Elastic Deep Learning in Multi-Tenant GPU Clusters

Yidi Wu*, Kaihao Ma*, Xiao Yan*, Zhi Liu*, Zhenkun Cai*, Yuzhen Huang*, James Cheng*, Han Yuan†, Fan Yu†
* Department of Computer Science and Engineering, The Chinese University of Hong Kong
† Huawei Technologies Co. Ltd
{ydwu, khma, xyan, zliu, zkcai, yzhuang, jcheng}@cse.cuhk.edu.hk, {yuanhan3, fan.yu}@huawei.com

Abstract—We study how to support elasticity, that is, the ability to dynamically adjust the parallelism (i.e., the number of GPUs), for deep neural network (DNN) training in a GPU cluster. Elasticity can benefit multi-tenant GPU cluster management in many ways, for example, achieving various scheduling objectives (e.g., job throughput, job completion time, GPU efficiency) according to cluster load variations, utilizing transient idle resources, and supporting performance profiling, job migration, and straggler mitigation. We propose EDL, which enables elastic deep learning with a simple API and can be easily integrated with existing deep learning frameworks such as TensorFlow and PyTorch. EDL also incorporates techniques that are necessary to reduce the overhead of parallelism adjustments, such as stop-free scaling and dynamic data pipeline. We demonstrate with experiments that EDL can indeed bring significant benefits to the above-listed applications in GPU cluster management.

Index Terms—Deep learning system, elastic deep learning, GPU cluster management

1 Introduction

Due to the huge success of deep learning (DL), many organizations have built large GPU clusters for deep neural network (DNN) training. A GPU cluster typically serves many concurrent users. Users submit deep neural network (DNN) training jobs and the respective resource requirements (e.g., the number of GPUs) to the cluster. A multi-tenant GPU cluster is usually managed by a traditional cluster manager (e.g., YARN [1], Mesos [2]) or a scheduler tailored for GPU clusters (e.g., Optimus [3], Gandiva [4], Tiresias [5], Themis [6], SLAQ [7]), with objectives such as *high throughput, high GPU efficiency* [1]*, short job completion time (JCT), and good responsiveness for small jobs* [2].

Through an analysis of the trace data from Microsoft's production GPU cluster [8], we found that *elasticity, the ability to adjust the parallelism (i.e., the number of GPUs) of a DNN training job*, is beneficial to multi-tenant GPU cluster management in many aspects.

- **Adapting to cluster load variations.** GPU clusters can be heavily loaded in long periods of time while under-utilized on other periods as shown in Figure 3a. With elasticity, DNN training jobs can be scaled out (i.e., increase the parallelism) to achieve high throughput when the cluster is not busy or has transient idle resource, and scaled in (i.e., reduce the parallelism) to improve GPU efficiency when the cluster is heavily loaded.
- **Enforcing priority-based scheduling.** Elasticity can be used to gracefully enforce priority. Instead of preempting low-priority jobs, we may scale in low-priority jobs to obtain sufficient resources for high-priority ones. One application scenario is prioritizing small jobs to mitigate head-of-line blocking and improve job responsiveness and average JCT.
- **Supporting flexible job management.** Elasticity is useful for cluster management functionalities such as *straggler mitigation*, *performance profiling*, and *worker migration*: 1) stragglers are detrimental to the throughput of synchronous training and can be removed from a job by scaling in; 2) the performance of a job under different parallelism can be easily profiled using a series of scale-in operations; and 3) using scale-in and scale-out, we can migrate workers for a job from one machine to another machine to reduce resource fragmentation.

Elasticity has been explored for other types of workloads such as graph processing [9], batch processing [10] and parameter-server based machine learning [11], [12]. However, the unique characteristics of DL training jobs and multi-tenant GPU clusters pose challenges to attain the aforementioned benefits of elasticity. Specifically, *the overhead of elasticity needs to be low* to support frequent parallelism adjustment (e.g., in scheduling) and effectively utilize transient idle resources. Parallelism can be trivially adjusted

- *Xiao Yan is the corresponding author.*

1. Here, *throughput* is the average number of training samples processed per second. Let $t(p)$ be the average per-GPU throughput of a job using p GPUs, and $p^* = \arg\max_p t(p)$. *GPU efficiency* is defined as $t(p)/t(p^*)$, which is an indicator of how close the current average per-GPU throughput (using p GPUs) is to the optimal one (using p^* GPUs). Both throughput and GPU efficiency are job-level performance metrics.

2. Following Tiresias [5], we define *job size* as *(parallelism×running time)*, where the unit of job size is *(GPU * seconds)* in this paper.

by *Stop-Resume* [3], which checkpoints a job and restarts it with the desired parallelism. Stop-Resume is supported by most DL systems but the running job typically needs to be stopped for more than 30 seconds. To amortize the large overhead, Optimus [3] uses *Stop-Resume* only every 10 minutes. As we will show in §2.3, the overhead of Stop-Resume is too high to gain a performance improvement from transient idle GPUs. Moreover, *elasticity should be compatible with existing DL systems and require minimum user efforts* to enable its adoption in real applications. Finally, *a balance between consistency and efficiency needs to be maintained*. For example, to make elasticity transparent to users (e.g., algorithm designers), critical hyper-parameters such as the aggregated batch size of a job should be fixed under scaling but GPU efficiency can severely degrade in this case as we will show in §2.

We propose **EDL** to support *elastic deep learning* in multi-tenant GPU clusters. EDL is a light-weight coordination layer between a cluster scheduler and a DL system. EDL delegates single-machine execution to the underlying DL systems (e.g., TensorFlow [13], PyTorch [14], MindSpore [15], MXNet [16]). The DL system only needs to retrieve the meta data of a block of training data from EDL and notifies EDL after finishing a mini-batch. EDL can be used as a simple plug-in to different DL systems and maintains good usability, i.e., users only need to add a few lines to their original script (e.g., TensorFlow script) to enjoy elasticity and EDL hides all the details (e.g., dynamic parallelism adjustments) from users. The scheduler can instruct EDL to remove/add any worker for a training job using a simple API, e.g., *scale_in()* and *scale_out()*. Most critically, EDL significantly reduces the overhead of elasticity compared with Stop-Resume.

EDL is designed to ensure both the correctness and efficiency when any worker may join/leave a job at any time. In EDL, each job is managed by a leader process and EDL uses a distributed transaction-based mechanism for fast leader election. To scale out, EDL proposes *stop-free scaling*, which allows existing workers to continue training while newly added workers are being prepared for execution. This hides most of the scaling overhead. To scale in, EDL uses *graceful exit* to remove workers at the end of a mini-batch training with a negligible overhead. For data preparation, EDL uses a *dynamic data pipeline* to assign blocks of data to workers in an on-demand fashion and leverages data pre-fetching to avoid starvation of GPUs. The data pipeline also ensures that the training goes over a dataset once without repetition and omission in each epoch.

We conducted extensive experiments to validate the performance of EDL. We first show that EDL has low overhead for normal training (without elastic scaling) compared with Horovod — a state-of-the-art distributed DL training framework. For elastic scaling, we show that EDL reduces the overhead of scaling out by an order of magnitude and has a negligible overhead for scaling in compared with Stop-Resume. In addition, we show that using EDL brings significant benefits to applications such as straggler mitigation, performance profiling, and job migration. With some simple modifications, we enable Tiresias [5], a state-of-the-art DL scheduler, to efficiently apply elasticity in DNN job scheduling, which achieves a reduction in the average JCT by 89.5%.

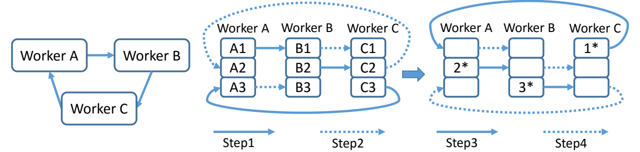


Fig. 1: An illustration of ring all-reduce

The rest of the paper is organized as follows. In §2, we give the motivation of the work. In §3 and 4, we present the API and the system design. In §5, we discuss the use cases of EDL. In §6, we report the experimental results. In §7 and §8, we give the related work and conclusions.

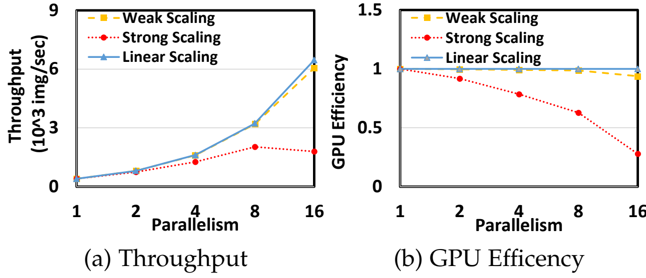
2 Motivation

2.1 Background

A DNN model is trained by going over a dataset many times (called epochs), and in each epoch the dataset is randomly shuffled and partitioned into a number of mini-batches. For each mini-batch, the model is updated using stochastic gradient descent (SGD), or its variants such as Adam and AdaGrad, with $w^{(t+1)} = w^{(t)} - \frac{\eta_t}{|\mathcal{B}_t|}\sum_{i \in \mathcal{B}_t} \nabla f(x_i, w^{(t)})$, where $w^{(t)}$ is the current model and $\mathcal{B}_t$ contains the training samples of the mini-batch. As calculating the gradient $\nabla f(x_i, w^{(t)})$ involves computation-intensive kernels such as matrix multiplication, DNN training is usually conducted on GPUs.

