Analysis of MLaaS GPU cluster trace: Final Report

Hantang Li, Haoteng Wang

Abstract

GPU computing is becoming more than necessary with the rise of machine learning products. To support the massive demand, companies like Alibaba are deploying ML-as-a-Service clouds. However, many problems arise in the deployment, including the fairness of GPU scheduling and GPU utilization. The paper [1](MLaaS in the Wild) provides a two-month workload trace collected from a production MLaaS cluster with over 6,000 GPUs in Alibaba. The authors explained theoretically how scheduling decisions are made for each task but did not provide a detailed practical analysis nor consider the fairness of their scheduling algorithm. Further, the MLaaS cluster used GPU sharing but only showed its benefit and analyzed its contention. So, there is more to explore.

In this project, we will apply data analysis, data visualization and machine learning techniques to understand the GPU cluster from a data perspective. We will evaluate the GPU sharing

infrastructure's performance and evaluate the system's fairness, and provide possible suggestions to improve the system. First, we will use various diagrams to interpret how some scheduling decisions are made in detail, such as tasks that use GPU sharing. We created interactive diagrams for users like the system manager to explore the system's data in an intuitive way. Second, we will evaluate how different task performs in the GPU cluster and how GPU sharing performs in terms of failure rate, wait time and run time. Third, we will evaluate the system's fairness at a specific time range, and we will provide an interactive plot to show how fairness changes in different settings. Further, we have attempted to modify the paper's simulator[1] to support FIFO and analyze the fairness but failed. We have explained the difficulty we faced while modifying the simulator.

Explore GPU sharing through the system's data

For Interactive plots, please see attached notebook "CSC2233_Interactive.ipynb"

GPU sharing has many variations based on how it is implemented, such as the hardware level implementation of NVIDIA's Multi-Instance GPU (MIG) [6] supports NVIDIA H100, A100, and A30 GPUs. MIG can partition the GPU into as many as seven instances, each fully isolated with its own high-bandwidth memory, cache, and compute cores. Another way of implementing it is on a software level, such as Salus[7], modifying the machine learning platform and acting as a virtual machine by efficient job switching and dynamic memory sharing, which could also achieve GPU sharing. The cluster we will explore implements AntMan[4], similar to Salus, using scheduling, job switching and memory paging while accommodating the fluctuating resource demands of deep learning training jobs by dynamic scaling. The area of sharing GPU by scheduling on the software level is quite new; with Salus

being published in 2019, there is no detailed data analysis report on how a software-level GPU sharing architecture performed in a GPU cluster. So, we will use the trace collected from the PAI GPU cluster to explore the relationship between GPU sharing and all the other variables.

The data[1] provides information about the GPU, CPU and GPU memory resource the task used, the status of the task, and wait time, duration, start and end time. We analyzed how different requested resources are being assigned to different GPU types, what level of resource do different tasks used, and how the wait time is distributed for different tasks. We created interactive plots that are attached to the document to further explore the data.

1. **Explore GPU type and GPU sharing**

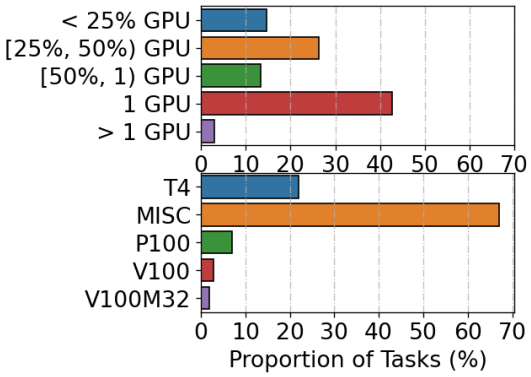


Figure 1: **Proportion of tasks[1]**

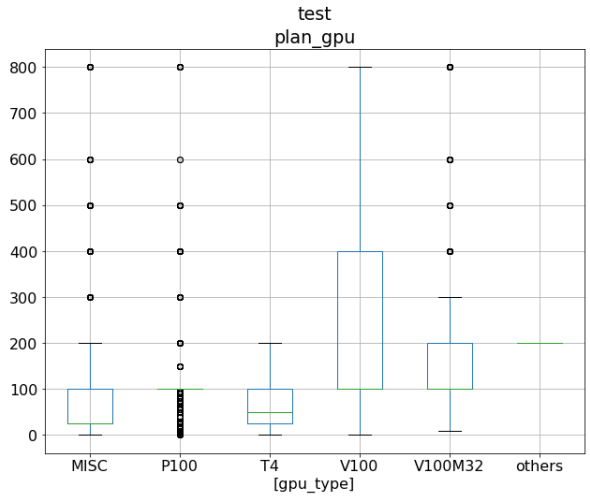


Figure 2: GPU usage, grouped by type

The integration between hardware and software could be challenging, and the software-level GPU sharing could perform differently on different GPU hardware architectures. The GPU cluster has a diversity of GPU types, and we want to explore whether GPU type and GPU sharing are related.

From figure 1, around 50% of tasks are using less than one GPU, and around 90% of tasks are running on mid to low-level GPUs in the category MISC or T4. From figure 2, most tasks running under MISC and T4 use less than one GPU, and most tasks running using P100, V100, and V100M32 use one or more GPUs. This could be caused by the simple reserving-and-packing policy that is being used in the server, in which high-end GPUs that support NVlink are reserved for tasks that require more GPU.

Further, we observed most tasks that used less than one GPU (GPU sharing) are on MISC or T4. Since MISC is short for "miscellaneous, in cases where analytics want to control variables by GPU type, we could use jobs that selected T4, as 20% of tasks are using T4 which provides sufficient data.

2. How do we define GPU sharing?

Since Antman[4] is the GPU-sharing infrastructure using scheduling, job switching and memory management to achieve GPU sharing, we do not have the data that Antman provides showing which job is being switched frequently and sharing GPU with other tasks.

We decided to categorize tasks that requested non-whole GPU, such as 50% or 150%, as tasks that involved a GPU sharing process. For this trace, there are 30633 tasks that do not share GPU with other tasks and 21094 tasks that share GPU with other tasks.

