FaPES: Enabling Efficient Elastic Scaling for Serverless Machine Learning Platforms

Xiaoyang Zhao
The University of Hong Kong
xyzhao@cs.hku.hk

Siran Yang
Alibaba Group
siran.ysr@alibaba-inc.com

Jiamang Wang
Alibaba Group
jiamang.wang@alibaba-inc.com

Lansong Diao
Alibaba Group
lansong.dls@alibaba-inc.com

Lin Qu
Alibaba Group
xide.ql@taobao.com

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

ABSTRACT

Serverless computing platforms have become increasingly popular for running machine learning (ML) tasks due to their user-friendliness and decoupling from underlying infrastructure. However, auto-scaling to efficiently serve incoming requests still remains a challenge, especially for distributed ML training or inference jobs in a serverless GPU cluster. Distributed training and inference jobs are highly sensitive to resource configurations, and demand high model efficiency throughout their lifecycle. We propose FaPES, a **Fa**aS-oriented **P**erformance-aware **E**lastic **S**caling system to enable efficient resource allocation in serverless platforms for ML jobs. FaPES enables flexible resource loaning between virtual clusters for running training and inference jobs. For running inference jobs, servers are reclaimed on demand with minimal preemption overhead to guarantee service level objective (SLO); for training jobs, optimal GPU allocation and model hyperparameters are jointly adapted based on an ML-based performance model and a resource usage prediction board, alleviating users from model tuning and resource specification. Evaluation on a 128-GPU testbed demonstrates up to 24.8% job completion time reduction and ×1.8 Goodput improvement, as compared to representative elastic scaling schemes.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

ACM ISBN 979-8-4007-1286-9/24/11
https://doi.org/10.1145/3698038.3698548

KEYWORDS

Distributed System, Cluster Scheduling

ACM Reference Format:
Xiaoyang Zhao, Siran Yang, Jiamang Wang, Lansong Diao, Lin Qu, and Chuan Wu. 2024. FaPES: Enabling Efficient Elastic Scaling for Serverless Machine Learning Platforms. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3698038.3698548

1 INTRODUCTION

Serverless computing, also known as function-as-a-service (FaaS), has emerged as a new computing paradigm adopted on major cloud platforms such as Amazon Lambda [2], Google Cloud Functions [4], and Microsoft Azure [7]. Serverless computing features ease of computing job deployment, high scalability, and cost effectiveness for users. In a serverless platform, users can run applications without managing infrastructure configurations. Instead, the cloud provider is responsible for allocating resources on-demand for the application (i.e., when function events are triggered) and automatically scaling resources based on resource utilization and changes in workload [20].

With the proliferation of machine learning (ML) services (e.g., training/fine tuning/serving of various deep neural network models [16]), leveraging GPU devices in a serverless platform for model training and inference is in demand. GPU-based serverless platforms [12, 13, 29, 37] often rely on NVIDIA device plugin to expose GPUs for hosting worker containers, and Kubernetes for resource management and job deployment [13]. However, efficiently serving ML workloads in a serverless environment poses a number of challenges.

First, hybrid workload is typically involved in ML services, comprising both offline training jobs (e.g., model developing) and online inference jobs (e.g., recommendation serving) [34]. Deploying them on the shared multi-tenant cluster often leads to performance degradation due to significant resource contention [9]. One common solution is reserving dedicated GPU servers as resource pools for training jobs and inference jobs, respectively. But according to our observations over a productive cluster, resource consumption of online serving requests often follows tidal patterns based on the time of day and workload submitted by users, causing most of the reserved inference servers underutilized. Global management for GPU resources is required to accommodate both types of workloads, maximizing the throughput of training tasks while guaranteeing the service level objectives (SLOs) of inference tasks.

Second, the auto-scaling decisions for ML training jobs should be model performance-aware. In a serverless platform, users rely on the platform to effectively allocate GPU resources to their jobs. Several elastic schedulers have been introduced for distributed machine learning [12, 19, 33]. Optimus [26] leverages predictive models to estimate DNN training throughput under different GPU allocations. Antman [34] adopts dynamic scaling and fine-grained GPU sharing to enhance cluster utilization. Themis [21] achieves finish-time fairness among concurrent jobs by periodically reallocating GPUs. In general, as more GPUs are allocated to a job, the communication overhead grows, and the global batch size may also rise, affecting both throughput and statistical efficiency [27, 38]. There is a need for strategically reconfiguring resources among concurrent jobs and co-adapt their hyperparameters (e.g., global batch size, learning rate) for better overall performance.

Third, monitoring of server resource usage and job demands should be forward-looking. While monitoring tools [6, 8] provide useful server utilization data for informed scaling decisions, current serverless platforms tend to focus only on the current resource utilization when allocating resources to each job [25, 35]. Without knowledge of completion times of existing jobs and the available time of GPU servers for hosting jobs, new job placements face a dilemma - whether to wait for more or better resources or be deployed with insufficient or non-ideal resources. This decision could further influence the scaling of latter arrival jobs, due to potential resource fragmentation or relaunch overhead of resource adjustment. Globally optimal resource allocation thus becomes challenging over long run of the serverless system. In addition, though serverless platforms can track traffic demands for serving applications, there is a lack of monitoring tools for ML training jobs on the training status (e.g., the gradient noise that reflects model convergence), making it incapable for timely resource adjustments.

To address the challenges mentioned above, we propose a **Fa**aS-oriented **P**erformance-aware **E**lastic **S**caling system for effective resource management in a serverless ML platform. FaPES exposes **FaPES-Interfaces** for developers to trigger hybrid ML functions in a serverless platform and enables efficient resource scaling for ML tasks. In FaPES, **FaPES-Manager** divides GPU servers into virtual clusters (VCs) - one for inference tasks and one for training tasks. Servers are dynamically loaned from the inference cluster to the training cluster as needed, and reclaimed back on demand with minimal migration costs. **FaPES-Scheduler** allocates resources for inference jobs to meet SLOs and generates resource schedule plans for training jobs (i.e., GPU allocation over time to fulfill the respective workload), with throughput prediction by a Goodput model under any resource and hyperparameter configurations. **FaPES-Pod** records the training status of a running training job and potentially reconfigures its resources as needed. To predict resource usage, FaPES maintains a **FaPES-Board** that collects the schedule plan for each job and estimates the release time of GPU resources.

In a nutshell, our contributions in designing FaPES are summarized as follows:

▹ We propose FaPES-Manager with the flexible resource loaning ability between VCs for inference and training workloads, and design a built-in mechanism to select GPU servers to be reclaimed for reduced preemption overhead.

▹ A GNN-based performance model is devised for Goodput metric on a given global batch size and a graph abstraction of allocated GPUs. Predicting the performance of an ML training task allows FaPES-Scheduler to co-optimize the hyperparameters and resource configuration, and estimate the completion time of each job to update the GPU occupation time on the FaPES-Board.

▹ We design FaPES-Scheduler with a comprehensive auto-scaling mechanism for hybrid ML jobs. Especially for training jobs, it prioritizes pending jobs and selects a subset of running jobs to perform resource reconfiguration. With forward-looking resource utilization from the FaPES-Board, the schedule plans of training jobs are generated using a primal-dual optimization framework to minimize the average job completion time (JCT).

▹ We implement FaPES on Kubernetes and evaluate its performance on a testbed of 128 GPUs. FaPES outperforms numerous existing scheduling schemes by 24.8% and ×1.8 in terms of average JCT and training efficiency, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Serverless Computing

Serverless computing is a cloud computation paradigm that allows developers to deploy and run applications in the cloud without the need of configuring detailed infrastructure. Developers provide service functions, which are self-contained units of code, such as processing logic for data streaming or neural network models written in PyTorch. The functions are then packaged into containers and deployed on demand when triggered. During the application's lifecycle, auto-scaling is provided to adjust resource configuration as

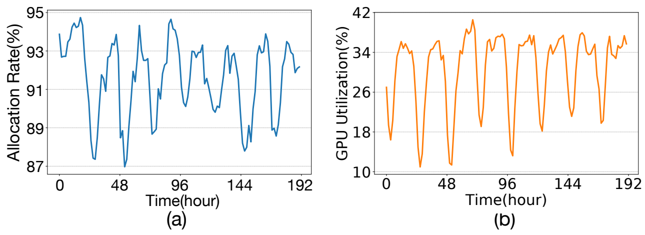


Figure 1: GPU allocation rate and utilization over 192 hours.

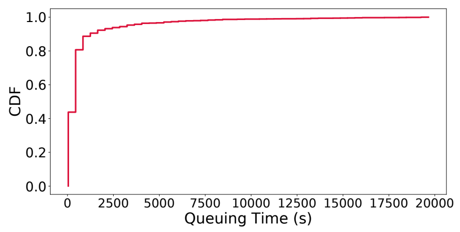


Figure 2: CDF of queuing time in the training cluster.

needed [13]. For conventional serving applications (such as Web services and video streaming) that receive varying request loads, the existing serverless platforms can scale resources in or out as required [35]. For ML training jobs which are resource intensive, it becomes more important to effectively coordinate resources among concurrent jobs for optimized resource utilization and model performance. Unlike on-demand cloud offerings such as AWS EC2 which support customized instance configurations, publicly available serverless platforms have only recently started to support accelerators like GPUs to meet the growing demand for ML training and inference [20].

