNOWLEDGEMENTS

This article was written with the financial support of the Grant Agency of the University of South Bohemia.

REFERENCES

Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., & Zaharia, M. (2020). Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20),* 481–498. https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

Li, J., Xu, H., Zhu, Y., Liu, Z., Guo, C., & Wang, C. (2023). Lyra: Elastic Scheduling for Deep Learning Clusters. *Proceedings of the Eighteenth European Conference on Computer Systems*, 835–850. https://doi.org/10.1145/3552326.3587445

Gu, J., Song, S., Li, Y., & Luo, H. (2018). 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),* 469–476. https://doi.org/10.1109/BDCloud.2018.00077

Shi, L., Chen, H., Sun, J., & Li, K. (2012). vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 61(6), 804–816. https://doi.org/10.1109/TC.2011.112

Han, M., Zhang, H., Chen, R., & Chen, H. (2022). Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22),* 539–558. https://www.usenix.org/conference/osdi22/presentation/han7

Nvidia multi-instance GPU, seven independent instances in a single GPU. (2023). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/

Hu, Q., Sun, P., Yan, S., Wen, Y., & Zhang, T. (2021). Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* https://doi.org/10.1145/3458817.3476223

Weng, Q., Yang, L., Yu, Y., Wang, W., Tang, X., Yang, G., & Zhang, L. (2023). Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. *2023 USENIX Annual Technical Conference (USENIX ATC 23),* 995–1008. https://www.usenix.org/conference/atc23/presentation/weng

Weng, Q., Xiao, W., Yu, Y., Wang, W., Wang, C., He, J., Li, Y., Zhang, L., Lin, W., & Ding, Y. (2022). MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22),* 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

Yeh, T.-A., Chen, H.-H., & Chou, J. (2020). KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. *Proceedings of the 29th International Symposium on High-Performance*

Parallel and Distributed Computing, 173–184. https://doi.org/10.1145/3369583.3392679

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., & Zhou, L. (2018). Gandiva: Introspective Cluster Scheduling for Deep Learning. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18),* 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

Turkkan, B., Murali, P., Harsha, P., Arora, R., Vanloo, G., & Narayanaswami, C. (2024). Optimal Workload Placement on Multi-Instance GPUs. https://arxiv.org/abs/2409.06646

Choi, S., Lee, S., Kim, Y., Park, J., Kwon, Y., & Huh, J. (2021). Multi-model Machine Learning Inference Serving with GPU Spatial Partitioning. https://arxiv.org/abs/2109.01611

Li, B., Patel, T., Samsi, S., Gadepally, V., & Tiwari, D. (2022). MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. Proceedings of the 13th Symposium on Cloud Computing, 173–189. https://doi.org/10.1145/3542929.3563510

Cui, W., Zhao, H., Chen, Q., Zheng, N., Leng, J., Zhao, J., Song, Z., Ma, T., Yang, Y., Li, C., & Guo, M. (2021). Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction (p. 15). https://doi.org/10.1145/3458817.3476143

De Bock, Y., Altmeyer, S., Huybrechts, T., Broeckhove, J., & Hellinckx, P. (2018). Task-set generator for schedulability analysis using the TACLebench benchmark suite. SIGBED Rev., 15(1), 22–28. https://doi.org/10.1145/3199610.3199613