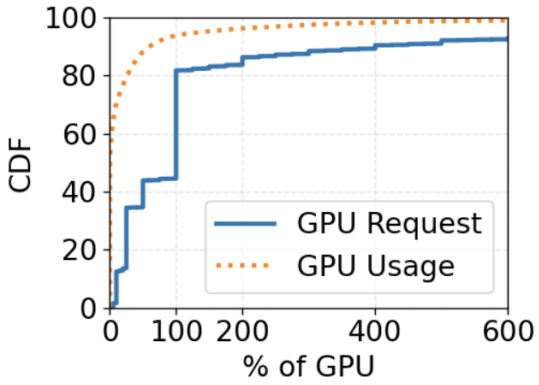


Figure 3: CDF of the GPU requested and GPU usage [1]

This definition is not perfect, as a task that requests a fragment of GPU can still hold a single GPU with no other tasks being allocated, and not all tasks being labelled as "Using Shared GPU" is sharing GPU with other tasks. And we are not sure how the cluster records the real GPU usage if the system records the burst GPU usage, then the task may still share GPU when no burst happens as the Antman selects GPUs with the utilization of less than M (set as 80% for now) in the past 10 seconds as candidates. But based on the paper "MLaaS in the Wild" [1], most users request more resources than they need, so we are confident that tasks that plan to use non-whole GPU will use less than what they planned, and the infrastructure is likely to select a GPU to share.

3. How tasks using shared GPU are distributed by GPU type

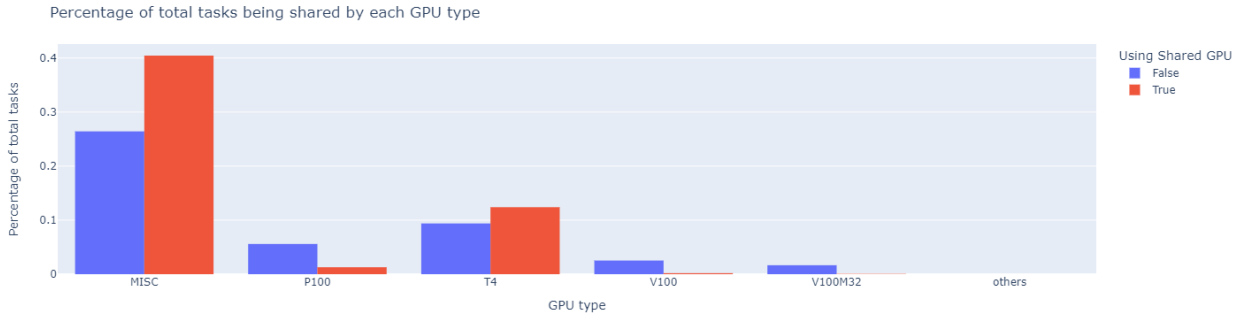


Figure 4: Percentage of tasks using share/non-share GPU are assigned to each GPU type

So, based on the definition, in figure 4, we categorized each task by whether they are estimated to share GPU with other tasks or not and counted for each GPU type. We found out that estimated that more than 50% of tasks are possibly sharing GPU with other tasks, and low-end GPU MISC is used mostly on tasks requested to share GPU, while other high-end GPUs, except T4, are being used less. T4 is an exception, as it is being used on tasks associated with different categories. While tasks using other high-end GPUs are less likely to be shared. So, in summary, most tasks use lower-end GPUs and occupy less than one GPU and possibly share GPU with other tasks, while some high-GPU tasks use high-end GPUs such as V100 and are not sharing GPU with others.

I have plotted a 3D interactive scatter plot showing what resources each task use. The plot and other interactive plots are attached to the “CSC2233_Interactive.ipynb.”

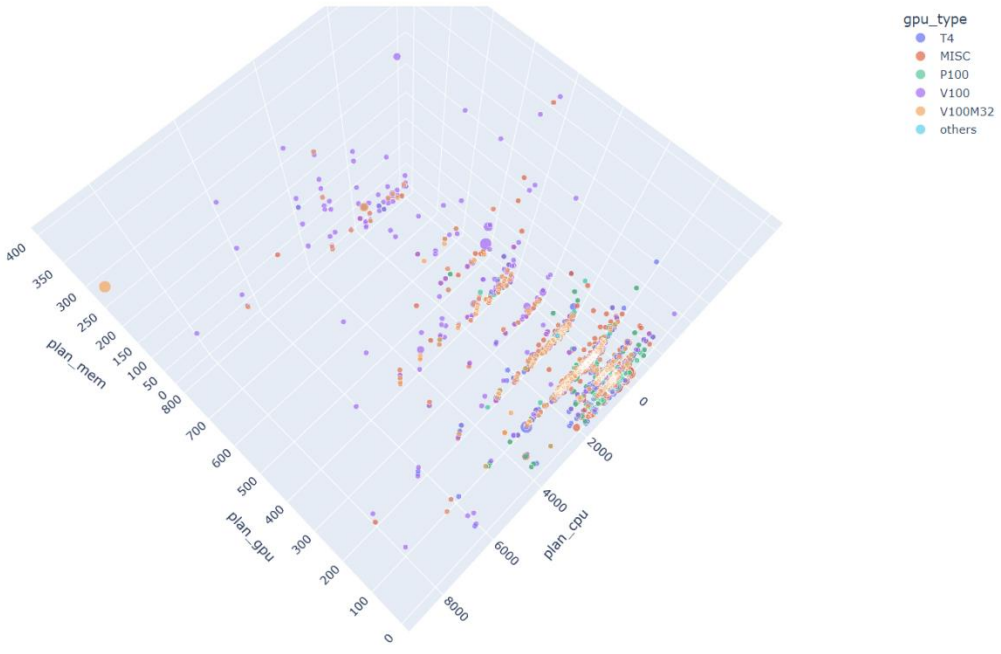


Figure 5: 3D interactive scatter plot for resources and GPU type

Based on the plot, it is clear that most tasks have low GPU, low memory and low CPU request. You could explore further by using the interactive plot for specification on each GPU type.

4. How tasks using shared GPU are distributed by task name

Based on the paper, we know that high GPU ML tasks tend to use more than one high-end GPU, and low GPU ML tasks tend to use shared GPU. And the GPU-sharing infrastructure Antman [4] only supports TensorFlow and PyTorch tasks. But we can only obtain information about the task name, and whether high or low GPU ML tasks are running on TensorFlow or PyTorch, so we need a detailed analysis to understand how the cluster decides which GPU is being used on which tasks and how each GPU are being shared.

