Lyra: Elastic Scheduling for Deep Learning Clusters

Jiamin Li[*]
City University of Hong Kong

Hong Xu
The Chinese University of Hong Kong

Yibo Zhu
Google

Zherui Liu
ByteDance Inc.

Chuanxiong Guo
Unaffiliated

Cong Wang
City University of Hong Kong

Abstract

Organizations often build separate training and inference clusters for deep learning, and use separate schedulers to manage them. This leads to problems for both: inference clusters have low utilization when the traffic load is low; training jobs often experience long queuing due to a lack of resources. We introduce Lyra, a new cluster scheduler to address these problems. Lyra introduces capacity loaning to loan idle inference servers for training jobs. It further exploits elastic scaling that scales a training job's resource allocation to better utilize loaned servers. Capacity loaning and elastic scaling create new challenges to cluster management. When the loaned servers need to be returned, we need to minimize job preemptions; when more GPUs become available, we need to allocate them to elastic jobs and minimize the job completion time (JCT). Lyra addresses these combinatorial problems with principled heuristics. It introduces the notion of server preemption cost, which it greedily reduces during server reclaiming. It further relies on the JCT reduction value defined for each additional worker of an elastic job to solve the scheduling problem as a multiple-choice knapsack problem. Prototype implementation on a 64-GPU testbed and large-scale simulation with 15-day traces of over 50,000 production jobs show that Lyra brings 1.53x and 1.48x reductions in average queuing time and JCT, and improves cluster usage by up to 25%.

CCS Concepts: • **Computer systems organization** → **Cloud computing**;

Keywords: GPU cluster scheduling, deep learning

[*]Work done at ByteDance Inc.

ACM Reference Format:
Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic Scheduling for Deep Learning Clusters. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3587445

1 Introduction

Recently, Deep Neural Networks (DNNs) have seen wild successes in many applications [25]. Hyperscale online service providers have adopted DNN, and build large-scale GPU clusters to accelerate DNN workloads for both training and inference. GPU cluster scheduling is a fundamental and critical task to utilize the expensive infrastructure efficiently, by optimizing job resource allocation and task placement.

It is common practice today to separately build and manage two types of GPU clusters, one for training and one for inference. This is because, for the same model, inference requires less computation and GPU memory than training and is less likely to utilize the numerous cores of training GPU [7, 35, 38]. Inference clusters usually use weaker GPUs, like Nvidia T4, with a fraction of the resources of the training GPUs, such as Nvidia V100 and A100.

This separation creates problems for both sides (§2). Our observations are based on experiences of operating production clusters with O(10k) GPUs for training and even more for inference. Specifically, inference cluster utilization is usually low (<40%) for an extended period of time due to the diurnal traffic pattern. At the same time, training jobs experience long queuing time before they can start, with an average of over 3,000s and 95%ile of almost 10,000s as seen from a 15-day trace with over 50,000 jobs. The long queuing time is due to both the high cluster utilization and the GPU resource fragmentation.

To address these problems, we propose *capacity loaning* to allow the inference cluster to loan the idle GPU servers during low-traffic periods to run training jobs, and reclaim them back when inference workloads increase again (§2.1). Capacity loaning mitigates both the utilization problem for inference and queuing problem for training. It is feasible for training jobs that do not have strict requirements on GPU type. Then to ensure the on-loan servers are rapidly utilized by training jobs when they become available, we draw inspiration from *elastic scaling* [16, 36, 42] (§2.2). Elastic scaling enables a running job to scale out or scale in to better utilize the dynamically changing resource pool. It also helps reduce queuing delay since an elastic job can start with a small number of workers first and increase its workers when more resources become available.

Capacity loading and elastic scaling create new degrees of freedom for cluster scheduling. As we navigate the new design space, we meet several new challenges that must be addressed before we can reap the benefits.

First, though loaning decisions can be solely made by the inference cluster scheduler to ensure inference workloads are not affected, reclaiming is more intricate. When the inference cluster needs to reclaim on-loan resources, the training scheduler has to preempt all running jobs on those servers. Given the high overhead of preemption and prolonged running time to the jobs, the scheduler must carefully select the servers in order to minimize the total preemptions.

Second, job scheduling is inherently more complicated with elastic scaling. Resource allocation has to consider a mix of inelastic jobs with fixed demand and elastic jobs with *variable* demand. We show that classical scheduling policies such as shortest job first (SJF) no longer work well with elasticity, and finding the JCT-optimal solution for merely two jobs is difficult. Given the allocation results, the scheduler still needs to determine the worker-server placement to minimize fragmentation, where servers are now heterogeneous with different GPUs because of capacity loaning.

Our key insight in solving these challenges is to prioritize the minimum resources needed by a job over its elastic demand, and to prioritize the dedicated training servers over the on-loan inference servers. This makes sense because the minimum demand of an elastic job is equivalent to an inelastic job to which not allocating resources is detrimental, but the elastic part can be fulfilled later without stalling the job.

Our solution, therefore, exhibits a two-phase structure following the above insight. For reclaiming, we first kill the elastic workers running on on-loan servers since stopping them does not lead to any job-level preemption. When preemption becomes inevitable, we characterize the problem as a knapsack problem with dependent item values [32] and develop an efficient heuristic to solve it (§4).

For resource allocation, we first allocate for both inelastic jobs and elastic jobs' base demand, with the aim of launching as many jobs as possible. We then scale out the scheduled elastic jobs if resources permit. The first phase can be solved using SJF to reduce queuing time and the second phase is formulated as a multiple-choice knapsack problem [48] to minimize running time, which in practice can often be solved using dynamic programming (§5.2). We then tackle the placement problem by placing the inelastic jobs on training servers, and elastic jobs on on-loan servers as much as feasible. Jobs are ordered based on the best-fit-decreasing policy to address the bin packing nature [6] and minimize fragmentation (§5.3).

Putting everything together, we design (§3–§5), implement (§6), and evaluate (§7) Lyra, a new cluster scheduler that realizes capacity loaning with elastic scaling. Lyra has an orchestrator that manages capacity loaning by executing instructions from the inference scheduler on when and how much to loan or reclaim, and by deciding which on-loan servers to return for reclaiming. Then a job scheduler periodically determines allocation and placement, and scales new and existing elastic jobs in response to resource and job dynamics. To be pragmatic, Lyra considers elastic scaling only for large DNNs whose training throughput scales well in our experiments.

The results of Lyra are promising (§7). We build a high-fidelity simulator, and replay a 15-day job trace collected from 3,544 training GPUs and 4,160 inference GPUs. We find that compared to a FIFO scheduler, Lyra can reduce the average and 95%ile JCT by up to 1.48x and 1.47x, respectively, and improve GPU usage by 25%. In terms of job scheduling, Lyra also outperforms state-of-the-art Pollux by 1.28x and 1.27x in median JCT and 95%ile JCT when elastic jobs occupy 36% training resources.

We summarize our contributions as follows.

- With production traces, we report the problem of separate management of training and inference clusters, i.e. low utilization in the inference cluster and long queuing time in the training cluster.
- We propose cluster-level capacity loaning and job-level elastic scaling, two new control knobs for cluster scheduling to address the above problems.
- We study the resulting cluster scheduling problems, develop a key insight to prioritize the minimum resources needed by each job to address elasticity, and use a principled approach to characterize and solve each problem.
- We design and implement Lyra, a novel cluster scheduler that integrates our solutions. Lyra works with existing resource management frameworks and is ready for deployment. Evaluation using testbed experiments and large-scale simulations validates its effectiveness.

2 Motivation

We start by presenting our motivation of Lyra.

2.1 Why Capacity Loaning?

Large GPU clusters are built to accommodate inference and training workloads with distinct requirements. Customer-facing inference jobs are latency-sensitive [7, 38]. Training jobs are much more resource-heavy and run for extended periods of time. Thus they emphasize completion times instead. Operators usually deploy separate clusters with different GPUs for training and inference, and manage them independently to minimize interference. Our production environment, for example, mainly uses Tesla V100 in the training cluster and T4 in the inference cluster. Job traces show that this practice leads to low utilization of inference resources and sub-optimal performance for training jobs.

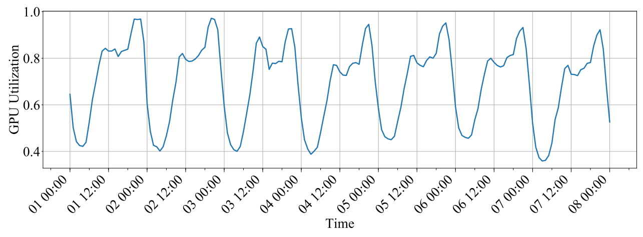


Figure 1. Inference cluster GPU utilization, i.e. fraction of GPUs serving at least one request in our inference cluster. The measurement spans one week from Oct 1 to Oct 7, 2020. The cluster has about 4,000 GPUs. The utilization changes from 42% in bottom hours to 95% in peak hours.