As compared to classic ML job scheduling architecture [9, 21, 26, 33, 34], FaPES exposes APIs for easy deployment of hybrid ML tasks, designs integration between the job scheduling component (FaPES-Scheduler) and resource management component (FaPES-Manager), and provides job status tracking and hyperparameter auto-tuning.

2.2 ML Applications

Both ML training and inference jobs can be submitted as functions to run in a serverless computing platform. Online inference tasks typically need to meet specific SLOs (e.g., in terms of response latency), which must be guaranteed with sufficient GPU resources. Inference workloads often show strong tidal patterns according to peak and off-peak usage times. Fig. 1 gives the GPU allocation ratio (out of all GPUs) and average GPU utilization per allocated GPU in an ML inference resource pool with 6000 NVIDIA A10 GPUs, on a production system hosting searching services in e-commerce platforms. During peak periods, the GPU allocation rate and utilization reach 95% and 41.2%, respectively. During night time when the workloads subside, these numbers drop to 86.3% and 10%. As this resource pool is reserved for inference services, the time-varying workloads result in a waste of expensive GPU resources. The underutilized resources can well be exploited for training tasks, which typically suffer from extended queuing time, waiting to be run. Fig. 2 shows the queueing times of recommendation training jobs on a resource pool with V100 GPUs. The average queuing time is 1460 seconds, accounting for 13.5% of the lifecycle of a training job; jobs in tail cases could be starved for hours.

Instead of reserving servers for each workload type, it is more efficient for a serverless ML platform to manage all servers as a shared pool. That is, when inference demands surge, resources should be quickly returned from the training pool to meet the serving objectives, while more resources can be used for training jobs during off-peak serving times. The key challenge in such serverless management lies in minimizing the migration and relaunch cost of preempting any running training jobs on these resources. FaPES seeks to dynamically adjust server allocation between training and inference jobs and minimize the preemption overhead by identifying servers with the least impact on running training tasks.

2.3 Performance-aware Auto-scaling

Prior to the rise of serverless computing, users would typically over-reserve resources beyond the actual requirements of their applications, leading to a significant waste of expensive resources [20]. Serverless computing eliminates the need for manual resource configuration, but generally adopts simple, classic scheduling methods for executing functions. For example, AWS Lambda uses a packing strategy to deploy applications [2]. Apache OpenWhisk uses a hashing method to schedule functions within a distributed cluster [1]. Some others [35] rely on Kubernetes's default load balancing scheme, which scores each server based on its resource load. When it comes to ML workloads, GPU allocation greatly affects the training throughput of a training job. Co-located jobs within the same server or under the same network switch can experience varying levels of performance interference.

Auto-scaling on a serverless platform expects an elastic strategy to determine resource configurations of jobs. Most elastic scaling strategies for ML clusters [26, 33, 34] only deal with elastic resource allocation, while users need to provide well-tuned hyperparameters (e.g. batch size, learning rate) before submitting their jobs to the platform. These hyperparameters impact training convergence, whose optimal settings change as a job scales. Pollux [27] and Shockwave [38] co-adapt hyperparameters with the job scale. Pollux introduces "Goodput" - a metric calculated as the product of the

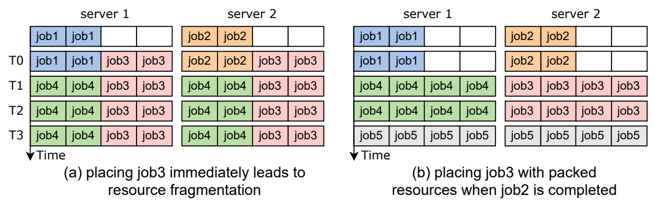

server 1
server 2
job1
job2
job3
job4
job5
T0
T1
T2
T3
Time
(a) placing job3 immediately leads to resource fragmentation
(b) placing job3 with packed resources when job2 is completed

Figure 3: Job3 and job4 arrive at time T0 and T1, respectively. By predicting job2's completion time, job3 and job4 can be better scheduled with packed resources and competed in shorter time.

throughput (trained samples per second) and statistical efficiency (achieved loss convergence progress per sample), to reflect the training efficiency. It solves a linear problem to decide the resource allocations of concurrent jobs for minimized average Goodput. Shockwave extends the throughput model of Pollux and targets the Nash fairness objective. These elastic scaling designs are insufficient for a serverless ML platform.

First, job performance modeling is a key to support effective elastic scaling. Server architecture and data center topology can be intricate in a modern cluster, and GPU placements of an ML job greatly impact the performance of the communication phase of a distributed ML job. Pollux models the training throughput using a linear online fitting model, which may not capture the complex behavior of tensor transmission within a cluster. Other embedded performance models [15, 24, 26] ignore the influence of GPU placements on the communication or the potential overlap between computation and communication phases.

Next, the elastic scaling process may introduce large overhead. Most methods [12, 15, 24, 27] reallocate resources of all concurrent jobs when a scheduling event occurs, which is not affordable (due to preemption and migration overhead) when the demands change all the time. Some allocate available GPUs to jobs with higher priority (e.g., shortest remaining time first) whenever a job is completed [14], but ignore the internal training status of running jobs (e.g., loss convergence, gradient noise).

FaPES employs a GNN-based performance model that encodes resource configuration into a graph and maps it to job performance. We also seek an efficient scheduling mechanism that identifies running jobs with sub-optimal performance based on runtime tracking, and tunes their hyperparameters and optimizes their resource allocation together with new arrival jobs.

2.4 Forward Looking of Resource Usage

Predicting resource usage of running jobs is crucial for making globally optimal scheduling decisions for new arrival jobs. Consider the scenario depicted in Fig. 3, where each

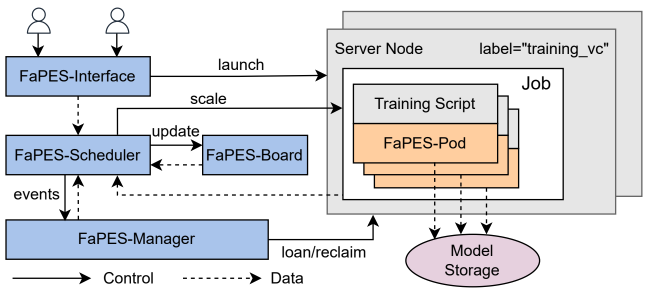


Figure 4: FaPES Overview

server has four GPUs and 2 GPUs are available on each server initially. When a new job (job3) arrives at time T0, there are two scheduling options: 1) immediately assign job3 to the 2 servers, or 2) wait for a dedicated 4-GPU server to become available once job2 is completed. Without forward visibility into future resource utilization, the first, more myopic option is likely to be chosen due to less queuing time. However, this short-sighted resource allocation decision is not globally optimal: it leads to higher cross-server communication overhead in job3 and potential performance interference between co-located job2 and job3; it also causes resource fragmentation for later arrival jobs (e.g., job4 at T1), prolonged job execution time and cluster usage inefficiency (e.g., job5 at T3 is queued). This motivates us to predict each running job's completion time and resource demand over time until job completion, and make informed, better decisions on new jobs' scheduling. As far as we know, none of existing ML scheduling systems consider forward-looking resource usage and make decisions accordingly.

3 FAPES SYSTEM

3.1 Overview

The overall architecture and workflow of FaPES are given in Fig. 4. The architecture follows the Kubernetes' design with serverless compatibility, consisting of five components to enable efficient elastic scaling for ML functions: FaPES-Interface, FaPES-Manager, FaPES-Scheduler and FaPES-Board in the control plane, and FaPES-Pod on GPU servers.

In the cluster, GPU servers are deployed with kubelet and kubeproxy to be visible to FaPES-Manager through kube-apiserver. Servers are divided into two virtual clusters (VCs) and identified by specific server labels - one VC for running training functions and the other for inference functions. FaPES-Interface receives requests from developers and launches each job as a function to perform ML training or inference tasks on the GPU servers. Each worker container is wrapped as a FaPES-Pod. The training status (e.g., gradient noise, remaining epochs) of each training job is collected by FaPES-Scheduler, and the intermediate checkpoints are saved to the centralized Model Storage. FaPES-Scheduler controls resource allocation and auto-scaling of concurrent jobs. The determined schedule plans are collected by FaPES-Board, which predicts future resource usage and provides this information to the FaPES-Scheduler's decision-making process in turn. FaPES-Manager is triggered by FaPES-Scheduler with server loaning or reclaiming events between the two VCs, based on the scaling requirements for each inference job.

3.2 FaPES-Interface

APIs. FaPES-Interface exposes high-level serverless APIs for deep learning (DL) developers to submit training or inference function. A training function includes the following parameters:

- *model*, the ML training model used to identify the corresponding container (e.g., docker) image for model creation and execution.
- *length*, the training workload in terms of the number of epochs of training. It is used as the termination condition for model training.
- *dataset*, the training dataset to use.