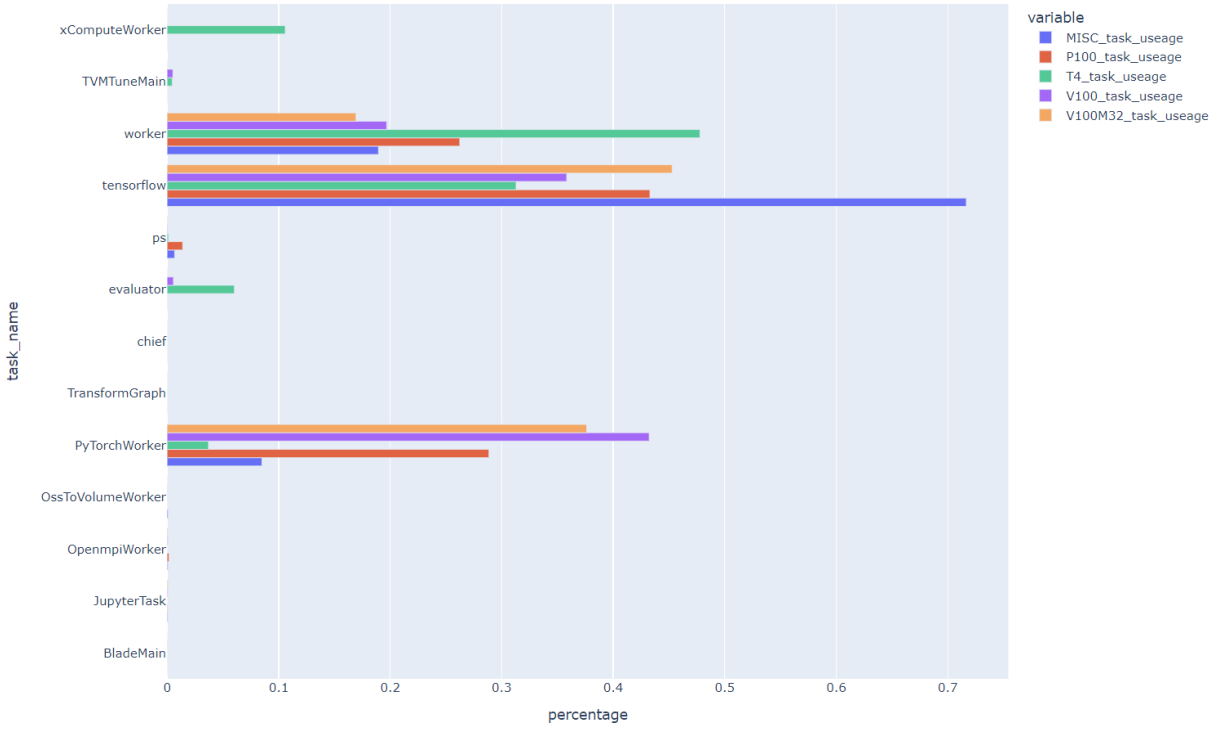


Figure 6: Percentage of tasks for each task name, grouped by GPU

First, we could inspect how each type of GPU is being used and analyze the workload by comparing the number of each task on each GPU.

From the plots, we can see that most low-end GPUs are assigned to be TensorFlow workers, such as 71% of MISC's tasks are being TensorFlow tasks, mid-range GPUs, such as T4, are being shared among multiple tasks, such as worker, TensorFlow and xComputeWorker. While highend-GPUs are mostly being used as a tensorflow worker or PyTorchWorker. One interesting point is that 78% of P100 tasks are used 100 percent of GPU exactly, and it could be caused by nodes that support P100 only having two GPUs and supporting NVLink, so those servers may be reserved for high-GPU tasks and less being shared.

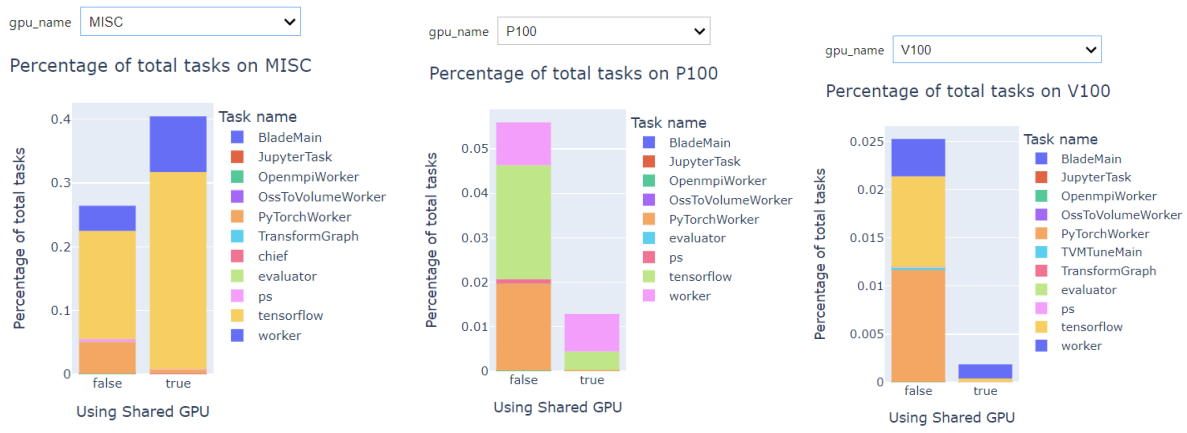


Figure 7: Percentage of tasks by MISC, P100, V100

We created an interactive plot to show how task name and GPU type are correlated. We found that low-end GPUs tend to have more TensorFlow tasks and are likely to share GPU, while high-end GPUs tend to have balanced TensorFlow and PyTorch tasks and are less likely to share GPU.