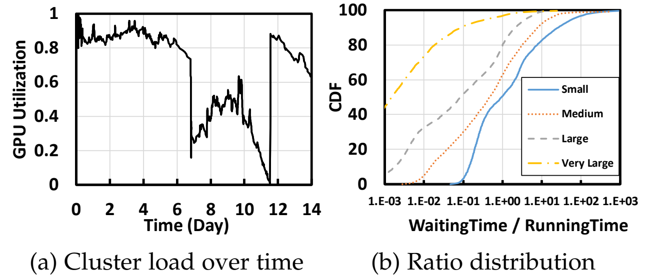
Due to the growing volume of data and the high complexity of DNN models (e.g., ResNet [17], VGG [18], Inception [19]), DNN training usually cannot be finished within a reasonable time on a single GPU and thus distributed training on multiple GPUs offers a good alternative. Among the various distributed training schemes, *synchronous data-parallel* is the most popular one [20], which partitions a dataset among GPUs and each GPU (i.e., a worker) calculates the gradient for some training samples in parallel. When all workers finish the gradient computation in a mini-batch, the local gradient from the workers are aggregated and then added to the model before the next mini-batch starts. As a synchronization barrier is enforced at the end of every min-batch, *stragglers* are detrimental to the performance of synchronous training.

Allreduce [21], [22] is a popular protocol for coordinating model updates from distributed workers and has been widely adopted in TensorFlow, PyTorch, MXNet and Horovod thanks to its simplicity and its efficiency in network communication. We present the implementation of Ring-Allreduce as follows. Workers form a ring communication topology and each worker communicates only with its two neighbors on the ring as illustrated in Figure 1. When one gradient tensor is ready, each worker sends, receives and aggregates $1/N$ (where N is the number of workers) of the tensor to the adjacent worker in a round-robin fashion in each step. After $N-1$ steps, each worker has $1/N$ of the tensor that aggregates the updates from all workers. In the next $N-1$ step, each worker passes its aggregated part of the parameters along the ring such that the gradient on all workers will be updated. *Parameter server* [23], [24], [25], [26]



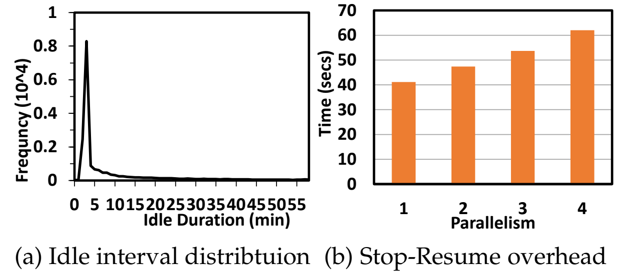
(a) Throughput (b) GPU Efficency

Fig. 2: The influence of parallelism on ResNet50, *linear scaling* is an ideal case which assumes that throughput scales linearly with parallelism and we included it for reference



(a) Cluster load over time (b) Ratio distribution

Fig. 3: Cluster load variation and job responsiveness



(a) Idle interval distribtuion (b) Stop-Resume overhead

Fig. 4: Idle interval and scaling overhead of Stop-Resume

is also widely used in distributed machine learning, which provides a key-value interface for model update/lookup. However, configuration is more complicated for parameter server as performance strongly depends on the number and location of the servers as well as the skewness of tensor sizes in the models [3], [5], [27], [28].

Horovod [29] is the state-of-the-art framework for distributed DNN training based on Allreduce. Horovod does not have native support for elastic training. To adjust the parallelism dynamically, Horovod may use the Stop-Resume mechanism provided by the underlying DL system. It delegates single machine execution to existing DL systems and adopts the synchronous data-parallel computation model. In a Horovod job, a leader process coordinates the order and granularity of gradient synchronization among workers.

2.2 The Benefits of Elasticity

We show the benefits of elasticity for multi-tenant GPU cluster management from observations in our experiments and the trace data from Microsoft [30]. The trace data contains scheduling events (e.g., job submission/finish time) and brief descriptions of jobs (e.g., user id, number and location of allocated GPUs) collected over two months from Microsoft's production GPU cluster (with approximately 2,300 GPUs). Note that all jobs use a static parallelism in the trace data.

Adjusting throughput and GPU efficiency. We show the influence of parallelism on training throughput and GPU efficiency in Figure 2. Two scaling schemes are considered, i.e., *weak scaling* and *strong scaling*. Weak scaling keeps the per-GPU batch size fixed, which means that the aggregated batch size increases with parallelism. Figure 2 shows that for weak scaling, training throughput increases almost linearly with parallelism and GPU efficiency stays almost constant, which is in line with previous findings [31], [32]. However, there are studies [31], [32], [33] observed that the convergence quality degrades when the batch size becomes too large and some theoretical results [34], [35] also support this observation. Strong scaling keeps the aggregated batch size fixed and adjusts the per-GPU batch size according to parallelism. Strong scaling helps make elasticity transparent to algorithm design but Figure 2 shows that GPU efficiency drops linearly as parallelism increases. This is because the communication cost increases with parallelism and it is difficult to overlap computation with communication when the per-GPU batch size is small.

The results in Figure 2 show that DNN training jobs can usually be processed with different parallelism, and both throughput and GPU efficiency change with parallelism. Thus, elasticity can be used to dynamically adjust the parallelism of DNN training jobs according to different objectives of cluster scheduling (e.g., shorter JCT, higher throughput or GPU efficiency).

Improve cluster utilization and JCT. We plot the changes in the load of the Microsoft cluster in a period of 2 weeks in Figure 3a. The cluster is almost fully loaded in some periods and many jobs are queuing to be processed, while in other periods the cluster load is relatively low. To further investigate in the idle resources, we define the *idle interval* of a GPU as the time elapsed between the completion of the previous job and the beginning of the next job on the GPU. We plot the frequency distribution of the idle intervals in Figure 4a, which shows that idle intervals follow a power-law distribution and the majority are short intervals. Specifically, 39.62% of the idle intervals are less than 4 minutes, which takes up 41.5% of the idle resources during peak hours (when >90% of the GPUs are occupied). As discussed in Section 1, elasticity can be used to scale in/out jobs to adapt to cluster load variations and utilize the transient idle resources. By improving cluster utilization, the JCT of the jobs can also be reduced.

Improve responsiveness. There exists a large variation in job sizes in the trace data, where *job size* is defined by *(parallelism×running time) [5], expressed as (GPU*sec)*. We sort jobs according to their sizes and partition them into four categories: small ($\leq$ 20%), medium (20%-50%), large (50%-90%) and very large (> 90%) according to the size distribution. We found that the largest of small jobs takes 85 GPU*sec, while the smallest of very large jobs takes 58,330 GPU*sec. As the small jobs are usually used for correctness

checking or parameter tuning, quick response is important. However, head-of-line blocking caused by long-running jobs can severely degrade the responsiveness for small jobs. We show this phenomenon in Figure 3b, which plots the cumulative distribution function (CDF) of the *(waiting-time / running-time)* ratio of each category of jobs. From Figure 3b, the curve "small" shows that for small jobs, nearly 50% of them have a *(waiting-time / running-time)* ratio greater than 1, which indicates that the waiting time of these small jobs is longer than their running time. Thus, we consider that small jobs have poor responsiveness because half of them require us to wait longer than to actually run them. In fact, nearly 20% of small jobs have a waiting time 10 times longer than their running time. To address this problem, we can scale in some long running jobs to make room for the small jobs when the cluster is overloaded.

2.3 Design Principles

Based on our observations in §2.2, we come up with the following system design principles in order to enjoy the benefits of elasticity. First, ***elasticity should come with low overheads***. Elasticity can be trivially achieved via *Stop-Resume*, which checkpoints a job and restarts it with the desired parallelism. We report the cost of adjusting a 1-GPU job to different parallelism for TensorFlow in Figure 4b. The result shows that the overhead of Stop-Resume ranges from 40 to over 60 sec [3]. This high overhead limits the ability to adapt to dynamic resource availability and job requirements as scaling can only be conducted infrequently, e.g., Optimus only uses Stop-Resume every 10 min [3]. The overhead also hinders the effective utilization of transient idle resources. Consider a job running with 4 GPUs and we have a transient GPU that will be idle for 4 min. Stop-Resume needs to first adjust the parallelism from 4 to 5 and then back to 4. Assume that each parallelism adjustment takes 30 sec, training is conducted with 5 GPUs for at most 3 min. Thus, the effective training time is at most (5 GPUs * 180 sec) = 900 GPU*sec. In contrast, the effective training time is (4 GPUs * 240 sec) = 960 GPU*sec if we do not use the idle GPU at all. Therefore, we need to design a set of parallelism adjustment procedures that carefully hide the overheads of parallelism adjustment.

Second, ***compatibility and usability are critical***. DL systems such as TensorFlow and PyTorch are widely used and hence elasticity provision should be built on these efforts, rather than developing a new system from scratch. From users' perspective, it must incur little to no effort to run their scripts written in existing DL systems as elastic training jobs. It should also be simple for a scheduler to adjust the parallelism of jobs in a cluster. This requires a good positioning of our system in the software stack and a careful design of the API.

Third, ***transparency and consistency should be considered***. We should ensure that the execution of a job applying elasticity is equivalent to its execution under a fixed parallelism regardless of the specifics of the parallelism adjustments, such that elasticity can be made transparent to algorithm design. For this purpose, we need to provide proper consistency semantics for DNN training jobs without incurring much overhead.