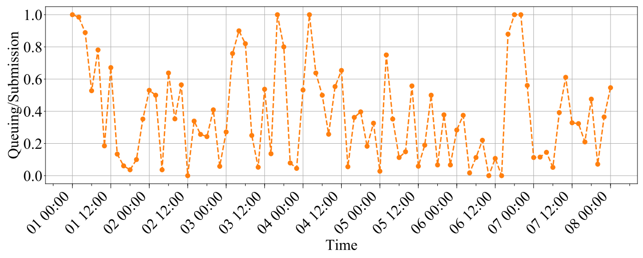


Figure 2. The fraction of queuing jobs among all the newly-submitted jobs in each hour in our training cluster for one week. A job suffers queuing time when the scheduler fails to satisfy its resource demand on the first try. If the ratio is high, it means that most of the jobs submitted in that hour are queued. The cluster has ~3,500 GPUs, and the average utilization is 82%.

Inefficient inference cluster utilization. Similar to other web services [27], the inference cluster is overprovisioned in order to handle the peak traffic. Inevitably, its resources are often underutilized due to the dynamic inference requests generated by customers.

We plot the GPU utilization in one of our inference clusters with 5-minute intervals for one week's time in Figure 1. Utilization is defined as the fraction of GPUs occupied by at least one inference job. We observe a clear diurnal pattern: peak traffic lasts about four hours at night, and demand trough occurs before dawn. The peak-to-trough ratio is ~2.2 within a day, and the average utilization is ~65%, both implying that there are abundant resources to be exploited in the inference cluster for extended periods of time.

Long queuing time for training jobs. Turning to the training cluster, a salient observation we make is that many training jobs experience long queuing before they can be dispatched with enough resources. Figure 2 depicts the hourly queuing job ratio in our training cluster for the same week as in Figure 1. A significant fraction of jobs (as high as 100%) still has to wait for resources from time to time. The average queuing time is longer than 3,000 seconds and certainly non-negligible.

The long queuing time is not only due to lack of resources. In fact, the average GPU utilization across the same period of time is 82%, which means there are often idle GPUs. The dynamic training demand certainly also contributes to the long queuing time. In addition, training demand does not exhibit clear patterns for prediction.

Capacity loaning. We propose to exploit the unused inference resources to run training jobs temporarily, i.e. loaning inference capacity for training. It mitigates both problems above at the same time: The inference cluster is better utilized, and training jobs have more resources to help reduce queuing time. The on-loan capacity can be reclaimed dynamically in case the inference traffic spikes to ensure the quality of service.

Though training jobs typically request specific GPUs, we find that up to 21% of jobs in our production traces do not do so. These *fungible* workloads can work with any GPU types in different execution runs. Lyra can launch these jobs on the loaned inference servers rather than waiting for training servers. To ensure feasibility, we may need to adjust the local batch size of the training job so that the models and the intermediate data can fit into the smaller inference GPU memory. We increase the number of workers so that the global batch size does not change to ensure the same model performance. This is straightforward since we know the GPU memory differences.

Another more aggressive way to exploit the loaned servers is to run a training job on heterogeneous GPUs, i.e. run on both training and inference GPUs (e.g. V100 and T4) at the same time. Heterogeneous training further improves scheduling flexibility with more potential gains. However, it requires delicate systems and algorithm support to work well, since the workers have to adopt different hyperparameter settings and inherently progress at different paces [4, 33, 39, 58]. Given that heterogeneous training remains an active research topic, our production training system only provides experimental support for it at the moment. Lyra's design does not depend on it, and we evaluate its effect in §7.2 when it is enabled for a small fraction of our jobs with a performance penalty.

2.2 Elastic Scaling for the Full Potential

To better cope with the constantly changing cluster capacity and further exploit the loaned inference resources, Lyra considers *elastic scaling*. Recently, elastic scaling has been introduced into ML frameworks [16, 36, 42] where a job can take a variable number of workers according to resource availability. One can even adjust the number of workers on-the-fly when the job is running.

Elastic scaling can greatly facilitate capacity loaning. With additional resources, training jobs can dynamically scale out to use more workers with more inference GPUs to accelerate training (provided they are running on inference GPUs already). When the cluster experiences high loads, some jobs could scale in to free some servers. In addition, when vacating the inference servers so they can be returned, the scaling-in operation reduces the need to completely preempt the jobs which incurs high overheads with checkpointing, re-launching containers, etc.

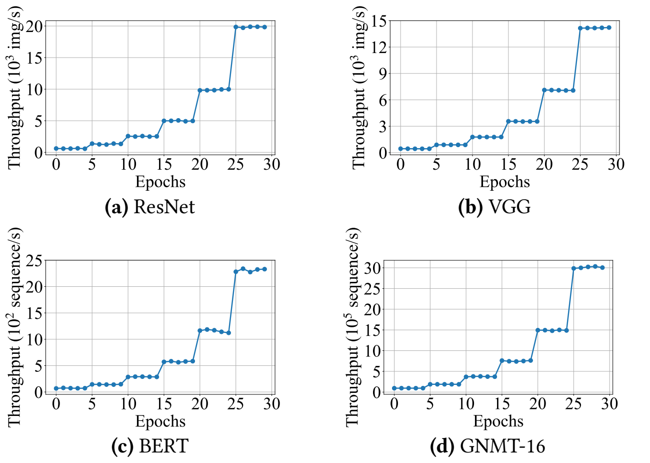


Figure 3. Throughput of four elastic training jobs using Tesla V100 GPUs. The workers are doubled every five epochs, starting from 1 worker. In our testbed, each server hosts 8 GPUs connected by NVLink. Servers use 100G InfiniBand interconnects. Each worker container uses 2 GPUs.

An acute reader might be wondering about the feasibility and benefit of elastic scaling in general. Indeed, besides the scalability issue of distributed training systems [18, 20, 41, 57], when we change the number of workers on-the-fly, the training hyperparameters may have to be updated as well. This can be fairly complex: for example, simply fixing the local batch size and linearly increasing the global batch size may impede the model convergence [11].

Thus in Lyra, elastic scaling is only adopted for jobs that scale well to the number of workers without updating the local batch size. Existing studies [1, 5, 19, 26, 29, 30, 45, 59] show that certain models like ResNet [15] and BERT [8] satisfy this requirement. We also find that, as shown in Figure 3, ResNet-50 [15], VGG16 [47], BERT [8], and GNMT-16 [54] all enjoy good throughput scalability and are well-suited for elastic scaling. Our traces reveal that the large jobs (~5% of all jobs) from these model families account for 36% of training cluster resources with an average running time of 14.2 hours, suggesting ample potential gains using Lyra. For these jobs, Lyra also restricts itself to *limited elasticity* where the worker number varies within a range.

2.3 Existing Cluster Schedulers

Much prior work exists on GPU cluster scheduling amid the proliferation of DL workloads. Lyra differs in two aspects.

First, capacity loaning represents a new angle to cluster scheduling that few have studied. Though the shared infrastructure is exploited by recent systems [28, 49, 50, 52, 53, 62], their focus is to schedule different types of workloads in a single cluster. Lyra instead focuses on virtually loaning resources between two different clusters. Specifically, it considers the problem of how to reclaim the transient on-loan resources while minimizing its negative impact on training jobs running on them (§4), which has not been considered

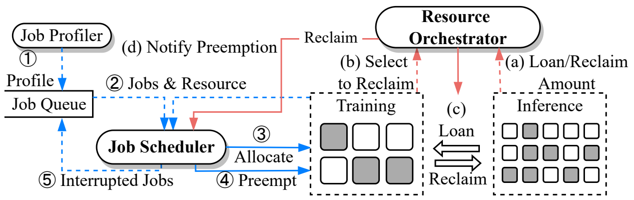


Figure 4. Lyra system architecture. Solid lines indicate control flow and dashed ones data flow. Red lines represent capacity loaning workflow, while blue ones elastic scaling workflow. Each square represents a GPU server; the gray ones are in use.

before. Further, Lyra takes advantage of the elasticity of training jobs to better utilize the dynamic cluster resources.

Second, some recent studies also considered scheduling elastic jobs. Gandiva [55] adopts an opportunistic approach to grow or shrink the number of GPUs used by a job without considering cluster-wide efficiency. AFS [18] greedily prioritizes jobs with the highest marginal throughput gain per GPU. Pollux [43] co-optimizes resource allocation and training hyperparameters to achieve high resource efficiency.

Compared to them, Lyra exploits the interplay between elastic scaling and capacity loaning to further improve the performance. In terms of technical approach, Lyra preserves the problem nature of scheduling elastic jobs and treats it as a variant of the knapsack problem, enabling it to make globally good allocation decisions and outperform greedy local heuristics in prior work. Though Lyra does not consider tuning hyperparameters, it can be readily integrated into Lyra (§7.4).

3 Design Overview

In this section, we describe Lyra's overall architecture and the key design questions we need to address.

Overall architecture. Lyra is a GPU cluster scheduler that exploits capacity loaning with elastic job scheduling. It runs on top of a cluster resource manager such as YARN [51] and Kubernetes [23] to execute its decisions.