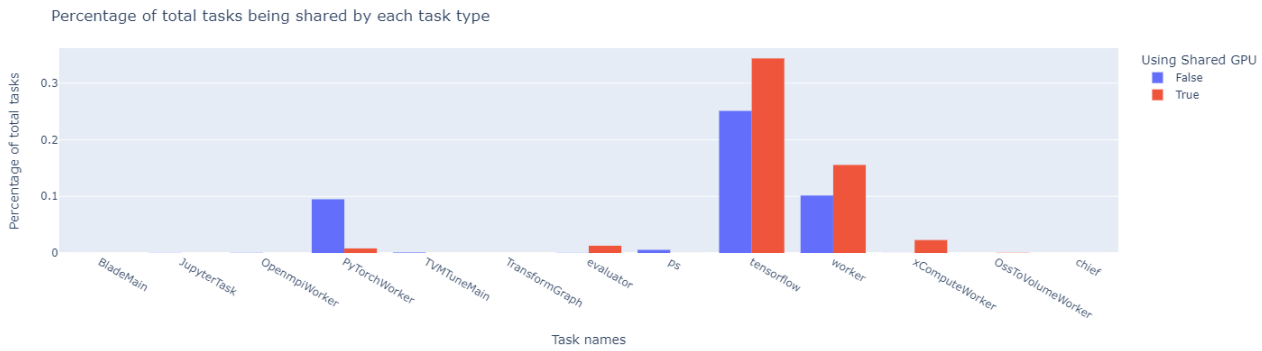


Figure 8: Percentage of total tasks being shared by each task type

Based on figure 8. We can see that more TensorFlow tasks and worker's tasks are using shared GPU while only around 10% of the PyTorch tasks are using shared GPU. In summary, TensorFlow tasks and their associated tasks are likely to use shared GPU. Around 47 percent of total tasks are running on low-end GPU MISC, in which around 70% of the total tasks are TensorFlow tasks, interestingly, among the 47 percent, 30 percent are using shared GPU, and 17 percent are not using shared GPU. But for high-end GPUs, most TensorFlow tasks are not using shared GPU, and the Reserving-and-packing strategy could partially explain this they used in which the cluster reserves high end-GPU for high-GPU tasks. While PyTorch tasks are less likely to use shared GPU and mostly run on high-end GPUs.

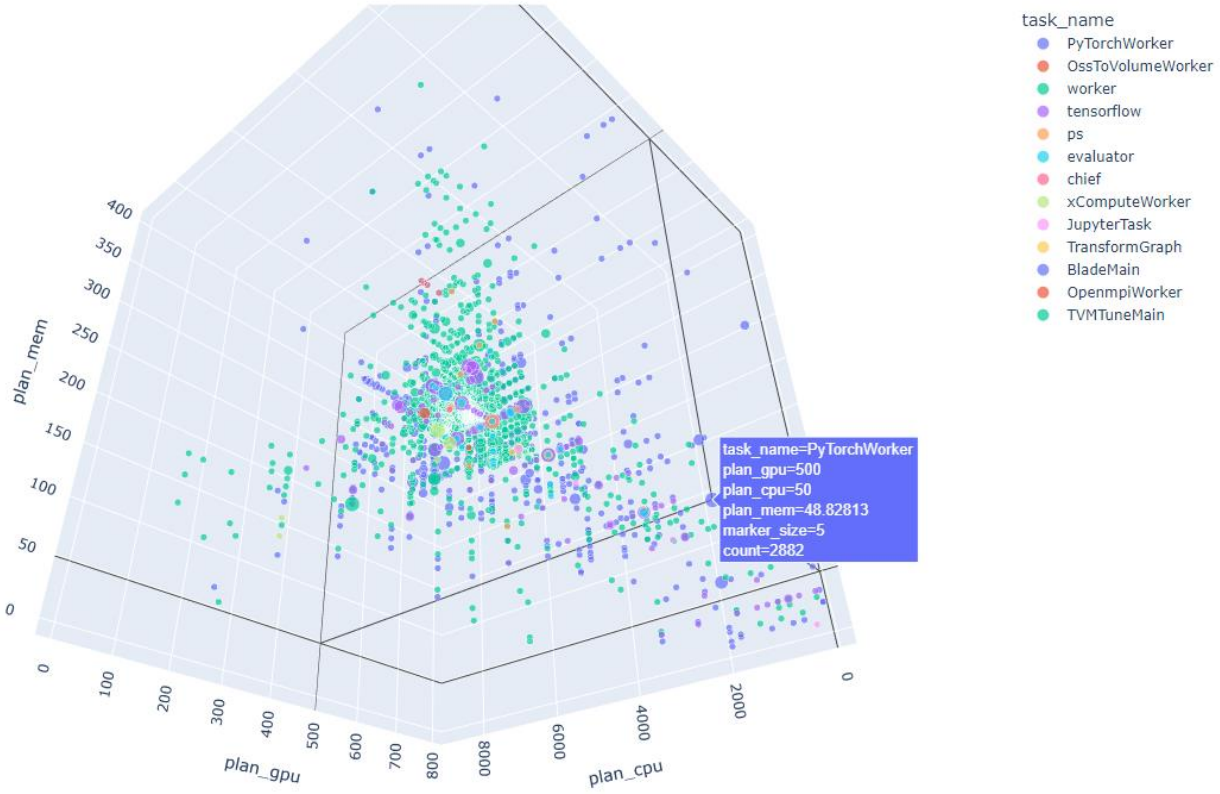


Figure 9: 3D interactive scatter plot for resources and task name

I also created an interactive plot showing how different tasks consume resources. By only selecting TensorFlow from the right list, you could observe that all TensorFlow tasks consume low memory, with maximum memory being 29.3 and mostly distributed on low-CPU and low-GPU side. In contrast, PyTorch tasks have a less skewed distribution in terms of plan GPU and may request more memory and CPU. As TensorFlow also can train large models, the system manager may want to inspect why TensorFlow is less used on GPU-heavy tasks.

Now we have some understanding of how the GPU cluster works, it has low-end to high-end GPUs that are applied to different tasks, low-end GPUs are mostly shared, and most TensorFlow tasks use shared GPU. We will have some further analysis regarding the GPU's scheduling system's performance, including the wait time, running time, and the actual GPU usage for those shared tasks.