3. The scaling overhead increases with parallelism as TensorFlow initializes the GPU devices in one machine sequentially.

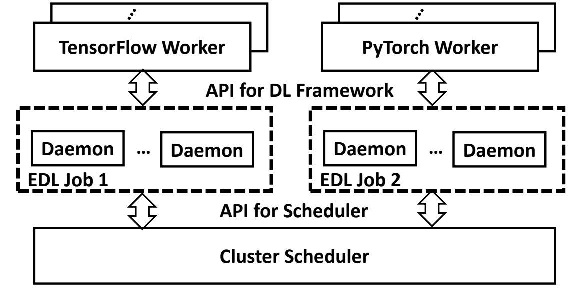


Fig. 5: The positioning of EDL

3 System Architecture and APIs

We focus on *data-parallel, synchronous training* as it is the dominant paradigm of distributed DL [13], [14], [16], [20], [29]. As users of DL are mostly familiar with popular systems such as TensorFlow, PyTorch and MXNet, it would be desirable if the core logic of elasticity can be shared among different DL systems. The shared component could be a new elastic communication library like Nvidia NCCL [22] or parameter server [23], [24], [25], [26]. However, supporting elasticity not only requires synchronizing the model among an elastic set of processes but also involves dynamically partitioning the training data and modifying parameters such as per-GPU batch size. Thus, we design EDL as a coordination layer sitting between DL systems and the GPU cluster manager as shown in Figure 5. The key APIs of EDL are summarized in Table 1. The cluster manager can use EDL's scheduler API to adjust the parallelism of jobs without knowing the details of the parallelism adjustment procedures. Users write training scripts using existing DL systems and only need to add a few lines to use EDL. This design incurs minimal change to existing infrastructures and results in good usability. We have integrated EDL with Huawei's DL system MindSpore [15]. MindSpore supports efficient single-device and distributed training through a tensor compilation module and automatic model partitioning. Some state-of-the-art training frameworks for emerging workloads, e.g., Seastar [36] and DGCL [37] for training graph neural network, have used MindSpore as the primary DL backend. In the following, we explain the integration with TensorFlow's API as concrete example.

In EDL, each job is executed by a group of worker processes and each process is associated with an EDL daemon. A *leader* is elected among the workers to schedule the order and granularity of gradient synchronization (the synchronization process is similar to Horovod [29]) and coordinate the parallelism adjustment (§4.1). Each worker process is attached with one GPU and runs in the single-machine mode using a DL system to compute gradient on some training samples for a mini-batch. We ingest communication operators (e.g., using the Grapler graph edit APIs in TensorFlow or hooks in PyTorch) in-between the computation and accumulation of gradients. Within the communication operator, an EDL daemon sends tensor synchronization requests asynchronously to the leader. After

TABLE 1: Key APIs of EDL

API for cluster scheduler	Description
scale_in(*job_handle, rmv_GPU_info*)	remove GPUs from a job
scale_out(*job_handle, add_GPU_info*)	add GPUs to a job
profile(*job_handle, min_p, max_p*)	profile a job
API for DL systems	**Description**
elastic_shard_generator()	generate the next shard's info
notify_batch_end()	check the need of scaling

receiving ready-to-reduce message from the leader, the EDL daemon delegates the synchronization task to a dedicated thread to avoid blocking message handling and the thread conducts gradient synchronization using communication libraries such as NCCL.

3.1 API for Cluster Scheduler

We assume that there is a centralized cluster scheduler (e.g., YARN), which has knowledge of resource availability and job status to make scheduling decisions. The scheduler may instruct EDL to adjust the parallelism of a job, identified by a unique *job_handle*, using the *scale_in*() and *scale_out*() operators. When a scaling operator is called, a message is sent to the leader of the workers that execute the job. The leader then coordinates the removal/addition of the specified GPU(s) and replies an acknowledgment message to the scheduler after the adjustment completes. Scaling operations are committed sequentially in EDL and if a scaling request is received in the middle of a parallelism adjustment, the leader sends a retry message to the scheduler. The leader may fail to reply in case of failure (either the leader itself or a worker). In either case, the scheduler may retry the scaling operation after a specified time (e.g., 60 seconds). The *profile*() operator measures the throughput and GPU efficiency of a job under a range of parallelism specified by [*min_p, max_p*]. It can be used to find the optimal parallelism of a job or collect information for scheduling by running profiling tasks on a dedicated small cluster [3], [5]. For a job that is already running, *profile*() can be called to report its throughput and GPU efficiency under the current parallelism without specifying the range.

EDL automatically recovers a job from failure using the remaining resources without intervention from the scheduler (§4.2), which eliminates delays due to re-scheduling and re-launching. EDL supports both weak scaling and strong scaling but uses strong scaling by default, which keeps the aggregate batch size of all the workers constant and decides the per-worker batch size according to the parallelism. Weak scaling is also supported and users may specify an optional parallelism range for weak scaling to be applied. Moreover, EDL ensures that training goes over the dataset once in each epoch without repetition and omission. The above consistency semantics is sufficient for most DNN training jobs [31], [38], [39].

3.2 API for DL systems

EDL provides a simple API for users of popular DL systems to run their scripts as elastic jobs. Some of them are standard and similar to the ones in Horovod, e.g., *init*(), *shutdown*() and *all_reduce*(), while *elastic_shard_generator*() and

```
1  import tensorflow as tf
2  import edl.tensorflow as edl
3  edl.init() # initialize EDL daemon
4  # create a generator object
5  ds = tf.data.Dataset.from_generator(
6    edl.elastic_shard_generator())
7  loss = Resnet50(ds) # construct a Resnet50
      model
8  # ingest Allreduce into graph within
      Optimizer
9  opt =
      edl.Optimizer(tf.train.AdamOptimizer(...))
10 # create optmization objective
11 obj = opt.minimize(loss)
12
13 with tf.train.Session() as s:
14   while not s.Done():
15     s.run(obj, feed_dict={...})
16     edl.notify_batch_end()
```

Listing 1: An example code of EDL with TensorFlow

notify_batch_end() are specifically introduced for elasticity. *elastic_shard_generator*() returns a generator object, which gives the meta-data of a chunk of training samples to a worker when its *next*() method is called, and a DL system can use it to load training samples dynamically from a list of partitions. This operator ensures the efficient distribution of the training samples to a dynamic set of workers under scaling (§4.3). EDL adds/removes workers for a job at the end of a mini-batch so that no training progress is lost and users can call *notify_batch_end*() to notify EDL of the mini-batch boundary. The end of a mini-batch can be identified trivially in users' training script in existing DL systems, for example, after *session.run()* in TensorFlow and the end of the *for-loop* for each mini-batch in PyTorch. Since each mini-batch typically takes hundreds of milliseconds, the delay of waiting for the end of a mini-batch is usually short.

Putting things together, we illustrate with an example that uses EDL with TensorFlow in Listing 1. Line 3 initializes the EDL daemon and Lines 5-6 construct a TensorFlow dataset object from the *elastic_shard_generator*() method of EDL. The *edl.Optimizer* in Line 9 is a helper class that inherits TensorFlow's optimizer class and we ingest the Allreduce operation into the computation graph in *edl.Optimizer*. In Line 16, users indicate the end of one mini-batch with *notify_batch_end*(). It can be seen that using EDL is easy and it only requires adding a few lines (i.e., the lines containing "edl") to a user's script.

4 System Design and Implementation

The design of the EDL system has three goals: *flexibility*, *efficiency*, and *consistency*. Flexibility means that EDL should allow any process, either a worker or the leader, to leave or join a job at any time, which enables the scheduler to flexibly adjust parallelism. Efficiency means that EDL should significantly reduce the parallelism adjustment overheads compared with stop-resume, and should introduce negligible overheads to training under static parallelism without scaling. Consistency means that the execution of

a job under scaling should be equivalent to its execution without scaling [31], [38], [39].

To achieve these goals, EDL adopts three key designs: *automatic job management* (§4.1), *efficient parallelism adjustment* (§4.2), and *dynamic data pipeline* (§4.3).

4.1 Automatic Job Management

Each job has a leader to manage its workers. However, the leader may leave the job due to scaling or failure. Each worker runs a leader election/discovery procedure whenever the leader is not known to the worker, which ensures that there is always a leader to manage the job. Specifically, when a job is launched, each worker first performs the leader election procedure, which is implemented as a distributed *compare_and_swap* transaction using an external coordination system such as ZooKeeper [40] or etcd [41]. The workers query the leader's connection information (e.g., hostname and port number) in the external coordination service using the *job_handle* as key. If the connection information is void or expired, a worker writes its own address into the information and becomes the leader. The leader needs to periodically refresh its address information, which is configured to expire automatically if the leader fails to do so. Upon expiration, workers will be notified so that they will perform leader election again.

After a leader is elected, it establishes an RPC server accepting connections, while other workers connect to the leader and send a registration message to join the job. According to our measurement, leader election took 7ms on average and 33ms at maximum when 256 workers used etcd for distributed coordination. During job execution, the leader infers the liveness of the workers from the gradient synchronization requests in every mini-batch and thus explicit heartbeat message is not needed. When *scale_out*() or *scale_in*() is called, the leader communicates with the new or exiting workers to prepare them for joining or leaving the job. The leader also constructs a new communication topology for distributed training with/without the new/exiting workers. More details will be presented when we discuss *scale_out*() and *scale_in*() in §4.2.