An inference function includes the following parameters:

- *model*, the ML inference model. An inference task is completed when the developer terminates it by stopping the corresponding containers.
- *min_scale*, the minimal number of workers estimated by users to meet customized SLOs (latency, throughput, precision, etc.) during peak request demand, based on historical serving records.
- *max_scale*, the maximum number of workers that a function could be scaled up to in case the request demand hits an unexpected peak beyond what has been provisioned for. For example, one might set *min_scale* to 8 given historical estimation while setting *max_scale* to 12, so that the task can tolerate unexpected request traffic or underestimation of the required resources, thereby further improving the serving metrics (e.g., latency, throughput, etc.). It is usually set based on developer experience or task urgency [20].

The above APIs allow a developer to focus on the task, while the system-level resource management and the scaling of each job is handled by other components of FaPES. For inference tasks, we let users provide scaling range for two reasons: *first*, how GPU allocation affects various SLO metrics is more unpredictable than training tasks, as they are sensitive to hyper-parameters and influenced by request traffic demand; *second*, it allows those inference tasks that don't have an SLO value (e.g., not for online serving) to be started, when the minimum number of GPUs are allocated.

Automated Processing. When a function is triggered, FaPES automates pre-processing before launching it to the respective VC. A training function undergoes a profiling stage first, by being assigned to a dedicated server and running there for a few training iterations. This profiling stage gathers essential information about the job's execution timeline, including the order and duration of each operator within one training iteration. The information is used by FaPES-Scheduler to construct the job performance model for scaling decision making. An inference function is assigned a priority according to the request submission time, i.e., the earlier job has a higher priority.

After pre-processing, jobs are pushed into the queues of the two VCs, respectively, and wait for the decision from FaPES-Scheduler to run with allocated resources.

3.3 FaPES-Scheduler

FaPES-Scheduler makes resource scaling decisions every 10 minutes (denoted as one time round).

Scaling for Inference Jobs. FaPES guarantees the SLO of pending inference jobs by satisfying the respective *min_scale* resource requirement set by developers. FaPES-Scheduler allocates and places required GPUs close enough inside the inference VC [3]. As illustrated in Fig. 5, when resources are sufficient to serve concurrent job, FaPES-Scheduler scales jobs up to *max_scale* to available servers in order of their priorities. If there are remaining idle servers after each job has scaled up to *max_scale*, the *loan* event is triggered to FaPES-Manager, which moves the idle servers from the inference VC to the training VC to improve the throughput of training jobs. When there are inference jobs whose *min_scale* cannot be satisfied due to the limited number of available servers, those that have been scaled up are scaled down to conserve resources for *min_scale* fulfillment. If still not all concurrent inference jobs can be served with *min_scale*, the *reclaim* event is triggered and FaPES-Manager moves loaned servers from the training VC back to the inference VC [18]. Both events take one server as the moving unit, to avoid colocation interference between different types of jobs within a server.

Scaling for Training Jobs. For performance-aware scaling of training jobs, FaPES-Scheduler relies on a carefully built performance model to predict the Goodput performance under arbitrary resource configurations and hyperparameters (i.e., batch size and learning rate), to be detailed in **Sec. 4**. The scaling of each training job undergoes three phases (further detailed in **Sec. 5**):

Phase 1 - Filtering: FaPES-Scheduler exploits the performance model to filter out training jobs with sub-optimal performance, which join the rescheduling process with new arrival jobs.

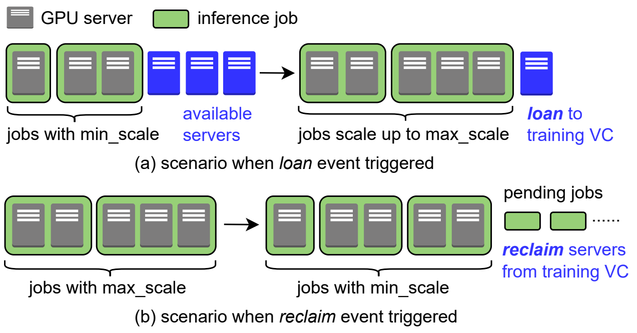

GPU server
inference job
jobs with min_scale
available servers
jobs scale up to max_scale
loan to training VC
(a) scenario when loan event triggered
pending jobs
......
jobs with max_scale
jobs with min_scale
reclaim servers from training VC
(b) scenario when reclaim event triggered

Figure 5: Loan/Reclaim events triggered to FaPES-Manager.

Phase 2 - Prioritizing: Priorities are given to jobs based on well-designed rules.

Phase 3 - Planning: In job priority order, FaPES-Scheduler determines a schedule plan for each job, i.e., GPU allocation and co-adapted hyperparameters over future time periods, based on resource occupation provided by FaPES-Board.

3.4 FaPES-Manager

FaPES-Manager receives events from FaPES-Scheduler and labels GPU server for VC identification accordingly. Servers are relabelled when loaned to the training VC and when reclaimed back to the inference VC. Flexible reallocation of server between the training VC and the inference VC is efficiently enabled by such label switches.

The challenge lies in which loaned servers should be reclaimed back when a *reclaim* event occurs (which specifies N_r servers to be reclaimed). Reclaiming servers involves a series of time-consuming steps: (1) gracefully terminate the containers of the training jobs running on the servers; (2) checkpoint the respective models and upload them to model storage for training recovery; (3) allocate new servers from the training VC to relaunch containers for continuous training. Choosing the right servers with a specific number is a 0-1 knapsack problem [18], where the decisions are binary variables associated with every loaned server, indicating whether to reclaim or not, and the objective is to minimize the total reclaiming overhead. The unique nature of distributed training makes the reclaiming decision for each server dependent. Suppose one job is occupying four servers, when we reclaim one server, the preemption time cost occurs on the remaining three servers as the job is scaled down and its training needs to be recovered. Then we can further reclaim any of the remaining three servers without additional cost.

FaPES-Manager uses two metrics to evaluate each server in reclaiming decision making: *reclaim cost* and *reclaim gain*. The *reclaim cost* is defined to be the number of running jobs on that server; the *reclaim gain* is the total number of servers that can be reclaimed without any additional cost when preempting all jobs on that server. In the example in

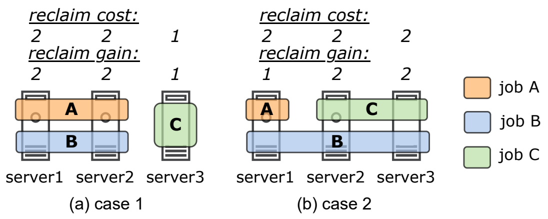


Figure 6: Optimal choices in example scenarios with $N_r = 2$: (a) reclaim server1 and server2 with a total cost of 2; (b) reclaim server2 and server3 with a total cost of 2.

Fig. 6(a), suppose two servers need to be reclaimed ($N_r = 2$) according to the triggered event. Reclaim costs of both server 1 and server 2 are 2, since they are holding job A and job B. The reclaim gain of server 1 is 2, since server 1 and server 2 become available when suspending jobs A and B on server 1. After reclaiming server 1, the reclaim cost of server 2 reduces to 0, because job A and job B are both terminated. Thus, the optimal decision is reclaiming server 1 and server 2 with a total cost of 2.

FaPES-Manager iteratively makes reclaiming decisions in two steps: (1) select a server with the largest *reclaim gain*, but smaller than N_r; when there are multiple candidate servers, choose the one with the least *reclaim cost*. A larger *reclaim gain* means that when suspending jobs running on the server, more additional servers holding the same jobs could become available, which can be reclaimed subsequently as well (due to lowered reclaim costs). Those servers are then selected to be reclaimed and will not join the following iterative steps. (2) update the *reclaim cost* and *reclaim gain* of remaining servers, after suspending jobs on the server chosen in step (1). The iterative selection process terminates when N_r servers to be reclaimed are decided.

4 TRAINING PERFORMANCE MODEL

FaPES-Scheduler relies on the performance model to make scaling decisions for training jobs. For topology-aware scaling, on each server, a device plugin exposes the GPU information and the bandwidth of NVLinks and NICs to FaPES-Scheduler. The intra-server architecture and inter-server connections are exploited in our graph-based throughput model (**Sec. 4.1**).

FaPES-Pod asynchronously calculates the gradient noise of each training job and reports them to FaPES-Scheduler using RPC. The training status information is exploited by the statistic efficiency model to predict the Goodput (**Sec. 4.2**).

4.1 GNN-based Throughput Model

We first analyze how the global batch size and GPU allocation affect the training throughput. The time of one training iteration is split into the computation phase and communication phase. We estimate the computation time based on the profiling results after automated processing of FaPES, and model the communication time based on the interconnect graph among allocated GPUs and a graph neural network (GNN) model.

Computation Phase. During the profiling stage, a data-parallel job run on a dedicated server with 2 workers, each occupying one GPU. The local batch size of each worker is m^*, and all-reduce communication is performed for gradient synchronization between both workers. We collect the timeline of each training iteration including the total computation duration T^*_{comp} and the precedent computation operators before launching each communication kernel. For actual model training on allocated resources in the training VC, we estimate the actual computation cost as follows, where M is the global batch size and V_{gpu} is the set of allocated GPUs to the training job ($|V_{gpu}|$ is the number of workers involved):

$$T_{comp} = \frac{M}{|V_{gpu}|m^*} T^*_{comp}$$

as the computation time grows linearly with the batch size [27].