Figure 4 presents Lyra's architecture. At the cluster level, the *resource orchestrator* receives instructions from the inference cluster about the number of servers to loan or reclaim (a), determines which servers shall be returned for reclaiming (b), and commands the underlying resource manager to move the selected servers virtually across management boundaries (c). When the orchestrator reclaims on-loan servers, it may need to preempt the training jobs running on them (d). Job preemption is executed via the job scheduler.

At the job level, jobs are submitted to the queues. The job profiler estimates the workload ① after jobs are enqueued. The *job scheduler* ② periodically collects job status and resource usage of the training cluster. Then it ③ computes the resource allocation and placement decisions for each job. Meanwhile, it gets preemption instructions from the orchestrator, interrupts the running jobs ④, and puts them back into the job queues ⑤. Job launching, scaling and interruption actions are again executed by the resource manager. Job

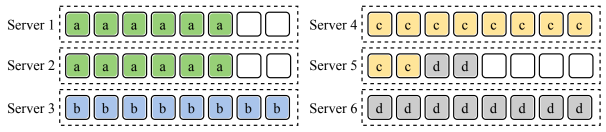


Figure 5. A reclaiming example. Each server has 8 GPUs. GPUs in-use are indicated by the job ID inside each square.

Server	# running jobs	sum of job's GPU fraction	sum of job's server fraction
1	1	0.5	0.5
2	1	0.5	0.5
3	1	1	1
4	1	0.8	0.5
5	2	0.4	1
6	1	0.8	0.5

Table 1. Different definitions of server preemption cost for the reclaiming example in Figure 5.

scheduler works periodically in a much smaller interval than the orchestrator in order to better handle job dynamics.

Since Lyra mainly deals with the training cluster and does not interfere with inference cluster scheduling, we use "jobs" to simply refer to training jobs hereafter without ambiguity. The basic unit of capacity loaning is a physical server. This is to prevent training jobs from interfering with the inference jobs on the same server.

Key questions. Lyra's design is centered around two key questions.

- **Server reclaiming.** Which servers should be returned so that the number of preempted jobs is minimized, when some on-loan servers need to be reclaimed?
- **Job scheduling.** How should we determine resource allocation across jobs, and how do we place a job's workers on servers, when some jobs are elastic and some servers are loaned from the inference cluster?

We now present how we address them with Lyra's detailed design in §4 and §5, respectively.

4 Capacity Loaning and Reclaiming

Lyra moves resources dynamically across inference and training clusters to improve utilization and training performance.

Assumptions. We presume that the inference cluster scheduler dynamically estimates the capacity needed to meet the latency, GPU utilization [24], or other performance targets, based on the predicted inference traffic [7, 14, 44]. Inference workloads are able to grow or shrink their containers along with the incoming traffic. Inference scheduler informs Lyra's resource orchestrator of (1) the amount of resources available for loaning when traffic is low, and (2) the amount of resources to be reclaimed from training in busy hours if any. That is, the inference cluster scheduler *autonomously* determines when and which servers to lend, and when and how many servers to ask back, based on its own policy. The inference performance is *not* affected by capacity loaning.

The key question for the training scheduler is the reclaiming mechanism as mentioned in §3, i.e. which on-loan servers should be returned given the number of servers needed by the inference scheduler. This matters because reclaiming a server entails preempting all its running jobs immediately. A job with checkpointing would incur overheads to save and load the checkpoint when resuming training later. If the job does not perform checkpointing [31], which is common in practice in our environment, its entire progress is lost and training has to restart from the very beginning. Clearly, both are undesirable and we strive to minimize preemptions by strategically picking the servers to return.

We propose a solution to reduce the negative impact on the training jobs hosted by the on-loan servers.

Minimizing preemptions. Vacating an on-loan server means its jobs are preempted in a cluster with no elastic jobs. We start with how Lyra minimizes inevitable preemption under this case. and will explain how elastic scaling plays its part in minimizing preemptions in §5.3.

Denote the number of servers that need to be returned at this point as N_R. Our problem is to pick N_R on-loan servers—which host inelastic jobs' workers—in order to minimize preemptions. More concretely, we choose to minimize the number of preempted jobs so fewer users are affected. This implies when reclaiming a server, we prefer the one with a big job to the one with a few small ones.

The problem closely resembles the classic knapsack problem (i.e. the 0-1 knapsack problem): The number of servers to reclaim N_R can be considered as the capacity of the knapsack; each server consumes one unit capacity, and the number of running jobs is each server's preemption cost (i.e. value). However, the server's preemption cost actually has interdependencies that make the problem more difficult.

Consider an example as depicted in Figure 5. Table 1 shows each server's preemption cost as the number of its running jobs (second column). Suppose we need to reclaim two servers. Servers 1 and 2 are obviously the optimal choice with one preemption. Yet the corresponding knapsack problem would select any two 1-cost servers such as 3 and 4 which lead to more preemptions. The issue here is that in our problem the costs of servers are coupled when they host the same job(s), whereas in the 0-1 knapsack problem the cost is independent of each other. Reclaiming server 1 for instance results in an idle server 2 whose cost becomes 0 instead of 1.

Knapsack problem with dependent item values is known to be NP-hard [32]. When N_R is one server, selecting the one with the fewest preemptions is simply by iterating all the on-loan servers. Given an N_R larger than a single server, we propose to resolve the dependency by treating it as part of the server preemption cost. One possible way is to define server preemption cost as the sum of the GPU fractions of each job on the server. For instance, server 4's cost would be 0.8 as it hosts 80% of job c's GPUs, and server 5's cost is 0.4 (0.2+0.2) as shown in Table 1. One can immediately see that this does not work well as it does not capture the job count. It causes server 5 to be selected with the least cost, which

Job	w^{min}	w^{max}	Min. running time
A	2	6	50
B	2	6	20

Table 2. Two elastic jobs and their demand information. Jobs complete in min. running time when allocated with w^{max} workers.

actually leads to two preemptions. Thus we choose to define server preemption cost as the sum of the server fractions of each job as shown also in Table 1. Server 4's cost would thus be 0.5 as it hosts job c and the workload of job c is shared by two servers as shown in Figure 5. This way server 5's cost is 0.5+0.5=1, i.e. the highest.

Once the preemption cost of each server is computed, the orchestrator selects the servers using the following heuristic: it iteratively picks the server with the lowest preemption cost, preempts its jobs by removing them from all their servers, and updates the cost of these servers correspondingly, until N_R servers are vacated. In case of a tie in the preemption cost, we additionally consider the collateral damage incurred if the server is reclaimed.

5 Job Scheduling

Lyra schedules jobs—both inelastic and elastic—to reduce overall JCT by utilizing resources as efficiently as possible. We start by explaining the challenge due to elasticity (§5.1). Then we present the solutions to the two facets of our scheduling problem. The first is *resource allocation* with both elastic and inelastic jobs, i.e. how to determine the number of workers each job gets (§5.2); the second is *placement*, i.e. which servers to place a job's workers on (§5.3).

Throughout this section, we assume that training throughput scales linearly with the number of workers within the scaling range, i.e. job's running time is inversely proportional to resources allocated, as discussed in §2.2.

5.1 Challenge of Elasticity

Job elasticity presents a unique challenge to resource allocation. Conventional schedulers either deal with jobs with fixed demands, or ones that can arbitrarily scale [18, 43]. However, for jobs with *limited elasticity* [36], the question of how to arbitrate resources so as to minimize average JCT is intricate.

Let us consider a simple example as shown in Table 2. There are two elastic jobs with different minimum running times when allocated their maximum demand. Assume the cluster has eight workers in total. Table 3 shows three common allocation strategies and the corresponding JCT performance. In solution 1, we favor job A by giving it the maximum demand; in solution 2, we favor B instead; and in solution 3 we equally allocate resources to them. All three strategies lead to different JCTs and the difference between the worst and best is 24%, demonstrating that inefficient allocation can lead to poor JCT performance.

Solution	Initial allocation		JCT		Average JCT
	A	B	A	B	
1	6	2	50	53.33	51.67
2	2	6	63.33	20	41.67
3	4	4	60	30	45

Table 3. Possible resource allocation results for the two jobs when they share a cluster that can host 8 workers. Only the initial allocation is shown; once the first job finishes, the other is immediately allocated more resources as much as possible. Three solutions lead to very different JCTs.

Job	w^{min}	w^{max}	Min. running time	JCT when favored	Avg. JCT
A	2	3	100	A: 100, B: 24	62
B	2	6	20	A: 106.67, B: 20	63.33

Table 4. A counter example with two elastic jobs, where prioritizing A with longer running time is actually better for JCT.

Classic algorithms are not optimal. One may be wondering if the classic shortest (or smallest) job first strategies would work here. At least in the example of Table 2, the optimal allocation is indeed to first satisfy job B, which has the shortest running time. Yet, we can construct a counter example as depicted in Table 4 to show that this does not always work. We slightly modify job A to have a maximum demand of 3, and minimum running time of 100; the other setup is identical to Table 2. In this case, if we satisfy B first, the average JCT (63.33) is actually worse than satisfying A's demand first (62).

Intuitively, shortest job first, or SJF, is designed for fixed job running times with the intuition that each job should be given the least queuing time, which is the only variable in computing JCT. In our case, job running time itself varies along with the resource allocated, which in turn affects the overall JCT and makes the problem more complex.