An alternative to the leader discovery mechanism is to launch a dedicated process (not attached with GPU) as the leader (similar to an application master [1]). Such a design has the advantage that *scale_in*() operations will not affect the leader. However, using multiple types of processes complicates the current single-program-multiple-data (SPMD) execution pattern. Deployment is more complicated as the leader requires different resource configurations.

4.2 Efficient Parallelism Adjustment

To reduce the overheads of parallelism adjustments, EDL uses *stop-free scaling* to hide the high cost of *execution context preparation* during *scale_out*() and applies *graceful exit* to make the overhead of *scale_in*() negligible.

Scale out. Adding new workers to a running job takes three steps: *execution context preparation*, *communication topology construction*, and *model preparation*. Execution context preparation involves loading dynamic libraries (e.g., cuDNN, cuBLAS), preparing training data, allocating space on both

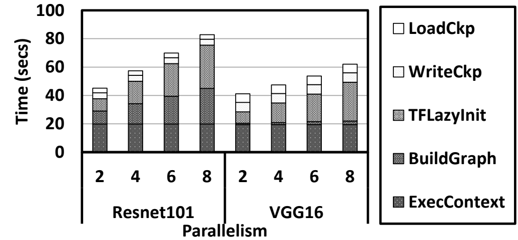


Fig. 6: Scaling overhead decomposition for TensorFlow

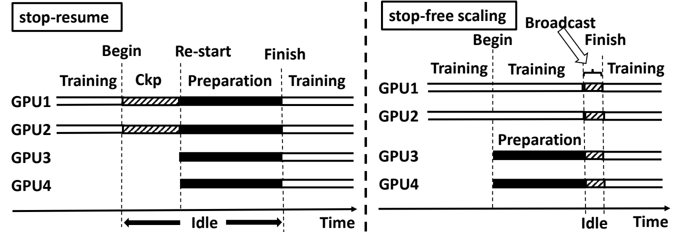


Fig. 7: An illustration of stop-free scaling

GPU memory and main memory, and so on. Declarative DL systems such as TensorFlow also need to build and optimize the computation graph. For communication, new workers need to connect to the leader for coordination and all the workers need to form a new ring topology for model synchronization. New workers also need to acquire the up-to-date model before joining the training. We provide a breakdown of the time for scaling out a 1-GPU job on TensorFlow in Figure 6 with the execution context preparation overhead marked in dark colors. The result shows that the cost of execution context preparation dominates the scaling overhead. This observation is consistent for all the models we experimented.

Motivated by this observation, we propose *stop-free scaling*. The key insight is that the training on the existing workers does not need to be stopped when the new workers conduct execution context preparation. Each new worker launches two separate threads, a main thread and a background thread. The main thread conducts execution context preparation while at the same time the background thread performs leader discovery and sends a registration request to the leader. The leader constructs a new communication topology involving the new workers after receiving their registration requests and broadcasts it to all the workers. Note that the original communication topology is not destructed yet, and thus the existing workers can continue the training without being affected. A new worker sends a ready message to the leader when it finishes execution context preparation and receives the new communication topology, but is blocked until it receives an OK message from the leader.

Once the ready messages from all the new workers have been received, the leader broadcasts an OK message and a future timestamp to all the workers. The existing workers check at the end of each mini-batch indicated by *notifyBatchEnd*() and switch to the new communication topology when its next local timestamp reaches the timestamp specified by the leader. The timestamp is implemented as the mini-batch count and we set the future timestamp as

$t_cur + k$, where t_cur is the current mini-batch count of the leader. k is determined as T_a/T_b, in which T_b is the current per-mini-batch time for the job and T_a is a predefined time allowance (500ms by default) to tolerate fluctuations in network latency. One existing worker is chosen to broadcast its model to the new workers as using only one worker for broadcasting reduces the time for model synchronization. After the new workers obtain the latest model, *scale_out*() completes and the training continues with the new parallelism.

An illustration of stop-free scaling, contrasting with stop-resume, is given in Figure 7, where we add two more GPUs to a job. With stop-free scaling, existing workers only need to stop and wait until the model is broadcast to the new workers, which can complete within 1 second for most models according to our experiments. Compared with stop-resume, the long execution context preparation time for new workers is now hidden behind the normal execution of the existing workers.

Scale in. For *scale_in*(), we apply *graceful exit*, in which the scheduler gives the exiting workers a short time allowance (e.g., 30 seconds, but usually a few seconds is enough) to leave. On receiving the *scale_in*() request, the leader constructs a new communication topology and broadcasts it to the remaining workers. Similar to the case of *scale_out*(), the leader also sends a future timestamp to all the workers, at which the exiting workers should leave and the remaining workers should switch to the new communication topology. Before reaching this timestamp, training continues with all the workers. If the leader is instructed to leave, it will erase its address in the external coordination system such that a new leader can be elected using the leader election protocol. The old leader will send the job meta-data (e.g., batch size, data loading progress, etc.) to the new leader before exiting and all the remaining workers will connect to the new leader at the scheduled timestamp. With graceful exit, the overhead of *scale_in*() is negligible as the exiting workers just need to leave and the remaining workers do not need to stop and wait.

Failure recovery. We consider *forced exit*, including process failure, as a special case of scaling in. Worker failure can be detected if a worker fails to send the gradient synchronization request for a mini-batch and leader failure can be detected by the leader election/discovery protocol. When failure happens, the model may be inconsistent. For example, if a worker fails before finishing synchronizing all gradients, the model on the other workers would be partially updated. EDL provides two protocols to recover from failure, i.e., *consistent recovery* and *approximate recovery*. Consistent recovery requires the leader to write a checkpoint to persistent storage such as HDFS [42] periodically (e.g., every 1000 mini-batch or every 10 minutes). Upon failure, the job is resumed by loading and restarting from the latest checkpoint, which ensures model consistency. As DNN training is known to be robust to bounded errors, approximate recovery can also be used to simply construct a new communication topology for the surviving workers and redo the current mini-batch. Users can choose one of the two protocols. By default, EDL uses consistent recovery.

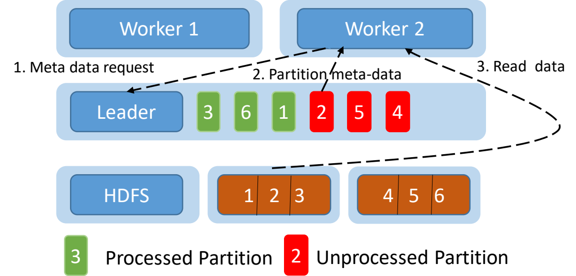


Fig. 8: An illustration of dynamic data pipeline

4.3 Dynamic Data Pipeline

Existing DL systems partition a dataset among workers before training starts, and each worker goes over its assigned partitions in each epoch [13], [14], [16]. This *static data allocation* method works well in practice, but we show that static data allocation lacks flexibility and results in complicated data management for elastic DL.

Consider a dataset with 1M samples, which is partitioned into 1K partitions each with 1K samples, and there are 10 workers each getting 100 partitions. If we want to add 5 GPUs to this job, two options are possible under static data allocation. First, we can wait until the end of the current epoch and re-assign the partitions among the 15 workers, which is inflexible as parallelism adjustment is only possible at the end of the current epoch (instead of the current mini-batch as in EDL). Second, we can re-assign only those unprocessed partitions in the current epoch among the 15 workers and conduct a global re-allocation when the current epoch ends. However, if another scaling instruction (e.g., removing 3 out of the 5 added GPUs because they are transient resources) comes before the re-assignment finishes, a new data allocation plan needs to be constructed on the partially re-assigned data within the current epoch. Some other issues, such as hiding the delay of data re-assignment and handling partition fragmentation or imbalance, also need to be considered, which make the design and implementation complicated.

To support elasticity, EDL assigns data partitions to workers dynamically in an on-demand fashion as shown in Figure 8. The dataset is logically divided into d partitions, where d is sufficiently larger than the number of workers while the size of a partition is still large enough to allow high-bandwidth data reading in batches. The partitioning is only conducted at the meta-data level, e.g., recording file names and offsets, and the dataset is not physically partitioned. The leader generates a random permutation of the indexes of the partitions and uses it for dynamic data assignment. When a worker needs a new partition, it sends a data-read request to the leader by calling the *next*() method of the generator object returned by *elastic_shard_generator*(). The leader replies the request with the meta-data (e.g., file path, offset and length) of the next unassigned partition. The worker then issues asynchronous I/O request to the distributed file system (e.g., HDFS [42]) for reading this partition.

For the purpose of progress tracking, each worker records an offset in its current partition, which indicates where the next mini-batch should start. The workers report

their offsets to the leader at the end of each mini-batch and this information is attached to the gradient synchronization request with negligible overhead. When new workers join a job, the leader simply assigns some unprocessed (or partially processed) partitions to them. When a worker leaves under graceful exit, it reports to the leader the meta-data of the current partition and its offset in the partition such that the leader can assign the remaining unprocessed data in this partition to another worker. If the leader needs to leave, it sends the partition permutation list and the progress of all the workers to the new leader before it exits. EDL also writes the partition permutation list and the worker progresses to checkpoint such that a job can be restored properly.