Communication Phase. Conventional communication cost models estimate communication time by dividing total communicated data size by the aggregated bandwidth [12, 26, 27], which do not capture the complex intra-connections and inter-connections of GPU servers. For example, within a DGX-1 server of 8 GPUs, the locations of allocated GPUs decide whether NVLinks or PCIe links are used in inter-GPU paths [31]. The whole cluster often adopts a hierarchical fat-tree topology to connect GPU servers [12]. Behavior of communication libraries is not negligible either, e.g., NCCL chooses between ring or tree communication topologies, and determines the chunking protocol and the number of parallel channels based on the communicated data size and GPU allocations. We use a GNN model to capture the inter-GPU topology, learn the behavior of the NCCL library and predict the communication time. We collect over 5,000 training samples using NCCL benchmarks by varying GPU allocations and tensor sizes to avoid overtuning. This prediction is generic to different workloads, since ML jobs of different models widely adopt NCCL as backend [32].

The complete cluster topology can be represented by a graph $G = (V, E)$ (an example in Fig. 7(a)). $V = \{V_{gpu}, V_{cpu}, V_{sw}\}$ is the set of nodes containing GPUs, CPUs, and network switches. $E = \{E_{nv}, E_{pcie}, E_{net}\}$ is the set of links among different nodes. GPUs on the same server are inter-connected with NVLinks and each GPU is connected to its affinity CPU on the server with a PCIe link. If one CPU has an affinity NIC, there is a link in the graph abstraction connecting the CPU

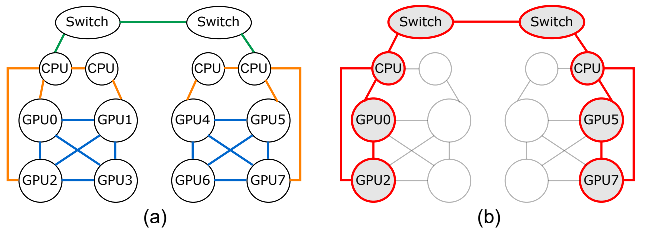


Figure 7: (a) Graph abstraction of the cluster: blue lines - NVLinks, orange lines - PCIe links, green lines - network links. (b) Extract subgraph when allocating GPUs 0, 2, 5, 7 to a training job.

to the switch that the NIC is connected to, whose bandwidth is the NIC bandwidth capacity.

We consider the following features for each GPU node $v \in V_{gpu}$:

$$h_v = (\overrightarrow{type}, data\_size, data\_type, cpu, mem)$$

$\overrightarrow{type}$ is a one-hot encoding of GPU types in the cluster; the other four values indicate communicated data size, tensor data type (e.g., int8, fp16, fp32), number of available CPU cores and memory bandwidth on its affinity CPU, respectively. These features reflect the traffic volume and the transmission capacity of hardware. NCCL takes this information as input to select the communication topology from ring and tree, determine the chunking protocols, and set the number of parallel channels. For each CPU node in V_{gpu} or switch node in V_{sw}, the node features are zero vectors, as they do not produce tensor data but join the feature aggregation process. The edge feature e_{vu} between two nodes v and u is a scalar representing the bandwidth capacity in-between, normalized against the maximal link speed in the cluster.

For any given set of allocated GPUs (e.g., to run a training job), we can extract a subgraph G_{sub} connecting those GPUs (among which collective communication is run to synchronize gradients). An example subgraph is given in Fig. 7(b). A K-layer GCN model is used to learn node embeddings from the subgraph constructed. Node feature aggregation is done on the subgraph only. In convolutional layer $k+1$, a node v in the subgraph aggregates layer-k representations of the nodes in its neighborhood set $\mathcal{N}_v$ in the subgraph. To capture the edge information, edge features are used as the weights in aggregation:

$$h_v^{k+1} = \sigma\big(b^k + W^k \sum_{u \in \mathcal{N}_v} \frac{e_{vu}}{c_{vu}} h_u^k\big)$$

where σ is the non-linear activation function, b^k and W^k are learnable parameters of layer k, and $c_{vu} = \sqrt{|\mathcal{N}_v|}\sqrt{|\mathcal{N}_u|}$ is the product of square roots of node degrees. The rationale behind using the square root of node degrees is to prevent larger feature values of high-degree nodes (e.g., one Top-of-Rack switch could connect dozens of servers) from dominating the convolution operation. The final representations obtained at layer K are node embeddings of the subgraph, used for generating the subgraph-level embedding.

We further conduct three-level pooling of node features embeddings obtained from GCN modules to capture the subgraph structure. Each CPU node $v \in V_{cpu}$ aggregates feature embeddings of its affinity GPU nodes. At each switch, feature embeddings of attached CPU nodes are aggregated. Then at the cluster level, all aggregated feature embeddings of the switches are further aggregated. The final feature embedding vector of the subgraph is passed through a linear neural network layer, producing a scalar value as the communication time estimation, T_{comm}.

By utilizing GNNs, the performance model can learn how NCCL reacts and chooses the communication strategies (e.g., topology, chunking protocol, channel number) under different communication requests and GPU inter-connections.

To learn the model, we collect actual communication time samples of the NCCL kernel on various subgraphs, by varying GPU allocations and communicated data properties (e.g., data size, data type).

Iteration Time. We insert the estimated communication time of each communication operator into the computation timeline, according to profiled communication triggering points. Since computation and communication are overlapped in most cases, duration of the critical path in the timeline of one training iteration is regarded as the iteration time. Given GPU allocation subgraph G_{sub} and the global batch size M, training throughput is estimated as:

$$Throughput(G_{sub}, M) = \frac{M}{CriticalPathTime(T_{comp}, T_{commu})}$$

4.2 Goodput Performance Metric

Statistic Efficiency. We use the metric in Pollux [27] to model statistic efficiency of a training job, interpreted as the training progress in terms of convergence achieved per sample. Let $\phi = \frac{tr(\Sigma)}{|R|^2}$ be the gradient noise scale (GNS) observed from FaPES-Pod, which says that the noise scale is equal to the sum of the variances of the individual gradient components, divided by the global norm of the gradient R [22]. Statistical efficiency under global batch size M can be computed relative to the smallest batch size $M_0 = 1$:

$$Efficiency(\phi, M) = \frac{\phi + M_0}{\phi + M}$$

Goodput. We formulate the Goodput performance of a training job as the product of throughput and statistic efficiency, which indicates the training progress achieved per second:

$$Perf(G_{sub}, \phi, M) = Throughput(G_{sub}, M) \times Efficiency(\phi, M)$$

This performance model is used by FaPES-Scheduler to estimate the training performance under any GPU allocation and global batch size, and find the best global batch size and GPU allocation that maximizes the Goodput of a job. The learning rate is linearly scaled with the batch size change [27].

5 AUTO-SCALING DESIGN

We next present the detailed mechanism for FaPES-Scheduler to scale training jobs. When resource allocation to a job is decided, FaPES-Scheduler controls join and leave of GPU workers; FaPES-Pod detects these events and relaunches training jobs efficiently with adjusted hyperparameters.

5.1 Filtering Running Jobs

FaPES-Scheduler selects running training jobs for resource adjustment upon the start of a time round.

Inefficiency of Strawman Selection. One strawman approach is to select running jobs with poor performance achievement, defined as the average Goodput performance achieved over time rounds in its current schedule plan (detailed definition in Sec. 5.3) divided by its optimal performance according to the performance model (obtained by grid search). Jobs whose performance achievement is lower than a threshold are selected for rescheduling. However, several issues arise for such a strategy. *First*, jobs that will be completed soon could be selected, resulting in delayed completion or even job starvation after rescheduling. Consider a scenario in Fig. 8(a). In the current time round T_0, 8 GPUs are occupied by four distributed training jobs, and job4 and job5 are waiting in the queue. Job0 is estimated to complete in T_1 and job2 will scale up in T_1 according to their respective schedule plan. If job0 is under poor performance achievement and selected to be rescheduled, it will have a larger completion time or even be starved in the queue. *Second*, the resource recycled from the selected jobs may not be enough to serve pending jobs, hurting overall performance. For example, if job1 is selected, only 1 GPU becomes available for job4 and job5 in the queue. *Third*, it is also possible that all running jobs achieve good performance, preventing admission of new jobs for potential global performance improvement.

