AntMan: Dynamic Scaling on GPU Clusters for Deep Learning

Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia, ***Alibaba Group***

https://www.usenix.org/conference/osdi20/presentation/xiao

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

AntMan: Dynamic Scaling on GPU Clusters for Deep Learning

Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou,*
Zhi Li, Yihui Feng, Wei Lin, Yangqing Jia

Alibaba Group

Abstract

Efficiently scheduling deep learning jobs on large-scale GPU clusters is crucial for job performance, system throughput, and hardware utilization. It is getting ever more challenging as deep learning workloads become more complex. This paper presents AntMan, a deep learning infrastructure that co-designs cluster schedulers with deep learning frameworks and has been deployed in production at Alibaba to manage tens of thousands of daily deep learning jobs across thousands of GPUs. AntMan accommodates the fluctuating resource demands of deep learning training jobs. As such, it utilizes the spare GPU resources to co-execute multiple jobs on a shared GPU. AntMan exploits unique characteristics of deep learning training to introduce dynamic scaling mechanisms for memory and computation within the deep learning frameworks. This allows fine-grained coordination between jobs and prevents job interference. Evaluations show that AntMan improves the overall GPU memory utilization by 42% and computation utilization by 34% in our multi-tenant cluster without compromising fairness, presenting a new approach to efficiently utilizing GPUs at scale.

1 Introduction

Over the past years we have witnessed the great success of Deep Learning (DL) with GPUs. DL already powers several widely-used products today, spreading across fields including computer vision, language understanding, speech recognition, recommendation, advertisement, etc. Therefore, it has become a vital workload integrated into the production pipeline at scale. Large companies often build multi-tenant GPU clusters for DL workloads, similar to shared clusters for big-data analytics.

At Alibaba, we have observed low utilization of GPU hardware in shared multi-tenant DL clusters, while queuing many jobs waiting for resources. Such low utilization of DL cluster arises from two main aspects. Firstly, most DL-production training jobs cannot fully utilize all the GPU resources throughout their execution. Training a DL model often requires a mixture of computations, some of which can hardly be parallelized using GPU, such as graph sampling in graph neural network [21, 54], feature extraction in advertisement [15, 23], data augmentation in computer vision [56], etc. Besides, when scaled to distributed training, 90% of the time can be spent on networking [32]. Secondly, the common reservation-based approach for cluster scheduling results in significant GPU idling because DL jobs often cannot consume partial resources. For example, stochastic gradient descent (SGD) is synchronous and requires all resources to be available simultaneously for gang-scheduling [27]. The cluster scheduler thus forces partially available resources to idle in reserve until the final request is satisfied.

Packing jobs on shared GPUs can boost GPU utilization and make the same cluster accomplish more jobs overall. However, this approach is rarely used in production clusters. The reason is that although improving GPU utilization is beneficial, it is also critical to guarantee the performance of important *resource-guarantee* jobs (*i.e.*, jobs with resource quota). Co-executing multiple jobs on the same GPU can result in interference, which leads to significant performance slowdown of the resource guarantee jobs [48]. What's more, the job packing strategy can introduce memory contention on concurrent jobs, which could even cause the failure of the training jobs if the resource demands of a job abruptly increase. Therefore, it is typical in existing production GPU clusters to perform exclusive allocation of resources on jobs [27].

We present AntMan, a DL system that improves GPU cluster utilization while ensuring fairness and performance of resource-guaranteed jobs by doing cooperative resource scaling to minimize job interference. New mechanisms are introduced in DL frameworks to allocate the exact required amount of GPU memory and computation unit dynamically during the job training. Any spare GPU resources, including GPU memory and compute cycles, could be leveraged by oversubscription jobs. AntMan co-designs the cluster scheduler and DL frameworks to adapt to the inherent fluctuating re-

*Co-first author

source characteristics in production jobs, through framework information aware scheduling, transparent memory extension, and fast continuous inter-job coordination. With this architecture, AntMan opens a space for policy design of co-executing DL jobs using GPU resources. In the GPU clusters of Alibaba, AntMan adopts a simple and practical strategy to maximize the cluster throughput. While providing performance guarantee on *resource-guarantee* jobs, AntMan dispatches *opportunistic* jobs to best-effort utilize GPU resources at a low-priority without any resource guarantees.

We have implemented AntMan by modifying two most popular DL frameworks, PyTorch [35] and TensorFlow [8], to expose necessary new primitives for the cluster scheduler to leverage at runtime. Our scheduling policy is implemented in a scheduler prototype on top of Kubernetes for evaluation, and the complete system is fully implemented in Fuxi [52], the internal scheduler of Alibaba, to serve the production DL jobs in the GPU clusters.

We evaluate AntMan on a 64 V100-GPU Kubernetes cluster to show the advantages of the new scheduling primitives and policies with micro-benchmarks and real workloads. The trace evaluation shows that AntMan can preserve the performance of resource-guarantee jobs ideally without preemption. Moreover, it improves the average Job Completion Time (JCT) of all jobs by up to 2.05x compared to current production cluster scheduler, and 1.84x compared to Gandiva [48], a state-of-the-art DL cluster scheduler. We also deploy AntMan in real production clusters and report the evaluations and statistics on a heterogeneous cluster with over 5000 GPUs. The cluster statistics shows that AntMan improves the overall throughput by offering up to 17.1% more GPUs for DL jobs, significantly reduces the average queuing delay by 2.05x, and raises the GPU memory and computation unit utilization by 42% and 34% respectively.

The key contributions of this paper are as follows.

- We investigate the comprehensive characteristics of production DL clusters to understand low utilization from three aspects: hardware, cluster scheduling, and job behavior (Section 2).

- We introduce two new dynamic scaling mechanisms in both memory and computation unit management for DL frameworks to address the challenges of GPU sharing. The new mechanisms leverage DL job characteristics to dynamically adjust the resource usage of DL jobs efficiently during the job execution (Section 3.1).

- Through co-designing the cluster scheduler and DL frameworks to utilize dynamic scaling mechanisms, we introduce a new industrial method to GPU sharing. This maintains the job service-level agreement (SLA) in a multi-tenant cluster while improving the cluster utilization with opportunistic scheduling (Section 3.2 and 3.3).

- By deploying AntMan in Alibaba to serve tens of thousands of daily jobs, we conduct experiments and report the performance improvement in a cluster with more than 5000 GPUs, demonstrating a productive approach in managing multi-tenant DL cluster fairly and efficiently at scale (Section 5).

2 Motivation

In this section, we start by introducing essential DL terminologies as the background. We then highlight our observations by characterising the GPU production cluster to motivate the design of AntMan. We end by discussing opportunities to leverage the DL training characteristics.

2.1 Deep Learning Training

Deep learning training often consists of millions of iterations, and each iteration processes a few samples, called a *mini-batch*. Usually, a training mini-batch can be divided into three phases. Firstly, samples and model weights are calculated to produce a set of scores, known as a *forward* pass. Secondly, a loss error is calculated between the produced scores and the desired ones using an objective function. The loss is then spread backwards through the model to compute gradients, called a *backward* pass. Finally, the gradients are scaled by a learning rate, as defined by an optimizer, to update the model parameters. The computation output of a forward pass usually includes many data outputs, each of which is called a *tensor*. These tensors should be temporarily held in the memory and consumed by the backward pass to calculate gradients. Usually, to monitor the model quality in training, evaluations are periodically triggered.

To train models with massive data, DL generally adopts data parallelism in multiple GPUs where each GPU is responsible for processing a subset of data in parallel while performing gradient synchronizations per mini-batch before the model update.

In large companies, multi-tenant clusters are commonly used to improve hardware utilization, where users can sometimes oversubscribe GPU resource quota, especially when GPU demands burst [33].

2.2 Characterizing Production DL Cluster

We study resource usage in production clusters from three perspectives: hardware, cluster scheduling, and job behavior.

Low utilization of in-use GPUs. Figure 1 illustrates a one-week statistic of GPU memory usage and computation unit utilization. The numbers are collected from one of the production clusters with thousands of heterogeneous GPUs. GPU memory consumption is normalized by the memory capacity of the running GPU due to the heterogeneity in the GPU

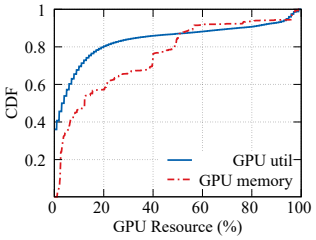


Figure 1: GPU resource statistic on a GPU production cluster.