The above procedure of dynamic data pipeline in EDL ensures that training goes over the dataset once in each epoch without repetition and omission regardless of whether scaling out and in are performed. However, different runs of an algorithm may not produce the same result as scaling may affect the order in which the samples are used in the training. In essence, the change in the processing order of the samples caused by scaling can be viewed as an additional source of randomness in the sample permutation and thus the consistency guarantee by dynamic data pipeline is sufficient for most deep learning tasks [31], [38], [39].

4.4 Implementation Details

We modified Horovod v0.16.1 and implemented the EDL daemon and plugins using Boost.asio with around 4K lines of code. We use NCCL v2.4.8 and TensorFlow v1.14.1. TCP is used to connect the leader with the workers and the cluster manager. We observed that usually tens of coordination messages are exchanged between the leader and the workers in each mini-batch training and the size of each message is within a few hundred bytes. As each mini-batch training usually takes only a few hundred of milliseconds, reducing the messaging latency is critical to avoid wasting GPU cycles. Therefore, we disabled the Nagle's algorithm [43], [44] in the TCP socket and the average latency of sending one message is 56 μs according to our measurement. We are also investigating to use RDMA to further reduce the latency.

To hide the latency of reading training data from the file system, each worker runs a producer-consumer data pipeline. A ping-pong buffer (or double buffer) is maintained between CPU and GPUs. The buffers are blocks of pinned memory to avoid disk swapping and enable fast data transfer to GPUs. A background thread serves as the producer and asks the leader for the meta-data of a new partition once a partition is dequeued from one of the buffers by the consumer. We overlapped host-to-device data movement with GPU computation by pre-fetching multiple mini-batches of training samples from main memory to GPU.

5 Elasticity In Use

In this section, we discuss how EDL can benefit DL cluster scheduling and be used to implement a number of important system functionalities such as straggler handling, performance profiling, and worker migration.

Algorithm 1: Compaction and Expansion

Input: Job pending queue groups: G, compaction threshold: N, parallelism for job j: p_j, minimum number of GPUs for job j: min_j, resource manager that manages the free GPUs in cluster: R. $\emptyset$: empty set.

```
1  if G.num_pending_job() > N then
2      for J̃ in G.pending_jobs() do
3          r = Argmax(Gain(J̃, r_j, G_i)), where
             p_j − r_j >= min_j, r_j is non-negative integer
             and i ≠ 0.
4          schedule_job(r, J̃)
5      end
6  end
7  else if G.pending_job_num() == 0 then
8      while R.num_free_gpus() > 0 do
9          G_free = R.free_gpu().pop()
10         J̃ = Argmax((S(G_free, p_j+1) − S(∅, p_j)) / S(∅, p_j))
11         schedule_job(G_free, J̃)
12     end
13 end
```

5.1 Elasticity-Aware DL Scheduling

According to §2, EDL can be used to (1) adjust the trade-off between throughput and GPU efficiency, (2) improve cluster utilization and JCT by adapting to the variations in cluster load, and (3) make good use of transient idle resources. One way to enjoy all of these three benefits is an elasticity-aware DL scheduler based on EDL.

As developing a new scheduler is out of the scope of this paper, we extend Tiresias [5], a state-of-the-art GPU cluster scheduler based on the shortest-job-first principle. Tiresias manages jobs in multiple groups, $G_0, G_1, \ldots$, and the group with a smaller index has higher priority. Scheduling is conducted by allocating resources to jobs in the higher-priority groups first. Each group G_i has a service quantum t_i for its jobs, meaning that a job can only consume up to t_i GPU*sec and after that it will be moved to G_{i+1}. When a job is submitted to the cluster, it is first placed into G_0 and gradually moved to a lower-priority group as it keeps running. If a job is not scheduled for a long time, it will be moved to G_0 to prevent starvation. Tiresias computes a new scheduling plan for all jobs whenever there is a new event (e.g., a new job is received or some job changes its priority). A running job will be preempted if it cannot be scheduled (i.e., its required resources cannot be allocated) in the new plan. Tiresias achieves good responsiveness for small jobs because they can be completed in the first few groups, i.e., groups with higher priority (e.g., jobs that take less than t_0 GPU*sec are always scheduled first). Readers may refer to [5] for details.

To enable elasticity-aware scheduling for Tiresias, we extend its scheduling protocol with algorithm 1. Compaction is triggered (Lines 1-6) if the number of waiting jobs exceeds a threshold N. We define $Gain(\tilde{J}, r_j, G_i)$ as the gain in GPU efficiency by removing r_j GPUs from running job j in group

G_i ($i \neq 0$) using scale-in and allocating these GPUs to $\tilde{J}$ [4]. Expansion is triggered (Lines 7-13) if there is no pending jobs but some GPUs are idle. For each GPU, we select from the running job that has the largest gain when allocated the GPU in consideration, where the gain is defined as $\frac{S(G_{free},\ p_j+1)-S(\emptyset,\ p_j)}{S(\emptyset,\ p_j)}$, in which $S(G_{free},\ p_j)$ is the training throughput with the current parallelism p_j by adding a GPU from G_{free} to job j and $\emptyset$ means no GPU is added.

We call the new scheduling algorithm **Elastic-Tiresias**. Intuitively, the two rules of Elastic-Tiresias aim to improve GPU efficiency when the cluster load is high and try to fully utilize the idle resources when the load is low. With these two simple modifications, Elastic-Tiresias achieves significantly better performance compared with the original Tiresias (§6.3). If users do not want the scheduler to change the parallelism of a job, they can mark the job as inelastic and Elastic-Tiresias simply skips it when conducting parallelism adjustment.

5.2 Additional Use Cases of EDL

EDL can also be easily used to provide important system functionalities such as follows.

Straggler mitigation. Some workers may become stragglers due to reasons such as high GPU temperature (which leads to clock frequency drop) and strong interference from co-located jobs. Stragglers are a major cause of performance degradation in synchronous training as a synchronization barrier is enforced at the end of each mini-batch. EDL detects stragglers by monitoring the time that workers spend on a mini-batch via the gradient synchronization requests. If a worker is consistently slower than other workers in a few consecutive mini-batches (e.g., its per-mini-batch time is longer than 1.2 times of the median for 10 mini-batches), the leader may trigger a ***scale_in***() operation to remove this worker from training with negligible overhead. Note that a smaller parallelism without straggler can lead to better performance as we will report in §6.2. A replacement worker, or the straggler machine itself (e.g., after cooling down or the completion of co-located jobs), can easily join the job using ***scale_out***() to restore to the original parallelism.

Performance profiling. Building an analytical model for the performance of DNN training jobs under different parallelism and placement plans is important but generally challenging [3], [4], [5]. There are many factors such as model architecture, global batch size and network bandwidth, that may affect performance in different ways. Therefore, it is common and also often necessary to run profiling jobs to measure the performance under different configurations to collect information for performance tuning and/or job scheduling. The *profile*() method in EDL can be easily used to measure the runtime performance under a range of parallelism, defined by (min, max). As ***scale_in***() has much lower overhead than ***scale_out***(), EDL starts a profiling job with the maximum parallelism and gradually scales in to the minimum parallelism. At each parallelism, the job is run for a few mini-batch iterations (e.g., 20) to measure the performance.

4. We enforce a locality constraint that the p GPUs to be allocated to $\tilde{J}$ must come from no more than $\lceil P_{\tilde{J}}/m \rceil$ machines, where $P_{\tilde{J}}$ is the user-specified parallelism for $\tilde{J}$ and m is the number of GPUs on each machine.

TABLE 2: Model statistics

	Inception3	ResNet50	VGG16	Transformer
Model Size(MBs)	92	98	528	24
Batch Size(per GPU)	64	64	32	2,048

Worker migration. Sometimes the scheduler needs to move one worker of a job from one machine to another machine, e.g., to co-locate the workers of this job to reduce communication cost or to make room in a machine so that it can be dedicated to some other purposes. Worker migration can be easily operated in EDL by first scaling in to remove the workers on the destination machine and then scaling out to add new workers from the target machine, without stopping the job. We further optimize this procedure by merging the scale-in and scale-out operations into one single migration operation, in which the communication topology is switched only once.

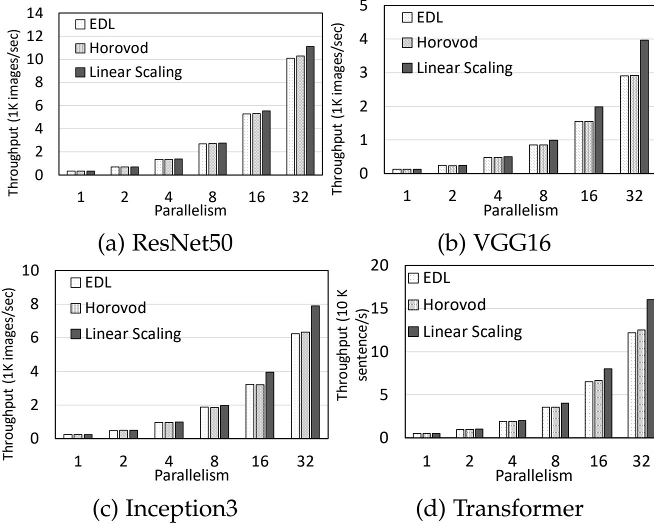
6 Experimental Results

We evaluated EDL on a cluster with 8 machines, each equipped with a 96-core Intel CPU, 8 NVIDIA Tesla V100 SMX2 GPUs and 256 GB RAM. The machines are connected with 100 Gbps infiniband. We used 4 popular DNN models in the experiments, and their model sizes and batch sizes are reported in Table 2. Among them, ResNet50 [17], VGG16 [18] and Inception3 [19] are designed for computer vision tasks while Transformer [45] is widely used for NLP tasks. We used strong scaling by default. The aggregated batch size was fixed to per-GPU batch size times the initial number of GPUs and remained unchanged regardless of scaling. We did not test convergence time and model quality (e.g., classification accuracy). This is because under strong scaling and synchronous training, parallelism adjustments do not affect the quality of the models. Convergence time is determined by both the number of epochs (which in turn depends on the algorithm and task) and training throughput. As a system paper, we focus on training throughput as it directly translates into convergence time with a given number of epochs. We summarize the key results of the experiments as follows.