Dynamic Filtering. To avoid these problems, we design the job selection following two rules: (1) *Remaining time awareness.* Jobs to be completed in the next time round are not selected for rescheduling, to avoid increasing their total execution time. (2) *Dynamic selection instead of a deterministic threshold.* FaPES-Schedule recycles resources of running jobs in increasing order of their performance achievement, until better average performance achievement is estimated when allocating the recycled resources to all jobs to be (re)scheduled (new arrival jobs and running jobs selected

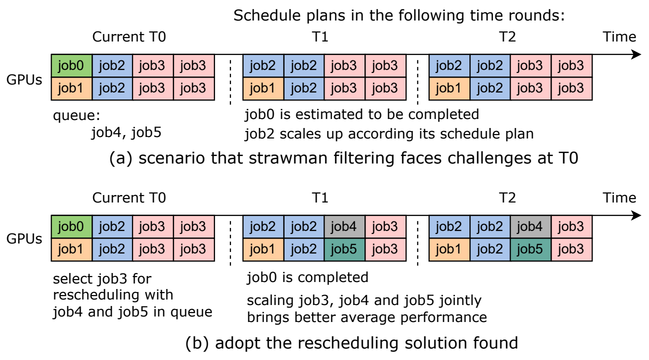

Schedule plans in the following time rounds:
Current T0
T1
T2
Time
GPUs
job0
job1
job2
job3
job4
job5
queue:
job4, job5
job0 is estimated to be completed
job2 scales up according its schedule plan
(a) scenario that strawman filtering faces challenges at T0
select job3 for rescheduling with job4 and job5 in queue
job0 is completed
scaling job3, job4 and job5 jointly brings better average performance
(b) adopt the rescheduling solution found

Figure 8: Filtering running jobs for rescheduling at T0.

to be rescheduled). To quickly estimate whether recycled resources can lead to better overall performance, FaPES-Schedule allocates recycled resources evenly among jobs to be (re)scheduled, estimates their performance achievements, and compares the average performance achievement with the current average of selected running jobs. If there is no potential performance improvement over existing resource allocations of running jobs, no running jobs will be selected for rescheduling. In Fig. 8(b), job3 is selected to be rescheduled with jobs in the queue for better overall performance. As job arrivals change over time in a severless platform, this dynamic selection enables efficient resource reconfiguration as needed.

5.2 Prioritising Jobs

In each time round, we (re)allocate GPUs to running jobs that are selected to be rescheduled and new arrival jobs waiting in the queue, referred to as a *scheduling cohort*. Optimal resource allocation to the scheduling cohort that minimizes the average job completion time is commonly a NP-hard problem (Sec. 5.3), with a solution space growing exponentially with the size of the cohort. Prioritizing jobs in resource allocation helps reduce the solution complexity, to be linear to the number of jobs in the scheduling cohort.

Priority Assignment. FaPES-Scheduler prioritizes jobs following three rules: (i) new arrival jobs have higher priorities over running jobs, to reduce job queuing time (practically, developers may like to have their jobs running first, and then scaled up when available resources allow); (ii) new arrival jobs are prioritized in decreasing order of the queueing time, i.e., higher priority to longer-waiting jobs (e.g., those not scheduled in the last few rounds); (iii) among running jobs, jobs with lower performance achievement have lower priority than those with higher performance achievement.

Starvation Avoidance. After a running job is selected to be rescheduled, the job may wait in the queue due to its lower priority than new arrival jobs. To avoid starvation, FaPES-Scheduler uses *pop up* operations to give the highest priority to such old jobs that have been waiting for a long time (i.e., 3 time rounds in our experiments).

5.3 Scheduling Jobs

In each time round, FaPES-Scheduler makes a schedule plan for each job j in the current scheduling cohort J. A schedule plan is described by a set of binary decisions $\{x_j^g(t)\}$ over future time rounds, indicating whether job j has a worker placed on GPU g in time round t, for all $g \in \mathcal{G}$ ($\mathcal{G}$ is the set of all available GPUs), $t \in T$ (T is the set of time rounds starting from the current one). We assume no GPU sharing among different jobs.

Schedule Planning Problem. Let L_j be the set of feasible schedule plans for job j. Binary variable y_{jl} indicates whether the schedule $l \in L_j$ is chosen for job j or not. d_{jl} is the completion time of job j when choosing schedule l. $v_{jl}(t)$ is the training throughput of job j in time round t predicted from the performance model of the job when choosing schedule l, i.e., given its GPU allocation at t, the corresponding throughput when tuning the global batch size M to achieve the best performance. Let S be the set of all servers in the serverless cluster, and $\mathcal{G}_s$ be the number of available GPUs on server s. F_j is the maximal number of training epochs of the job, specified by the *length* argument in the job function; N_j is the total number of data samples per training epoch according to the *dataset* in use. $\hat{t}$ denotes the duration of each time round (600 seconds in our experiments). The schedule planning problem that FaPES-Scheduler solves in each time round can be formulated into the following optimization problem, aiming at minimizing the completion time of all jobs in the scheduling cohort:

$$\min \quad \sum_{j\in J}\sum_{l\in L_j} y_{jl} d_{jl} \tag{1}$$

$$\text{s.t.:} \quad \forall j \in J: \sum_{l\in L_j} y_{jl} \le 1 \tag{2}$$

$$\forall t \in T, \forall s \in S: \sum_{j\in J}\sum_{l\in L_j}\sum_{g\in\mathcal{G}_s} y_{jl} x_{jl}^g(t) \le \mathcal{G}_s \tag{3}$$

$$\forall j \in J: \sum_{t\in T} \hat{t} v_{jl}(t) \ge F_j N_j \tag{4}$$

$$\forall j \in J, l \in L_j: y_{jl} \in \{0,1\} \tag{5}$$

(2) specifies that at most one schedule plan is chosen for each job. (3) ensures that the number of allocated GPUs on a server is less than the server capacity. (4) guarantees completion of each job's training workload with the chosen schedule plan: in each time round, the finished training workload equals the product of the time duration $\hat{t}$ and the training throughput $v_{jl}(t)$; the total number of trained samples over the scheduled time rounds should be no smaller than the total workload $F_j N_j$.

d_{jl}, $x^g_{jl}(t)$ and $v_{jl}(t)$ are given in each feasible schedule plan l, and the decisions to solve are y_{jl}'s. The number of decision variables increases exponentially with the numbers of servers and jobs. We design an efficient algorithm to derive good solutions in polynomial time, exploiting job priorities and the primal-dual optimization framework [39]. We relax integrity constraint (5), associate dual variable u_j and $\alpha_s(t)$ with constraint (2) and constraint (3), respectively. We then obtain the dual problem of the relaxed optimization problem, where the complicating resource constraint (3) (due to resource sharing among jobs on the same server) is decomposed into the following:

$$\forall j \in J, l \in L_j : u_j \le d_{jl} + \sum_{t\in T}\sum_{s\in S}\sum_{g\in\mathcal{G}_s} \alpha_s(t) x^g_{jl}(t) \tag{6}$$

$\alpha_s(t)$ can be interpreted as the unit resource price for allocating one GPU in server s in time round t. Then the RHS of (6) can be regarded as the total resource price of adopting the schedule l for job j, which should be minimized to minimize the dual objective (and hence optimize the primal objective according to duality) [39].

Making schedule plans. We propose a primal-dual optimization approach to minimize the RHS of (6) and obtain the schedule plan of individual jobs. Iteratively, we compute the best schedule for each job according to the current dual price values ($\alpha_s(t)$), and then estimate the resulting load on each server s and updates the dual prices $\{\alpha_s(t)\}$ accordingly.

Best schedule. Observe the RHS of (6): if we enforce a completion time $d_{jl} = \tau_j$ for job j, the best schedule for job j should achieve the minimal total resource price $P(\tau_j, F_jN_j)$ accumulated over time rounds before the enforced completion time τ_j.

$$P(\tau_j, F_jN_j) = \sum_{t\in T}\sum_{s\in S}\sum_{g\in\mathcal{G}_s} \alpha_s(t) x^g_{jl}(t)$$

To compute the schedule plan achieving this minimal total resource price, we use dynamic programming to decide the required training workload in each time round:

$$P(\tau_j, F_jN_j) = \min_{w\in[0,F_jN_j]} P\_t(\tau_j, w) + P(\tau_j - 1, F_jN_j - w)$$

where $P\_t(\tau_j, w)$ represents the minimal resource price incurred to finish workload w during time round t. We compute $P\_t(t, w)$ by greedily allocating GPUs on servers with the smallest price $\alpha_s(t)$.

Specifically, for the first time round of a job with a new schedule plan (new arrival or rescheduled), we gradually allocate GPUs on servers in increasing order of their prices $\alpha_s(t)$ until the required training workload of the job in t can be completed on the allocated GPUs, according to throughput prediction with the performance model. If several servers

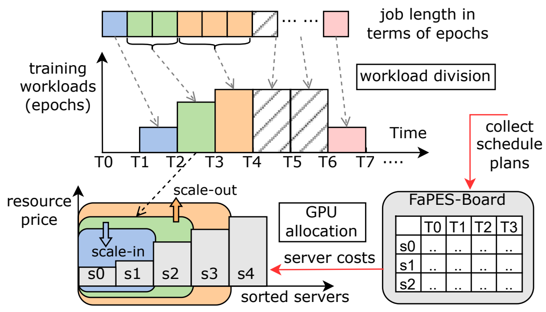


Figure 9: Compute GPU allocation with least overall resource price, when enforcing a job completion time T7.

have the same resource price, we prefer ones that could bring more performance gain, as guided by the performance model. For later time rounds of the job, GPU allocation is determined based on GPU placements of the job in the previous time round $t-1$, in order to reduce migration overhead. If the existing GPU allocation (aka allocation in the previous round) can finish a larger training workload than the required in t, we gradually recycle GPUs allocated to the job (i.e., scaling down) from servers with the highest prices until the remaining GPU allocation can just fulfill the required training workload; if the existing allocation cannot fulfill the required workload in t, we gradually add more GPUs to the job (i.e., scaling up) on servers with lowest prices.