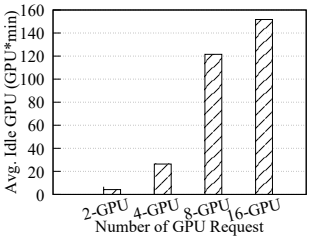


Figure 2: Average GPU idle waiting waste from gang-schedule.

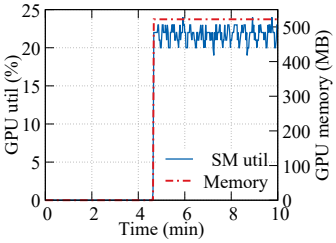


Figure 3: DeepFM on Criteo dataset.

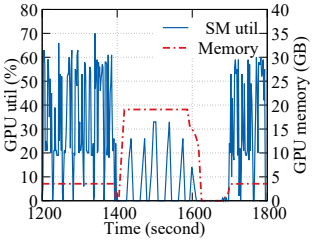


Figure 4: ESPnet on text-speech dataset.

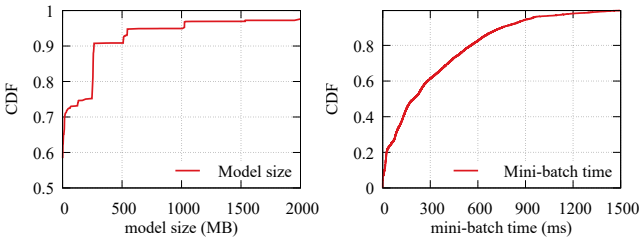
memory capacity. As shown in the figure, only 20% of the GPUs are running applications that consume more than half of the GPU memory. With regards to the usage of computation unit, only 10% of the GPUs achieve higher than 80% GPU utilization. This statistic indicates that both the GPU memory and computation units are not being fully utilized, and are thus wasting the expensive hardware resources.

Idle waiting for gang-schedule. To train deep learning with massive amounts of data, distributed multi-GPU training is essential. Multi-GPU training jobs require gang-scheduling, which means a job will not start training unless all required GPUs are simultaneously available [19, 27]. However, in a cluster, GPU resources can hardly be satisfied simultaneously. (*e.g.*, three GPUs might need to be held and then wait for the last one before launching a 4-GPU job, leaving the three GPUs in idle waiting mode). The more resources a job requires, the more GPU cycles are wasted when in idle waiting mode due to partial resource reservation. To understand the resource waste due to idle waiting, the timestamp of every resource grant for every gang-scheduled job was recorded. The idle waiting time of each GPU (*i.e.*, the gap between the job launching time and the resource granting time) is summed up to calculate the total resources wasted in idle waiting for a job. Figure 2 illustrates the average idle waiting resource waste for different sizes of jobs. The more GPUs a job requires, the higher the cost the cluster must pay for holding idle resources.

The unpredictable arrival of upcoming resources is the reason that reserved resources are left idle. A naïve approach to improving utilization is to launch other jobs on idle waiting resources. However, this can cause the large jobs to become starved and break the scheduling fairness. In addition, once all resources are satisfied, the burst GPU demand of this resource-guarantee job can lead to inter-job resource conflicts with the ones that are currently running in GPUs, which may cause the jobs to fail. Recently, elastic training (*e.g.*, TorchElastic [7]) is proposed to adapt to the incrementally available resources. However, it is rarely used in production because of the non-determinism it introduces to the accuracy [18, 47].

Dynamic resource demand. In addition to the idle wasting from job scheduling, our observation finds that DL jobs usually cannot fully utilize GPU resources during their life cycle. Figure 3 illustrates the first 10 minutes of resource usage when running DeepFM [20] on Criteo dataset. At the beginning, preprocessing on the dataset only requires CPU. However, both GPU Streaming Multiprocessor (SM) utilization and memory usage are boosted at 275 seconds. Such dynamic resource demands also commonly exist in other jobs. Figure 4 illustrates a 10-minute (1200∼1800 seconds) profiling on ESPnet [46], an end-to-end speech model training job. The model training pipeline could contain several phases. During the training phase, ESPnet consumes 3.6 GB GPU memory with a dynamic GPU SM utilization up to 70%. At 1400 seconds, decoding on GPU (around 1400∼1600 seconds) and synthesis (around 1600∼1700 seconds) on the CPU are issued in order to evaluate the model. It is worthy of note that, the decoding phase requires up to 19 GB GPU memory. After the evaluation phase, the model training continues. Such intra-job dynamic resource demand is common in production DL pipelines, making it hard to predict desired resources. We also find some jobs periodically become CPU bound, which is consistent with the observations in neural machine translation tasks [49]. We omit the result due to space limitation.

The dynamic resource demand actually conflicts with the fixed resource allocation and the potentially long running time in the training of deep learning jobs. Jobs requiring sufficient resources according to their peak usage make expensive hardware underutilized. If not granted sufficient resources, the job performance may be limited and thus the job completion time could be delayed. In addition, the memory caching design in existing DL frameworks (*e.g.*, TensorFlow and PyTorch) also conceal the temporal memory usage variations [50], which prevents GPU memory from potential sharing.



(a) Model size distribution. (b) Mini-batch time distribution.

Figure 5: One-week deep learning tasks statistic.

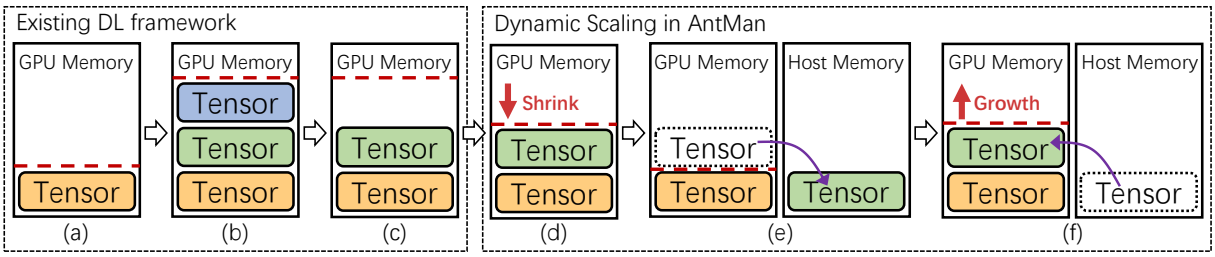


Figure 6: Dynamic scaling universal memory in AntMan

2.3 Opportunities in DL Uniqueness

The preceding characterization of the production DL cluster shows that low utilization is common for both GPU memory and GPU computation unit (*i.e.*, SM). It shows great opportunities to improve the cluster throughput with resource over-subscription. However, the unpredictable inter-job and intra-job demand burst introduces challenges to safe resource sharing. Jobs could run out of memory due to resource contention. Besides, in multi-tenant clusters, it is important to provide performance isolation for jobs holding a resource quota when the jobs are executed in a resource-sharing approach. To cater to these challenges when scheduling deep learning jobs, AntMan leverages the opportunities in the uniqueness of DL training.

We sample 10K tasks in a week of our production cluster to understand DL characteristics. We measure model size and mini-batch size during model training, both shown in Figure 5. Even though DL training could potentially use as much as 32 to 40 GB GPU memory (*e.g.*, V100 and A100), only a small portion is used to store the persistent DL model. 90% of DL models occupy only 500 MB GPU memory.[1] The majority of GPU memory is allocated and freed within the same mini-batch. Moreover, the DL training cycle is also rather small. As much as 80% of tasks consume a mini-batch within 600 *ms*.

We exploit such unique characteristics in several ways to schedule jobs on shared GPUs. Firstly, due to the small model size in common, the majority of GPU memory could be scheduled among the co-executing jobs. Secondly, mini-batch cycles are generally quite small, allowing fine-grained GPU memory and computation scheduling at every mini-batch boundary. This could further allow fast resource coordination between jobs. Thirdly, mini-batches apply mostly similar computations that can be utilized to profile the job performance, therefore their progress rate can be created as a performance metrics to quantify interference.