- EDL is efficient as it introduces negligible overheads to normal training when there is no scaling and significantly reduces the overheads of parallelism adjustment compared with Stop-Resume.
- The efficient elasticity enabled by EDL provides efficient support for scheduling primitives such as profiling, straggler mitigation and job migration.
- By utilizing elasticity, the Elastic-Tiresias algorithm introduced in §5.1 achieves higher GPU utilization and significantly shortens JCT compared with the original Tiresias.

6.1 The Overheads of Elasticity

Performance under static parallelism. As DNN training jobs run with a static parallelism (i.e., fixing the number of



(a) ResNet50 (b) VGG16

(c) Inception3 (d) Transformer

Fig. 9: Performance under static parallelism

GPUs for running a job during its entire execution) most of the time, it is crucial that the designs for elasticity in EDL (e.g., RPC-based coordination, dynamic data pipeline) incur little overhead on normal training. We validate this by comparing EDL with the state-of-the-art distributed DL framework, Horovod. We measured the throughput (averaged over 500 mini-batches) of EDL and Horovod for training different DNN models using up to 32 GPUs. As a common practice of testing the scalability of distributed DNN training systems [31], [46], we increased the total batch size linearly with the number of GPUs (i.e., weak scaling). We report the results for four popular models in Figure 9, which shows that EDL achieves comparable throughput with Horovod for normal training (without elastic scaling), where the throughput of EDL and that of Horovod differ by at most 3%. The slight difference in throughput is mainly caused by the difference in the communication layer: Horovod leverages MPI with RDMA support while EDL uses TCP, which has a relatively larger message latency.

Scaling overheads. To scale in, EDL does not stop training and uses graceful exit to remove the exiting worker(s). To scale out, EDL needs to stop training for a short period of time to broadcast the latest model to the new worker(s) (§4.2). In comparison, Stop-Resume needs to stop all workers for the entire parallelism adjustment period for both scaling in and out.

We report the *stopping time* of scaling out for EDL and Stop-Resume in Figure 10 and Figure 11 (averaged over 20 trials), respectively. Stopping time is how long all workers for a job are stopped from training during scaling. In both figures, $x \rightarrow y$ means scaling a job from x GPUs to y GPUs, and *local* means the added GPUs are on the same machine as the original GPUs, while *remote* means the added GPUs are on another machine. The results show that in all cases (i.e., the number and location of the added GPUs), the stopping time of EDL is orders of magnitude shorter than that Stop-Resume thanks to EDL's efficient parallelism adjustment procedure (§4.2). For EDL, the stopping time of ResNet50 is long because it has a complicated model architecture and contains many small tensors, and it is

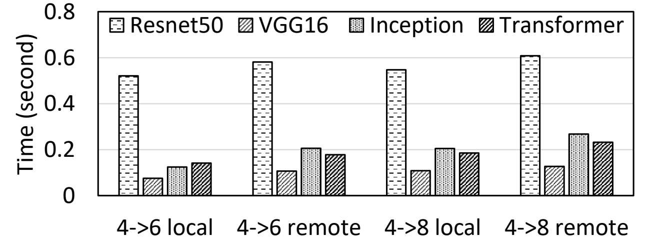


Fig. 10: The stopping time of EDL

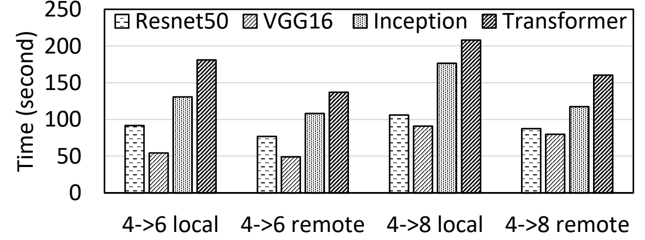


Fig. 11: The stopping time of Stop-Resume

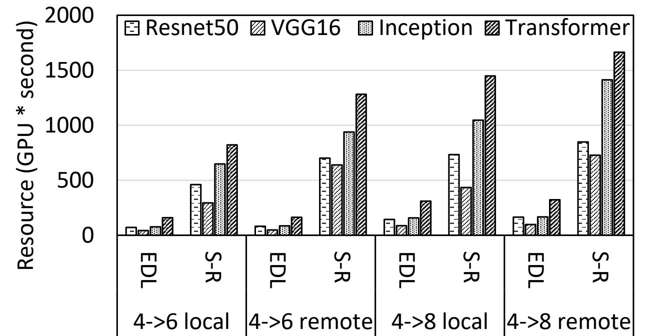
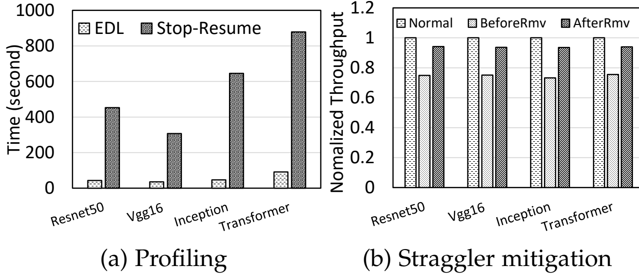


Fig. 12: Resource loss comparison

difficult to utilize the communication bandwidth efficiently when transmitting small tensors. The stopping time of EDL increases slightly when adding more GPUs (e.g., 4→6 local vs. 4→8 local) or the added GPUs are on a remote machine (e.g., 4→6 local vs. 4→6 remote). This is because adding more GPUs increases the number of workers used for model broadcast and inter-machine communication is slower than intra-machine communication. As the size of the models is small w.r.t. our network bandwidth, the stopping time of EDL is less than 1 second in all cases.

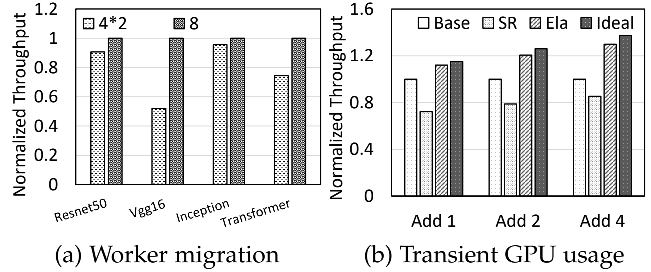
The stopping time of Stop-Resume is significantly longer than EDL because it re-launches a job for scaling and needs to pay expensive initialization cost. For Stop-Resume, scaling to a remote machine is faster than the same machine (e.g., 4→6 local vs. 4→6 remote) because TensorFlow initializes the GPUs on the same machine sequentially and GPUs on different machines in parallel. Thus, scaling to a remote machine overlaps part of the initialization overhead.

We report the *resource loss* (in GPU * time) during scaling out for Stop-Resume and EDL in Figure 12. Resource loss is calculated by multiplying the number of GPUs allocated for a job with the time period in which these GPUs are not used for training during scaling. The results show that the resource loss of EDL is significantly smaller than that of Stop-Resume. This is because Stop-Resume stops all GPUs



(a) Profiling (b) Straggler mitigation

Fig. 13: Performance for profiling and straggler migration



(a) Worker migration (b) Transient GPU usage

Fig. 14: Worker migration and transient GPU usage

during the entire scaling operation. In contrast, for EDL, only the newly added GPUs are not used during scaling while the existing GPUs only need to stop for a short period for model broadcast. We will show in the next subsection, the small resource loss of EDL translates into efficient utilization of transient idle resources. We omit the overhead analysis for scaling in as Stop-Resume has similar overhead for both scaling in and scaling out, while scaling in incurs almost no overhead in EDL thanks to graceful exit.

6.2 The Benefits of Using EDL

In this set of experiments, we demonstrate the benefits brought by EDL in various applications.

Performance profiling. We report the time taken by EDL and Stop-Resume for a profiling job (testing the training performance with 2 to 8 GPUs and running for 10 mini-batches under each parallelism) in Figure 13a. EDL first started the job with 8 GPUs and then gradually scaled in to 2 GPUs. In contrast, Stop-Resume started a new job under each parallelism to measure the performance. The results show that EDL used approximately 10% of the time taken by Stop-Resume to do the same profiling jobs. This is because Stop-Resume needs to pay the expensive context initialization cost repeatedly for each parallelism, while EDL pays the context initialization cost only once at the beginning and then uses low-overhead scale-in operations to adjust the parallelism.