Take Fig. 9 as an example, to complete more training workload in T3 (than T2), the job chooses to scale up with server S3 joined (servers are sorted with prices). Practically, the minimal allocation unit within each server is 2 GPUs under the same PCIe switch, for the purpose of reducing resource fragmentation and mitigating PCIe interference between different jobs. Jobs in the scheduling cohort are allocated in order, based on predefined priorities.

We enumerate completion times τ_j, compute GPU allocation and overall resource prices accordingly, and choose the one with the least overall resource price as the best schedule plan for job j.

Price updates. The schedule plan made for each job is collected by FaPES-Board and the load (i.e., GPU occupation) on each server is updated. We update the resource price of each server according to the server load, such that a higher load incurs a larger cost, for cluster-wide server load balance and resource interference mitigation among concurrent jobs. Especially, we update the server resource price to $H(\frac{U}{H})^{load/capacity}$, where U and H are upper bound and lower bound of completion time among all jobs, respectively, load and capacity indicate the occupied and total numbers of GPUs on the server. In this way, we unify the arithmetical unit of price with the completion time in (6). We obtain bound values by allocating one GPU and all GPUs to jobs in advance and predicting the corresponding makespan with the throughput model.

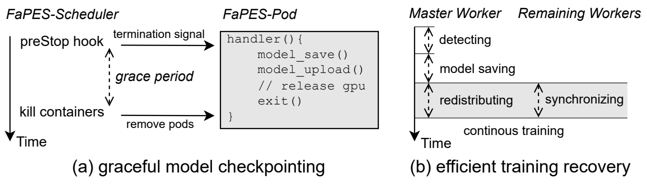
Whenever a job is determined with a schedule plan, it will be collected by FaPES-Board (as shown in Fig. 9) to update the resource occupation and prices over future time rounds, which are provided as the forward-looking information for FaPES-Scheduler to make schedule plans for the following jobs.

Compensation for Estimation Error. FaPES-Scheduler relies on the performance model to predict the training throughput and co-adapt training hyperparameters. To track the actual training progress and correct the estimation error, FaPES-Scheduler collects the actual remaining workload of each job from FaPES-Pod and redistributes the remaining workload among future time rounds when a job is to be rescheduled. It is possible that a job will not be rescheduled along its life cycle and the accumulated estimation error may cause two situations. First, if the job is completed earlier than the scheduled time, GPUs are wasted since the resource board makes them available only when a job's schedule plan has been done. To counter this, we let FaPES-Scheduler notify FaPES-Board to recycle the resources, once it identifies the job has been completed (no more remaining samples to train). Next, if the actual training takes more time than the predicted, when the planned completion time is reached, FaPES-Board could have reported availability of the job's GPUs to FaPES-Scheduler, leading to GPU conflicts. In such a case, we allow the job to maintain its current GPU allocations until the tail samples are trained and retain the GPU occupation status in FaPES-Board.

6 FAPES IMPLEMENTATION

We implement FaPES prototype using Python with 4,000+ LoC to be compatible with k8s of version 1.22. Developers submit jobs with a specific tag on the scheduler field, indicating that they will not be scheduled by the default k8s scheduler. FaPES-Board is implemented as a daemon process on the same server as FaPES-Scheduler, it receives update and fetch requests from FaPES-Scheduler and maintains a table recording the load and price of each server. FaPES-Scheduler achieves the placement of FaPES-Pod onto GPUs by binding specific servers with affinity, using the k8s client APIs. When co-adapting the hyperparameters, FaPES passes them as a key-value format in environment variables, which can be read by k8s containers and exploited for training. For running jobs that need to be scaled in or scaled out, FaPES optimizes model checkpointing and training recovery procedures as follows.

First, instead of periodically checkpointing model parameters, we enable each job to checkpoint its model and upload



```
handler(){
    model_save()
    model_upload()
    // release gpu
    exit()
}
```

Figure 10: FaPES-Pod supports efficient scaling.

it to the Model Storage when its containers are to be terminated (e.g., resources preempted by new jobs or claimed back by the inference VC). To ensure graceful termination, FaPES-Scheduler sets up a preStop lifecycle hook for each container as shown in Fig. 10(a). When FaPES-Scheduler decides to stop a job, a termination signal is triggered to FaPES-Pod, whose handler function will upload the model within a grace period (15 seconds in our setting) and release resources. After the grace period, the container is removed successfully.

Second, when FaPES-Scheduler decides to scale a job, it removes workers from or adds new workers to the job, instead of relaunching the whole job. As illustrated in Fig. 10(b), FaPES-Pod on the master worker relies on a Rendezvous handler to detect worker joins and leaves. When a scaling event is detected by the master worker, distributed training is interrupted and the master worker in the job uploads the current model parameters to Model Storage, which will be fetched by the remaining workers for model parameter synchronization. The data loader and rank information of all workers are then redistributed based on the new hyperparameters and number of workers. This distribution process can be fully overlapped in time with model synchronization, allowing training to be efficiently recovered in a short time.

7 EVALUATION

7.1 Experimental Settings

Testbed. We evaluate FaPES on a serverless cluster containing 16 servers and 128 GPUs in total. Each server is equipped with Intel Xeon 8163 CPU@2.50GHz with 96 cores, 8 Nvidia V100 GPUs with 300GB/s NVLinks, 16 lanes of 16GB/s PCIe3.0 bus, and a Mellanox 200Gbps NIC. Linux-4.19.91 OS and Kubernetes 1.22 are installed for container management. The servers are inter-connected by three switches: one switch is connected to 6 servers, the second switch is connected to other 6 servers, and the third switch is to the rest 4 servers. The switches are interconnected with 400GbE high-bandwidth links.

Baselines. We compare FaPES-Manage against two baseline strategies for resource loaning between inference VC and training VC: (1) Dedicated partition - N_r^{max} servers are reserved exclusively for inference workloads in the shared cluster, where N_r^{max} is the maximal number of servers required to meet SLOs of the inference workloads (i.e., the maximal number of the sum of *min_scale* of inference jobs); the remaining servers are dedicated to the training workload; (2) One-way resource loaning - only servers loaned from inference VC to training VC will be reclaimed back for inference workload when needed.

We compare FaPES-Scheduler with the following representative scheduling strategies for ML clusters:

- **First-in-First-out (FIFO)**. The training functions are pushed into the tail of the job waiting queue, and the system fetches from the head of the queue for training job execution. Elastic scaling is not supported. The number of GPUs is set from 4 to 64, the learning rate is set to 10^{-4}, and the local batch size is set from 1 to 128.
- **Shortest Job First (SJF)**. It prioritizes jobs with shorter training length, and always executes jobs in the current queue with the shortest training length first. Elastic scaling is not supported. Number of GPUs and hyperparameters are set the same as FIFO.
- **Pollux** [27]. It co-adapts hyperparameters of each training job with allocated GPU resources, using a built-in Goodput performance model. It employs a population-based search algorithm to find optimal GPU allocations among all concurrent jobs, that maximize overall Goodput. In each time round, all running jobs are terminated, and GPU resources are completely reallocated based on the optimization results.
- **ElasticFlow** [12]. It is a serverless framework that supports elastic scaling for ML training jobs. Each job has a specific deadline that must be met. ElasticFlow has three modules: (1) admission control - it accepts jobs that are estimated to be able to meet the deadlines with available GPUs; (2) resource allocation - it greedily allocates GPUs to jobs with higher marginal throughput gain; (3) placement - a packing strategy is used to reduce the communication cost. To implement it in our system, admission control only accepts new jobs whose initial GPU demand can be satisfied; in each time round, resource allocations to all concurrent jobs are reconfigured.
- **Optimus** [26]: It reschedules all concurrent training jobs in each time round based on an online fitting throughput model. It first allocates at least one GPU for each running job to avoid starvation, and then gradually allocates additional GPUs to the job with the largest marginal throughput gain. It employs a packing strategy to place allocated GPUs of each job together.

Workloads. We collect training and inference workload traces from a production serverless ML cluster. We extract the tidal pattern of inference requests from the productive cluster (Fig. 1) for inference job injections in our experiments. The inference requests contain searching and recommendation models, and the resource demand varies as the min_scale and max_scale parameters of inference jobs.

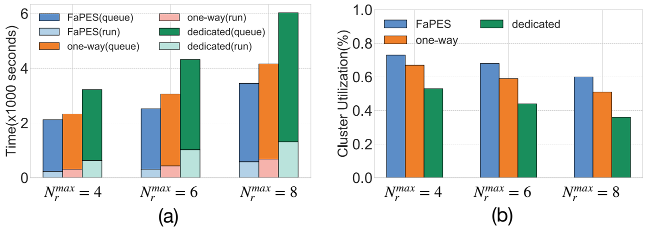


Figure 11: Performance of resource loaning strategies.

We use the observed arrival pattern (i.e., the number of submitted requests per minute) to inject training jobs in our experiments as well. Each training job function specifies: (1) the DNN model, which is randomly chosen from four models, ResNet50 [17], VGG16 [30], GPT-2 [28], and ViT [10]; (2) length, set from 100 epochs to 200 epochs (for resource conservation); (3) dataset, ImageNet for vision tasks ResNet50, VGG16 and ViT and a personal chat dataset for natural language processing tasks GPT-2. Both datasets are down-scaled for resource conservation. We train the models using PyTorch 1.12 with CUDA 11.3 and NCCL version 2.14.