More specifically, the above examples reveal two characteristics of elastic job's running time that SJF cannot handle. (1) Elastic scaling complicates the job sorting decision of SJF. Since job running time varies with the resources allocated, it is no longer apparent that we simply sort them based on their minimum running time. As shown already, doing so does not lead to an optimal result. (2) The resource efficiency of each job is different. In Table 4, job A has a larger workload (i.e. product of maximum demand and minimum running time) than B, implying that the running time improvement of A is larger than that of B if both are given the same number of workers. Even though the resource allocation difference is merely one when we prioritize different jobs, job A's running time contributes to a 6.67-second JCT reduction while job B's only increases by 4 seconds.

In the simplest two-job case, we can analyze the outcome of different allocation strategies. The complete theoretical analysis is omitted here for brevity and can be provided upon request. Allocation in the general case is undoubtedly more complicated with more elastic jobs plus inelastic jobs, as the optimal strategy requires enumerating the exponentially many possible resource allocations. Our quest in the following is, therefore, to find a good heuristic for the problem.

5.2 Two-Phase Resource Allocation

Intuition: Prioritize inelastic workload. To ease the challenge of elasticity, our insight is that an elastic job has two types of demand: a *base demand* that is inelastic in nature, i.e. the minimum demand, and a *flexible demand* that is elastic. They should be treated separately: The base demand essentially corresponds to an inelastic job whose allocation strategy is binary, and not allocating resources to it incurs more queuing delay to the job. In contrast, the flexible demand can be unfulfilled without serious impact since the job is still making progress with base demand.

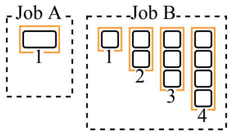
Therefore, we treat the *inelastic* workload, including elastic jobs' base demands and inelastic jobs, as the first class citizen. We schedule them first with all available resources to minimize the average JCT. This also avoids starvation. Then in phase two, we consider the flexible demand of elastic jobs to fully utilize the remaining resources from phase one.

Setup and assumptions. We focus on solving the offline setting myopically where the set of jobs and resources are given, and cope with the job dynamics and cluster capacity change by periodically performing scheduling in high frequency. This is common in the literature [10, 60]. Our scheduling solution is non-preemptive to minimize disruptions to training; preemption only happens during reclaiming when it becomes inevitable as in §4. Thus at a scheduling epoch, the set of available resources refers to idle GPUs and GPUs being used by flexible workers for resizing (including on-loan GPUs), and the set of jobs includes those waiting in the queue and running elastic jobs (only flexible workers). The on-loan inference GPUs are normalized relative to training GPUs when calculating the resource capacity.

We rely on job's running time information (minimum running time for elastic jobs), which can be predicted with profiling and ML methods [17, 61].

Two-phase heuristic design. We now elaborate our heuristic. The problem in phase one is how to minimize average JCT for jobs with fixed demands and known running times, for which we adopt the shortest job first (SJF) policy [9] which is a sensible and commonly used solution. As long as there are idle GPUs and pending jobs, we schedule job j^* with the smallest running time. If the demand of j^* exceeds the remaining capacity, we remove it from the pool and continue.

Phase two is more interesting. We must determine the number of additional GPUs elastic jobs get to maximize the JCT reduction. Elastic jobs here include those already running. It turns out this problem can be transformed into *multiple-choice knapsack problem* [48]: The knapsack's capacity is the number of remaining GPUs. An elastic job j is a group with $w_j^{max} - w_j^{min}$ items, each representing a possible allocation for j's flexible demand. An item's weight is the number of GPUs in this allocation, and its value is its JCT reduction over the job's maximum running time. Figure 6 illustrates this transformation with the two-job example in Table 4. The problem is to pack the items into the knapsack so that the total value is maximized, with the constraint of taking exactly one or zero items from each group.



Group	Item	Weight	JCT Reduction Value
A	1	2	50
B	1	1	20
	2	2	30
	3	3	36
	4	4	40

Figure 6. Item weights and JCT reduction values for jobs in Table 4. Here, we assume job A needs 2 GPU per worker and job B 1 GPU per worker.

The multiple-choice knapsack problem, similar to the classical knapsack, is NP-hard and often solved by dynamic programming which runs in pseudo-polynomial time [48]. With a moderate number of GPUs and jobs, dynamic programming can usually solve the instance efficiently. We find that the longest solution time in our evaluation is 0.02s with 354 items and 245 GPUs which is much shorter than a typical job's time use.

5.3 Worker Placement

Given the allocation results, i.e. number of workers each job gets, we still need to determine the placement of each worker to complete scheduling. Our goal is to reduce fragmentation. The primary concern is the mix of inelastic and elastic jobs as well as the transient on-loan servers with different GPUs.

Our fundamental strategy is bin packing with best-fit decreasing (BFD) heuristic [37]. Jobs are sorted in decreasing order of their per-worker GPU demand as GPU is most likely the bottleneck resource for training. Starting from the largest job, we place each worker of the job into a non-empty server that best fits its demand; if none has sufficient remaining resources, we place it on a new server. If the job is elastic, we prefer to place it on inference servers in order to maximize the potential for scaling in during reclaiming and reduce job preemptions. If it is inelastic, we prefer to place it on training servers. When placing elastic jobs, we also place their base and flexible demands on separate groups of inference servers so that during reclaiming (§4), Lyra can release the server group for flexible demands first without any preemption to see if this alone is sufficient.

6 Implementation

We have implemented a prototype of Lyra with about 3500 lines of Python. The prototype works with our existing YARN and Kubernetes deployment to move servers across clusters virtually, manage worker containers for training, and monitor the status of servers and workers.

We highlight key details of the implementation as follows.

Interface for capacity loaning. During loaning, resource orchestrator need to update the available resource of each cluster once the operation is decided. We create a *whitelist* API to facilitate capacity loaning operations. Both Lyra's scheduler and the inference scheduler maintain their own whitelist of servers under their control. During loaning, the orchestrator adds on-loan servers to Lyra scheduler's whitelist according to inference scheduler's instructions. In reclaiming, the orchestrator removes the selected servers from Lyra's whitelist after its scheduler confirms they no longer have running workers.

Inference resource usage predictor. We develop a simple NN model to predict the inference resource usage. The predictor is an LSTM model with a window size of 10 and two hidden layers. We apply Adam optimizer and use MSE to compute loss. We predict the resource usage of the next five minutes and compare the average resource usage to the ground truth. The average loss is 0.00048 over 1440 data points. With the predictor, Lyra can initiate reclaiming decisions in advance before the inference resource usage increases.

Data locality and resource isolation. Lyra performs capacity loaning only between clusters in the same datacenter to ensure the network bandwidth across servers is consistently high [2, 46]. Also, the basic unit of loaning is a physical server so co-location of inference and training jobs is not possible, and no additional isolation mechanisms are needed.

Enable elastic scaling. We enable elastic training in our environment with a few modifications to the ML frameworks. Chiefly, we embed a controller process to each elastic job that coordinates the worker join and departure. As presented before, an elastic training job has a base demand and a flexible demand. Base demand guarantees the gang scheduling of minimum requests and the flexible demand shortens the running time whenever possible while preserving loss convergence. Recent work [36, 56] developed more complete scaling solutions that our implementation could also utilize.

Handle heterogeneous GPU training. As discussed in §2, a small portion of training jobs can run on heterogeneous GPUs experimentally. When this feature is turned on, Lyra's job scheduler considers these jobs with the lowest priority on the remaining servers after all other jobs are scheduled. The actual scheduling logic for these jobs remains the same as we discussed in §5, except that if they are elastic, their base demands are placed on training servers, and flexible demands on inference servers whenever possible.

7 Evaluation

We evaluate Lyra using large-scale simulations and testbed experiments with traces from our production clusters.

The highlights of our findings are:

- In simulations, Lyra shows salient benefits with 1.53x and 1.48x reductions on average queuing time and JCT, respectively, compared to the baseline FIFO scheduler. When working alone, capacity loaning has 1.39x and 1.31x reductions in average queuing time and JCT, and elastic scaling has 1.35x and 1.38x reductions in the same metrics.
- Compared to state-of-the-art scheduler Pollux [43], Lyra's scheduling algorithm brings 1.35x average queuing time and 1.42x average JCT reductions when both consider tuning the training hyperparameters. Lyra's reclaiming algorithm performs comparably to the optimal solution with only 1–3ms running time.
- In testbed, Lyra improves average job queuing time by 1.38x and average JCT by 1.22x over the Baseline without loaning or scaling. Preemption only happens to ~9% of the jobs in reclaiming with an average 63-second overhead.

7.1 Setup

Traces. We rely on a 15-day job trace from one of our production training clusters with 3,544 GPUs (443 8-GPU servers). There are 50,390 training jobs, and job running time range from minutes to days. We also use a GPU utilization trace from the inference cluster for the same time period. Part of the traces has been shown in Figures 1 and 2 already.

Simulator. We built a discrete-event simulator for evaluating Lyra at scale using job traces from production. It simulates the cluster scale, hardware configuration, and all job events including arrival, completion, scaling, and preemption. Job's running time in the simulator is derived from actual training time in the traces. For elastic jobs, we compute its actual training time based on the traces which is inversely proportional to its resource allocation as discussed in §5. We also consider jobs with imperfect scalability in §7.2.

Testbed. Our testbed consists of four 8-GPU training servers and four 8-GPU inference servers. Each training server uses Nvidia V100 GPUs with 32GB GPU memory and has 92 vCPU with 350 GB memory. Each inference server uses Nvidia T4 GPUs with 16GB GPU memory and has 92 vCPU and 210 GB memory. The resource management framework is YARN, and training data is stored in HDFS.

Headroom in inference cluster. To handle unexpected traffic surges in the inference cluster, we leave a headroom of 2% of the inference cluster capacity. These machines are never to be loaned. This is chosen based on our empirical observations. Lyra's resource orchestrator runs every five minutes; and we find that the median inference traffic burst within five minutes is ~2% of the inference cluster capacity based on our GPU utilization trace (§2.1).

Training job types. Based on the resource requirements, training jobs can be:

- *Fungible*: 21% jobs can be executed on different GPU types in different runs, i.e. ideal for capacity loaning.
- *Elastic*: Jobs can take a variable number of workers that can be adjusted on-the-fly. They are ideal for elastic scaling.
- *Heterogeneous*: Jobs can run on different GPU types at runtime.

Scenarios. We consider various scenarios with different degrees of support for elastic scaling and heterogeneous training, both of which are not widely used today.

#	Scenario	Scheme	Queuing Time (s)			JCT (s)			GPU Usage		Preemption
			Mean	Median	95%ile	Mean	Median	95%ile	Training	Overall [1]	Ratio [2]
1	—	Baseline [3]	3072	55	8357	16610	791	82933	0.72	0.52	0
2	Basic	Lyra	2010	26	3358	11236	568	56477	0.86	0.65	12.24%
3	Advanced		1835	24	3238	10434	525	56553	0.86	0.68	7.35%
4	Heterogeneous		1944	27	3574	12113	604	57392	0.78	0.64	11.23%
5	Ideal		1157	22	3204	8891	422	41146	0.93	0.72	5.72%
6	Capacity Loaning (Basic)	Opportunity	2788	22	5256	14828	744	67843	0.74	0.63	19.35%
7		Random	2901	23	5478	14678	731	62923	0.76	0.64	20.89%
8		SCF	2783	24	4994	14923	695	62456	0.76	0.64	17.48%
9		Lyra	2212	23	3427	12947	662	57987	0.76	0.65	14.94%
10	Elastic Scaling (Basic)	Gandiva	3035	49	6632	15912	755	80567	0.79	NA	NA
11		AFS	2284	47	3488	15045	686	60883	0.95	NA	NA
12		Pollux	2791	58	5883	14534	721	72123	0.93	NA	NA
13		Lyra	2275	47	3475	12048	602	57597	0.92	NA	NA
14		Lyra+TunedJobs	2054	43	2749	10229	564	52458	0.91	NA	NA

(1) Overall GPU usage denotes the GPU utilization in both training and inference cluster. It is applied when the training cluster size is changing in capacity loaning.
(2) Preemption ratio is the ratio between the total number of preemptions and the total number of job submissions.
(3) No capacity loaning or elastic scaling is considered. We use the FIFO job scheduler in Baseline §7.1.

Table 5. Simulation results in different scenarios using different schemes.

- *Basic*: Here fungible jobs are used for capacity loaning (21% of total training load), and elastic jobs are used for elastic scaling (~5% of all jobs accounting for 36% of total training resources). No heterogeneous training. This corresponds to the status quo in our environment (recall §2.1) and is the default scenario.
- *Advanced*: On top of *Basic*, 10% of jobs can use heterogeneous GPUs with non-ideal performance. The jobs are randomly selected and distributed evenly across 15 days. Specifically, heterogeneous training jobs only achieve at most 70% of the ideal results. We experimentally confirm such a performance gap which has also been reported by prior work [4, 39].
- *Heterogeneous*: Different from the *Advanced* scenario, we disable the 21% fungible training load and consider the 10% heterogeneous training non-ideal performance solely.
- *Ideal*: All jobs support scaling and heterogeneous training with ideal performance. For jobs without a pre-defined scaling range, we consider its requested demand to be the base demand, and its scaling range is twice that.

Schemes compared. We compare Lyra to the following schemes that represent the state-of-the-art and/or the most common solutions to each sub-problem of Lyra.

We first compare capacity loaning to a simple opportunistic scheme:

- *Opportunistic Scheduling*: We disable capacity loaning, and queue the 21% fungible training jobs to the inference cluster with a lower priority than inference jobs, so they can opportunistically use the idle servers.

We also consider two basic strategies for server reclaiming:

- *Random*: On-loan servers are randomly selected.
- *Smallest (Job) Count First (SCF)*: The top-k servers that host the smallest number of jobs are chosen.

We consider several solutions to elastic scheduling. Some are slightly modified to conform with our setup for elastic jobs.

- *Gandiva* [55]: Elastic scaling is also mentioned in Gandiva. It exploits elasticity by scaling out jobs to utilize the remaining resources on servers whenever they are underutilized. We consider under-utilization to be the period when there are available resources but no pending jobs.
- *AFS* [18]: Starting from one GPU per job, it iteratively adds one more GPU to the job with the largest marginal throughput gain. We implement AFS by allocating base demand to each job first and allocating one more worker to the job with the largest throughput gain per GPU.
- *Pollux* [43]: Pollux computes the goodput of training jobs and applies genetic algorithms to find the resource allocation. It also adjusts batch size to maximize goodput and learning rate based on Adascale [21]. We adopt the model distribution listed by Pollux to capture the model goodput.

We notice that Pollux's idea of tuning the hyperparameters according to allocated resources is orthogonal to job scheduling. To compare with Pollux fairly, we integrate this idea into Lyra in §7.4:

- *Lyra+TunedJobs*: Use Lyra's job scheduler and adapt Pollux's job agent for job-level hyperparameter-tuning within the scaling range. Job agent adjusts model batch size and learning rate whenever job resource allocation changes.

Lastly, our baseline scheme is:

- *Baseline*: A FIFO cluster scheduler with no capacity loaning or elastic scaling.

Metrics. We consider queuing time and JCT to evaluate Lyra. We report Lyra's performance improvements using the

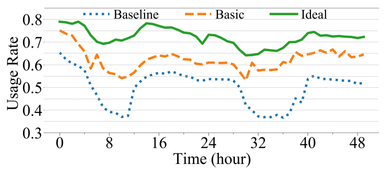


Figure 7. Overall resource usage rate of Baseline and Lyra in Basic and Ideal scenario.

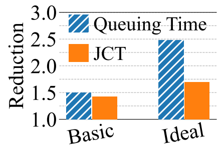


Figure 8. Average queuing time and JCT against Baseline in imperfect scalability.

following method:

$$\text{Reduction} = \frac{\text{Duration of a scheme compared}}{\text{Duration of Lyra}}$$

Both the average queuing time and the average JCT are the arithmetic mean.

7.2 Overall Performance in Simulation

We evaluate Lyra thoroughly with large-scale simulation. We provide its overall performance here. Analyses of its individual components are presented in §7.3 and §7.4.

Simulator calibration and fidelity. To first establish its fidelity, we evaluate our simulator against the prototype system in a testbed with a small trace.

We calibrate our simulator by comparing the scheduling logs between the testbed and the simulator. We carefully build several tiny training job traces (20 minutes – 2 hours) to cover all possible job and resource allocation status. We run the traces on the testbed and record the timestamp of every activity (e.g. job launching, start and end of training, scheduling decision). The same traces are replayed on the simulator. We compare the timestamp and decision of each activity, and find the first wrong decision or the first activity with a larger-than-two-seconds time difference. We resolve the time difference and replay the trace repeatedly until all activities on the simulator match with the testbed records.

We add a fixed overhead according to our testbed experiments (§7.5) whenever a job is preempted in simulation. The simulation results are similar to testbed results, with a difference of 6.2% and 3.4% in average and 95%ile JCT, and 3.5% and 4.4% in average and 95%ile queuing time. The small difference mainly stems from the overhead of placing and removing workers and moving resources between clusters which the simulator does not capture.

Cluster and workload. We use the full 15-day trace and the same cluster configuration as our production clusters.

Queuing time, JCT, and cluster usage. Table 5 records the performance of Lyra in different scenarios. Overall, queuing time and JCT are improved by 1.53x and 1.48x when compared to Baseline in the Basic scenario (row 2). The overall cluster usage is improved by 25%. In the Advanced case with non-ideal heterogeneous training, queuing time and JCT are reduced by 1.67x and 1.59x over Baseline and by 1.10x and 1.08x over Lyra itself in the Basic scenario. In the Ideal case which represents the performance upper bound, the average combined usage of inference and training clusters is improved by 38.5% (to 72%) over Baseline. Compared to the Basic case, average queuing time and JCT in the Ideal case show additional 27% and 14% improvements by virtue of complete job flexibility and perfect performance scalability.

Scheme	Queuing Time (s)			JCT (s)		
	Mean	Median	95%ile	Mean	Median	95%ile
Baseline	4573	1283	23351	11547	2122	60170
Lyra	1119	274	7256	6887	1373	35776

Table 7. Queuing time and JCT of jobs running on on-loan servers.

Since the training cluster's resource is dynamically changing, we depict the hourly combined cluster usage for 48 hours in Figure 7. The Baseline usage curve shows a clear diurnal pattern mostly attributable to the inference cluster. When capacity loaning is enabled, Lyra improves the usage and flattens the curve; the most significant improvement is a 14% usage increase between Basic and Baseline. Notice the combined usage does not reach 100% since the inference cluster needs a 2% headroom to gracefully handle the latency SLA.

How scaling helps capacity loaning? We now seek to understand how our two key ideas interact and complement each. Scaling helps capacity loaning, especially in reducing preemptions in reclaiming the on-loan servers. With elastic scaling disabled, Table 5 shows that preemption as percentage of running jobs increases from 12.24% (row 2) to 14.94% (row 9). We also observe that on average the flexible server group (hosting elastic workers only) alone satisfies 53.5% of reclaiming demand each time. With more aggressive flexibility (row 5), preemption is reduced to 5.72% and satisfies 83.5% of reclaiming demand each time.

In §5.3, we discussed how Lyra places elastic and inelastic jobs with on-loan servers in the cluster. In Table 6, we compare the placement performance in different scenarios without special treatment to elastic jobs, i.e. instead of grouping their flexible demand and placing them to on-loan servers as much as possible, the scheduler places them to training servers first just like inelastic jobs. The most significant difference is in preemption ratio. Without grouping the flexible demand, preemption ratio increases by up to 91% in Ideal (compared to Table 5 row 5). Preemptions also incur degradation to job runtime; for example average queuing time and JCT in the Basic case increase by up to 11.1% and 15.2%.

Impact of imperfect scaling. Thus far we have assumed linear scalability of elastic jobs based on our empirical analysis in §2.2. Here we also evaluate Lyra when elastic jobs scale non-linearly with throughput loss. When one more worker is added to a job, we add a 20% loss to the throughput brought by this worker. Figure 8 presents Lyra's gains over Baseline with non-linear scaling. In Basic, average queuing time and JCT are 3.03% and 5.82% higher than those with linear scalability (Table 5 row 2). The degradation is mild because most training jobs are inelastic in Basic scenario and Lyra always satisfies their base demands. In Ideal, JCT is inflated by 10.54% to 9,828 seconds (compared to Table 5

Scenario	Avg. Queuing Time (s)	Avg. JCT (s)	Preemption Ratio
Basic	2231	13872	13.22%
Advanced	1944	12474	10.04%
Ideal	1273	9982	10.93%

Table 6. Performance without special placement of elastic jobs. Lyra naively places jobs based on the BFD heuristic.

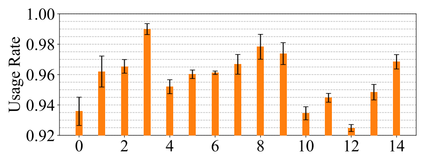


Figure 9. The daily average resource usage of on-loan servers (monitored every 5 minutes).

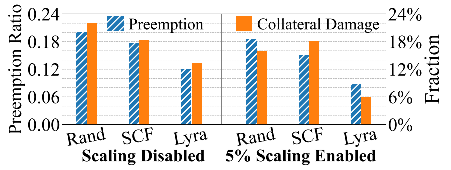


Figure 10. Preemption ratio and average collateral damage (defined in §7.3, Reclaiming heuristic).

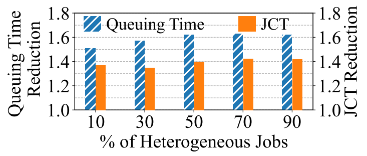


Figure 11. Average queuing time and JCT against Baseline in Heterogeneous.

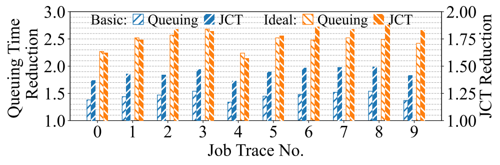


Figure 12. Average queuing time and JCT against the respective Baseline in Basic and Ideal for ten 10-day traces.

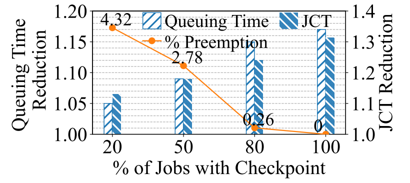


Figure 13. Average queuing time and JCT when jobs with checkpointing increase in Ideal.

row 5) due to the increase in job running time; the gain over Baseline is ~1.7x.

Heterogeneous training. In Table 5, Lyra in the Heterogeneous scenario shows 1.58x and 1.37x reduction over Baseline in average queuing time and JCT. However, the preemption ratio is only 1% lower than Basic compared to 4.89% reduction in Advanced. We also manually enable heterogeneous training for more jobs in Figure 11. Intuitively, more jobs capable of heterogeneous training could bring more benefits to cluster efficiency. Job resource allocation could be more flexible. However, heterogeneous training leads to throughput loss and uses more resources to maintain the training progress than homogeneous training. Moreover, the availability of inference servers is subject to inference cluster traffic, and jobs may have to wait when few resources are available. Therefore, the reduction of average queuing time approaches its asymptotic limit when 50% or more jobs support heterogeneous training.

Reproducibility of results. We also validate the reproducibility of the results. Here we compose ten 10-day training job traces based on the full 15-day trace in §7.1 using the bootstrapping technique. The cluster size remains the same. Figure 12 shows the results. Lyra's gains in queuing time and JCT are 1.45x and 1.44x in Basic, and 2.47x and 1.78x in Ideal. Lyra's performance is better when the training cluster has a long job queue. On weekends, training cluster is less busy. We notice that the gain in traces No.0 and No.4 is lower (10%) than others because two weekends are selected. On weekends, training cluster is less busy. Lyra's performance is better when the training cluster is busy and has a long job queue. Excluding these two traces, Lyra's improvement is statistically significant and consistent with results in Table 5 (rows 2 and 5). The average JCT improvement in Basic and Ideal shows a less than 4% gap with the performance improvement on the complete trace.

7.3 Deep-Dive: Capacity Loaning

We now dive into the two components of Lyra. We first focus on capacity loaning, aiming to understand its sources of gain and how our knapsack-based reclaiming heuristic compares to other schemes. The results here are obtained without elastic scaling.

Sources of gain. Table 5 (row 9) shows that loaning alone reduces average queuing time and JCT by 1.39x and 1.31x over Baseline. Loaning also improves the combined cluster usage from 52% to 66%. The JCT improvement mainly comes from the reduction in queuing time as jobs now can run on the loaned resources instead of waiting in the queue. Table 7 shows the statistics of queuing time and JCT for jobs running on the on-loan servers. The median and 95%ile queuing time is improved by 4.68x and 3.22x, respectively, compared to Baseline. The resource usage rate of on-loan servers throughout the experiment is consistently above 92% as depicted in Figure 9, which proves the effectiveness of resource loaning.

We observe that JCT improvement of capacity loaning is not as significant as elastic scaling (Table 5 row 13). This is because (1) loaning depends on idle inference resources and its gain is less stable, and (2) compared to scaling, loaning itself does not affect job running time.

Opportunistic scheduling. We then discuss why capacity loaning is more efficient than simple opportunistic scheduling. Table 5 row 6 shows the performance when the fungible jobs are scheduled opportunistically in the inference cluster. This does improve average queuing time and JCT over Baseline, but suffers 26.0% and 14.5% loss compared to Lyra (row 9). This is mainly because when fungible jobs are blindly put to inference servers, they suffer lower resource efficiency.

Reclaiming heuristic. We compare our reclaiming heuristic to Random and SCF. We use two metrics, the percentage of preempted jobs among running jobs, and collateral damage as the fraction of GPUs vacated in excess of the reclaiming demand. It is clear from Figure 10 Lyra outperforms others with and without elastic scaling. Without scaling, Lyra's knapsack-based heuristic reduces preemption and collateral damage by 1.51x, 1.68x and 1.36x, 1.59x over SCF and Random, respectively. With scaling, Lyra scales elastic jobs on the flexible server group first which further widens the gap. From Table 5, it is clear that reducing preemptions is beneficial: Lyra reduces the average queuing time and JCT by 1.26x, 1.15x and 1.31x, 1.13x over SCF and Random.

We also run an exhaustive search to find the optimal reclaiming solution. Lyra results in the same number of preemptions as optimal when reclaiming fewer than 60 servers, and incurs 19% more preemptions otherwise. We compare the servers reclaimed by Lyra with the optimal solution. An average 84% of servers in the optimal solution are picked by Lyra's reclaiming decision. The average running time of the optimal solution, however, is 420k times that of Lyra.

Use of checkpointing. Checkpointing can effectively help a preempted job recover the training progress and resume from where they are interrupted. Since the scheduler cannot determine if a job has proper checkpointing or not, in our default setup we have made a conservative assumption that no jobs have checkpointing. Here we gradually increase the fraction of jobs with checkpointing enabled and present its impact on performance against the default case without checkpointing (Table 5 row 9). Figure 13 shows that prevalent checkpointing consistently improves Lyra: for example the preemption ratio is reduced to 0.26% and average JCT is reduced by 1.24x when 80% jobs have checkpoints.

7.4 Deep-Dive: Job Scheduling

We evaluate job scheduling in more detail here. The results are obtained without capacity loaning in Basic scenario.

Sources of gain. Table 8 shows the queuing time and JCT distributions of all schemes. Our key insight in solving the scheduling problem is to prioritize the inelastic workload (§5.2). Gandiva does not improve Baseline much due to its opportunistic nature: it only scales jobs in low-utilization periods. Both Lyra and AFS allocate the minimum demand to each job initially. From Table 8, they have similar median queuing time. Though Pollux considers job's minimum demand and favors those with large goodput, it does not explicitly launch as many jobs as possible, thus incurring longer queuing time. Lyra outperforms Pollux by 1.23x and 1.69x in median and 95%ile queuing time.

Turning to JCT, we find from Table 8 that Pollux tends to prolong the large-and-long jobs by shrinking their resources towards the end of training to yield for newly-started jobs that make rapid progress with the same resources. Moreover, Pollux's performance heavily hinges upon the problem scale and the number of iterations allowed for its genetic algorithm. In a large cluster of over 3,500 GPUs with heavy workload, the preset 100 iterations are not sufficient to get an efficient allocation result. To keep the scheduling overhead acceptable, we set the number of iterations to 250 and

Scheme	%ile Queuing Time (s)				%ile JCT (s)			
	50	75	95	99	50	75	95	99
Baseline	55	1892	8357	14323	791	29163	82933	376513
Gandiva	49	1764	6632	11806	755	27244	80567	323626
AFS	58	1297	5883	11124	721	12304	72123	323513
Pollux	47	772	3488	9031	686	20143	60883	247435
Lyra	47	697	3475	8731	602	12072	57597	223815
Lyra+TunedJobs	43	566	2749	7112	564	9293	52458	194391

Table 8. 50%ile, 75%ile, 95%ile and 99%ile of queuing time and JCT (Basic).

% Wrong Prediction	Queuing Time Reduction	JCT Reduction
20%	2.21	1.52
40%	2.17	1.49
60%	1.76	1.38

Table 9. Queuing time and JCT reduction with incorrect running time estimation. The fraction of incorrect estimation varies from 0% to 60%. We assume each incorrect prediction has a random error margin within 25%.

Lyra still has 1.20x and 1.25x improvements in median and 95%ile JCT. AFS assumes unbounded elasticity and shows a higher resource usage. However, unlimited elasticity and greedy allocation implicitly favor jobs with better throughput at the cost of others. Its average JCT is 1.2x that of Lyra which balances the resources each job gets by making global allocation and considering limited elasticity.

Sensitivity analysis: Proportion of elastic jobs. We wish to analyze whether Lyra is sensitive to the proportion of elastic jobs in the mix. Figure 14 shows the performance comparison when elastic jobs grow from 20% to 100% of the population. All schemes show improvements as a result. Lyra delivers the largest gains in both queuing time and JCT compared to other schemes with more elastic jobs, demonstrating that its scheduler most efficiently exploits job elasticity. AFS also has good gains in queuing time as it initially allocates minimum demand to each job. Its JCT gains, however, are much lower due to the greedy heuristic in ordering the jobs for allocation. Pollux's queuing time performance is poor as queuing time is not considered in its design. Its JCTs are much better because it auto-tunes the hyperparameters for the best performance.

Sensitivity analysis: Error in running time estimation. Our second sensitivity analysis concerns the running time prediction which Lyra's scheduler relies on. Table 9 shows the performance under different estimation accuracy. Lyra improves queuing delay by 1.76x over Baseline even when there are 60% wrong predictions (each with at most 25% error). Its gain is consistent with less than 60% wrong predictions, which demonstrates its robustness.

Sensitivity analysis: Imperfect scaling of elastic jobs. In Figure 16, we plot Lyra's performance with non-linear scalability of training throughput, following the same setup discussed in §7.2. The average JCT improves by 1.86x when all the elastic jobs scale non-linearly. When the fraction of elastic jobs is less than 50%, non-linear scalability has less than 5% impact on JCT compared to linear scalability. Yet its impact on JCT grows (up to 9%) as elastic jobs become the

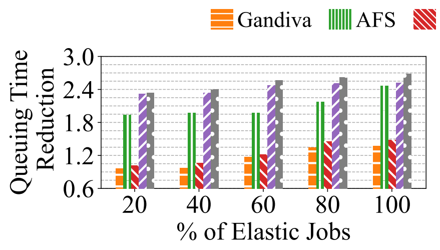


Figure 14. Queuing time reduction of Baseline as elastic jobs increase.

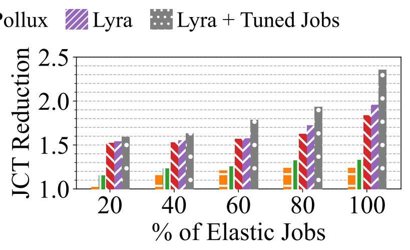


Figure 15. JCT reduction of Baseline as elastic jobs increase.

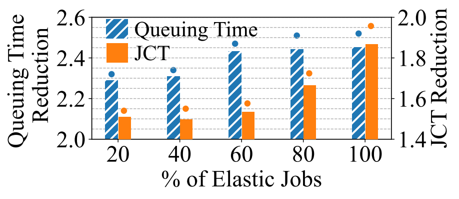


Figure 16. Lyra with non-linear scaling. Dots indicate the results with linear scaling.

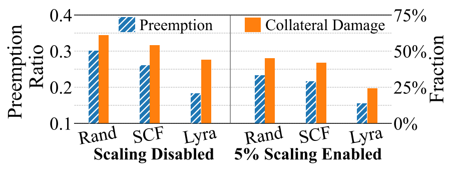


Figure 17. Preemption Ratio and average collateral damage comparison in testbed.

Scenario	Scheme	Queuing Time (s)			JCT (s)			Preemption
		Mean	Median	95%ile	Mean	Median	95%ile	Ratio
Overall	Baseline	1532	772	1003	4078	2183	3096	0
	Lyra	1109	503	738	3335	1747	2731	18%
Capacity Loaning	Random	1527	658	993	3893	2046	3015	34%
	SCF	1473	614	864	3857	1994	3001	30%
	Lyra	1230	594	823	3748	1946	2864	22%
Elastic Scaling	Gandiva	1443	645	1002	3882	2015	2893	NA
	AFS	1338	534	882	3521	1836	2803	NA
	Pollux	1405	576	937	3552	1934	3004	NA
	Lyra	1318	546	798	3413	1791	2794	NA

Table 10. Testbed results using different schemes in Basic scenario.

primary workload, because they run slower due to non-linear scalability. Meanwhile, the newly-arrived jobs have to wait longer for running jobs to vacate the resources, resulting in up to 7% increase in average queuing time.

Hyperparameter tuning. We study Lyra+TunedJobs now which adapts Pollux's job agent to tune jobs' hyperparameters as explained in §7.1. In the Basic scenario, Lyra+TunedJobs (row 14 in Table 5) enjoys additional 18% and 13% gains over Baseline in 95%ile and 99%ile JCT. This gain is more significant when all the jobs are elastic as seen in Figures 14–15.

More importantly, Lyra+TunedJobs allows for a fair comparison with Pollux as both have hyperparameter tuning. It outperforms Pollux by 1.32x and 1.37x in median and 95%ile JCT in Basic scenario (Table 8). Lyra's gain over Pollux is larger here which shows that Lyra's scheduling policy performs better in JCT. The main reason is that Lyra specifically optimizes JCT while Pollux optimizes goodput for resource efficiency. Thus JCT for some jobs is affected, especially near the end of training when the marginal gain of resources becomes smaller (i.e., goodput is lower) and resource allocation is decreased. Another side-effect of goodput-based scheduling is back-and-forth scaling as goodput varies as soon as the hyperparameter or allocation changes. We find the number of scaling operations in Pollux is 1.76x that of Lyra+TunedJobs in the Ideal scenario, and many are scaling-out followed immediately by scaling-in in the next interval.

7.5 Testbed Results

We use our prototype in testbed experiments to schedule jobs and YARN to run, scale, and preempt them.

Workload. We use a scaled-down version of the traces with 180 training jobs (10 elastic ones, similar to Basic scenario); jobs with (maximum) demand larger than 16 GPUs (50% cluster) are excluded. Job submission lasts for 8 hours and training time varies from 2 minutes to 2 hours. The inference trace is also scaled down according to the testbed capacity.

JCT and queuing time. Table 10 shows the statistics of queuing time and JCT. Lyra improves average and 95%ile queuing time by 1.38x and 1.36x over Baseline . In terms of JCT, Lyra improves the median and 95%ile by 19.9% and 11.7% over Baseline. The gains come from both capacity loaning and elastic scaling: the orchestrator performed 6 loaning and 8 reclaiming operations involving a total of 10 servers, and the scheduler issued 73 scaling operations. In capacity loaning, Lyra outperforms Random and SCF by 19% and 15% in average queuing time. In elastic scaling, Lyra's tail queuing time is 10% shorter than AFS. Its JCT gain is 1.19x over Baseline compared to 1.14x and 1.15x for AFS and Pollux. The results here show that Lyra is highly effective in reducing queuing time. The JCT improvements are relatively small due to the inference cluster's limited resources compared to job demand. We observe the inference cluster loaned at most three servers which is equivalent to one training server in computational capability, while it is common for a job to demand an entire training server in our trace.

Preemption. Figure 17 shows the total number of preemptions and the corresponding collateral damage in testbed. Lyra reduces preemption significantly by over 1.3x compared to Random and SCF reclaiming schemes (row group 2). We also measure the preemption overhead, including the time to save a checkpoint to the disk, terminate containers, launch new containers on different servers, and load the checkpoint before training starts. The average overhead is 63 seconds, which is adopted in our large-scale simulation.

8 Discussion

Fine-grained resource sharing. Lyra uses physical machines as the basic unit of loaning and reclaiming. Our intention is to avoid interference between training and inference.

This concern can be alleviated by improvements from the infrastructure (e.g. better isolation mechanisms). Then one may consider fine-grained sharing on the GPU level, which allows more sharing opportunities but also demands a more careful scheduling design because of the larger problem scale.

Performance under scaling. We assume the elastic job's training throughput is linear in the allocated resources within the scaling range. In practice training throughput is likely to scale sub-linearly due to factors such as network communication and synchronization overhead. An improved approach may be to empirically profile the throughput and running time of the workloads as a non-linear function of resources. Lyra's scheduling algorithm still works with non-linear scaling which does not change the combinatorial nature of the problem; we provided simulation results in §7.2.

Heterogeneous GPU training. Training with heterogeneous GPUs is an active area of research and current mechanisms are primitive [4]. We observe that though adjusting the batch size can roughly synchronize the workers, it may prolong the training convergence in some cases. More effort is needed to improve training efficiency with heterogeneous GPUs and to automate hyperparameter adjustment [3, 34].

9 Related Work

GPU cluster schedulers. We have discussed Pollux, AFS, and Gandiva extensively in §2.3 and §7.2. Tiresias [13] applies least-attained-service to minimize average JCT. It does not consider elastic scaling. Optimus [40] predicts the training time by modeling the loss convergence speed and designs a heuristic to minimize average JCT. Predicting a DNN's convergence, however, is challenging as discussed in [15]. PAI [53] introduces a scheduler which reserves high-end GPUs for high-GPU tasks and packs low-GPU tasks on less advanced GPUs. These works all schedule jobs in a cluster with a fixed capacity.

Systems support for elastic scaling. There is emerging interest in exploiting resource elasticity in distributed training. Systems such as [16, 22, 42] extend various ML frameworks to support elasticity. [36] proposes an auto-scaling policy by considering both cost and scaling efficiency. AntMan [56] provides a scaling mechanism to micromanage computation and GPU memory during training, and a job scheduler for performance guarantees. They are complementary to Lyra as they provide practical solutions for scaling DNN jobs.

Dynamic resource allocation. Graphene [12] and PriorityMeister [62] dynamically adjust resource allocation to fit job's time-varying demand and utilize resources more efficiently. In Lyra, we consider scaling for jobs that can work with a range of resources, which are taken as constraints to the scheduling problem. Lyra schedules jobs with an extra dimension of how much resource should a job get and its impact on cluster performance.

10 Conclusion

We have presented Lyra, an elastic GPU cluster scheduler for deep learning. The key idea is to exploit cluster-level elasticity by loaning idle inferences servers for training, and job-level elasticity by scaling jobs to better utilize the dynamic resource pool. In designing and evaluating Lyra, we have addressed new challenges in cluster management, by introducing heuristics to reduce job preemption cost due to loan-reclaiming, and to minimize job completion time when elastic jobs are presented. We plan to investigate information-agnostic scheduling without knowing jobs' running time a priori in future work.

Acknowledgments

We thank the anonymous EuroSys'23 reviewers and our shepherd Tim Harris for their constructive and valuable comments. We also thank the anonymous reviewers from ATC'22, NSDI'22, SOSP'21, and OSDI'21 for their feedback that helped shape the paper. This work was supported in part by funding from the Research Grants Council of Hong Kong (N_CityU139/21, C2004-21GF, R1012-21, R6021-20F, and 11209520), and from CUHK (4937007, 4937008, 5501329, 5501517 and 8601677).

References

[1] Ammar Ahmad Awan, Khaled Hamidouche, Akshay Venkatesh, and Dhabaleswar K Panda. 2016. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*.

[2] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *Proc. USENIX OSDI*.

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

[4] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. 2020. Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments. In *Proc. ACM SoCC*.

[5] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *Proc. USENIX OSDI*.

[6] Edward G. Coffman Jr., János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. 2013. Bin Packing Approximation Algorithms: Survey and Classification. In *Handbook of Combinatorial Optimization*.

[7] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI*.

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

[9] Samuel Eilon and IG Chowdhury. 1977. Minimising waiting time variance in the single machine problem.

[10] Dror G Feitelson and Larry Rudolph. 1998. Metrics and benchmarking for parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*.

[11] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming

He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv:1706.02677
[12] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. USENIX NSDI*.
[13] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *Proc. USENIX NSDI*.
[14] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proc. USENIX OSDI*.
[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proc. IEEE/CVF CVPR*.
[16] Horovod. 2021. Elastic Horovod. https://horovod.readthedocs.io/en/latest/elastic_include.html.
[17] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. 2022. dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training. In *Proc. MLSys*.
[18] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *Proc. USENIX NSDI*.
[19] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. arXiv:1807.11205
[20] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *Proc. USENIX OSDI*.
[21] Tyler B Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. 2019. AdaScale SGD: A Scale-Invariant Algorithm for Distributed Training.
[22] Kubernetes. 2021. ElasticDL: A Kubernetes-native Deep Learning Framework. https://github.com/sql-machine-learning/elasticdl.
[23] Kubernetes. 2021. Kubernetes. https://kubernetes.io/.
[24] Kubernetes. 2021. Kubernetes Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning.
[26] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proc. USENIX OSDI*.
[27] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. 2011. Dynamic right-sizing for power-proportional data centers. In *Proc. IEEE INFOCOM*.
[28] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proc. ACM ISCA*.
[29] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2019. Mlperf training benchmark. arXiv:1910.01500
[30] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. 2018. Massively distributed SGD: ImageNet/ResNet-50 training in a flash. arXiv:1811.05233
[31] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *Proc. USENIX FAST*.
[32] Davoud Mougouei, David MW Powers, and Asghar Moeini. 2017. An integer linear programming model for binary knapsack problem with dependent item values. In *Australasian Joint Conference on Artificial Intelligence*.
[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proc. ACM SOSP*.
[34] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *Proc. USENIX OSDI*.
[35] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. 2018. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*.
[36] Andrew Or, Haoyu Zhang, and Michael Freedman. 2020. Resource elasticity in distributed deep learning. In *Proc. MLSys*.
[37] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. 2011. *Heuristics for vector bin packing*. Microsoft Research Technical Report. Microsoft Research.
[38] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. arXiv:1811.09886
[39] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *Proc. USENIX ATC*.
[40] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*.
[41] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proc. ACM SOSP*.
[42] PyTorch. 2021. PyTorch Elastic. https://pytorch.org/elastic/0.2.0rc1/distributed.html#module-torchelastic.distributed.launch.
[43] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *Proc. USENIX OSDI*.
[44] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proc. ACM SOSP*.
[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv:1909.08053
[46] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. 2022. Singularity: Planet-Scale, Preemptible, Elastic Scheduling of AI Workloads. arXiv:1403.1349
[47] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556
[48] Prabhakant Sinha and Andris A Zoltners. 1979. *The multiple-choice knapsack problem*. Technical Report. Operations.
[49] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proc. USENIX OSDI*.

[50] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. ACM EuroSys*.

[51] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proc. ACM SoCC*.

[52] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proc. ACM EuroSys*.

[53] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *Proc. USENIX NSDI*.

[54] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv:1609.08144

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

[56] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Proc. USENIX OSDI*.

[57] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. 2015. Performance modeling and scalability optimization of distributed deep learning systems. In *Proc. ACM SIGKDD*.

[58] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, Guoping Long, Lansong Diao, Chuan Wu, Zhen Zheng, Jun Yang, and Wei Lin. 2020. Optimizing distributed training deployment in heterogeneous GPU clusters. In *Proc. CoNEXT*.

[59] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. arXiv:1904.00962

[60] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. In *Proc. VLDB Endow.*

[61] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *Proc. USENIX ATC*.

[62] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2014. Prioritymeister: Tail latency qos for shared networked storage. In *Proc. ACM SoCC*.