Straggler mitigation. We manually created a straggler for a job running with 16 GPUs, by delaying its gradient synchronization requests by 1/3 of the per-mini-batch time, which is equivalent to limiting its computation capability to 75% of the maximum. Figure 13b shows that the overall throughput also degrades to approximately 75% of the normal case, as all workers need to wait for the straggler in synchronous training. We configured EDL to detect stragglers based on the statistics of the past 10 mini-batches. For all the jobs we tested, EDL took less than 10 seconds to detect the straggler and removed it within 5 seconds using scale-in. After the straggler was removed, the training throughout returned to about 94% of the normal case (with 1 less GPU, i.e., the removed straggler). Note that when there are more stragglers, the detection time and removal time do not increase since they can be removed using one scaling operation.

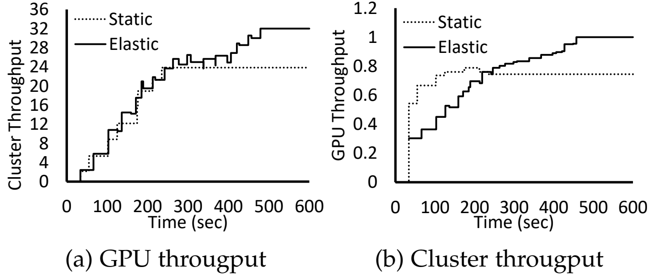
Worker migration. Co-locating GPUs for a job is important for training large models due to costly inter-machine communication. We considered a job running on 2 machines, each using 4 GPUs. We used EDL to migrate the job to one of the machines and run on the 8 GPUs on that machine. We report the training throughput before and after the migration in Figure 14a. For large models, e.g., VGG16, there was an significant increase in throughput (nearly 50%) after migration, though the increase was not obvious for small models (e.g., 5% for Inception). We found that the cost of worker migration was similar to scaling out and training on the target machine was only stopped for less than a second.

Use of transient resources. To validate the benefit EDL can bring out of the transient idle resources, we conducted an experiment using a job that trained ResNet50 with 4 persistent GPUs and considered the cases that add 1, 2 and 4 idle GPUs on the same machine. The idle GPUs were revoked every 4 minutes to simulate the transient idle resources reported in §2.2. Four schemes were used: (1) *Baseline*, which did not use the idle GPUs and used the 4 persistent GPUs for training at all time; 2) *Stop-Resume* (*SR*), which used Stop-Resume for scaling out and scaling in when using the transient idle GPUs; 3) *EDL*, which used EDL for scaling; 4) *Ideal*, which assumed that the scaling completed instantly without any overhead. Note that scaling needed to be conducted twice for each idle interval, i.e., scaling out to add the idle GPUs to training and scaling in to remove these GPUs after the transient period.

Figure 14b shows that EDL achieved at least 97% of the throughput of Ideal. In contrast, Stop-Resume performed even worse than Baseline due to its high scaling overheads, which is in line with our analysis in §2.2. We found that 11.7 minutes is the shortest transient interval needed for Stop-Resume to outperform Baseline with 1 idle GPUs, while EDL only requires the idle interval to be longer than the launch-up time of a worker to outperform Baseline. This result shows that the low scaling overhead enables EDL to utilize idle resources more effectively.

6.3 Performance on Cluster Scheduling

Synthetic workload. To demonstrate the benefits of using EDL in scheduling, we created a synthetic workload to evaluate the performance with/without elasticity. We submitted a job to our cluster using 4 machines, each with 8 GPUs, at every 30 seconds, until 16 jobs were submitted (no job left in the middle). Each job trained a model randomly chosen from the 9 popular DNNs in TensorFlow's official benchmarks [47] (e.g., ResNet, VGG variants) and all jobs ran using 4 GPUs by default. This synthetic workload models different loading conditions that can appear in a production



(a) GPU througput (b) Cluster througput

Fig. 15: Performance on synthetic workload

cluster, i.e., the load was low at the beginning when only a few jobs were running, and gradually the cluster was overloaded.

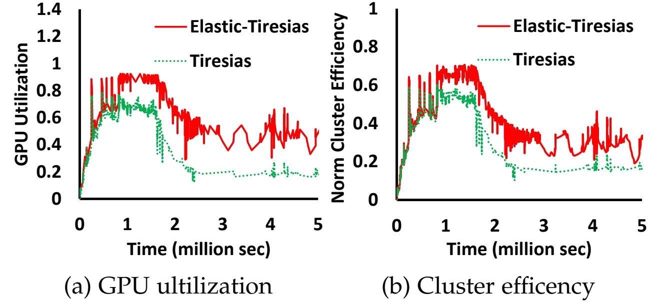
We compared two scheduling strategies: *Static* and *Elastic* (i.e., using EDL). Static ran each job with a static parallelism of 4 and occupied all the GPUs for the first 8 jobs. After that, new jobs were put in a pending queue. Elastic allocated a new job to the least loaded machine (measured by the number of running jobs) and assigned the GPUs on a machine to its jobs uniformly. Elastic also scaled out a job to use any idle GPUs on the machine the job was running, as long as the scale-out does not decrease its throughput [5]. When all GPUs were occupied and a new job was submitted, Elastic scaled in the running job(s) to release GPUs for the new job following the algorithm introduced in §5.1.

We report the *normalized cluster throughput* (cluster throughput for short) and the *average normalized GPU throughput* (GPU throughput for short) of Static and Elastic in Figure 15. For a job running with p GPUs, its normalized throughput is defined as $S(p)/S(1)$, in which $S(q)$ is its training throughput with q GPUs. The cluster throughput is the sum of the normalized throughputs of all running jobs in the cluster, while the GPU throughput is the ratio between the cluster throughput and the number of utilized GPUs. Intuitively, the cluster throughput measures the efficiency of the entire cluster while the GPU throughput measures the efficiency of the utilized GPUs. Figure 15a shows that Elastic achieved higher cluster throughput than Static almost all the time, while Figure 15b shows that the GPU throughput of Elastic was lower than Static at the beginning. This is because Elastic scaled out the jobs to use idle GPUs when the cluster load was light and strong scaling was employed [6], which resulted in lower per-GPU efficiency but higher cluster efficiency. The small spikes on the curves of Elastic were caused by the scaling operations. Both GPU and cluster throughput of Elastic approached their maximum when 16 jobs were running, while those of Static reached their maximum when approximately 8 jobs were running. The results thus verify that using EDL improves the cluster efficiency under different loading conditions.

Production cluster simulation. To show the benefits of using EDL in scheduling a large GPU cluster, we compared Elastic-Tiresias (presented in §5.1) with Tiresias [5]. We used the simulator provided in [5], which has been shown to produce results close to actual execution. The simulation was based on the trace data collected from Microsoft's production cluster [8], [30]. The trace data contains more than 100,000 training jobs, but the model architectures of the jobs are not disclosed. Thus, we followed the same approach in [5] and generated models chosen uniformly at random from TensorFlow's official benchmarks. Both Tiresias and Elastic-Tiresia were configured with three queues (also called *groups* in §5.1) and the service quantum for G_0 and G_1 are 500 GPU*sec and 10,000 GPU*sec, respectively. Elastic-Tiresia uses $N=10$ for the threshold of waiting jobs and $r=0.5$ for the quality of service guarantee.

TABLE 3: Statistics of job completion time (sec)

	Tiresias	Elastic-Tiresias	Reduction (%)
Mean	235,068	24,658	89.5%
Median	1,080	561	48.1%
95th	1,914,470	88,886	95.4%



(a) GPU ultilization (b) Cluster efficency

Fig. 16: Performance on cluster scheduling

We report some statistics of the JCTs of Tiresias and Elastic-Tiresias in Table 3. With elasticity enabled, the JCTs of Tiresias are significantly reduced. To further examine the scheduling performance of Tiresias and Elastic-Tiresias, we plot the GPU utilization rate (i.e., the fraction of GPUs in use) and the cluster efficiency (normalized by the total number of GPUs) in Figure 16. The results show that Elastic-Tiresias achieves higher GPU utilization rate and cluster efficiency than Tiresias. The GPU utilization rate of Elastic-Tiresias is higher because it scales out the jobs to utilize the idle GPUs. The cluster efficiency curve is highly correlated with the curve of the GPU utilization rate, which shows that utilizing the idle GPUs also leads to higher cluster efficiency, which in turn leads to improved JCTs.

7 Related Work

Deep learning schedulers. Instead of using traditional cluster manager such as Yarn [1], Mesos [2], Omega [48] and Borg [49], a number of DL-specialized schedulers are proposed for multi-tenant GPU clusters recently, e.g., Optimus [3], Gandiva [4] and Tiresias [5]. Optimus adjusts the number of parameter servers/workers of MXNet periodically using the Stop-Resume approach to minimize JCT. Gandiva [4] introduces various mechanisms such as *migration, grow-shrink, profiling and suspend-resume* to adjust resource allocation according to runtime measurements.

5. We assume profiling was conducted beforehand such that the scheduler knew the performance of the jobs under different parallelism.

6. As illustrated in Figure 2, when increasing parallelism under strong scaling, throughput usually increases but GPU efficiency decreases.

As *grow-shrink* adjusts the batch size of a job along with the parallelism, Gandiva only uses it when a job is declared to be parallelism insensitive. As introduced in §5.1, Tiresias approximates the shortest-job-first strategy with a priority discretization framework to alleviate head-of-line blocking. EDL positions itself as a system that provides low-overhead elasticity, and can cooperate with existing GPU schedulers by enabling more frequent parallelism adjustments and supporting scheduling mechanisms such as *migration and profiling* efficiently. EDL also provides consistency semantics under elasticity which helps generalize grow-shrink to all jobs.