7.2 FaPES-Manager: Resource Loaning

We first evaluate how our resource loaning design between inference VC and training VC benefit the cluster throughput. By varying N_r^{max} and proportionally scaling resource demands of inference jobs, we evaluate FaPES-Manager under different levels of inference request traffic. To focus on inspecting performance gains of resource loaning, we inject but do not perform elastic scaling for the training workloads in this experiment. The number of GPUs is set from 4 to 64, the learning rate is set to 10^{-4}, and the local batch size is set from 1 to 128, according to experiences.

Fig. 11(a) reports the average queuing time and completion time of training jobs under different resource loaning strategies. Compared to the baselines, FaPES reduces the queuing time of training workloads by 14.7% to 69.6%, and job completion time by 9.0% to 42.8%. Fig. 11(b) further shows that FaPES improves the overall cluster utilization (the number of occupied GPUs divided by the total number of GPUs) by a factor of ×1.09 to ×1.67. With its dynamic resource loaning between inference VC and training VC, under-utilized servers can be fully utilized for training jobs waiting in the queue, leading to reduced queuing times compared to the dedicated server partition baseline. Additionally, with FaPES's selective server movement from training VC to inference VC for inference workload, less training job preemption and relaunching are incurred, contributing to reduced job completion times compared to the one-way resource loaning baseline.

7.3 FaPES-Scheduler: Elastic Scaling

We next focus on comparing FaPES-Scheduler's performance on scheduling training jobs, and use all 128 GPUs for training jobs.

Cluster Utilization. In Fig. 12, we observe FaPES achieves up to 51.5% improvement in cluster utilization, compared to non-elastic scheduling baselines. When compared to elastic scheduling baselines, FaPES demonstrates a 6.5% improvement, except Optimus. Optimus tends to launch as many jobs as possible by assigning one GPU to each job first. This greedy scheme results in the highest GPU allocation rate, leading to better cluster utilization than others. Nonetheless, when both inference and training jobs are run in the cluster, FaPES-Scheduler can still achieve the best cluster utilization, together with resource management with FaPES-Manager, to be shown in Sec. 7.5.

Job Completion Time. In Fig. 13(a), FaPES achieves an average job completion time (JCT) reduction of 42.3% to 47.9%, as compared to non-elastic scheduling baselines. The tail jobs (e.g., 95th percentile) incur significantly longer completion times with the baselines, up to 2.8X longer than FaPES's 95th percentile jobs. With the non-elastic baselines, jobs are more likely to be queued and starved, as they prioritize running jobs with large GPU demands. FaPES outperforms the elastic scheduling baselines by 10.2% to 24.8%, attributed to several key factors: (i) Unlike other elastic methods that require extensive resource reconfiguration across all concurrent jobs, FaPES tracks runtime performance of running jobs and only adjusts resources for selected scheduling cohort, incurring less job relaunching overhead; (ii) FaPES employs a more sophisticated, topology-aware throughput model that captures communication patterns and behavior across the cluster, helping better scheduling decision making; (iii) FaPES allocates GPUs based on training workload requirements in future time rounds, with less GPU resources allocated to a job predicted to complete soon and effective resource allocation to jobs with more needs.

Queuing Time. In Fig. 13(b), Optimus incurs no queuing time as all jobs are assigned with one GPU first. FaPES reduces job queuing time by 20.8% to 86.4% as compared to other baselines, due to its design property and starvation avoidance to avoid queuing jobs for an unexpected time.

Statistic Efficiency. Except Pollux, the existing scheduling schemes primarily focus on the throughput impact when scaling jobs, but neglect the training efficiency implications. FaPES co-adapts the global batch size to ensure the best Goodput at each job scale. As shown in Fig. 13(c), FaPES demonstrates a 1.4× to 1.8× improvement in average Goodput, compared to other baselines. Compared to Pollux, FaPES achieves a 10.6% Goodput gain. This advantage is enabled by FaPES's more comprehensive performance model, which is aware of the effects of GPU placements on training efficiency. When setting the schedule plan, FaPES allows jobs to be queued opportunistically, letting other running jobs complete first if their remaining workloads are not much. This reduces contention among many concurrent jobs, enabling global optimal resource utilization and training efficiency.

Table 1 summarizes the detailed performance statistics of each completed training job.

7.4 System Overhead

We next report the overhead existing in the system workflow.

Scheduling delay. The scheduling delay is collected as the duration from the start of a time round to when FaPES-Scheduler has made schedule plans for jobs in the current scheduling cohort. Fig. 14(a) shows that FaPES-Scheduler is able to make scheduling decisions within 580ms in 90% of the cases. This shows the efficiency of FaPES's scheduling algorithm, whose complexity is only linear to the number of selected jobs in the scheduling cohort.

Negotiation delay. The negotiation delay between FaPES-Pod container and FaPES-Scheduler for gradient noise data over RPC is measured as the duration from the start of sending the training status update from the container to when feedback from FaPES-Scheduler is received back at the container. Fig. 14(b) reveals that the RPC negotiation can be done within 2ms among 90% of collected time samples, indicating that FaPES-Scheduler can efficiently track training status from FaPES-Pod containers in a timely manner.

Migration overhead. The auto-scaling process of a job involves several steps: model parameters need to be saved in a checkpoint format and uploaded to Model Storage; then when the job is restarted, it undergoes container creation (assuming the container image is cached locally) and the model is downloaded from Model Storage. Table 2 gives the average time spent on each of these phases during VGG16 model training (with a model size of 528MB). The overhead is acceptable as compared to the running time of jobs.

Prediction accuracy. During our experiments, we measure training jobs' actual communication time and compare the actual time with estimated communication time using our performance model. We collect more than 5,000 data samples for training and the average error is 36%, which stems from two factors: (i) NCCL uses GPU kernels for communication, contending with training kernels in terms of GPU SM and memory bandwidth; (ii) cross-traffic generated by concurrent jobs in the shared cluster could cause the network volatile and unstable. In FaPES, we compensate for accumulated estimation errors and avoid any GPU conflict or waste, using our design at the end of Sec. 5.3.

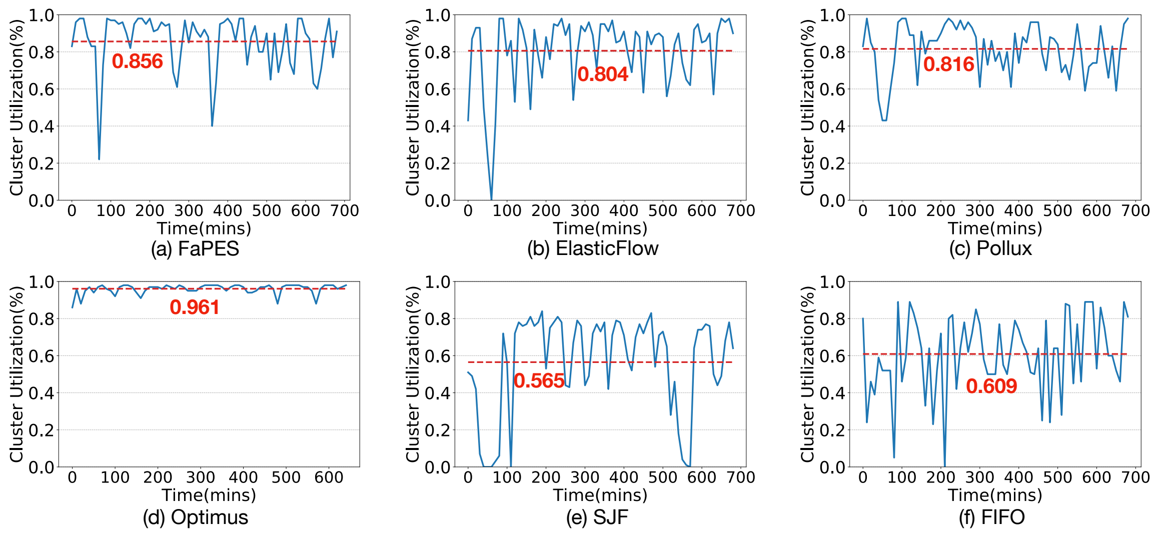


Figure 12: Cluster utilization over time under different scheduling strategies.

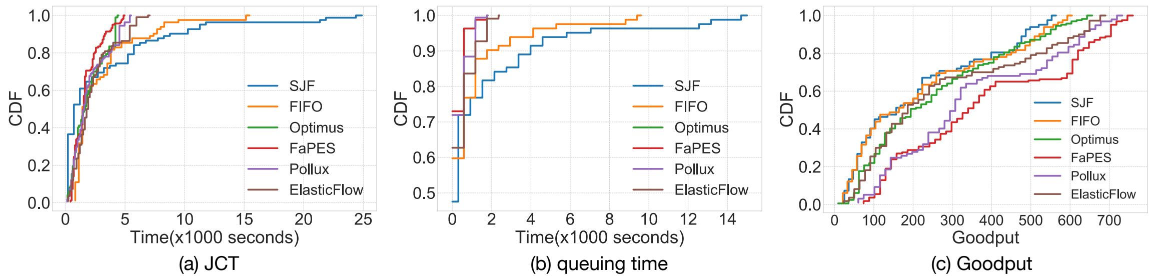


Figure 13: Cumulative distribution function (CDF) of job performance statistics under different scheduling strategies.

Table 1: Detailed job performance statistics.

	JCT ($\times$1000 seconds)				Queuing time ($\times$1000 seconds)				Goodput			
	avg.	**min.**	**max.**	**95th.**	**avg.**	**min.**	**max.**	**95th.**	**avg.**	**min.**	**max.**	**95th.**
FaPES	1.76	0.43	4.97	4.03	0.19	0.00	1.80	0.60	387.86	74.04	760.29	718.01
ElasticFlow	2.34	0.24	7.07	5.75	0.37	0.00	2.40	1.80	277.54	23.00	690.49	655.58
Pollux	2.07	0.14	5.52	4.99	0.24	0.00	1.80	1.20	350.47	60.56	733.33	674.24
Optimus	1.96	0.17	4.46	4.24	0.00	0.00	0.00	0.00	257.31	9.13	656.49	582.87
SJF	3.38	0.21	24.89	11.56	1.40	0.00	15.00	5.94	215.87	25.23	564.27	523.41
FIFO	3.05	0.82	15.45	8.40	0.78	0.00	9.60	4.14	226.25	20.98	605.04	567.89

Table 2: Time overhead in job migration.

Phase	**Time Spent (ms)**
model checkpointing	1580
model upload	402
model download	140
container creation	3558

7.5 Large-scale Simulation

To evaluate FaPES in larger cluster settings, we further conduct trace-driven simulation. We simulate 1000 servers, each equipped with 8 GPUs. The servers are interconnected in a hierarchical topology: each rack contains 10 servers, which are connected to a ToR switch; every 10 ToR switches are further connected to an edge switch; and a total of 10 edge switches are linked to the aggregation switch in the top tier. The bandwidth capacity of links from bottom to top

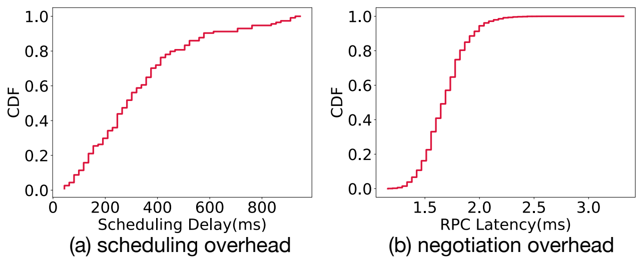


Figure 14: System overhead.

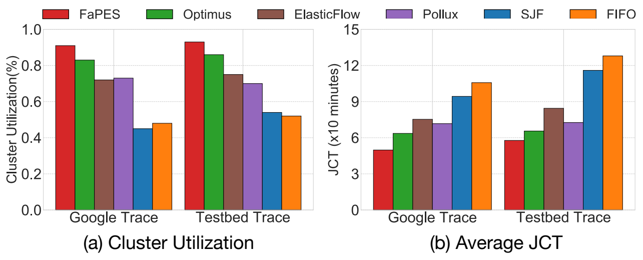


Figure 15: Performance under different workload traces.

tiers of the topology are 200Gbps, 400Gbps, and 800Gbps, respectively. We inject inference requests with $N_r^{max} = 200$, accounting for 20% of all servers. To verify the workload generality, we use the task arrival pattern from the public Google trace [5] to inject training models described in the production trace above.

We apply profiled computation timeline of each job from our textbed experiments. We apply job migration overhead as given in Table 2 in every auto-scaling event of a job as well. In the simulation, FaPES adopts its complete inter-VC resource loaning and job scheduling designs, while different scheduling baselines adopt one-way resource loaning between inference and training VCs.

Fig. 15(a) shows that FaPES achieves 9.6% to 26% improvement in terms of cluster utilization as compared to elastic baselines, and up to 2× as compared to non-elastic baselines. Fig. 15(b) reveals that FaPES reduces the average JCT by 11.9% to 33.8% as compared to elastic baselines, and by 47.2% to 54.8% as compared to non-elastic baselines. These validate efficiency of FaPES's resource loaning and auto-scaling designs.

We further adjust the duration of the time round, and investigate FaPES's cluster utilization and job completion time, when its inter-VC resource loaning is enabled and not. In the case without loaning (denoted as w/o loan), servers that are underutilized by the inference jobs will not be moved to the training cluster for training jobs. In Table 3, we observe that when the duration is longer, though impact of job migration overhead is smaller and more jobs can be scheduled

Table 3: Performance under different time round durations

	10min		20min		30min	
	Util.	**JCT**	**Util.**	**JCT**	**Util.**	**JCT**
w. loan	93%	38.5 min	81%	42.7 min	62%	53.0 min
w/o loan	82%	41.6 min	66%	59.3 min	51%	67.9 min

jointly, the resource adjustments are less timely, which leads to poorer overall performance.

7.6 Discussions

Besides data parallel jobs, FaPES can also be extended to serve jobs with various parallelism strategies like pipeline parallel [23] or tensor parallel [36]. One can exploit FaPES by integrating a newly designed performance model into FaPES-Scheduler, while keeping the whole architecture and the remaining components (i.e., FaPES-Board, FaPES-Manager, FaPES-Pod) just the same. LLM workload with hybrid parallelism [11] is not the target of FaPES. It is more efficient to reserve dedicated servers and pack a large-scale model onto them with local affinity [16].

8 CONCLUSION

We present FaPES, a FaaS-oriented performance-aware elastic scaling system for serverless ML platforms running both training and inference workloads. FaPES advocates flexible two-way loaning of GPU resources between training and inference virtual clusters, achieving improved overall cluster utilization and better serving SLO fulfillment. Additionally, FaPES carefully designs a training performance model and a job scheduling mechanism to minimize training job completion time and maximize training efficiency globally. We extensively evaluate FaPES with testbed and simulation experiments, using ML job traces from a production cluster. Experimental results demonstrate that FaPES achieves higher cluster throughput and better individual job performance compared to representative resource schedulers.

ACKNOWLEDGMENTS

This work was supported in part by Alibaba Group through Alibaba Innovative Research (AIR) Program and grants from Hong Kong RGC under the contracts HKU 17207621, 17204423 and C7004-22G (CRF).

REFERENCES

[1] 2024. Apache Openwhisk. https://github.com/apache/openwhisk.
[2] 2024. AWS Lambda. https://aws.amazon.com/cn/pm/lambda/.
[3] 2024. AWS-Placement Group. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.
[4] 2024. Google Cloud Function. https://cloud.google.com/functions.
[5] 2024. google cluster data. https://github.com/google/cluster-data.

[6] 2024. metrics-server. https://github.com/kubernetes-sigs/metrics-server.

[7] 2024. Microsoft Azure. https://azure.microsoft.com/en-us.

[8] 2024. prometheus. https://github.com/prometheus/prometheus.

[9] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 505–513.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[12] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[13] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. 2023. FaST-GShare: Enabling efficient spatio-temporal GPU sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*. 635–644.

[14] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 721–739.

[15] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.

[16] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling Large Language Model Training to More Than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[17] Brett Koonce and Brett Koonce. 2021. ResNet 50. *Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization* (2021), 63–72.

[18] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 835–850.

[19] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. 2023. EasyScale: Elastic Training with Consistent Accuracy and Improved Utilization on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[20] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James Larus, and Haibo Chen. 2024. Jiagu: Optimizing Serverless Computing Resource Utilization with Harmonized Efficiency and Practicability. *arXiv preprint arXiv:2403.00433* (2024).

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

[22] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162* (2018).

[23] 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 *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[24] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.

[25] Joe H Novak, Sneha Kumar Kasera, and Ryan Stutsman. 2019. Cloud functions for fast and robust resource auto-scaling. In *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*. IEEE, 133–140.

[26] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[27] 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 *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*.

[28] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[29] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. 2024. Mlless: Achieving cost efficiency in serverless machine learning training. *J. Parallel and Distrib. Comput.* 183 (2024), 104764.

[30] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[31] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems* 2 (2020), 172–186.

[32] Adam Weingram, Yuke Li, Hao Qi, Darren Ng, Liuyao Dai, and Xiaoyi Lu. 2023. xCCL: A Survey of Industry-Led Collective Communication Libraries for Deep Learning. *Journal of Computer Science and Technology* 38, 1 (2023), 166–195.

[33] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[34] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[35] Hanfei Yu, Athirai A Irissappane, Hao Wang, and Wes J Lloyd. 2021. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 31–40.

[36] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. 2021. Oneflow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032* (2021).

[37] Han Zhao, Weihao Cui, Quan Chen, Shulai Zhang, Zijun Li, Jingwen Leng, Chao Li, Deze Zeng, and Minyi Guo. 2024. Towards Fast Setup and High Throughput of GPU Serverless Computing. *arXiv preprint arXiv:2404.14691* (2024).

[38] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. 2023. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 703–723.

[39] Ruiting Zhou, Jinlong Pang, Qin Zhang, Chuan Wu, Lei Jiao, Yi Zhong, and Zongpeng Li. 2022. Online scheduling algorithm for heterogeneous distributed machine learning jobs. *IEEE Transactions on Cloud Computing* 11, 2 (2022), 1514–1529.