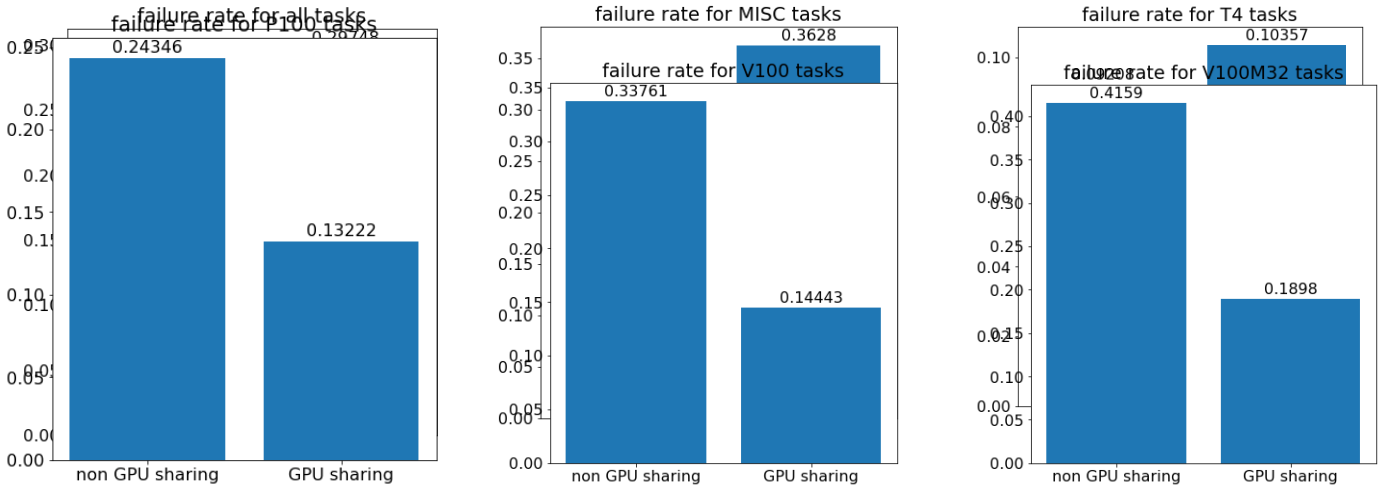
5. Analyze the success rate for tasks using shared GPU.

The paper mentioned that, unlike CPUs, GPUs do not natively support sharing between different jobs at the same time, and it does not mention any side effects of GPU sharing. In this section, we explore whether GPU sharing can cause the failure rate of jobs to rise.

One of the main challenges of sharing GPUs is that many popular machine learning frameworks, like TensorFlow, allocate the entire GPU memory upfront by default [https://www.run.ai/guides/multi-gpu/simplify-gpu-sharing-part-1]. So, code modifications are often required. However, this is not always possible, as the job can run inside a container. Moreover, memory fragmentation can also cause the system to be inefficient without proper arrangement.

Figure 10: Failure rate per GPU

We continue to categorize tasks into two separate groups: one uses GPU sharing and another one that doesn't, in which tasks with an integer number of requested GPUs are considered tasks with no GPU sharing involved and vice versa. From the diagram above, if we look at all jobs, we can see that tasks that do not share GPUs have a lower failure rate than those



that do. For tasks that do not share, the failure rate is at around 21.6%. The rate rises to about 36.2% for tasks that do share. It seems that sharing a GPU will cause the task more likely to fail and discourage the PAI from continuing to allow GPU sharing. However, if we re-visit the data in respect of different types of GPUs, we can see that the older GPUs (i.e. MISC) are the driving force behind this phenomenon. When only looking at the MSIC GPUs, the difference in failure rate grows to around 15%. However, if we are only looking at the newer generations of GPUs(i.e. T4, P100, V100, and V100M32), we clearly see that the trend does not hold any more or even be reversed. For T4, the difference falls to less than one percent, and as more and more new GPUs are put into use, and more and more old GPUs are retired, it is safe to say that sharing a GPU or not will not have a negative impact on the success rate anymore.

6. Explore the wait time, task runtime for tasks shared GPU

The paper didn't analyze the wait time and runtime for tasks that ran on shared GPUs. In this section, we explore whether allowing GPU sharing or not can have an impact on a task's wait time and run time. In the table below, we first analyze the wait time and run time according to whether a task is GPU sharing or not.

shared	wait_time	runtime
False	598.411026	5482.198652
True	204.444356	5356.928737

According to the table, we can see that although, on average, the run time of the two kinds of tasks is pretty similar, their wait times are drastically different. It could be due to GPU sharing, but it could also be due to the difference in the amount of GPU requested. For example, a task that requires 1 GPU is definitely expected to wait longer than a task that requests 0.1 GPU.

According to the data provided, tasks that do not request GPU sharing request around 1.17 GPUs per task, and 0.27 GPU is requested by each GPU-sharing task. Figure 10 represents the average wait time of tasks to get one unit of GPU grouped by the amount of GPU requested. For example, if a task requests 0.01 GPU and it waits 1 second, then to get one unit of GPU, the time is 100 seconds.

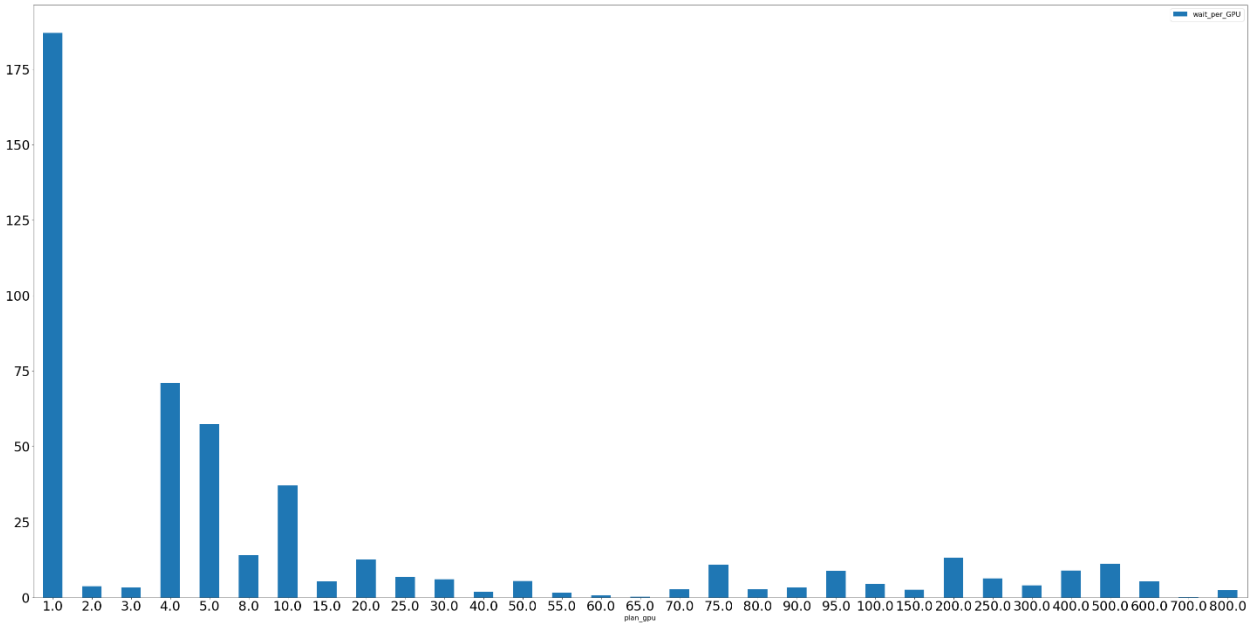


Figure 10: Wait time per unit GPU

After we look deeper into this by grouping tasks according to the amount of GPU required, we soon find that this is more likely due to the fragmentation of GPU requests. As we can see from the figure, the wait time per GPU is heavily skewed to the left, which means that as the request size becomes smaller and smaller, the time required per one unit of GPU becomes larger and larger. There is no observable difference in the wait time between GPU sharing and non-GPU sharing tasks in this case. Nicely expected to wait longer than a task that requests 0.1 GPU.

Analyze the fairness of the system based on wait time

To analyze fairness, we first need to define it. The GPU cluster can be considered a complex multi-server with a multi-queue system. However, we have no idea how the cluster schedules each job, so we can simplify the queueing system and try to define the system in an intuitive way. We could ignore the fairness of resource assignments such as GPU and CPU, as the user chooses what resource they use, and we can focus on the fairness of the wait time.

We will use the following methods to analyze fairness. First, the most simple way of defining fairness is to consider all tasks equal, and we should let each task wait the same amount of time. Second, we could use the idea of "proportional fairness," the metric captures the intuitive notion that it is fair for larger jobs to have larger response times and for smaller jobs to have short response times.[8]

1. **Analyze fairness between hour slots**

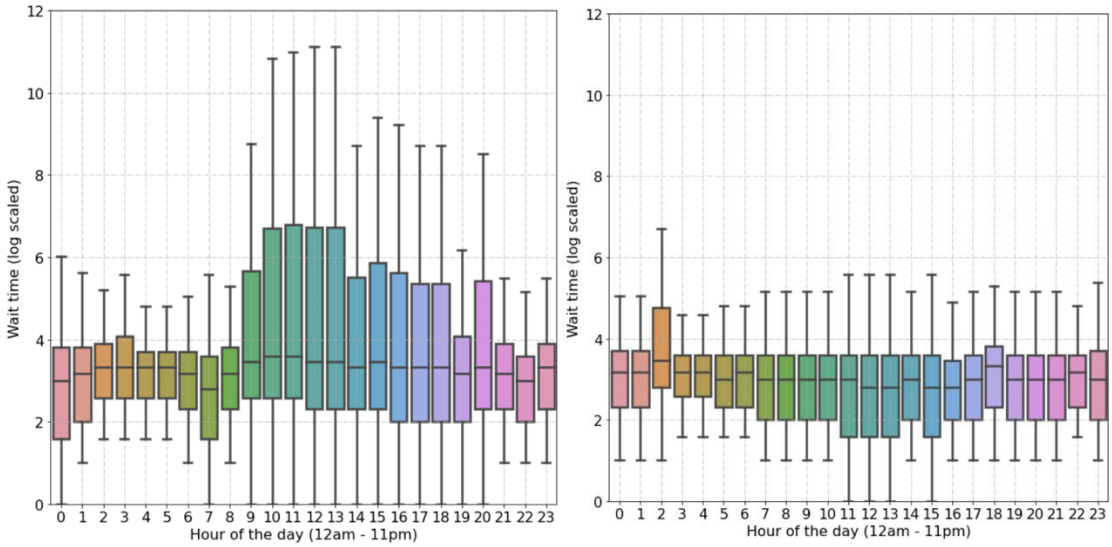


Figure 11: wait time for tasks used whole GPU wait time for tasks used shared GPU

We applied a log scale to wait time to visualize the distribution better. If a box plot has a long box than others, then the data set that the box plot present has a larger variance. But as we hope all people wait the same time, a larger variance in the data means the difference between most

people's wait times is larger and thus unfair. From figure 11, shared tasks' wait time does not change a lot during the day, while non-shared tasks' wait time fluctuates a lot during the work time from 9 am to 8 pm. Since the box plot for tasks using shared GPU is almost the same across each hour, based on the first definition, in which all tasks are equal, tasks using shared GPU are quite fair across different hour slots of a day. But tasks using the whole GPU across working time are not as fair as other times and are less fair compared to tasks using shared GPU in general.