Elastic ML/DL systems. Machine learning (ML) systems are usually based on the parameter-server architecture [23], [24], [25], [26], [26] and process distributed ML workloads such as Logistic Regression (LR) and Latent Dirichlet Allocation (LDA) [50] in CPU clusters. Elasticity has also been found useful in adapting to resource availability for such workloads. Litz [11] adopts designs such as update forwarding and executor migration to support the dynamic addition/removal of servers and workers. Based on a performance model, Cruise [12] dynamically adjusts the configurations of the parameter servers and workers for optimal performance. EDL focuses on DL workloads that require specialized accelerators such as GPUs. However, its designs for elasticity can also be applied to training ML models such as LR and LDA based on the AllReduce architecture.

Baidu's Paddle EDL [51] is a DL system based on the parameter-server architecture and integrated with Kubernetes. Very recently, Ant Financial also introduced an early-stage ElasticDL project [52], which is based on TensorFlow 2.0. Both systems are designed for asynchronous training and fall back to Stop-Resume if parallelism is adjusted during synchronous training. DL2 [53] supports elasticity on (parameter-server-based) MXNet but it is not clear how DL2 hides the overheads of adding new workers and how the training data is partitioned among a dynamic set of workers. A concurrent work, AutoScaling [54], also supports elasticity on the Allreduce architecture and tries to reduce the overheads of scaling. However, AutoScaling focuses on using elasticity to trade-off between cost (i.e., the number of GPUs) and efficiency (i.e., JCT) for a single job instead of improving multi-tenant cluster management. Some important system issues, such as the interaction with the DL systems and the cluster scheduler and failure recovery are also not considered in AutoScaling. Compared with these existing works, EDL conducted a comprehensive analysis of the potential benefits of elasticity for multi-tenant cluster management and demonstrated with extensive experiments that these benefits can be achieved with EDL. EDL also introduced comprehensive system designs such as user-friendly APIs, dynamic data partitioning and failure recovery to enable elasticity in a real production environment.

After the submission of our work, Horovod released Elastic Horovod, which supports elastic training. EDL differs from Elastic Horovod as follows. Elastic Horovod requires users to specify the details for scaling (e.g., the GPUs to add/remove), while EDL provides a simple API to allow the cluster scheduler to easily add/remove GPUs for a running job, which is more suitable for a multi-tenant cluster. Elastic Horovod assigns training data to workers in a coarse-grained manner and cannot ensure that all data samples are used for training in one epoch in the face of failure. In contrast, EDL's dynamic data pipeline provides such guarantee by carefully recording the worker progresses in the data partitions. More critically, Elastic Horovod does not hide the cost of warming up new workers, which is the main overhead for scaling out, while EDL uses wait-free scaling to eliminate the overhead.

Systems for transient resources. Due to the significantly lower price of preemptible instances on cloud than on-demand ones, many systems have been designed to utilize transient resources [55], [56]. Proteus [57] is a parameter server based ML system that manages models on reliable server nodes and allows workers to be dynamically added or removed to utilize the revocable resources. Hourglass [9] is a graph processing system that partitions a graph into micro partitions and reassigns these micro partitions among the machines when resource changes. Tributary [58] runs web servers using transient resources across different cloud markets to avoid correlated preemptions within one spot market and satisfy quality of service guarantees (e.g., low latency). Flint [59], Pado [60] and TR-Spark [61] focus on batch-processing jobs and use smart checkpointing and task scheduling strategies to minimize the impact of resource revocation. While transient workers usually last for hours in cloud spot markets, EDL considers a more stringent situation where it is common that transient GPU resources are only available for minutes, which necessitates elasticity with low overheads.

8 Conclusions

We presented EDL, which supports elastic GPU utilization with low overheads. EDL can benefit multi-tenant GPU cluster management in many ways, including improving resource utilization by adapting to load variations, maximizing the use of transient idle GPUs, performance profiling, straggler mitigation, and job migration. We showed in our experiments that significant performance benefits can be obtained using EDL in these applications.

Acknowledgments. We thank the reviewers for their constructive comments that help significantly improve the quality of the paper. This work was supported by GRF 14208318 from the RGC of HKSAR.

References

[1] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop YARN: yet another resource negotiator," in *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, 2013, pp. 5:1–5:16. [Online]. Available: https://doi.org/10.1145/2523616.2523633

[2] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[3] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, 2018, pp. 3:1–3:14. [Online]. Available: https://doi.org/10.1145/3190508.3190517

[4] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/xiao

[5] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, 2019, pp. 485–500. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[6] K. Mahajan, A. Singhvi, A. Balasubramanian, V. Batra, S. T. Chavali, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient GPU cluster scheduling for machine learning workloads," *CoRR*, vol. abs/1907.01484, 2019. [Online]. Available: http://arxiv.org/abs/1907.01484

[7] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: quality-driven scheduling for distributed machine learning," in *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 2017, pp. 390–404. [Online]. Available: https://doi.org/10.1145/3127479.3127490

[8] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, 2019, pp. 947–960. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jeon

[9] P. Joaquim, M. Bravo, L. E. T. Rodrigues, and M. Matos, "Hourglass: Leveraging transient resources for time-constrained graph processing in the cloud," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, 2019, pp. 35:1–35:16. [Online]. Available: https://doi.org/10.1145/3302424.3303964

[10] L. Liu and H. Xu, "Elasecutor: Elastic executor scheduling in data analytics systems," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 2018, pp. 107–120. [Online]. Available: https://doi.org/10.1145/3267809.3267818

[11] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing, "Litz: Elastic framework for high-performance distributed machine learning," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 631–644. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/qiao

[12] W.-Y. Lee, Y. Lee, J. S. Jeong, G.-I. Yu, J. Y. Kim, H. J. Park, B. Jeon, W. Song, G. Kim, M. Weimer, B. Cho, and B.-G. Chun, "Automating system configuration of distributed machine learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 2057–2067.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[14] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. (2017) Automatic differentiation in pytorch. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ

[15] Huawei, "Mindspore," https://e.huawei.com/us/products/cloud-computing-dc/atlas/mindspore, 2021.

[16] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 2818–2826. [Online]. Available: https://doi.org/10.1109/CVPR.2016.308

[20] S. Kim, G. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B. Chun, "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, 2019, pp. 43:1–43:15. [Online]. Available: https://doi.org/10.1145/3302424.3303957

[21] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel Distribitued Computing*, vol. 69, no. 2, pp. 117–124, 2009. [Online]. Available: https://doi.org/10.1016/j.jpdc.2008.09.002

[22] Nvidia, "Nvidia nccl," https://developer.nvidia.com/nccl. [Online]. Available: https://developer.nvidia.com/nccl

[23] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, "Flexps: Flexible parallelism control in parameter server architecture," *PVLDB*, vol. 11, no. 5, pp. 566–579, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p566-huang.pdf

[24] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 583–598. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[25] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, 2017, pp. 181–193. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang

[26] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, 2015, pp. 1335–1344. [Online]. Available: https://doi.org/10.1145/2783258.2783323

[27] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville,, ON, Canada, October 27-30, 2019.*, 2019, pp. 16–29. [Online]. Available: https://doi.org/10.1145/3341301.3359642

[28] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed DNN training," *CoRR*, vol. abs/1905.03960, 2019. [Online]. Available: http://arxiv.org/abs/1905.03960

[29] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018. [Online]. Available: http://arxiv.org/abs/1802.05799

[30] M. R. Asia. Project philly traces. https://github.com/msr-fiddle/philly-traces.

[31] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017. [Online]. Available: http://arxiv.org/abs/1706.02677

[32] Y. You, I. Gitman, and B. Ginsburg, "Scaling SGD batch size

to 32k for imagenet training," *CoRR*, vol. abs/1708.03888, 2017. [Online]. Available: http://arxiv.org/abs/1708.03888
[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[34] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=H1oyRlYgg
[35] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 1731–1741. [Online]. Available: http://papers.nips.cc/paper/6770-train-longer-generalize-better-closing-the-generalization-gap-in-large-batch-training-of-neural-networks
[36] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu, "Seastar: Vertex-centric programming for graph neural networks," in *Proceedings of the Fourteenth EuroSys Conference 2021, April 26-28, 2021*. ACM, 2021.
[37] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "Dgcl: An efficient communication library for distributed gnn training," in *Proceedings of the Fourteenth EuroSys Conference 2021, April 26-28, 2021*. ACM, 2021.
[38] J. Haochen and S. Sra, "Random shuffling beats SGD after finite epochs," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019, pp. 2624–2633. [Online]. Available: http://proceedings.mlr.press/v97/haochen19a.html
[39] L. Bottou, "Curiously fast convergence of some stochastic gradient descent algorithms," in *Proceedings of the symposium on learning and data science, Paris*, 2009.
[40] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010. [Online]. Available: https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems
[41] etcd. (2019) etcd. [Online]. Available: https://github.com/etcd-io/etcd
[42] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/MSST.2010.5496972
[43] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, 2013, pp. 439–455. [Online]. Available: https://doi.org/10.1145/2517349.2522738
[44] J. Nagle, "Congestion control in IP/TCP internetworks," *RFC*, vol. 896, pp. 1–9, 1984. [Online]. Available: https://doi.org/10.17487/RFC0896
[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008. [Online]. Available: http://papers.nips.cc/paper/7181-attention-is-all-you-need
[46] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds. ACM, 2016, pp. 4:1–4:16. [Online]. