MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters

Qizhen Weng, *Hong Kong University of Science and Technology and Alibaba Group;* Wencong Xiao, *Alibaba Group;* Yinghao Yu, *Alibaba Group and Hong Kong University of Science and Technology;* Wei Wang, *Hong Kong University of Science and Technology;* Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding, *Alibaba Group*

https://www.usenix.org/conference/nsdi22/presentation/weng

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters

Qizhen Weng[†*], Wencong Xiao[*], Yinghao Yu[*†], Wei Wang[†], Cheng Wang[*], Jian He[*], Yong Li[*], Liping Zhang[*], Wei Lin[*], and Yu Ding[*]

[†]Hong Kong University of Science and Technology [*]Alibaba Group

{qwengaa, weiwa}@cse.ust.hk, {wencong.xwc, yinghao.yyh, wc189854, jian.h, jiufeng.ly, liping.z, weilin.lw, shutong.dy}@alibaba-inc.com

Abstract

With the sustained technological advances in machine learning (ML) and the availability of massive datasets recently, tech companies are deploying large ML-as-a-Service (MLaaS) clouds, often with heterogeneous GPUs, to provision a host of ML applications. However, running diverse ML workloads in heterogeneous GPU clusters raises a number of challenges. In this paper, we present a characterization study of a two-month workload trace collected from a production MLaaS cluster with over 6,000 GPUs in Alibaba. We explain the challenges posed to cluster scheduling, including the low GPU utilization, the long queueing delays, the presence of hard-to-schedule tasks demanding high-end GPUs with picky scheduling requirements, the imbalance load across heterogeneous machines, and the potential bottleneck on CPUs. We describe our current solutions and call for further investigations into the challenges that remain open to address. We have released the trace for public access, which is the most comprehensive in terms of the workloads and cluster scale.

1 Introduction

Driven by recent algorithmic innovations and the availability of massive datasets, machine learning (ML) has achieved remarkable performance breakthroughs in a multitude of real applications such as language processing [23], image classification [33,55], speech recognition [32,56,62], and recommendation [30, 60, 74]. Today's production clusters funnel large volumes of data through ML pipelines. To accelerate ML workloads at scale, tech companies are building fast parallel computing infrastructures with a large fleet of GPU devices, often shared by multiple users for improved utilization and reduced costs. These large GPU clusters run all kinds of ML workloads (e.g., training and inference), providing infrastructure support for ML-as-a-Service (MLaaS) cloud [2–4, 7, 8].

In this paper, we share our experiences in running ML workloads in large GPU clusters. We present an extensive characterization of a two-month workload trace[1] collected from a production cluster with 6,742 GPUs in Alibaba PAI (Platform for Artificial Intelligence) [2]. The workloads are a mix of training and inference jobs submitted by over 1,300 users, covering a wide variety of ML algorithms including convolutional and recurrent neural networks (RNNs and CNNs), transformer-based language models [23, 37, 56], GNNs-based (graph neural network) recommendation models [31, 57, 75], and reinforcement learning [39, 43, 44]. These jobs run in multiple ML frameworks, have different scheduling requirements like GPU locality and gang scheduling, and demand variable resources in a large range spanning orders of magnitude. GPU machines are also heterogeneous (see Table 1) in terms of hardware (e.g., V100, P100, T4) and resource configurations (e.g., GPUs, CPUs, and memory size). In comparison, prior workload analyses focus mainly on training CNN and RNN models in homogeneous environments [18, 29, 36, 41, 65, 66, 72].

The large heterogeneity of ML workloads and GPU machines raises a number of challenges in resource management and scheduling, making it difficult to achieve high utilization and fast job completion. We present those challenges, describe our solutions to some of them, and invite further research on the open problems.

Low utilization caused by fractional GPU uses. In our cluster, a task instance usually can only use parts of a GPU. In fact, the median usage of streaming multiprocessors (SMs) of an instance is 0.042 GPUs. Existing coarse-grained GPU allocation schemes dedicate an entire GPU to one task instance [36, 41, 72], and would result in extremely low utilization in our cluster.

We address this problem with *GPU sharing*, a technique that allows multiple ML tasks to time-multiplex a GPU in a controlled manner [66]. Utilizing this feature, the scheduler consolidates a large volume of low-GPU workloads onto a small number of machines, using only 50% of the requested

[1]The trace was collected in July and August 2020, and is now open for public access as part of the Alibaba Cluster Trace Program [1].

GPUs on average. Such consolidation causes no severe interference: among high-utilization GPUs, only 4.5% run ML tasks with potential contention on SMs.

Long queueing delays for short-running task instances. Short-running task instances are prone to long queueing delays caused by head-of-line blocking. In fact, around 9% of short-lived instances spent more than half of their completion time waiting to be scheduled. An effective solution is to predict the task run-time and prioritize short tasks over the long ones. Existing approaches require specialized framework support to track and estimate the training progress [41,46,49], which is not always possible in production as users can run standard or customized ML frameworks without such feature.

However, there is a silver lining. In our cluster, the majority of workloads are recurring, with 65% of tasks repeatedly executed at least 5 times in the trace. Through careful feature engineering, we can predict the durations of most recurring tasks within 25% error, sufficient to make quality scheduling decisions as suggested by previous work [16]. Trace-driven simulations shows that using shortest-job-first scheduling with predicted task durations reduces the average completion time by over 63%.

Hard to schedule high-GPU tasks. Our cluster runs a small portion of compute-intensive ML tasks for business-critical, user-facing applications. These tasks request full GPUs (no sharing) and can attain dramatic speedup on high-end devices by exploiting advanced hardware features such as NVLink [12] (see Section 6.1)—these picky requirements make them difficult to schedule.

Our scheduler employs a simple *reserving-and-packing* policy to differentiate those hard-to-schedule high-GPU tasks from other tasks. It reserves high-end GPU machines (e.g., V100 with NVLinks) for a small number of high-GPU tasks with picky scheduling requirements, while packing the other workloads on less advanced machines, using GPU sharing. The reserving-and-packing policy reduces the average queueing delay by 68% for high-GPU tasks and 45% for all.

In our quest for optimized cluster management, a few challenges remain open, which have received less attention in the literature.

Load imbalance. We observe imbalanced load running in heterogeneous machines. In general, machines with low-end GPUs are more crowded than those with high-end GPUs: the former have over 70% CPUs and GPUs of these machines allocated on average, while the latter have only 35% CPUs and 49% GPUs allocated. There is also a *provisioning mismatch* between workloads and machines. On average, workloads running in 8-GPU machines demand 1.9× more CPUs per GPU than the machines can provide (12 CPUs per GPU), whereas those running in 2-GPU machines request 53% fewer CPUs per GPU than the machine specifications (32 or 48 CPUs per GPU).

Bottleneck on CPUs. While ML workloads perform training and inference on GPUs, many data processing (e.g., data fetching, feature extraction, sampling) and simulation tasks (e.g., reinforcement learning) involved in the pipeline run on CPUs, which can also become a bottleneck. In fact, we find that workloads running in machines with higher CPU utilization are more likely to get slowdown. For example, in T4 machines, those slowed tasks measure an average of 33.5% P75 CPU utilization, noticeably higher than that measured by the accelerated tasks (21.3%). Similar results are also found in V100 machines reserved for high-GPU workloads (50.6% P75 CPU utilization for slowed tasks and 42.4% for the accelerated), indicating that even GPU-demanding workloads can be harmed by CPU contention.

We believe the observations made in our cluster do not stand in isolation. We share the insights derived from our analysis and discuss potential system optimization opportunities in improving ML framework, adopting resource disaggregation, and decoupling data pre-processing from GPU training (see Section 7). We hope that the observations and experiences shared in our study, as well as the release of the PAI trace, can inspire follow-up research in optimizing ML workload scheduling and GPU cluster management.

2 Background

Fast growing data and GPU demand. The support for scalable machine learning has become increasingly important in production data processing pipelines. In our experience of operating general-purpose ML platforms for production workloads, we have witnessed the fast growing demand of both training data and GPU resources. In just a few years, the sheer volume of training data for an ML job has grown orders of magnitude, from the standard dataset of 100s GB (e.g., ImageNet [22]) to an Internet scale of 10s or even 100s TB. The massive volume of data forces ML jobs to scale out to a large number of GPU machines. In our cluster, the largest single ML job requests to run on over 1,000 GPUs, posing a significant gang-scheduling challenge to the cluster.

Alibaba PAI. To accommodate the fast growing computing demand of ML workloads, Alibaba Cloud offers Machine Learning Platform for AI (PAI), an all-in-one MLaaS platform that enables developers to use ML technologies in an efficient, flexible, and simplified way. PAI provides various services covering the entire ML pipeline, including feature engineering, model training, evaluation, inference, and autoML. Since its introduction in 2018, PAI has gained tens of thousands of enterprises and individual developers, making it one of the largest leading MLaaS platforms in China.

Figure 1 illustrates an architecture overview of PAI, where users submit ML jobs developed in a variety of frameworks, such as TensorFlow [14], PyTorch [48], Graph-Learn [75], RLlib [38]. Upon the job submission, users provide the application code and specify the required compute resources, such

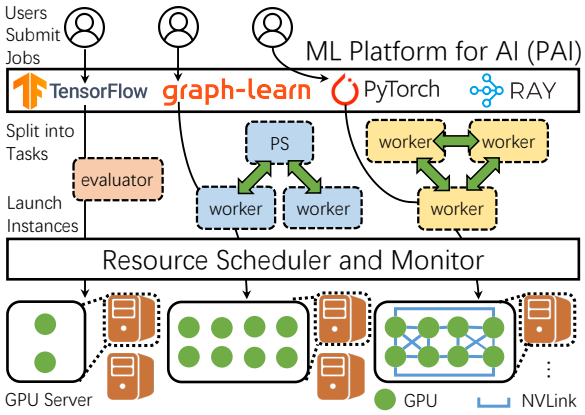


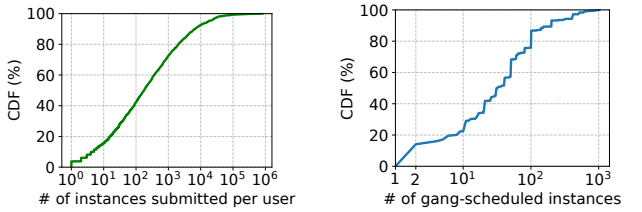
Figure 1: Architecture overview of PAI.

Table 1: Machine specs of GPU clusters in the existing trace analysis works. GPUs with † are equipped with NVLink [12]. The Philly trace does not reveal CPU specs and GPU types.

System	#CPUs	Mem (GiB)	#GPUs	GPU type	#Nodes
PAI	64	512	2	P100	798
	96	512	2	T4	497
	96	512	8	Misc.	280
	96	384	8	V100M32†	135
	96	512/384	8	V100†	104
	96	512	0	N/A	83
Philly [36]	Unk.	528/264	2	12GB GPU	321
	Unk.	528/264/132	8	24GB GPU	231
Tiresias [29]	20	256	4	P100†	15
$\text{Gandiva}_{\text{fair}}$ [18]	12	224	4	K80	32
	12	448	4	P100	12
	12	448	4	V100	6
Themis [41]	24	448	4	K80	12
	12	224	2	K80	8
HiveD [72]	24	224	4	K80	125
	24	224	4	M60	75
Antman [66]	96	736	8	V100M32†	8

as GPUs, CPUs, and memory. Each job is translated into multiple *tasks* of different roles, such as parameter servers (PS) and workers for a training job, and evaluator for an inference job. Each task may consist of one or multiple instances and can run on multiple machines. PAI employs Docker containers to instantiate tasks for simplified scheduling and execution on heterogeneous hardware.

Trace analysis. Running diverse ML workloads in shared GPU clusters at cloud scale raises daunting challenges. Trace analysis is essential to understand those challenges and provide new insights on system optimization. However, existing analyses are performed on GPU clusters with limited size, workload diversity, and machine heterogeneity, and hence cannot fully represent the state of the art (see Table 1). Take Microsoft's Philly trace [36] as an example. Whereas distributed training is now commonplace, the majority of Philly



(a) CDF of the number of instances run by a user.

(b) CDF of gang-scheduled task instances.

Figure 2: Heavily skewed distribution of task instances run by users and the prevalence of gang-scheduling requirements.

workloads ($> 82\%$) ran on a single GPU instance when the trace was collected in 2017. It is also unclear what types of GPUs were used to run those workloads, which may have significant impact to scheduling [41, 46]: the performance of new-generation GPUs can be 1.1–8× higher than the older generations [18]. Moreover, the Philly trace only includes the training workloads, whereas it is common to run both training and inference jobs in a shared MLaaS platform [47, 51, 69].

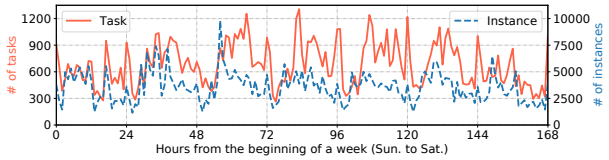
The insufficiency of existing works motivates the release of the PAI trace, which we examine next.

3 Workload Characterization

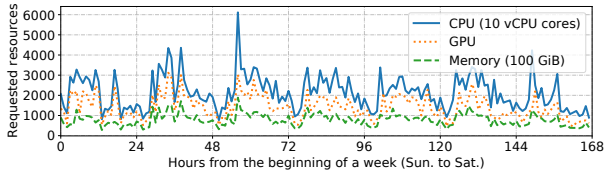
In this section, we analyze the ML workloads in the released PAI trace. We start with an overview of the trace, followed by a characterization of its temporal and spatial patterns.

3.1 Trace Overview

Trace information. The released PAI trace contains a hybrid of training and inference jobs running state-of-the-art ML algorithms in mainstream frameworks [14, 48, 75]. Most jobs request multiple GPUs. The trace records the arrival time, completion time, resource requests and usages in GPUs, CPUs, GPU memory and main memory of the workloads at various levels (e.g., job, task, and instance) (Sections 3.2 and 3.3). The application semantics, such as whether the code is performing training or inference, and in what ML framework, are not available as our cluster scheduling system Fuxi [26, 71] only sees the execution containers and is agnostic to the running applications. Nevertheless, we have manually examined some workloads and included their application names (e.g., click-through rate prediction and reinforcement learning) in the trace to provide some clues whenever possible (Sections 6.1 and 6.2). Machine-level information is also provided in the trace, including the hardware specs (Table 1) and time-varying resource utilizations (Section 4) collected by the daemon agents that periodically query the Linux kernel and GPU driver (e.g., NVIDIA Management Library [9]) in the host machines. The detailed schema and trace data are given in the trace repository [1].



(a) Number of tasks submitted and their instances in one week.



(b) Total resource requests of running tasks in one week.

Figure 3: Task submissions and resource requests roughly follow diurnal patterns.

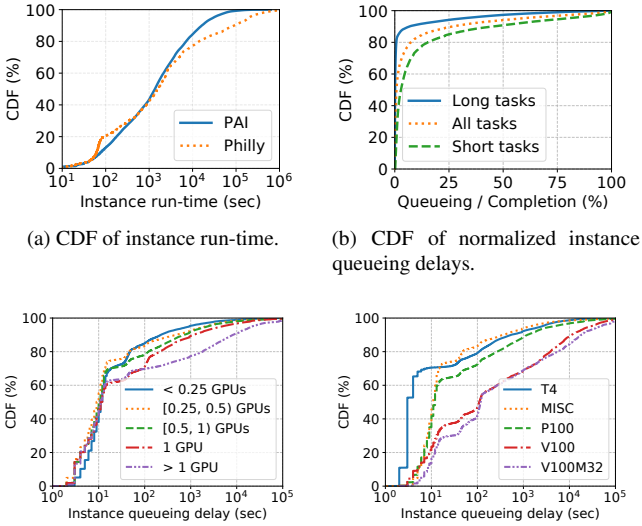
Jobs, tasks, and instances. In PAI, users submit *jobs*. Each job has one or multiple *tasks* taking different computation roles. Each task runs one or multiple *instances* in Docker containers. For example, a distributed training job may have a parameter-server (PS) task of 2 instances and a worker task of 10 instances. All instances of a task have the same resource demands and might be *gang-scheduled* (e.g., running simultaneously for all PyTorch workers). Our characterization in this subsection mainly focuses on task instances.

Heavy-skewed instance distribution. The PAI trace contains more than 7.5 million instances of 1.2 million tasks submitted by over 1,300 users. Figure 2a depicts the distribution of task instances run by users, which is *heavily skewed*. More specifically, around 77% of task instances are submitted by the top 5% users, each running over 17.5k instances, while the bottom 50% users run less than 180 instances each.

The prevalence of gang-scheduling. Our distributed ML jobs require gang-scheduling. As shown in Figure 2b, among all task instances, around 85% have such requirements, in which 20% must be gang-scheduled on more than 100 GPUs, some even requesting over 1,000. Together, tasks with gang-scheduled instances account for 79% of the total GPU demands. The prevalence of these tasks makes it difficult to achieve high utilization.

GPU locality. In addition to gang-scheduling, a task may request to run all its instances on multiple GPUs co-located in one machine, a requirement known as *GPU locality*. Although such requirement often leads to prolonged scheduling delays [29, 36, 72], it enables the use of high-speed GPU-to-GPU interconnect within a single node (e.g., NVLink and NVSwitch), which can dramatically accelerate distributed training [12, 15, 36]. In our cluster, enforcing GPU locality yields over 10× speedup for some training tasks (Section 6.1).

GPU sharing. PAI supports *GPU sharing* that allows multiple task instances to time-share a GPU at a low cost. With this feature, users can specify GPU request in $(0, 1)$ and run its task instances using parts of GPUs. We will show in Section 5.1 that GPU sharing enables considerable savings on GPU provisioning.



(a) CDF of instance run-time.

(b) CDF of normalized instance queueing delays.

(c) CDF of queueing delays w.r.t. GPU requests per instance.

(d) CDF of queueing delays w.r.t. GPU types.

Figure 4: CDF of instance run-time and queueing delays.

Various GPU types to choose from. PAI provides heterogeneous GPUs and allows users to specify the required GPU types to run their tasks. The available choices include NVIDIA Tesla T4, P100, V100, V100M32 (V100 SXM2 with 32 GB memory), and other GPUs of older generations (Misc in Table 1), e.g., Tesla K40m, K80, and M60. In our cluster, only 6% tasks require to run on specified GPUs, while the others have no such limitation and can run on any GPUs.

3.2 Temporal Pattern

We next examine the temporal patterns of the PAI workloads.

Diurnal task submissions and resource requests. Figure 3 depicts task and instance submissions as well as the overall resource requests in one week during the trace collection period. We observe rough diurnal patterns, where task submissions in weekdays (from the 24$^{\text{th}}$ to 144$^{\text{th}}$ hours) are slightly higher than in weekends. It is worth mentioning that in addition to the daytime, midnight is also a rush hour for task submissions (Figure 3a). Yet, most tasks submitted at midnight are less compute-intensive, having only a few instances and requesting a small amount of resources (Figure 3b).

Instance run-time in a wide range. Figure 4a shows the distribution of instance run-time (solid line). Similar to the Philly trace [36] (dotted line), instance run-time varies in a wide range spreading four orders of magnitude. The median run-time (23 minutes) is comparable with that of Philly (26 minutes), while their 90$^{\text{th}}$ percentile (P90) run-time (4.5 hours) is shorter than that of Philly (25 hours).

Non-uniform queueing delays. The queueing delay (aka wait time or scheduling delay), measured from the moment of task submission to the start of the task instance, varies greatly among instances. Compared to the long-running instances, short-running instances usually spend a larger portion of time in queueing. To see this, we use the median run-time as a threshold and divide instances into long-running and short-running ones, where a long-running (short-running) task instance has a longer (shorter) run-time than the median. In Figure 4b, We compare the queueing delays of these task instances relative to their completion times (queueing delay plus run-time). Around 9% short-running instances spend more than half of the completion time waiting to be scheduled; this number drops to 3% when it comes to long-running instances.

A task instance's queueing delay also depends on its GPU request. Figure 4c shows that instances willing to share GPUs (i.e., GPU request in $(0, 1)$) can be quickly scheduled, with the 90^{th} percentile (P90) queueing delay being 497 seconds. In comparison, instances that do not accept GPU sharing need to wait for a longer time, with the P90 delay being 1,150 (8,286) seconds for those requesting one GPU (> 1 GPU).

Long queueing delays are also seen in instances requesting high-end GPUs. As shown in Figure 4d, for instances running on advanced V100 GPUs (including V100M32), the median and P90 delays are 113 and 13,709 seconds, respectively. In comparison, for instances running on low-end miscellaneous GPUs, the median and P90 delays are only 11 and 360 seconds, respectively.

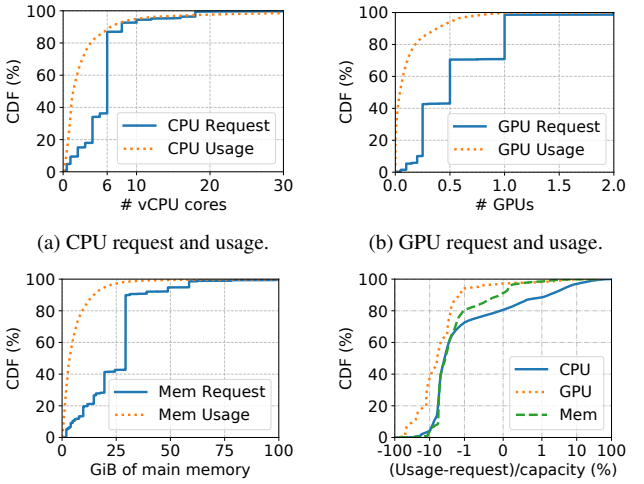
3.3 Spatial Pattern

We finally present the spatial patterns of the PAI task instances by analyzing their resource requests and usages. PAI collects the system metrics of running tasks every 15 seconds and provides visualization tools [2, 25] for users to analyze the workload patterns and figure out their resource requests.

Heavy-tailed distribution of resource requests. Figures 5a, 5b, and 5c (blue solid lines) respectively depict the distributions of the total CPUs, GPUs, and memory requested by all instances. All three distributions are heavy-tailed, with around 20% instances requesting large resource amounts and the other 80% requesting small to medium. More specifically, the P95 request demands 12 vCPU cores[2], 1 GPU, and 59 GiB memory, more than twice the median request (6 vCPU cores, 0.5 GPUs, and 29 GiB memory).

Uneven resource usage: Low on GPU but high on CPU. Most users tend to ask for more resources than they actually need, resulting in a low resource usage (dotted lines in Figures 5a, 5b, and 5c). In our cluster, the median instance resource usages are 1.4 vCPU cores, 0.042 GPUs, and 3.5 GiB memory, much smaller than the median request. We stress

[2]In our cluster, each physical processor core consists of two vCPU cores, using hyper-threading technology [42].



(a) CPU request and usage.

(b) GPU request and usage.

(c) Memory request and usage.

(d) Usage minus request, normalized by the machine capacity.

Figure 5: CDF of instance resource requests and actual usages.

that the low GPU usage is not caused by the low computing demand, but by contentions on other resource like CPU, making GPUs idle for most of the time (Section 6.2). Figure 5b also shows that around 18% instances barely use GPUs: they perform computations such as running parameter servers, fetching and pre-processing data, which are mostly on CPUs with small or no GPU involvement.

In PAI, instances of a task can use *spare resources* in the host machines, making it possible to *overuse* more resources than requested. Compared to GPU and memory, overuse of CPUs is more prevalent. To see this, for each instance we measure the difference between its resource usage and request for CPU, GPU, and memory—positive (negative) being overuse (underuse). We normalize the results by the machine's CPU, GPU, and memory capacity, respectively, and depict the distributions in Figure 5d. There are 19% task instances overusing CPUs (blue solid line with $X > 0$). In comparison, only 3% (9%) instances use more GPUs (memory) than they requested.

4 GPU Machine Utilization

Having studied the workload characterization, we turn to resource utilization in GPU machines.

4.1 Utilization of Compute Resources

We start to analyze the utilization of compute resources, including CPU, GPU, main and GPU memory. Our cluster has 1,295 2-GPU machines and 519 8-GPU machines (Table 1). Machines with 8 GPUs have a lower CPU-to-GPU ratio than those with 2 GPUs. In light of their different configurations, we perform measurement separately for the two types of machines. Each machine has time series data of resource utilization measured every 15 seconds by the monitoring system.

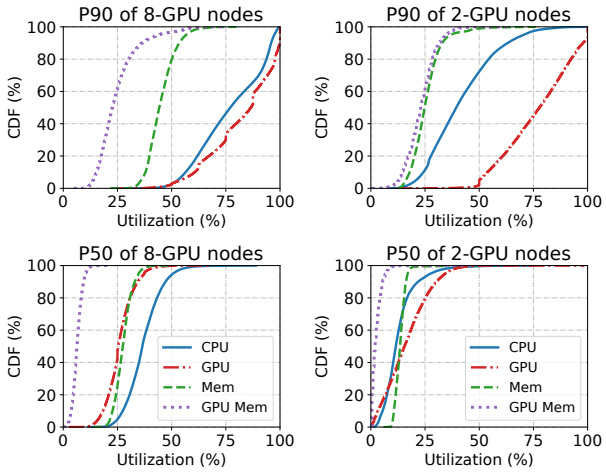


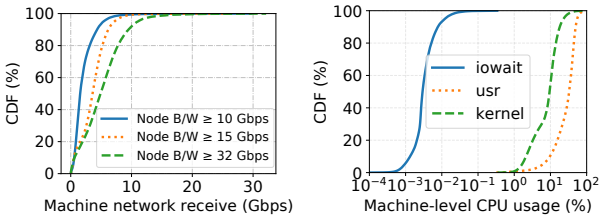
Figure 6: CDF of P90 and P50 (median) utilization of CPU, GPU, main and GPU memory in different machine groups.

At each timestamp, we collect the utilization of all 8-GPU machines and calculate the tail (P90) and the median (P50). Together, we obtain a sequence of P90 and P50 utilizations taken at different timestamps. We depict their distributions in Figure 6 (two subfigures on the left). We perform the same measurements in 2-GPU machines and depict the results in the right two subfigures. Compared to memory (main and the GPU's), GPU and CPU have higher utilization. In 8-GPU machines (upper-left in Figure 6), the average P90 utilization of GPU (red dash-dotted line) and CPU (blue solid line), i.e., the arithmetic mean of P90 values from all timestamps, reaches 82% and 77%, respectively. In 2-GPU machines (upper-right in Figure 6), the P90 GPU utilization remains high (77% on average), while the P90 CPU utilization drops to 42% on average due to the large CPU-to-GPU ratio (32 or 48 CPUs per GPU). In both types of machines, the P90 utilization of the main and GPU memory stays below 60% at almost all time, indicating that our tasks are less memory-intensive.

Compared to other resources, we measure a larger variation of utilization on GPUs. As shown in Figure 6, the distribution of P90 GPU utilization spans a wide range from less than 40% to 100% of the computing power provided by the streaming multiprocessors of the machine's GPUs; the difference between the tail and the median utilization is also larger on GPU than on other resources (comparing the top sub-figures with the bottom). The large variation is partly due to the bursty GPU usage patterns found in our ML workloads [65, 66]. It is also due to the design of our scheduler that prioritizes packing over load balancing (Section 6.3).

4.2 Low Usage of Network and I/O

In addition to compute resources, network and I/O are also frequently used in distributed ML. To understand their impact,



(a) CDF of machine network input. (b) CDF of machine CPU time.

Figure 7: Low usage of network and I/O.

we measure the network input rate[3] in machines with different bandwidth guarantees ($\geq$ 10 Gbps for P100 and Misc, $\geq$ 15 Gbps for T4, and $\geq$ 32 Gbps for V100) and depict their distributions in Figure 7a. The P95 network input rate only reaches 54%, 48%, and 34% of the guaranteed bandwidth provided in P100 (or Misc), T4, and V100 machines, respectively.

In terms of I/O, we collect machine-level CPU usage data, including the I/O waiting time (iowait) and the execution time in usr and kernel modes, respectively. Figure 7b shows their distributions. The CPU time spent on iowait is three orders of magnitude smaller than that in usr and kernel modes, meaning that CPUs are mostly busy processing data rather than waiting for the I/O to complete.

5 Opportunities for Cluster Management

In PAI, our goal of cluster management is two-fold: (1) achieving high utilization in GPU machines, and (2) completing as many tasks as fast as possible. In this section, we describe the opportunities and our efforts in achieving the two goals.

5.1 GPU Sharing

Unlike CPUs, GPUs do not natively support sharing and are allocated as indivisible resources in many production clusters [36, 72], where a single task instance runs exclusively on a GPU. Although such allocation provides strong performance isolation, it results in GPU underutilization, which is particularly salient in our cluster as most instances can only utilize a small portion of the allocated GPUs (Section 3.3).

To avoid this problem, the PAI cluster scheduler supports GPU sharing which allows multiple task instances to run on the same GPU in a space- and time-multiplexed manner. With this feature, a task instance can request a fraction of GPU ($<$ 1 GPU) and is guaranteed to allocate the specified fraction of GPU memory upon scheduling (space-multiplexed). When needed, an instance can also use unallocated GPU memory during execution. An instance, however, has no guaranteed allocation of compute units (i.e., SMs), which are dynamically shared among co-located instances (time-multiplexed).[4]

[3] Our trace does not log the network output. For most training and inference tasks, the network input is orders of magnitude larger than the output.

[4] Fine-grained sharing of compute units with isolation guarantee requires

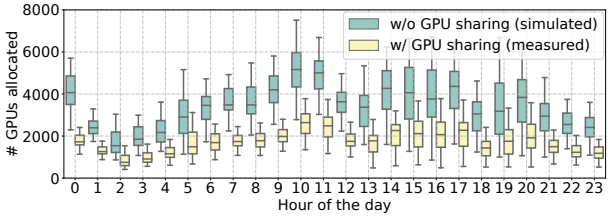
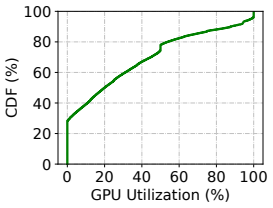
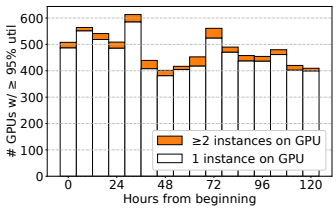


Figure 8: Box plots of the number of allocated GPUs with and without GPU sharing. The boxes depict the 25th, 50th, and 75th percentiles, respectively; the two whiskers are one interquartile range (IQR) past the low and high quartiles.



(a) CDF of GPU utilization.



(b) The number of GPUs with $\geq 95\%$ utilization in a 5-day period. The top-stacked bars (orange) show GPUs running ≥ 2 instances.

Figure 9: Heavy utilization is rarely measured in GPUs; most heavy-utilized GPUs run a single instance.

Benefits of GPU sharing. GPU sharing enables considerable savings on resource provisioning. To see this, we simulate the scenario of no GPU sharing, in which we replay the trace and count the number of allocated GPUs in each hour. Figure 8 compares the simulated results with the numbers measured in the real system, binned in hour of the day. On average, only 50% of GPUs are needed with sharing. In the peak hour at around 10 am, the savings can be up to 73%.

Does GPU sharing cause contention? As the utilization increases, instances running on a shared GPU start to contend for streaming processors (SMs), causing interference. To quantify how frequently the contention may occur, we collect the utilization data of all GPUs in two months and depict their distribution in Figure 9a. Heavy utilization ($\geq 95\%$) is rarely measured, which accounts for only 7% cases in the trace. We further examine those heavy-utilized GPUs in which running instances have a high chance to contend with each other. Figure 9b shows the number of heavy-utilized GPUs in a 5-day period, among which only a few (4.5% on average) run multiple instances (the top-stacked bars). As the majority of heavy-utilized GPUs run a single instance, no contention occurs. We therefore believe GPU sharing does not cause severe contention in our cluster.

high-level support of ML framework. In PAI, such support is provided by AntMan [66]. Yet, it only applies to tasks running in the frameworks where AntMan is implemented (currently supporting TensorFlow and PyTorch).

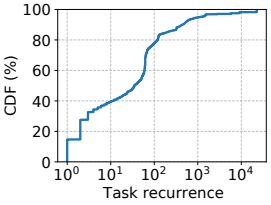


Figure 10: CDF of task recurrence.

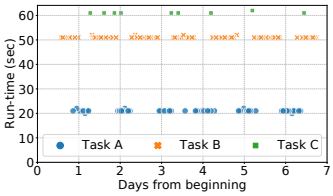


Figure 11: Submissions and instance run-times of three batch inference tasks using BERT.

5.2 Predictable Duration for Recurring Tasks

Knowing the duration (aka run-time) of ML task instances is the key to making better scheduling decisions. Existing schedulers for ML workloads predict the task instances duration based on the training progress (e.g., number of iterations, loss curve, and target accuracy) and speed of the task [29, 41, 46, 49]. Obtaining such information requires specific framework support (e.g., TensorFlow and PyTorch), which is not always possible in our cluster as users run a variety of frameworks of standard or customized version, and their submitted tasks may not perform iterative training (e.g., inference). In fact, our cluster scheduler [26, 71] is designed for container workloads and is agnostic to the task semantics.

The prevalence of recurring tasks. Despite the scheduler being agnostic to task progress, we find that most tasks are *recurring*, and their instance run-times can be well predicted from past executions. Yet, in our system, task recurrence cannot be simply identified from the task ID or name, which is uniquely generated for each submission. Instead, we turn to the meta-information consistently specified by a task across multiple submissions, such as the entry scripts, command-line parameters, data sources and sinks. Hashing the meta-information generates a unique Group tag, which we use to identify the recurrence of a task. Following this approach, we depict the distribution of task recurrences in Figure 10: around 65% tasks repeatedly run at least 5 times in the trace.

In addition to periodic training, many recurring tasks perform *batch inference*. These tasks aggregate data from incoming requests and then perform batch inference on a collective of data in one go. Users can configure the task launching interval, ranging from minutes to days. As an illustrative example, Figure 11 shows three recurring tasks identified in the trace that perform batch inference with pre-trained BERT [23] models. All three tasks run on a regular basis, with stable average instance run-times that can be accurately predicted.

Instance duration prediction for recurring tasks. A recurring task can be submitted by different users with different resource requests, and its instances may have different run-times. We therefore predict the duration from past runs based on three features, the task's username (User), resource requests (Resource, including GPU and other resources), and group tag (Group). Taking these features as input, we predict

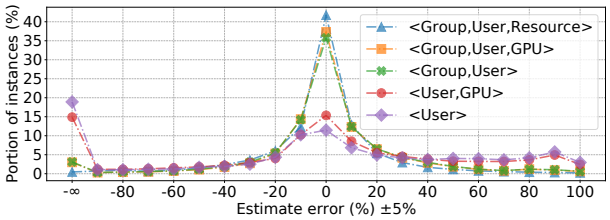


Figure 12: Percentage prediction error, i.e., $(true-pred)/true$ in percentage, of duration estimates with different features.

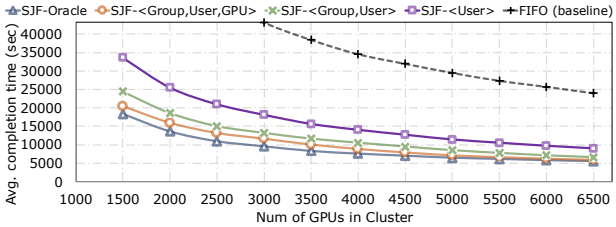


Figure 13: Average task completion time given different GPU cluster sizes and various scheduling policies in simulation.

the task's average instances duration using the CART (Classification And Regression Trees [17]) algorithm with a tree regressor. The regressor makes at most 10 splits for each tree and uses the mean absolute error (MAE) as the splitting criterion. We choose MAE instead of the standard mean squared error (MSE) because the former is more robust to extreme outliers in heavy-tailed distribution than the latter.

To evaluate the accuracy of our prediction, we consider tasks that recur at least 5 times in the trace. We use 80% of those tasks to train the predictor and the remaining 20% for testing. Figure 12 compares the accuracy of the predictor trained with different feature inputs, including Group, User, Resource, and GPU (requested GPU types and numbers). We use percentage prediction error [35] as the accuracy metric, defined as $(true-pred)/true \times 100\%$. Our evaluation shows that Group is the most important feature that greatly improves the prediction accuracy. Further complementing it with User and Resource (or GPU) results in less than 25% prediction error for 78% instances. According to prior studies [16], duration predictions with such accuracy is sufficient to make high-quality scheduling decisions.

Benefits for scheduling. We present a simple simulation study to evaluate how the prediction of task instance duration can help improve scheduling. We developed a discrete-time simulator and use it to replay the trace. We sample tasks from the trace and feed their resource requests, arrival times, real and predicted run-times into the simulator. We assume homogeneous GPUs in simulation and respect the real duration when scheduling a task instance to a GPU. Both the simulator and experiment scripts are released along with the trace [1].

We configure two scheduling policies, first-in-first-out (FIFO) and shortest-job-first (SJF), in simulation. Figure 13 shows the average task completion time in GPU clusters of different sizes using FIFO and four SJF schedulers, where SJF-Oracle makes scheduling decisions based on the real-measured task instance duration (ground truth) and the others use predictors trained with different input features. Compared to FIFO, the four SJF schedulers reduce the average task completion time by 63–77%, depending on the predictors they use. In particular, the predictors trained with the Group feature yield better performance; the more features are included, the more accurate the predictions are, and the closer the scheduling performance is to the optimum (SJF-Oracle). These results are in line with Figure 12.

6 Challenges of Scheduling

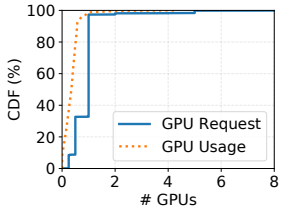
Compared to previous simulations, scheduling ML tasks of large heterogeneity in production clusters is far more complex. To understand the challenges posed by such heterogeneity, in this section we present case studies for two representative types of ML tasks with high and low GPU requests. We describe our scheduling policies deployed in production that differentiate between the two types of tasks in light of their different request and usage patterns. Yet, many challenges remain open, which we discuss in detail.

6.1 Case Study of High-GPU Tasks

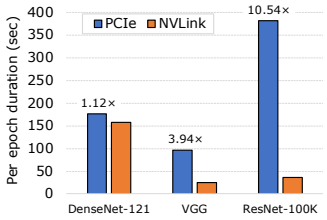
In our cluster, a small portion of tasks run compute-intensive instances with high GPU requests (Section 3.3). These tasks train state-of-the-art models or perform inference with trained models for business-critical, user-facing applications. They request powerful GPU devices with high memory or advanced hardware features (e.g., NVLink).

NLP with advanced language models. Around 6.4% tasks running in our cluster perform natural language processing (NLP) using advanced models, such as BERT [23], ALBERT [37], and XLNet [67]. Among them, 73% have large input and must run on GPUs with 16 GiB or higher memory (i.e., T4, P100, V100/V100M32). Figure 14a shows the distribution of GPU requests and usages of NLP instances, where 40% request more than 1 GPU and use over 0.4 GPUs in computing power. Comparing Figure 5b and Figure 14a, we observe much higher GPU requests and usages of NLP tasks than that of general workloads.

Image classification with massive output. In our cluster, some distributed training tasks request to run their worker instances in one machine with high-speed GPU-to-GPU interconnects (e.g., NVLink) for much improved performance, a requirement known as *GPU locality*. A typical example is to train a classification model that classifies images of goods into a large number of standard product units (SPUs). The model can be a modified ResNet [33] with the last output layer replaced by a softmax layer with 100,000 output of SPUs

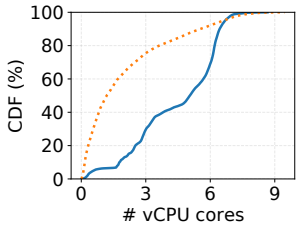


(a) CDF of GPU requests and usages of NLP task instances.

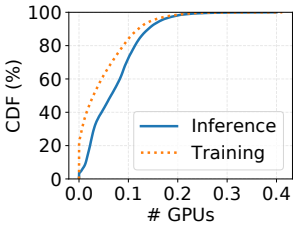


(b) Per-epoch duration of 3 classification models trained in 8-GPU machines with and without NVLink.

Figure 14: High-GPU tasks (NLP and image classification).

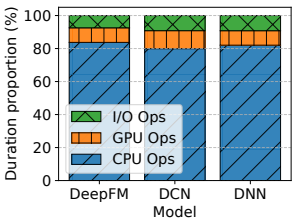


(a) CPU usage of instances.

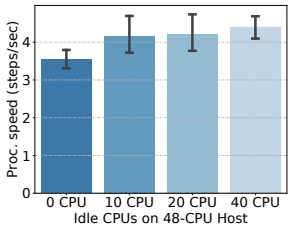


(b) GPU usage of instances.

Figure 15: CDF of the CPU and GPU usage of click-through-rate (CTR) instances.

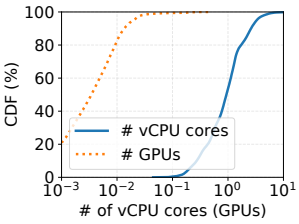


(a) Duration breakdown of CTR prediction instances.

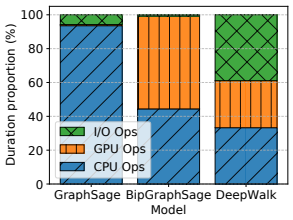


(b) DeepFM training instances interfered by the co-located load.

Figure 16: Microbenchmark of inference and training instances of click-through-rate prediction models.



(a) CDF of CPU and GPU usage.



(b) Instance duration breakdown.

Figure 17: Resource usage and duration breakdown of GNN training instances.

(ResNet-100k). The presence of such a large fully-connected layer mandates the exchange of massive gradient updates between worker instances, making communication a bottleneck. For these tasks, meeting GPU locality is critically important. Figure 14b compares the duration of a training epoch of three classification models with a large number of output in 8-GPU machines with and without NVLink (i.e., via PCIe). All three models achieve salient speedup with NVLink: ResNet-100k, the largest model, is accelerated by 10.5×.

6.2 Case Study of Low-GPU Tasks

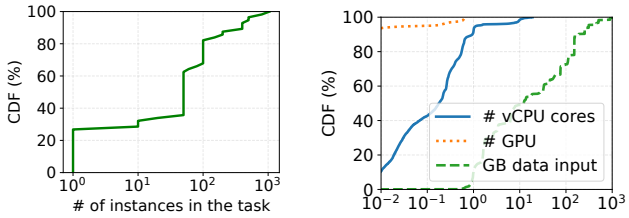
The majority of tasks running in our cluster have low GPU requests and usages (Section 3.3). To understand this somewhat unexpected result, we study three popular tasks. By profiling their executions, we find that they spend a considerable amount of time on CPUs for data processing (e.g., data fetching, feature extraction, sampling) and simulation (e.g., reinforcement learning), leaving GPUs under-utilized.

CTR prediction model training and inference. Among all tasks in the trace, over 6.7% are for advertisement click-through rate (CTR) prediction. These tasks use a variety of CTR models [30, 60, 73, 74], with around 25% instances performing training and the other 75% performing inference. Figure 15 shows the distributions of the CPU and GPU usages of these instances. Compared to training, inference instances have higher CPU utilization as they process a large volume of data continuously arriving. Both instances have low GPU utilization: over 75% instances use less than 0.1 GPUs.

We next profile the executions of three inference instances with DeepFM, DCN, and DNN models, respectively. Figure 16a shows the run-time breakdown of I/O, GPU, and CPU operations. The three instances spend around 80% runtime on CPUs to fetch and process the next input batch (IteratorGetNext in TensorFlow [20, 40]); GPU and I/O operations (e.g., MatMul, Sum, Cast, MEMCPYHtoD) only account for 10% of the execution time, respectively.

The high CPU usage of these instances makes them prone to interference from the co-located workload, especially in machines with high CPU utilization. To see this, we run training instances of a DeepFM model in containers with 8 vCPU cores. Together with an instance, we run some artificial load using spare cores of the host machine to create CPU stress. We configure varying load to control the level of stress. Figure 16b shows the instance training speed in a 48-core machine under varying stresses with 0 to 40 cores left idle (highest to no stress). Though the co-located load run on different vCPU cores not occupied by the instance, it still results in up to 28% slowdown of the training speed due to the contention of other shared resources, such as cache, power, and memory bandwidth [19, 21, 58].

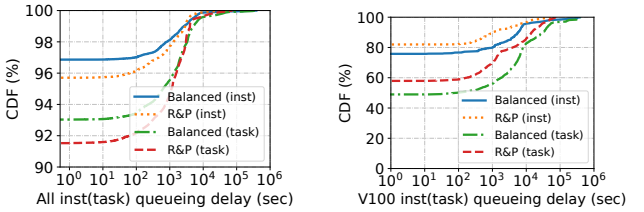
GNN training. Graph Neural Network (GNN) training comes as another popular computation, which accounts for 2% instances in our cluster, including GraphSage [31], Bipartite GraphSage [75], GAT [57], etc. Figure 17a shows the distribution of CPU and GPU usage of GNN training instances, where CPU is more heavily utilized than GPU. In production GNN models, a graph must undergo a sequence



(a) CDF of RL instances.

(b) CDF of resource usage.

Figure 18: Characterization of reinforcement learning instances.



(a) Queueing delays of all instances and tasks.

(b) Queueing delays of V100 instances and tasks.

Figure 19: Task queueing delays in simulation with load-balancing (Balanced) and reserving-and-packing (R&P).

of pre-processing, such as EdgeIteration, NeighborSampling, and NegativeSampling [75], before turning into an embedding (a computationally digestible format, usually vectors) of a deep neural network. Such graph pre-processing is currently cost-effective when performing on CPUs. As shown in Figure 17b, it accounts for 30–90% duration of each training iteration in different models.

Reinforcement learning. Our cluster also runs many reinforcement learning (RL) tasks. An RL algorithm iteratively generates a batch of data through parallel simulations on CPUs and performs training with the generated data on GPUs to improve the learning policy. Figure 18a shows that 72% RL tasks have at least 10 gang-scheduled instances, with the largest one running over 1,000 instances. Most RL instances are used to run simulations, eating up lots of CPUs and network bandwidth but only a small fraction of GPUs, as shown in Figure 18b. In fact, in the largest RL task, each instance requests only 0.05 GPUs.

6.3 Deployed Scheduling Policies

Compared to low-GPU tasks, high-GPU tasks have picky scheduling requirements and are usually run by business-critical applications. They are hence differentiated from other tasks and scheduled as first-class citizens.

Reserving-and-packing. In our cluster, the scheduler employs a *reserving-and-packing* policy. That is, it intentionally reserves high-end GPUs (e.g., V100/V100M32 with NVLinks) for high-GPU tasks, while packing the other workloads to machines with less advanced GPUs (e.g., T4 and Misc). Specifically, for each task, the scheduler characterizes its *computation efficiency* using a performance model that accounts for many task features, such as the degree of parallelism, the used ML model, the size of embedding [59, 64, 70], and the historical profiles of other similar tasks. Tasks with high computation efficiency larger than a certain threshold are identified as high-GPU.

For each task, the scheduler generates an ordered sequence of *allocation plans*; each plan specifies the intended GPU device and is associated with an attempt timeout value. The scheduler attempts allocation following the ordered plans: it waits for the availability of the intended GPU specified in the current plan until timeout, and then moves on to the next plan for another attempt. For high-GPU tasks, the allocations of high-end GPUs are attempted before the less advanced ones in the ordered plans; for other tasks, the order is reversed. Our GPU scheduler is implemented atop Fuxi [26, 71], a locality-tree based scheduling system.

Load-balancing. Given the potential resource contention and interference between co-located task instances (Section 6.2), maintaining load balancing across machines with similar specs is also important. Therefore, under reserving-and-packing, the scheduler also prioritizes instance scheduling to machines with low *allocation rate*, measured as a weighted sum of the allocated CPUs, memory, and GPUs normalized by the machine's capacity.

Benefits. Our scheduler prioritizes reserving-and-packing over load-balancing. To justify this design, we evaluate two scheduling policies using the simulator described in Section 5.2: ① simply load-balancing machines using progressive filling (always scheduling a task's instances to the least utilized node), and ② only performing reserving-and-packing without considering load balancing (R&P). We sample 100,000 tasks with over 500,000 gang-scheduled instances from the trace and feed them into the simulator. Figure 19a shows the CDF of the queueing delays of all instances and tasks under the two policies. Note that the queueing delay of a task is also the queueing delay of its gang-scheduled instances. Over 90% instances and tasks are launched immediately under the two policies. Compared to load-balancing, reserving-and-packing reduces the average task queueing by 45%, mostly attributed to the significant cutoff of the tail latency by over 10,000 seconds. Figure 19b further compares the queueing delays of business-critical tasks and instances requesting V100 GPUs under the two policies: reserving-and-packing reduces the average task queueing delay by 68%. The simulation results justify our design of prioritizing reserving-and-packing over load-balancing.

6.4 Open Challenges

However, our scheduler policy design is not without its problems, many of which remain open to address. We next discuss

Table 2: Mismatch between machine specs and instance requests, in terms of the provisioned/requested CPUs per GPU.

vCPU cores per GPU	All nodes	8-GPU nodes	2-GPU nodes
Machine specs	23.2	12.0	38.1
Instance requests	21.4	22.8	18.1

those open challenges, which we believe also stand in other GPU clusters with heterogeneous machines.

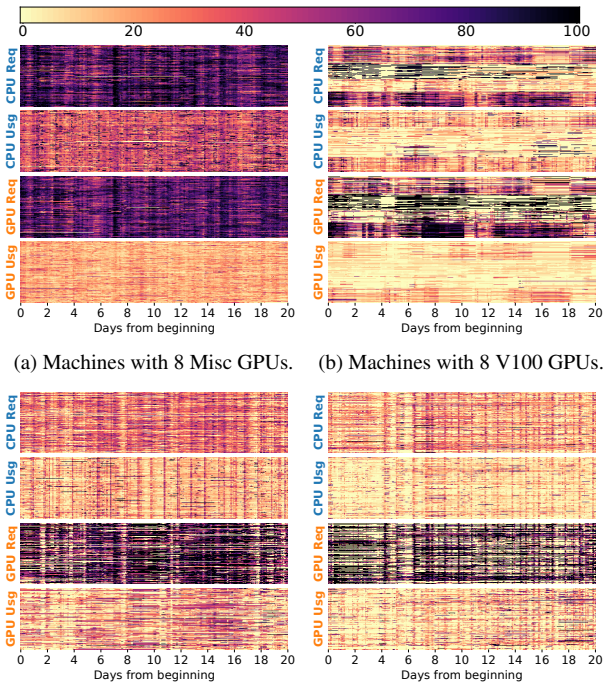
Mismatch between machine specs and instance requests. We observe a mismatch between machine specs and instance requests. Table 2 compares the average number of provisioned and requested vCPU cores per GPU in machines with 8 and 2 GPUs and their running instances. In 8-GPU machines, 12 vCPU cores are provisioned for each GPU. Yet, the instances running in those machines request 22.8 vCPU cores per GPU on average. On the other hand, CPUs in 2-GPU machines are over-provisioned, where the CPU-to-GPU ratio is more than twice of the instance requests.

To understand how the mismatch may affect the machine utilization, we randomly sample a number of nodes with different specs and depict the requests and usages of CPUs and GPUs in heatmaps shown in Figure 20, where each row corresponds to one machine, and all values are normalized to the machine's capacity. Compared to 8-GPU nodes, 2-GPU machines have substantially underutilized CPUs despite GPUs being heavily occupied. On average, P100 (T4) machines have 31% (20%) CPUs allocated with only 19% (10%) CPU utilization (Figures 20c and 20d).

We stress that the mismatch between machine specs and instance requests is not fundamental, as the cluster-wide CPU-to-GPU specs remains close to the overall instance requests (23.2 vs. 21.4 as shown in Table 2). We therefore believe that the mismatch can be avoided or at least mitigated by improved scheduling (e.g., rescheduling some high-CPU instances in 8-GPU machines to 2-GPU nodes).

Overcrowded weak-GPU machines. Compared to other machines, those with less advanced GPUs are overcrowded. The problem becomes even more salient in 8-GPU nodes (Misc GPUs) as shown in Figure 20a. On average, 77% CPUs and 74% GPUs are allocated in these machines. CPUs are better utilized than GPUs: the utilization of CPU is 43% on average, while the average utilization of GPU is 18%. This result is partly caused by our scheduling algorithm prioritizing weak-GPU machines for low-GPU tasks (Section 6.3), which account for a large instance population in our cluster.

Imbalanced load in high-end machines. Compared to other nodes, high-end machines with advanced V100 GPUs are less crowded (Figure 20b), with the average allocation ratios of CPUs and GPUs being 35% and 49%, respectively. These machines are usually reserved for a small number of important high-GPU tasks, thus suffering from low utilization. We also observe imbalanced load among V100 machines. In



(a) Machines with 8 Misc GPUs. (b) Machines with 8 V100 GPUs.

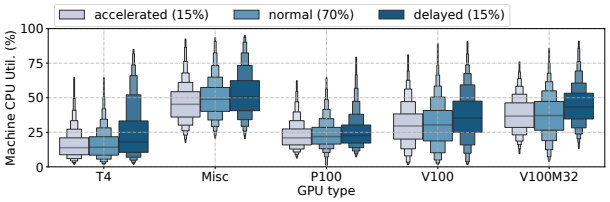
(c) Machines with 2 P100 GPUs. (d) Machines with 2 T4 GPUs.

Figure 20: Heatmap of requests and usages of CPU and GPU in machines with different specs. Each row corresponds to one machine.

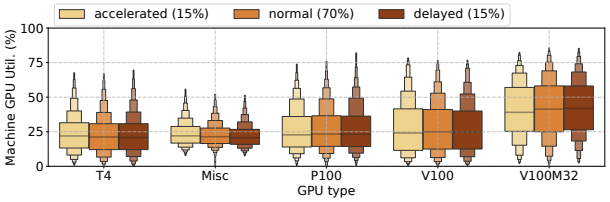
Figure 20b, the machines near the bottom are more crowded than the others. This suggests that the current load-balancing algorithm still has plenty room to improve (Section 6.3).

CPU can be the bottleneck. As shown in Section 6.2, a large number of ML tasks use CPUs more extensively than GPUs. These tasks are more likely to get slowdown in machines with high CPU contentions. To see this, we study the correlation between machine utilization and instance slowdown in the trace and depict the results in Figure 21. Our analysis focuses on the recurring tasks (Section 5.2). In each task recurrence, we divide the instances into three groups: 1) instances with *accelerated* execution whose duration is the shortest 15%, 2) *normal* execution whose duration is the middle 70%, and 3) *delayed* execution whose duration is the longest 15%. Figure 21a compares the CPU utilization in machines running accelerated, normal, and delayed instances. In general, machines running delayed instances measure higher CPU utilization than those running accelerated and normal instances. However, such correlation is not found on GPUs. As illustrated in Figure 21b, the distributions of GPU utilization show no substantial differences across machines running accelerated, normal, and delayed instances.

We next zoom in to the popular CTR prediction tasks with high CPU usage (Section 6.2). Figure 22 shows the CDF of CPU/GPU utilization in machines running accelerated and delayed instances, respectively. In machines with over 24% CPU utilization run 50% delayed instances but only 10%

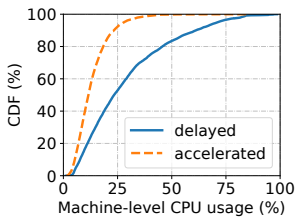


(a) CPU utilization of machines with various GPU types.

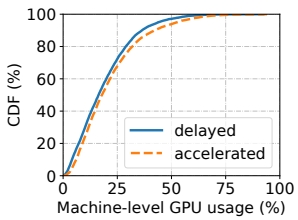


(b) GPU utilization of machines with various GPU types.

Figure 21: Correlation between machine utilization (CPU and GPU) and instance slowdown. Machines hosting delayed instances have higher CPU utilization than those hosting normal and accelerated ones. In contrast, such correlation is not found on GPUs. The boxes depict the $1/128$, $1/64$, ..., $1/4$, $1/2$, $3/4$, ..., $63/64$, $127/128$ quantile values [34, 61].



(a) CDF of machine CPU usage.



(b) CDF of machine GPU usage.

Figure 22: The impact of resource utilization to the execution of CTR prediction instances with high-CPU usages.

accelerated instances (Figure 22a), an evidence of strong correlation between CPU contention and instance slowdown. GPU contention, on the other hand, has no clear contribution to instance slowdown (Figure 22b).

To summarize, task instance scheduling in GPU clusters should also account for the potential interference caused by CPU contentions. This essentially calls for a multi-resource scheduler that jointly considers CPUs, GPUs, memory, I/O, and network when making scheduling decisions.

7 Discussion

Support of elastic scheduling. One fundamental challenge posed to GPU schedulers in heterogeneous clusters is the gang-scheduling requirement of distributed training. Some frameworks [6, 13] are hence developed to support elastic scheduling which allows a training job to dynamically adjust the number of workers on the fly. Compared to gang-scheduling jobs, elastic-scheduling jobs are easier to handle: they can start with a small amount of resources and later scale to more GPUs when the cluster becomes less crowded. However, elastic scheduling introduces non-determinism to final model accuracy [27, 63].

Machine provisioning and resource disaggregation. GPU schedulers should also account for machine provisioning: in our previous analysis, although 8-GPU machines provide abundant GPU processing power, 2-GPU machines can be a better fit to tasks with heavy CPU processing. To make the problem simplified, many system works propose to decompose monolithic machines into a number of distributed, disaggregated hardware components for improved hardware elasticity [53], despite the non-negligible communication overhead. TensorFlow has recently made a framework-level attempt towards this direction. It released an experimental data service [5] to decouple data pre-processing from GPU training so as to address the CPU bottleneck. However, it requires changing user's source code with non-trivial efforts.

8 Related Work

GPU sharing. GPU sharing can be supported at different levels. At the GPU hardware level, NVIDIA recently released the Multi-Instance GPU (MIG) [10] feature that enables partitioning a large GPU into multiple small GPU instances with isolated memory and bandwidth. However, MIG is only available on the latest A100 GPUs, and it does not support arbitrary GPU partition. At the GPU software level, GPU time-multiplexing can be implemented by intercepting CUDA APIs [24, 28, 54]. Yet, it usually introduces non-trivial context switching overhead and does not provide a good isolation between the co-located task instances. NVIDIA Multi-Process Service (MPS) [11] offers an alternative solution, but it cannot isolate failures among co-executed process. At the framework level, by extending standard ML frameworks such as TensorFlow and PyTorch, AntMan [66] and Salus [68] enable fine-grained GPU sharing and manage GPU memory for each task instance at a low cost. However, Salus requires users to adapt their code to the framework, while AntMan only supports training tasks.

GPU cluster scheduler. Many GPU cluster schedulers have been proposed recently (Table 1). Notably, Optimus [49] and Tiresias [29] schedule distributed training jobs with an objective of minimizing the average completion time; Themis [41], Gandiva$_{\text{fair}}$ [18], and HiveD [72] further consider completion-time fairness for the training jobs. All these works support no GPU sharing, with the minimum allocation unit being one GPU. The clusters used in evaluation are of limited size, workload diversity, and machine heterogeneity.

ML workload characterization. In addition to computation, communication and I/O are also important for distributed training and are thus the focus in the previous characterization

studies. For example, `tf.data` [45] reports that a majority of production ML workloads read many terabytes of data and spend a large proportion of time in data loading. Some ML schedulers [50, 52] study the training efficiency with different network bandwidth and propose to mitigate the communication overhead for accelerated training. An earlier characterization of ML training tasks in Alibaba PAI [59] suggests to replace the PS-Worker architecture with Ring AllReduce to better exploit the high-speed NVLink among GPUs. These works mainly focus on distributed training but leave aside the general MLaaS workloads and cluster resource management.

9 Conclusion

In this paper, we characterized a two-month production trace consisting of a mix of training and inference tasks in a large GPU cluster of Alibaba PAI. We made a number of observations. Notably, the majority of tasks have gang-scheduled instances and are executed recurrently. Most of them are small, requesting less than one GPU per instance, whereas a small number of business-critical tasks demand high-end GPUs interconnected by NVLinks in one machine. For those low-GPU tasks, CPU is often the bottleneck, which is used for data pre-processing and simulation. To better schedule the PAI workloads, our scheduler enables GPU sharing and employs a reserving-and-packing policy that differentiates the high-GPU tasks from the low-GPU ones. We also identified a few challenges that remain open to address, including load imbalance in heterogeneous machines and the potential CPU bottleneck. We have released the trace to facilitate future research on improved GPU scheduling.

10 Acknowledgment

We are deeply indebted to our shepherd John Wilkes, who has patiently gone through this work and helped shape the final version. We thank the anonymous reviewers of NSDI '22 for their valuable comments. We also thank colleagues from Alibaba Group, including Kingsum Chow, Yu Chen, Jianmei Guo, Guoyao Xu, Shiru Ren, Haiyang Ding, and many others, for their feedback and assistance in the early stage of this work. This work was supported in part by RGC GRF Grant 16213120 and the Alibaba Research Internship Program. Qizhen Weng was supported in part by the Hong Kong PhD Fellowship Scheme.

References

[1] Alibaba cluster trace program. https://github.com/alibaba/clusterdata, 2021.

[2] Alibaba machine learning platform for AI. https://www.alibabacloud.com/product/machine-learning, 2021.

[3] Amazon machine learning. https://docs.aws.amazon.com/machine-learning, 2021.

[4] Azure AI. https://azure.microsoft.com/en-us/overview/ai-platform/, 2021.

[5] Distributed tf.data service. https://github.com/tensorflow/community/blob/master/rfcs/20200113-tf-data-service.md, 2021.

[6] ElasticDL: A Kubernetes-native deep learning framework. https://github.com/sql-machine-learning/elasticdl, 2021.

[7] Google Cloud Vertex AI. https://cloud.google.com/vertex-ai, 2021.

[8] IBM Waston. https://www.ibm.com/watson, 2021.

[9] NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml, 2021.

[10] NVIDIA Multi-Instance GPU (MIG) user guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, 2021.

[11] NVIDIA Multi-Process Service (MPS). https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2021.

[12] NVIDIA NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/, 2021.

[13] TorchElastic. https://pytorch.org/elastic, 2021.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proc. USENIX OSDI*, 2016.

[15] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proc. ACM/IEEE SC*, 2017.

[16] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *Proc. USENIX ATC*, 2018.

[17] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[18] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proc. ACM EuroSys*, 2020.

[19] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: QoS-aware resource partitioning for multiple interactive services. In *Proc. ASPLOS*, 2019.

[20] Maxwell Collard. TensorFlow performance bottleneck on IteratorGetNext. https://stackoverflow.com/q/48715062, 2021.

[21] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ASPLOS*, 2013.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proc. IEEE CVPR*, 2009.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[24] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. HPCS*, 2010.

[25] Raj Dutt, Torkel Ödegaard, and Anthony Woods. Grafana: The open observability platform. https://grafana.com/, 2021.

[26] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. Scaling large production clusters with partitioned synchronization. In *Proc. USENIX ATC*, 2021.

[27] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[28] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. GaiaGPU: sharing GPUs in container clouds. In *ISPA/IUCC/BDCloud/SocialCom/SustainCom*, 2018.

[29] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proc. USENIX NSDI*, 2019.

[30] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247*, 2017.

[31] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[32] Tomoki Hayashi, Ryuichi Yamamoto, Katsuki Inoue, Takenori Yoshimura, Shinji Watanabe, Tomoki Toda, Kazuya Takeda, Yu Zhang, and Xu Tan. Espnet-TTS: Unified, reproducible, and integratable open source end-to-end text-to-speech toolkit. In *Proc. IEEE ICASSP*, 2020.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, pages 770–778, 2016.

[34] Heike Hofmann, Hadley Wickham, and Karen Kafadar. Value plots: Boxplots for large data. *J. Comput. Graph. Stat.*, 26(3):469–477, 2017.

[35] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.

[36] 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.

[37] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[38] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray RLlib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381*, 2017.

[39] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[40] Chi Keung Luk, Jose Americo Baiocchi Paredes, Russell Power, and Mehmet Deveci. Debugging correctness issues in training machine learning models, November 12 2020. US Patent App. 16/403,884.

[41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proc. USENIX NSDI*, 2020.

[42] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technol. J.*, 6(1), 2002.

[43] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proc. ICML*, 2016.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[45] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.

[46] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proc. USENIX OSDI*, 2020.

[47] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, 2019.

[49] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.

[50] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proc. ACM SOSP*, 2019.

[51] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. MLPerf inference benchmark. In *Proc. ACM/IEEE ISCA*, 2020.

[52] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *Proc. USENIX NSDI*, 2021.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proc. USENIX OSDI*, 2018.

[54] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.*, 61(6):804–816, 2011.

[55] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proc. IEEE CVPR*, 2016.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. NIPS*, 2017.

[57] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[58] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. Metis: learning to schedule long-running applications in shared container clusters at scale. In *Proc. ACM/IEEE SC*, 2020.

[59] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia. Characterizing deep learning training workloads on Alibaba-PAI. In *Proc. IEEE IISWC*, 2019.

[60] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proc. ACM ADKDD*, 2017.

[61] Michael L Waskom. Seaborn: statistical data visualization. *J. Open Source Softw.*, 6(60):3021, 2021.

[62] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. ESPnet: End-to-end speech processing toolkit. In *Proc. INTERSPEECH*, 2018.

[63] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proc. ACM SoCC*, 2016.

[64] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[65] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. USENIX OSDI*, 2018.

[66] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU cluster for deep learning. In *Proc. USENIX OSDI*, 2020.

[67] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. XLNet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[68] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.

[69] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

[70] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. AI Matrix: A deep learning benchmark for Alibaba data centers. *arXiv preprint arXiv:1909.10562*, 2019.

[71] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proc. VLDB Endowment*, 2014.

[72] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *Proc. USENIX OSDI*, 2020.

[73] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *Proc. AAAI*, 2019.

[74] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proc. ACM KDD*, 2018.

[75] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. AliGraph: a comprehensive graph neural network platform. In *Proc. VLDB Endowment*, 2019.