2. **Analyze fairness between GPU type and GPU resource occupations**

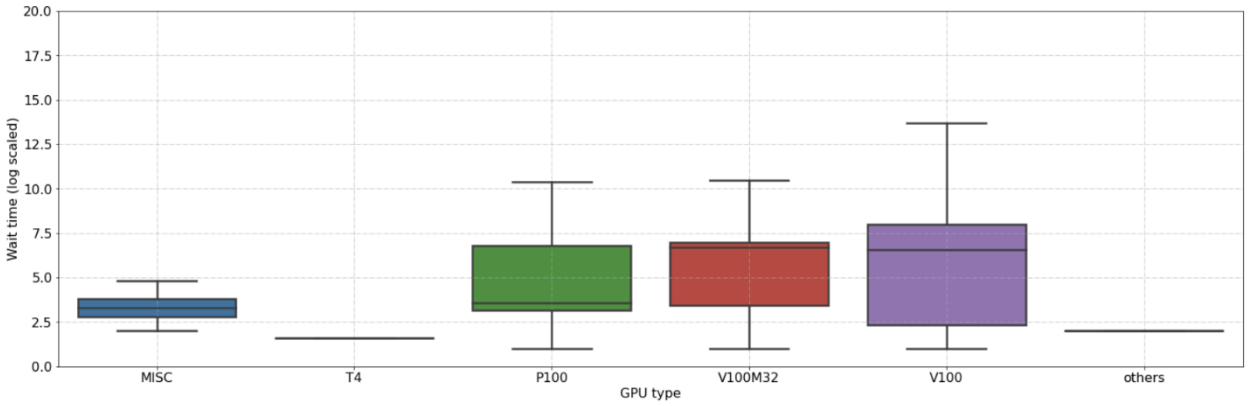


Figure 12: wait time by GPU type

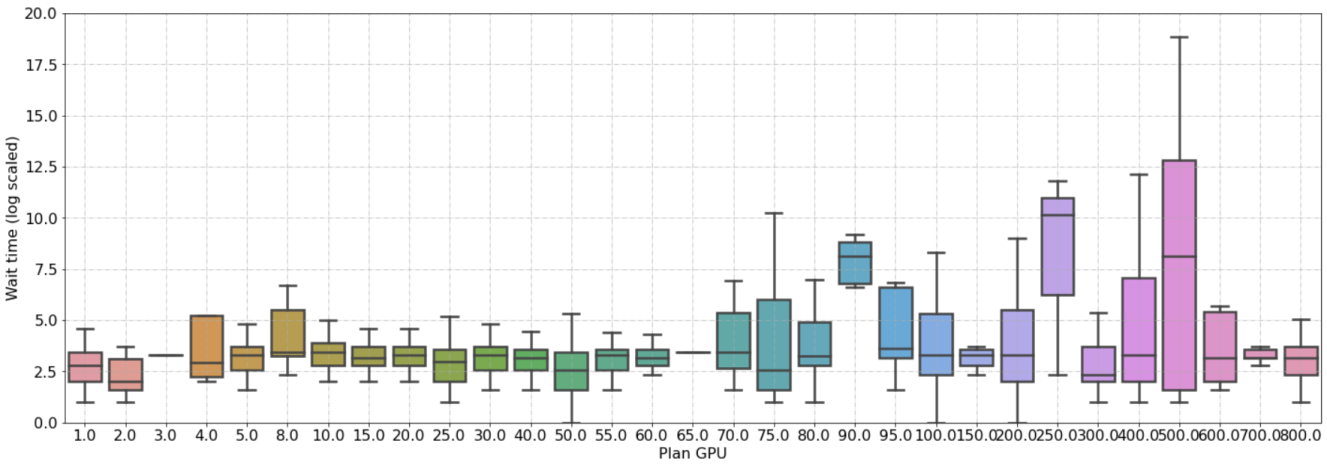


Figure 13: wait time by planned GPU usage (in percent)

Based on previous analysis, some GPUs are more likely to be shared, such as MISC and T4. And from Figure 12, we can see that tasks using MISC and T4 have almost the same wait time. Further from figure 13, tasks selected to run using less than 70% of a GPU will have less wait time, and each task waits a similar amount of time. Hence, if we consider all tasks equal, tasks using MISC and T4 GPU type and planning to run using less than 70% of GPU tasks are fairer than other tasks. If we consider tasks that occupy more resources should wait longer, then the wait time distribution becomes reasonable, as tasks requesting high-end GPU tend to wait longer, and tasks that use more GPU resources tend to wait longer. But the fluctuation in wait time for high-GPU tasks is large, and those users may experience unfairness.

3. Analyze the correlation between wait time and CPU-usage

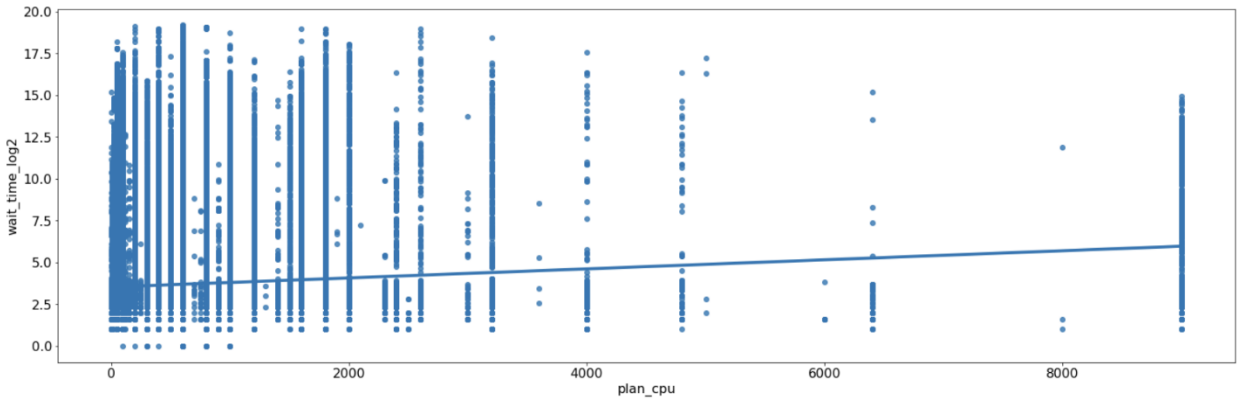


Figure 14: Linear relationship between plan_cpu and wait time.

By using ordinary least squares, we fitted a linear regression model between plan_cpu and wait time. Here we plotted the regression line between log scaled wait time and plan_cpu to show the positive relationship. The coefficient is 0.0107 with a p-value close to zero, which shows there is a strong positive relationship between plan_cpu and wait time. So we think the system is enforcing proportional fairness to the system, and the wait time is proportional to the resource they acquired. To improve, we could further explore a mathematical way of evaluating fairness and unfairness instead of intuitively.

Estimate wait time by using Machine Learning Algorithms

As we observed above, people could experience different wait times depending on the resource they chose. Estimating wait time could be beneficial for users to manage their work. So, we want to apply machine learning methods to create a model that estimates the wait time based on the resource user requested and the time that the user submitted.

We preprocess the data by selecting all the planned resources and the hour task initiated. We observed some wait times being negative, which is not possible, so we removed those rows with negative wait times. Then we drop NA and split the data for training.

1. Linear Regression

We would like first to try the linear regression model to estimate the wait time. Because linear regression is simple to understand, and based on our exploration, some variables seems to correlate with the score linearly, so we would like to try the linear regression model first.

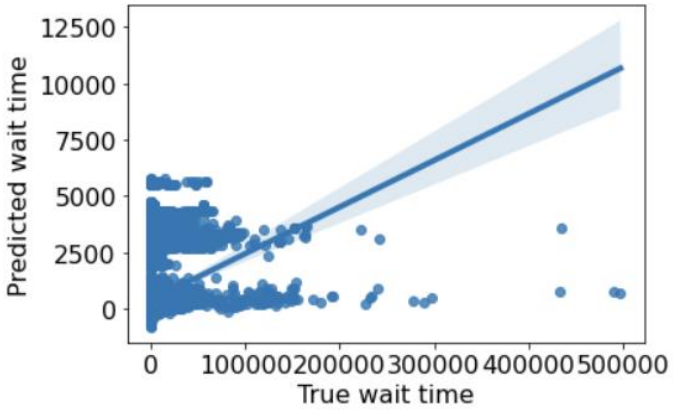


Figure 15: Linear regression

Based on the plot, we can see the model captures the linear relationship between each variable and the wait time, but when the actual wait time is small, there are a lot of variances, causing the model not to perform well. We use Root Mean Square Error (RMSE) to evaluate the result. The metric can describe how far off we should expect our model to be on its next prediction. The RMSE is 4010. This shows our model's prediction, and the true value has about 4000 seconds deviation, and it is a lot for most tasks with a short wait time, but for tasks that request a lot of resources and expected to wait longer according to proportional fairness, this model might be useful.

2. Random Forest

The second model we want to try is RandomForest. RandomForest fits several classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. We chose this algorithm because there could have some non-linearity between each variable and the wait time, and the decision tree could capture those non-linearities. Further, random forests could reduce variance by training on different samples of the data.

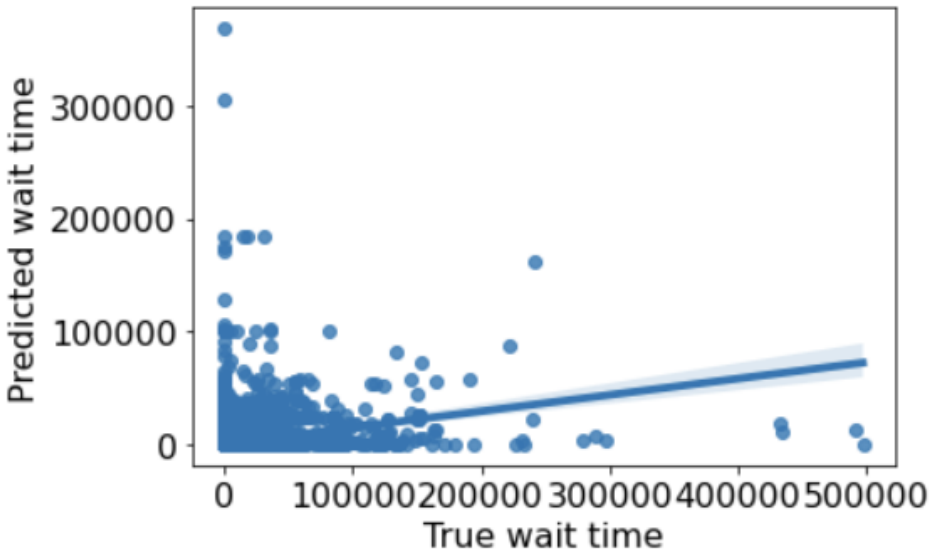


Figure 16: Random Forest

The RMSE is 4072 and is larger than the RMSE of the linear regression model. But from the plot, we observe that the random forest predicted wait time is around the same scale as the true wait time gets larger. So the model may perform better for tasks expected to wait longer, for example, more than 100000 seconds (about one day 4 hours).

Enhance the simulator to support Round Robin Scheduling

Round Robin scheduling is one of the scheduling algorithms that guarantee fairness. Round robin is a preemptive scheduling algorithm, while the simulator only supports two non-preemptive scheduling algorithms. Instead of directly modifying the logic of the simulator, we plan to simulate the round-robin by chopping the jobs with a duration longer than the scheduling window. Then, we feed the jobs into the simulator and use the FIFO runner to run. We have written a script that can chop the jobs based on the job number limit. For instance, upon receiving the original CSV file that contains 100k jobs, it can only chop the first n jobs based on user input and output them into a new CSV file so that the simulator can read it. It can also support two ways of simulating the extra time needed for context switching: one is an absolute value, and another is proportional to the job length. However, we encountered a problem: for a set number of jobs, the FIFO scheduling algorithm will be completed in a much shorter time than our round robin algorithm. We suspect that this may be caused by some integral logic of the scheduler. We will look into this issue more deeply. However, as there is no document for the simulator nor any comments in the code, we might need help to identify the cause and address it.

Summary

In summary, by using the data collected from the GPU cluster PAI[1], we have explored the GPU type, task names, and their affection for GPU sharing. We created interactive plots for users to view the data intuitively. We observed from the data that high-end GPUs are less likely to be shared, and TensorFlow tasks are more likely to use shared GPUs. And GPU sharing does not affect the failure rate. So the system's GPU sharing is working as expected in Antman[4]. But the technique could be further applied to PyTorch tasks, as not many PyTorch tasks use shared

GPU. Further, GPU sharing could be applied to tasks that request large resources, as tasks that request large resources experience some unfairness, and high-end GPUs are not utilizing GPU sharing. We further tried to estimate the wait time using linear regression and random forest. The models have the potential to estimate tasks that require large resources and are expected to wait longer. For future works, we could modify the simulator to support simulating round-robin and GPU sharing if the author provides clear documentation. And evaluate the fairness of the system mathematically would also be a good project.

Related Works

1. *MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters*. USENIX. (n.d.). Retrieved December 12, 2022, from https://www.usenix.org/conference/nsdi22/presentation/weng
2. Yu, P., & Chowdhury, M. (2019, February 12). *Salus: Fine-grained GPU sharing primitives for deep learning applications*. arXiv.org. Retrieved December 12, 2022, from https://arxiv.org/abs/1902.04610
3. Geng, Xin & Zhang, Haitao & Zhao, Zhengyang & Ma, Huadong. (2020). Interference-aware parallelization for deep learning workload in GPU cluster. Cluster Computing. 23. 1-14. 10.1007/s10586-019-03037-6.
4. *Antman: Dynamic scaling on GPU clusters for Deep Learning*. USENIX. (n.d.). Retrieved December 12, 2022, from https://www.usenix.org/conference/osdi20/presentation/xiao

5. Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In Proc. USENIX ATC, 2019. https://github.com/msr-fiddle/philly-traces.
6. NVIDIA Multi-Instance GPU (MIG) https://www.nvidia.com/en-us/technologies/multi-instance-gpu/
7. Yu, P., & Chowdhury, M. (2019, February 12). *Salus: Fine-grained GPU sharing primitives for deep learning applications*. arXiv.org. Retrieved January 5, 2023, from https://arxiv.org/abs/1902.04610v1
8. Wierman, A. (2011). Fairness and scheduling in Single Server Queues. *Surveys in Operations Research and Management Science*, *16*(1), 39–48. https://doi.org/10.1016/j.sorms.2010.07.002
9. Elliott, R. (n.d.). A measure of fairness of service for scheduling algorithms in Multiuser Systems. *IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No.02CH37373)*. https://doi.org/10.1109/ccece.2002.1012991