3 Design

AntMan deeply co-designs cluster schedulers and DL frameworks to address GPU sharing challenges. In this section, we first describe the new mechanism extensions in DL frameworks. We then introduce the collaborative scheduling design to leverage those new primitives. Finally, we present a new productive policy enabled in the cluster scheduler of Alibaba to manage DL jobs.

3.1 Dynamic Scaling in DL Frameworks

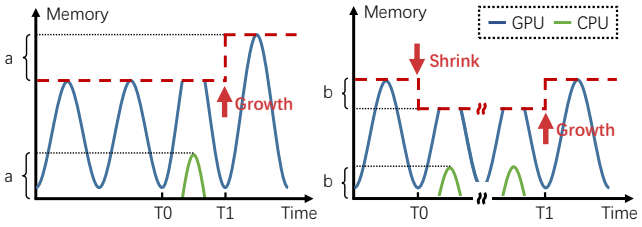
As mentioned in Section 2.2, DL training clusters exhibit low utilization due to unsaturated GPU usage in DL workloads and unique gang-schedule requirements during job scheduling, which contains great potentials that can be exploited to execute more jobs. However, some challenges need to be addressed, such as executing jobs at their minimal requirements while preventing GPU memory usage outbreak failures, adapting to the fluctuating computation unit usage while limiting potential interference. At its core, existing DL frameworks are designed for dedicated GPU executions, which lack key capabilities when collaborating with other jobs. Such conflicts between production DL cluster characteristics and DL framework limitation motivate the design of dynamic scaling mechanisms to enhance DL frameworks. The dynamic scaling mechanisms include the fine-grained dynamic control in two aspects, GPU memory and computation unit. We elaborate them next.

3.1.1 Memory Management

A dynamic memory management mechanism is introduced in AntMan to adapt the allocated memory on the fluctuating memory demands of a DL training job. This is achieved by allocating universal memory to DL application tensors, *i.e.*, switching tensors between GPU and CPU host machine DRAM across mini-batches. Modern operating systems support *paging* in memory management at the granularity of memory pages, where they use disk as memory when they run out of physical memory. AntMan adopts a similar approach, however, this is carried out in an application-specific granularity, tensor, which can be transparently migrated in universal memory addresses at runtime. In this way, DL frameworks can support the dynamic GPU memory upper limit.

Figure 6 illustrates the memory management in existing DL frameworks as well as the differences to AntMan. The total

[1] we omit the largest 2% jobs' model size as the number is business sensitive.



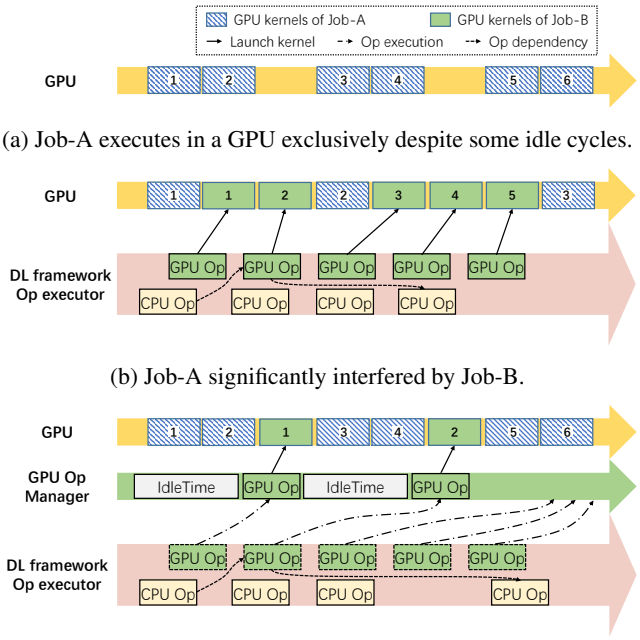
(a) Scaling for memory burst up. (b) Scaling to secure memory.

Figure 7: Leveraging mini-batch behavior to scale memory efficiently.

number of cached GPU memory size (*i.e.*, red dash line) increases with tensors created in DL frameworks (Figure 6a∼b). In order to eliminate the expensive overheads in memory allocations and de-allocations, and also to speed up training among mini-batches, the GPU memory is cached in a global memory allocator inside DL frameworks after tensors are destroyed. Prevalently, some tensors are used only in certain stages of DL training (*e.g.*, data preprocessing, evaluation), which are no longer required. However, this portion of cached GPU memory is not released (Figure 6c). This cached memory design in DL frameworks optimizes individual job performance at the cost of losing sharing potentials.

AntMan turns to the approach of scaling the GPU memory upper limit. It proactively detects in-used memory to shrink the cached memory to introspectively adjust GPU memory usage to an appropriate fit. This is done by monitoring application performance and memory requirements when processing mini-batches (Figure 6d). Furthermore, new primitives are provided to shrink the upper limit of GPU memory at runtime, even below the actual GPU memory demand of a job. AntMan uses its greatest effort to allocate tensors on GPU devices, however, tensors can be allocated outside of GPU with the host memory if GPU memory is still lacking (Figure 6e). With such universal memory support, jobs can continue to process even below their actual GPU memory requirements, where we find workloads slowdown the performance differently (Section 3.3). Tensors can be allocated back to GPU automatically when the GPU memory's upper limit increases (Figure 6f).

Paging in operating systems introduces costly page copy between the memory and disk. In contrast, thanks to the unique pattern of DL, tensor copy between the GPU and CPU host DRAM is explicitly avoided. Identical tensors are created across mini-batches, and therefore, AntMan exploits this pattern to adjust the upper limit of the memory at the boundary of the mini-batches. Figure 7a illustrates how memory scaling addresses the burst demand. At T_0, the memory requirement of a running DL training job increases, due to the limited upper-bound of GPU memory, some tensors cannot be placed in the GPU memory, and are instead created using the host memory. AntMan detects the usage of the host memory, and at T_1, it raises the GPU memory's upper limit for that job according to the usage of the host memory, which allows



(a) Job-A executes in a GPU exclusively despite some idle cycles.

(b) Job-A significantly interfered by Job-B.

(c) GpuOpManager of Job-B controls the interference.

Figure 8: Computation management to run two jobs in a shared GPU without interference.

the tensors to be fully allocated in the GPU device for the next mini-batch. Note that, the performance of this running job might slowdown in a mini-batch as tensors are placed in the host memory. However, such performance overheads are negligible, considering a typical DL training often requires millions of mini-batches. The overhead of memory shrinkage and growth is quantified in Section 5. Furthermore, AntMan provides fine-grained GPU memory scheduling at runtime. A training job might shrink to secure memory resources for other jobs, and grow back after other jobs are finished, as shown in Figure 7b. It illustrates that a DL job scales down at T_0 and scales up at T_1, at the cost of some tensors allocated on the host memory. Therefore, the usage of the remaining GPU memory between T_0 and T_1 for jobs running in the same shared GPU is secured.

3.1.2 Computation Management

Dynamic computation unit management is a mechanism introduced in AntMan to control the GPU utilization of a DL training job. Modern operating systems (*e.g.*, Linux) support cgroups, which limits, accounts for, and isolates the CPU resources that a process requires [1]. AntMan introduces a similar method of dynamically isolating the GPU computation resource access of DL-specific processes at runtime.

When multiple DL jobs are launched on the same GPU, the interference is mainly caused by the potential GPU kernel queuing delay and PCIe bus contention [14], which could

result in consistent performance downgrades across all jobs if packing jobs are running on the same model and configuration [48]. Our observation shows that jobs slowdown in different ways if different jobs are packed together (Section 5.1). This is because jobs have different capabilities at acquiring GPU computation units. Consequently, job performance can barely guarantee or predict in GPU sharing, resulting in difficulties on the deployment of GPU sharing for multi-tenant clusters. Figure 8 illustrates an example of GPU computation unit interference for two jobs that are executed on the same GPU. Figure 8a illustrates how Job-A executes on a GPU in a fine-grained manner. In short, GPU kernels will be placed in order and processed by the GPU computation unit one by one. Note that, in Figure 8, Job-A might not be able to fully saturate the GPU, resulting in idle GPU cycles and low GPU utilization which can potentially be used by other jobs. Therefore, Job-B is scheduled on this GPU (Figure 8b). The GPU operators of Job-B launch kernels (green blocks) executed in the GPU, which can fill it up, and thus delay the execution of other GPU kernels (blue blocks), leading to the poor performance of Job-A. The interference mainly comes from the lack of ability to control the execution frequency of GPU kernels. To address this issue, We introduce a GPU operator manager in DL framework(Figure 8c). Existing DL frameworks issue GPU kernels in the GPU operator once its control dependency is satisfied. In AntMan, the execution of GPU operator is dedicated to a newly-introduced module, called *GpuOpManager*. When a GPU operator is ready to execute, it is added to GpuOpManager instead of being directly launched. The main idea of GpuOpManager is to control the launching frequency by delaying the execution of GPU operators. In this way, AntMan introduces a new primitive to limit the GPU utilization of a DL training job using GpuOpManager. GpuOpManager continuously profiles the GPU operators execution time and simply distributes idle time slots before launching the GPU operators. Note that, GpuOpManager only delays the GPU kernel execution. Therefore, the potential dependencies among operators (including GPU operators and CPU operators) are retained, meaning that CPU operators can continue if possible. As illustrated in Figure 8c, the third CPU operator is not blocked, however, the fourth one is delayed as it depends on the second GPU operator, which has its execution delayed by the GpuOpManager.

3.2 Collaborative Scheduler

In this section, we describe how we co-design the cluster scheduler and DL frameworks to leverage the dynamic scaling mechanisms mentioned above for collaborative scheduling. We focus on the overall architecture of AntMan and how different modules operate. The detailed policy description is in the next section.

As shown in Figure 9, AntMan adopts a hierarchical architecture, where a global scheduler is responsible for job

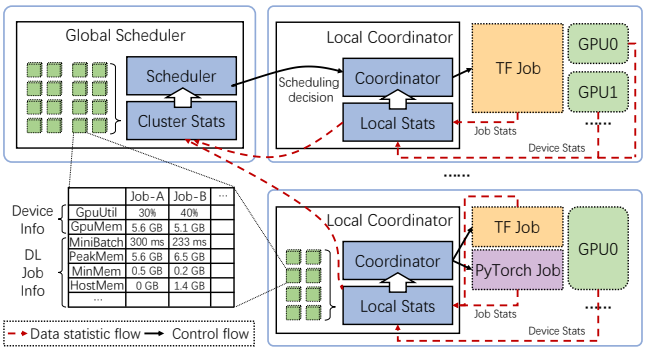


Figure 9: Collaborative scheduling workflow of AntMan.

scheduling. Each working server contains a local coordinator that is responsible for managing the job execution using the primitives of dynamic resource scaling through considering the statistics reported from DL frameworks. AntMan is designed for multi-tenant GPU clusters. In a multi-tenant cluster, each tenant usually owns certain resources, annotated as a resource quota (*i.e.*, number of GPUs), which is the concurrent performance guarantee resources that can be assigned to the jobs of that tenant. The sum of the GPU resource quota of each tenant is less equal to the total capacity of a GPU cluster. In AntMan, jobs are classified into *resource-guarantee* jobs and *opportunistic* jobs by global scheduler with different scheduling policies applied (Section 3.3). Resource-guarantee jobs consume a certain amount of GPU resources quota of their corresponding tenants while opportunistic jobs do not. Therefore, AntMan ensures that the performance of the resource-guarantee jobs should be consistent with that in exclusive executions.

In AntMan, similar to conventional cluster schedulers, the scheduling decision is dispatched from the global scheduler to the local coordinator. In addition, the local coordinator introspectively schedules the GPU resources to DL training jobs using the dynamic scaling mechanisms (Section 3.1). Therefore, the scheduling decisions can be treated as a top-down control flow. In contrast, data statistic flow information is collected by statistic modules of the local coordinator and aggregated on the cluster statistic module in a bottom-up approach to help make scheduling decisions, which is similar to Apollo [10]. Alongside with the hardware information (*e.g.*, GPU utilization, GPU memory usage), AntMan also leverages detailed job information reported by DL frameworks, including mini-batch duration, peak memory usage, minimal memory usage, and host memory consumption, etc. This information can also assist job scheduling decisions made by the global scheduler. For example, peak memory and minimal memory usage are used to indicate the GPU memory size that can be made available quickly. Mini-batch time shows how soon the GPU memory can be available for another DL training job, which can affect the scheduling decisions of the global scheduler when launching jobs.

Algorithm 1 scheduleJob(in job, out nodes)

```
1:  nodes0 ← findNodes(job.gpu, constraints ← job.topo)
2:  nodes1 ← findNodes(job.gpu, constraints ← M)
3:  nodes2 ← minLoadNodes(nodes1, job.gpu)
4:  if job.isResourceGuarantee:
5:      if numGPUs(nodes0) >= job.gpu:
6:          return nodes0
7:      else:
8:          reserve(nodes0)
9:  else:
10:     return nodes2
```

Once a job is launched on a GPU server, a local scheduler takes over the management of its end-to-end execution. Due to the load fluctuation of a DL training job, a local coordinator acts in an introspective mode to perform continual job control to DL frameworks. More specifically, it collects the statistics from the hardware and DL frameworks of all jobs, which is used to control job performance via resource usage adjustments (*e.g.*, shrink GPU memory) through the new primitives we introduced in Section 3.1.

3.3 Scheduling Policy

In this section, we first present the goal of our cluster scheduler. Then we describe the detailed policies applied in global scheduler and local coordinator. Finally, we introduce the job upgrade in our system.

Goal. There is an inherent tension between providing fairness (*e.g.*, to ensure SLAs of DL jobs with guaranteed resources) and achieving high resource utilization (*e.g.*, GPU utilization), because of the constant fluctuation in both the load on a cluster and the resource needs of a job. Prevalent production DL cluster schedulers often trade fairness in certain ways for efficiency. For example, spare resources are allocated to over-provision tenants. However, such GPU resources can hardly get back without preemption. Generally, preemption is rarely used as it fails running jobs while wastes expensive GPU cycles. Besides, [27] also reports the out-of-order behavior which discriminates large jobs (*i.e.*, allocating more GPUs), leading to unfairness by preferring small jobs. In AntMan, multi-tenant fairness is our primary goal, and the second priority is to improve the cluster efficiency therefore to achieve higher throughput. AntMan achieves fairness with the polices that are implemented in both the global scheduler and the local coordinator, powered by the dynamic scaling mechanisms. Furthermore, GPU opportunistic jobs are introduced in AntMan to steal idle cycles in GPUs so as to maximize cluster utilization.

Global scheduler. As a multi-tenant cluster scheduler, the global scheduler maintains multiple queues of tenants where jobs arrive and decides GPU locations allocated for jobs. For resource-guarantee jobs and opportunistic jobs, AntMan applies different scheduling polices as shown in Algorithm 1. *findNodes* is a function that returns the node and GPU candidates which satisfy the job request with an optional parameter to specify constraints. Global scheduler fairly allocates resource-guarantee jobs given sufficient GPU resources. In addition, resource-guarantee jobs are optimized to maximize the job performance using the free GPU resources, *i.e.*, GPUs that are not allocated to other resource-guarantee jobs (line 5-6). For instance, a distributed resource-guarantee job that uses all-reduce communication strategy (*e.g.*, NCCL [5]) can be scheduled on one server to utilize the NVLink [6] for high-performance communication. However, if the resource request of a job can partially be satisfied, the global scheduler reserves the resources for this job, and waits for others to meet the gang-scheduling requirement (line 7-8). Such insufficient resource reservation exists mainly for resource quota (*e.g.*, three GPUs left while there is a request for four) and resource fragmentation (*e.g.*, request four GPUs in the same server, however only four are available spread across servers). The reserved resources will never be occupied by other resource-guarantee jobs, however, they can be utilized by opportunistic jobs.

By default, the global scheduler will estimate the queuing time for jobs without GPU quota granted. Those jobs that suffer long queuing delay will be automatically executed as opportunistic jobs. To schedule opportunistic jobs, global scheduler aims to utilize free resources to the best of its ability. It allocates opportunistic jobs on GPUs by considering the actual GPU utilization, even when some other jobs run on those GPUs. Only GPUs with a utilization of less than M (set as 80% for now) in the past 10 seconds can be selected as candidates. AntMan adopts a heuristic strategy to allocate opportunistic jobs on the freest candidates (*i.e.*, *minLoadNodes*, line 9-10). In this way, there are some jobs allocated on the same GPU, where they are managed by the local coordinator. We will elaborate their coordinated execution next. Note that, although AntMan automatically selects opportunistic jobs by default, it also allows users to manually identify the job type at the point of submission; for example, as a resource-guarantee job explicitly to ensure SLAs. A job can also be specified as an opportunistic job that will never occupy the tenant's resource quota, and vice versa. In practice, users usually submit jobs in opportunistic mode to avoid the potential queuing delay, aiming to perform debugging and hyper-parameter tuning, which are both driven by early feedbacks [48, 51].

Local coordinator. The main responsibility of the local coordinator is to collaborate the execution of jobs on shared GPUs. Next, we first introduce how local coordinator ensures the performance of resource-guarantee jobs at shared execution. Then, we describe the approach to handle resource demand surges of a resource-guarantee job. Finally, we in-

troduce a greedy approach in AntMan to maximize the aggregated job performance when a GPU is only shared by opportunistic jobs. These approaches are achieved by utilizing the information reported from both GPU device and DL frameworks, and by instructing the memory management module (Section 3.1.1) and computation management module (Section 3.1.2) in DL frameworks.

A GPU is allocated to only one resource-guarantee job as it consumes GPU quota. However, in AntMan, it is possible that there are some opportunistic jobs executed on this GPU. As such, the local coordinator must prevent the resource-guarantee job from interfering by other co-located jobs at runtime. When a resource-guarantee job arrives on a GPU that runs with opportunistic jobs, the local coordinator first limits the opportunistic jobs in using GPU, for both GPU memory and GPU SM. By reducing the GPU usage of the opportunistic jobs, the newly launched resource-guarantee job will be capable of persistently initializing the training variables (*i.e.*, model) in the GPU memory. In addition, when launching a DL training job, the GPU device needs to be initialized by the DL framework, which takes more time if the GPU is in a high load. Once the resource-guarantee job is stably executed, the local coordinator will allocate the rest of the GPU memory to the opportunistic jobs. Furthermore, it gradually increases the GPU computation unit usage of opportunistic jobs without interfering with resource-guarantee jobs by monitoring the job performance (*i.e.*, mini-batch time). Similarly, when an opportunistic job arrives on a shared GPU, the local coordinator raises its GPU resource usage in a step-like fashion under the condition that the resource-guarantee job is not affected.

During the job execution, the resource demand of both the GPU memory and GPU computation unit might surge beyond the currently available resources (Section 2.2). To be aware of such dynamic resource demand, the local coordinator monitors the metrics that are reported by DL frameworks (*e.g.*, host memory usage, mini-batch time). Therefore, when a resource-guarantee job increases the GPU memory requirement, the tensors are temporarily stored using host memory, thanks to the universal memory (Section 3.1.1). The local coordinator shrinks the GPU memory usage of other opportunistic jobs and raises the GPU memory limit of the resource-guarantee job to recover its performance. It is similar for GPU computation unit usage coordination. Note that, AntMan relies on the application level metric (*i.e.*, mini-batch time) to indicate the job performance of resource-guarantee jobs. If it observes an unstable performance in the resource-guarantee job, it adopts a pessimistic strategy to limit the usage of GPU resources of other opportunistic jobs.

GPU resources can also be idle waiting without any resource-guarantee jobs (*e.g.*, due to gang-schedule as described in Section 2.2). In this case, if there is only one opportunistic job, the GPU resources can be fully utilized by this job without any constraints. Sometimes, it is possible that a GPU is occupied by multiple opportunistic jobs. Under this

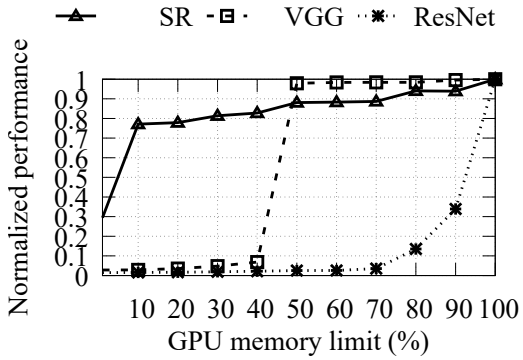


Figure 10: Workloads show diversity in performance sensitivity given insufficient memory.

scenarios, AntMan optimizes the aggregated job performance by maximizing GPU memory efficiency. With the dynamic scaling mechanisms enabled, we find that different workloads show differences in sensitivity regarding the performance slowdown from memory limitations. The peak memory usage of a job is limited using the dynamic memory scaling mechanism, and the host memory is thereby used for the remainder of the excess. As illustrated in Figure 10, Super Resolution (SR) model suffers only around 25% performance slowdown even with a 90% reduction in its device memory. VGG16 [43] model on Cifar10 dataset (VGG) can keep most of its original performance even after reducing its device memory by half. ResNet50 [22] on ImageNet dataset (ResNet) is sensitive to memory shrinkage; a 10% memory reduction introduces more than 60% slowdown. Therefore, when the total GPU memory demand of opportunistic jobs exceeds the GPU's memory capacity, AntMan adopts a simple heuristic approach which allocates GPU memory to the job that improves the normalized aggregated job performance at best. This is carried out via an introspective trial-and-error allocation.

Job upgrade. In AntMan, opportunistic jobs are executed at best-effort level to improve the cluster utilization. However, this is done without an SLA guarantee. The global scheduler upgrades these jobs given sufficient resources to complete them quickly. For distributed synchronous DL training, the partial upgrade does not help because the performance downgrade of a worker can be broadcast to the entire job. Thus, the global scheduler checks if all GPUs are filled up in opportunistic jobs. Once all task instances are ready to upgrade and the resource quota is sufficient, AntMan prefers to upgrade the opportunistic job rather than launch a new one. Global scheduler notifies local coordinator to tag it as a resource-guarantee job and consumes the tenant's GPU quota to accomplish the job upgrade.

4 Implementation

At Alibaba, DL training jobs are executed in Docker containers with our customized versions of DL frameworks. The APIs of the DL frameworks are compatible with the community version however with AntMan's features enhanced. A prototype custom cluster scheduler is implemented on Kubernetes [11] for evaluation. AntMan is fully implemented in our internal cluster scheduler, Fuxi [52], to serve the daily production training jobs on several clusters with thousands of GPUs each.

4.1 Deep Learning Framework

Dynamic scaling mechanisms are implemented in two popular deep learning frameworks, TensorFlow [8] and PyTorch [35], on versions v1.12 and v1.3.1 respectively. The implementation in TensorFlow takes 4000 lines of code (mostly in C++). The implementation in PyTorch takes about 2000 lines of code (500 lines in Python and 1500 lines in C++).

The modification of DL frameworks is mostly in three components: memory allocator, executor, and interfaces. As it adopts a similar implementation in both frameworks, we mainly use TensorFlow terminology to describe the details. To enable dynamic universal memory, `BFCAllocator` (`CUDACachingAllocator` in PyTorch) is modified to introduce an adjustable upper limit for memory. The memory allocator keeps track of the total bytes of memory allocation and triggers out-of-memory when total bytes exceed the upper limit. In addition, a new interface is introduced to the memory allocator to allow emptying of cached memory at any time. A new universal memory allocator, `UniversalAllocator`, is also added to wrap the GPU memory allocator and host memory allocator (*i.e.*, using `cudaHostMalloc` for memory allocation). When a memory allocation is triggered by the request of a tensor, UniversalAllocator tries to allocate the memory using the GPU memory allocator and treats the CPU memory allocator as a backup if there is insufficient GPU memory left over. Note that, the UniversalAllocator maintains a set data structure that records the pointers of memory regions allocated by GPU, which is used to classify the memory pointers for de-allocation.

To enable dynamic computation unit scaling, a `GpuOpManager` with an operator processing queue, which runs in a standalone thread, is introduced in DL frameworks. The operator executor of TensorFlow is modified accordingly to insert GPU operators to GpuOpManager queue in order so as to dedicate the execution of GPU operators to it. GpuOpManager may delay the actual execution of the GPU operators based on a limited percentage of the computation capacity.

The statistics of memory usage patterns and the execution information are aggregated for the local coordinator. The DL frameworks and local coordinator communicate through the file system. They both have a monitor thread to check the file for receiving either job statistics or control signals. To minimize the overhead of memory management, the dynamic scaling of memory is triggered at the mini-batch boundaries (end of `session.run()`).

4.2 Cluster Scheduler

A custom scheduler is implemented on Kubernetes [11] as a prototype to evaluate AntMan. The implementation requires around 2000 lines of code in Python. Overall, Kubernetes is responsible for cluster management and for executing jobs in Docker containers. Our global scheduler uses Python APIs to monitor the events in Kubernetes's API server for scheduling. Local coordinators are deployed as a DaemonSet in Kubernetes. Each coordinator monitors certain paths of the file system to collect the reported information for each job. The aggregated job and device information are stored in ETCD, a built-in distributed key-value store in Kubernetes. Therefore, global scheduler directly reads states in ETCD when making scheduling decisions.

AntMan has been fully implemented in Alibaba's internal cluster scheduler, Fuxi [52]. The implementation of global scheduler takes about 10000 LOC, including failover support and testing. The local coordinator implementation takes about 2000 LOC. Both of them are written in C++. The DL infrastructure is coupled with the big-data infrastructure, as DL jobs are part of the data pipeline. Fuxi adopts an architecture that optimizes for high performance scheduling, and it currently does not have ETCD. Global scheduler and local coordinator shall maintain their own aggregated device and job information and use RPC for communication.

5 Evaluation

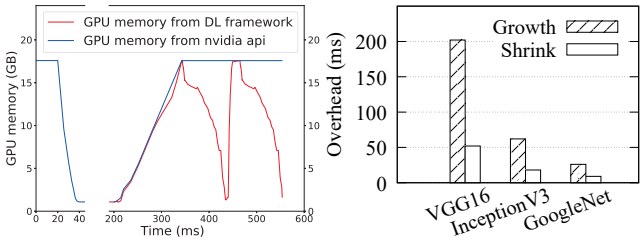
In this section, we first show micro-benchmark results to demonstrate the effectiveness and efficiency of AntMan mechanisms. We then evaluate the benefits of AntMan in a small cluster with 64 V100 GPUs to compare the policies with real workloads. Finally, we present the evaluation results on a production cluster with more than 5000 heterogeneous GPUs (V100 and P100). All the experiments are conducted on a cloud GPU cluster with 8 servers, unless explicitly stated. Every server is equipped with a 96-core Intel Xeon Platinum 8163 (Skylake) @2.50GHz with 736GB RAM, running CentOS 7.7. Each server has 8 NVIDIA V100 GPUs (32 GB GPU memory, with NVLink) powered by NVIDIA driver 418.87, CUDA 10.0, and CUDNN 7. The cloud GPU cluster is managed by Kubernetes; jobs are submitted through KubeFlow, and are executed in Docker containers. Only data-parallel is evaluated with synchronous training for jobs that require more than 1 GPU because they are common, although asynchronous training can also be supported. The trace in the experiment consists of 9 models, 2 of them implemented

	Model	Arrival	GpuMem	BS	Quota
Job-A	GCN	0 min	3.5 GB	1400	No
Job-B	ResNet	26 min	30.0 GB	360	Yes

Table 1: Setup and information of two jobs.

	Preempt	FIFO	Pack	UMem	AntMan
Job-A	Failed	43.0	43.1	43.4	43.9
Job-B	91.1	108.2	Failed	541.6	91.8

Table 2: Job status and JCT (min) of two jobs executing in different configurations.



(a) A shrink-growth profiling on ResNet-50.

(b) Overhead of GPU memory scaling for typical models.

Figure 11: Efficiency of GPU memory scaling in AntMan.

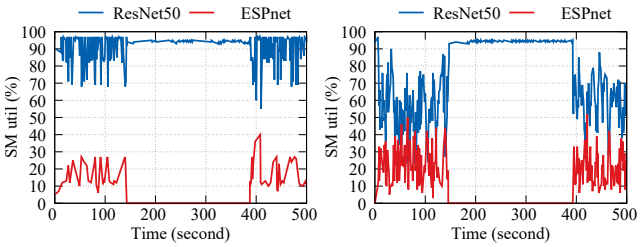
in PyTorch 1.3.1 and 7 of them implemented in TensorFlow 1.12.

5.1 Benchmark

In this section, we evaluate the dynamic scaling mechanism of AntMan in two aspects, memory and computation unit. We first demonstrate that dynamic memory scaling is indispensable in preventing failure and ensuring job performance. We then measure the efficiency of memory shrinkage and growth on typical workloads and detail the timeline on a ResNet-50 benchmark. Finally, we demonstrate the ability of dynamic computation unit scaling on avoiding job interference, by packing two jobs in a shared GPU.

Dynamic GPU memory scaling. To demonstrate that dynamic memory scaling is essential for sharing GPUs with multiple jobs, two typical jobs are chosen to construct a typical scenario. As shown in Table 1, Job-A is a GCN model that arrives at 0 minutes. Its peak GPU memory usage is 3.5 GB and is submitted by users without a resource quota. Job-B is a ResNet-50 task that arrives 26 minutes later. In total, it consumes 30 GB GPU memory and is submitted with a resource quota guarantee, which means it should run directly to meet the SLA requirements. The cluster has only one 32 GB GPU left and both jobs are scheduled on this GPU at arrival. Both jobs are run in the setup described above multiple times, but with different action policies when Job-B arrives. Table 2 shows the job status and job completion time (JCT) in minutes for both jobs with different configurations. At Job-B's arrival, the scheduler can choose to preempt Job-A. In this way, Job-B can be directly scheduled and finished in 91.1 minutes at the cost of Job-A's failure. The second choice is to run Job-B in a first-in-first-out (FIFO) mode. Job-B will not be launched until Job-A is finished, which introduces an extra 17.1-minute queuing delay. The third choice is to pack two jobs in the same GPU as proposed in Gandiva [48]. In this case, Job-B eventually fails because of the insufficient GPU memory (28.5 GB) granted. UMem indicates running Job-B in packing mode with the support of AntMan's universal memory, but without the coordinated scaling on the GPU memory limit (Section 3.1.1). Host memory are used when running out of GPU memory. Thus, Job-B will not fail from out-of-memory, however, it takes 514.6 minutes to finish and violates the SLA. AntMan leverages both universal memory and dynamic GPU memory scaling to coordinate job execution. It allocates sufficient device memory to Job-B as it runs with a resource quota, and offers the rest part of GPU memory to Job-A to allow it run as efficiently as possible. More specifically, when Job-B arrives, AntMan coordinates two jobs to shrink the GPU memory usage of Job-A and grow the GPU memory of Job-B. Job-B uses 30 GB GPU memory and Job-A uses the 2 GB left over, and 1.5 GB host memory. Note that, the performance of Job-B is still slightly slower compared to the preemptive scenario. This is because even though the required GPU memory is sufficient through dynamic scaling of AntMan, Job-B is still interfered in by the co-execution with Job-A in the computation unit.

Efficient memory shrinkage and growth. To demonstrate the efficiency of the dynamic memory scaling mechanism, a ResNet-50 job is run and the memory shrinkage and growth are manually triggered in order. As shown in Figure 11a, the performance is measured by monitoring the in-use GPU memory using both Nvidia API and memory statistics in DL frameworks. As Figure 11a indicates, the memory shrink from 17.6 GB to 1.3 GB takes only 17 *ms*. The GPU memory usage grows back to 17.6 GB in 143 *ms*, which is slower than the memory shrink. This is because GPU memory is allocated on demand with deep learning forward computation. Thus, the measured time includes both the forward computation time, which is essential to this mini-batch, and the memory allocation overhead. To understand the actual overhead, the time cost and memory usage of the next mini-batch are also plotted. The mini-batch with GPU memory growth takes 234 *ms* and the next mini-batch, which utilizes the cached memory, takes 119 *ms* to accomplish. Therefore, the growth overhead of ResNet-50 model is 115 *ms*. The same approach is applied to measure memory scaling overhead on other typical DL models. Figure 11b summarizes the overhead measured for VGG16 [43], Inception3 [45], and GoogleNet [44], which adjust GPU memory at a size of 17 GB, 16 GB, and 4 GB



(a) Packing mode.

(b) Adaptive computation adjustment mode.

Figure 12: The SM utilization rates of packing mode in Gandiva [48] and an adaptive computation adjustment mode in AntMan for a 500s segment of execution of ESPnet and ResNet-50.

respectively. Given a dynamic memory scaling interval of one minute, the largest overhead (*i.e.*, VGG16) is still negligible (only 0.4%).

Dynamic GPU computation unit scaling. To demonstrate the adaptive computation adjustment is essential for sharing GPU between multiple jobs, the SM utilization rates when running two typical jobs under packing mode and adaptive computation mode are characterized separately. As shown in Figure 12, the resource-guarantee job is an PyTorch job with ESPnet [46] model on the speech-text dataset. It co-executes with an opportunistic job which is a TensorFlow job with ResNet-50 [22] model on ImageNet [16]. Compared to ResNet-50, ESPnet consumes less SM and less memory. Therefore, packing these two jobs together into one GPU incurs a relatively higher GPU kernel queuing delay for the ESPnet and eventually leads to an SLA violation. Figure 12a illustrates that ESPnet is poor at competing GPU computation cycles compared to ResNet-50. The utilization of ESPnet remains mostly at 30% which is lower than in Figure 12b. ResNet-50 launches many more kernels per unit time than ESPnet, therefore, it consumes more GPU computation time. These results show that the end-to-end execution time of ESPnet increases dramatically from 20.1 minutes (when running on a dedicated GPU) to 105.2 minutes (when running together with ResNet-50).

Figure 12b illustrates that AntMan can leverage adaptive computation adjustment to utilize the left over resources as much as possible while still satisfying the SLA requirements. Specifically, AntMan introduces a feedback-based adjustment approach that continuously monitors the performance of resource guarantee jobs and uses performance feedbacks to adjust the GPU kernel launching frequency of opportunistic jobs. As shown in Figure 12b, the SM utilization rates of the training stage (the first 140 seconds) of ESPnet fluctuate between 5% and 50%. In this scenario, AntMan continuously adjusts the GPU kernel launching frequency of ResNet-50 to ensure the training performance of ESPnet. Therefore, the results reflected in this figure is that the SM utilization rates of ResNet-50 are constantly fluctuating between 30% to 90% within the first 140 seconds of execution. In contrast, the decoding stage (between 140 and 390 seconds) of ESPnet runs without consuming GPU computation cycles. Therefore, the SM utilization rates of ResNet-50 are relatively high at this stage As a result, by leveraging adaptive computation adjustments, the end-to-end execution time of ESPnet remains 20.8 minutes while ResNet-50 maintains 57% performance.

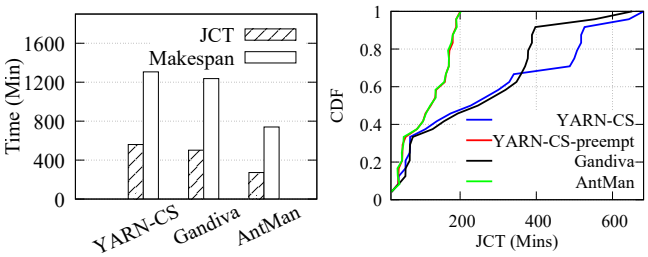
	Model	Type	Dataset
20%	ResNet-50 [22]	CV	ImageNet [16]
	VGG16 [43]	CV	Cifar10 [30]
	SuperResolution [42]	CV	BSD300 [34]
20%	Bert [17]	NLP	SQuAD [38]
20%	ESPnet [46]	Speech	Corp.Data
20%	GraphSAGE [21]	Rec.	PPI [55]
	GCN [29]	Rec.	Cora [41]
20%	DIN [53]	Ad.	Corp.Data
	Wide & Deep [15]	Ad.	Corp.Data

Table 3: Deep learning models and the ratios in the trace.

5.2 Trace Experiment

Workloads. Nine state-of-the-art deep learning models are selected from Github, together with open datasets, as summarized in Table 3. As the datasets of speech and advertisement are too small for evaluation, the internal datasets of Alibaba are used for the experiment. The models are classified into categories according to their application domains and they are evenly mixed up (20%). The job runtime of the trace is configured according to the distribution reported by Microsoft [48]. As a simplified multi-tenant setup, deep learning training jobs of the trace are randomly dispatched into two tenants. Tenant-A has 64-GPU quota and Tenant-B has no quota. Therefore, all Tenant-A's jobs are resource-guarantee jobs, and all jobs in Tenant-B are opportunistic jobs.

Baseline. The experiment compares AntMan to another GPU production cluster scheduler, Apache YARNs capacity scheduler (YARN-CS), which is used in Microsoft Philly [19, 28]. Gandiva [48], a state-of-the-art DL scheduling system, is also used for comparison. Gandiva introduces a series of primitives in DL for scheduling, including packing, migration, and time-slicing. The packing strategy of Gandiva is used in this experiment, which greedily schedules jobs to the GPUs with lowest GPU utilization and sufficient GPU memory. The migration and time-slicing proposed in Gandiva are to solve resource fragmentation and benefit AutoML, which are orthogonal to AntMan. Note that, Gandiva relies on job profiling information (*i.e.*, GPU utilization, GPU memory usage) for greedy packing decisions. Such profiling can hardly



(a) Comparison of YARN-CS, Gandiva, and AntMan.

(b) Job completion time of resource-guarantee jobs.

Figure 13: Trace experiment on 64 V100 GPUs.

be achieved in a production cluster, as its outputs might affect the successor tasks of DL pipeline. In the trace experiment, profiling information is unknown to both AntMan and YARN-CS.

Results. Figure 13a shows the average job completion time (JCT) and the makespan for the three schedulers when executing the same synthesized job trace in a cluster with 64 V100 GPUs. Compared to the capacity scheduler and Gandiva, AntMan improves average JCT by 2.05x and 1.84x. The total makespan is also reduced by 1.76x and 1.67x respectively. To understand the improvements brought about by AntMan, we config YARN-CS to run with preemption, which allows jobs in Tenant-A to preempt jobs in Tenant-B for execution. The JCT of resource-guarantee jobs (Tenant-A) are shown in Figure 13b. This shows the JCT of AntMan is almost the same as YARN-CS-preempt, however, YARN-CS-preempt achieves it with 46% of jobs being preempted. AntMan respects the jobs of Tenant-A and schedules them once their resource quota are satisfied, while conducting a performance control on the co-executing opportunistic jobs to avoid interference. Conversely, Gandiva delays the completion time of these jobs because of the lack of performance isolation and dynamic resource scaling.

5.3 Cluster Experiment

AntMan has been deployed on the production clusters of Alibaba to serve tens of thousands of daily deep learning training jobs. To verify the design and implementation of AntMan while ensuring it works properly, experiments and statistics are conducted on a heterogeneous GPU cluster with over 5000 GPUs.

To illustrate the cluster efficiency improvement provided by AntMan, one-week statistics were collected in December 2019, right before the deployment of AntMan, as the baseline. It is compared to the number collected in April 2020, after AntMan was fully deployed for weeks. However, as the jobs of these two weeks are different, the average JCT cannot be compared directly. Therefore, we focus on system metrics

	Avg.	90% tile	95% tile
Dec. 2019	1132	1978	5960
Apr. 2020	550	124	489

Table 4: One-week queuing delay statistic in seconds.

Interference	0%	0∼1%	1∼2%	2∼3%	3∼4%
# of jobs	9895	26	30	20	29

Table 5: Interference analysis on mini-batch time for 10K production jobs

comparison because the jobs of this cluster come from the same departments in Alibaba. The comparison shows that AntMan provides up to 17.1% extra GPUs for DL training jobs in this cluster. Hardware statistics show that AntMan achieves a 42% improvement on average for GPU memory usage and a 34% improvement on average for GPU utilization. Table 4 illustrates the queuing delay of jobs selected from a one-week period when roughly the same number of jobs arrive at the cluster. It illustrates that on average, the job queuing delay reduces by 2.05x and the tail latency significantly reduces by more than an order of magnitude, thanks to the cluster throughput improvement.

To measure the performance of resource-guarantee jobs in co-execution, 10000 jobs were randomly sampled from one week in April 2020 which both have the phases executing exclusively and co-executing with other jobs. For each job, the mini-batch time was recorded for both its dedicated execution and packing execution with other jobs. The mini-batch time difference between these two scenarios was calculated and any gaps larger than 10 *ms* were considered as interfered (10 *ms* is small enough to be considered as mini-batch fluctuation). In this way, the interference ratio for each job could be calculated. As shown in Table 5, 99% of the jobs suffer zero performance downgrades during job packing.

6 Related Work

GPU memory management. To optimize the limited and valued GPU memory for supporting larger batch-size DNN training, vDNN [39], Capuchin [36], CDMA [40], and Gist [26] adopts eviction, prefetching, and re-computation to reduce the GPU memory footprint, leveraging application-specific knowledge. Salus [50] packs multiple jobs in the same process to share the GPU memory management, however, with interference in co-execution. In addition, running multiple jobs in a process could potentially broadcast the failures, especially when given a significantly high failure ratio [27, 51]. AntMan provides a universal memory management design using dynamic GPU and CPU memory swapping at the granularity of tensors for the fluctuant load, which complements the memory swapping and re-computation policies.

Interference and performance isolation. Performance isolation is critical in modern operating systems and shared CPU clusters. Linux uses cgroups [1] to control the CPU and memory usage of a process. However, it rarely has support for general GPU applications. A series of research works, such as Quincy [25] and Entropy [24], optimize the job performance for fair sharing on CPU clusters. In AntMan, the characteristic of DL jobs is leveraged to provide fine-grained control on GPU memory and computation unit at runtime, which is similar to cgroups, but on an application level.

The interference issue of multiplexing jobs on a GPU has been well studied. Baymax [14] shares GPUs by mitigating queuing delay and PCIe contention. Prophet [13] tries to predict co-executed GPU workload performance using an analytical model. AntMan introduces an operator management module in the executor of the DL framework, leveraging the inherent periodical mini-batch iteration cycles as a metric for inter-job coordination. It controls the frequency of GPU kernel launches and resolves the contention in both the GPU computation unit and PCIe.

NVIDIA MPS can co-operate with multi-process CUDA applications in a GPU. MPS support is not production ready yet [4]. The resource limit cannot be changed at the runtime of a client process which violates the fluctuant characteristic. Moreover, MPS merges CUDA execution in only one context, resulting in the termination of all clients for any fatal GPU exceptions. rCUDA [37] and FlexDirect [3] of VMWare Bitfusion allow jobs to be remotely executed on a shared GPU.

GPU cluster scheduling Today, DL training jobs in multi-tenant production clusters are managed by infrastructures such as Kubernetes or YARN [9, 28], where jobs are allocated on dedicated GPUs, leading to common low utilization [27]. Gandiva [48] proposes time-slicing, migration, and packing to allow GPU sharing. Time-slicing and migration switch the GPU usage among jobs in coarse-grained, and therefore cannot improve GPU utilization. The packing approach proposed in Gandiva [48] could potentially introduce significant unpredictable resource contention, which violates the fairness requirements of a shared multi-tenant cluster. Themis [33] addresses the unfairness of placement-sensitive characteristic in DL jobs by proposing a long term fairness object. Gandiva$_{fair}$ [12] addresses the fairness issue of multi-size job time-slicing and proposes an automated trading mechanism. AlloX [31] efficiently and fairly schedules DL jobs in interchangeable resources by modelling the scheduling problem as a min-cost bipartite matching problem. AntMan introduces opportunistic DL jobs as low-priority jobs to best-effort utilize the GPU cycles, which is complementary to the fairness metrics and policies proposed above.

Elastic training. To utilize the idle GPUs introduced by gang-scheduling and to support fault-tolerance in DL training, TorchElastic [7] and ElasticDL [2] are designed to start training with any number of available GPUs. A common problem of these elastic DL frameworks is that the model training accuracy can hardly be guaranteed or reproduced, and are thus rarely used in production.

7 Conclusion

We present AntMan, a deep learning infrastructure deployed in the GPU production clusters of Alibaba. AntMan introduces dynamic scaling primitives in deep learning frameworks, allowing flexible fine-grained control of GPU resources for individual deep learning jobs at runtime. By utilizing the effective primitives mentioned above, AntMan co-designs cluster scheduler and deep learning frameworks for cooperative job management, allowing GPUs to be utilized by over-provision of opportunistic jobs at best-effort while avoiding the interference to other jobs. AntMan improves the overall GPU memory utilization and the computation unit utilization of Alibaba's GPU clusters by 42% and 34% respectively without compromising fairness.

Acknowledgements

We would like to thank our shepherd Roxana Geambasu and the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Chen Xing, Jin Ouyang, Xinyuan Li, Lixue Xia for their help in improving quality of writing.

References

[1] cgroups. https://en.wikipedia.org/wiki/Cgroups.

[2] ElasticDL. https://github.com/sql-machine-learning/elasticdl/.

[3] FlexDirect of VMware BitFusion. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/whitepaper/vmw-bitfusion-docs-flexdirect-whitepaper.pdf.

[4] MPS. https://github.com/NVIDIA/nvidia-docker/issues/419.

[5] NCCL. https://developer.nvidia.com/nccl/.

[6] NVLink. https://www.nvidia.com/en-us/data-center/nvlink/.

[7] TorchElastic. https://github.com/pytorch/elastic.

[8] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, volume 16, pages 265–283. USENIX Association, 2016.

[9] Scott Boag, Parijat Dube, Benjamin Herta, Waldemar Hummer, Vatche Ishakian, K Jayaram, Michael Kalantar, Vinod Muthusamy, Priya Nagpurkar, and Florian Rosenberg. Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs. In *Workshop on ML Systems, NIPS*, 2017.

[10] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.

[11] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14:70–93, 2016.

[12] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[13] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.

[14] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 681–696. ACM, 2016.

[15] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[18] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[19] 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 *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[20] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for CTR prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1725–1731. ijcai.org, 2017.

[21] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 1024–1034, 2017.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9, 2014.

[24] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009*, pages 41–50. ACM, 2009.

[25] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[26] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2018.

[27] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[28] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Tech. Rep.*, 2018.

[29] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[30] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[31] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[32] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training, 2020.

[33] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[34] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, volume 2, pages 416–423, July 2001.

[35] 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 *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[36] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[37] Javier Prades and Federico Silla. Gpu-job migration: The rcuda case. *IEEE Trans. Parallel Distrib. Syst.*, 30(12):2718–2729, 2019.

[38] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2383–2392. The Association for Computational Linguistics, 2016.

[39] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 18:1–18:13. IEEE Computer Society, 2016.

[40] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

[41] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29(3):93–106, 2008.

[42] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video

super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1874–1883, 2016.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[44] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[45] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[46] 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 *Interspeech*, pages 2207–2211, 2018.

[47] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter R. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 84–97. ACM, 2016.

[48] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[49] Wencong Xiao, Zhenhua Han, Hanyu Zhao, Xuan Peng, Quanlu Zhang, Fan Yang, and Lidong Zhou. Scheduling CPU for gpu-based deep learning jobs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, page 503. ACM, 2018.

[50] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

[51] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An empirical study on program failures of deep learning jobs. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20, pages 1159–1170, NY, USA, 2020. Association for Computing Machinery.

[52] 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 *Proceedings of the VLDB Endowment*, volume 7, pages 1393–1404. VLDB Endowment Inc., 2014.

[53] 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 *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1059–1068, 2018.

[54] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.

[55] Marinka Zitnik and Jure Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, 2017.

[56] Barret Zoph, Ekin D. Cubuk, Golnaz Ghiasi, Tsung-Yi Lin, Jonathon Shlens, and Quoc V. Le. Learning data augmentation strategies for object detection. *CoRR*, abs/1906.11172, 2019.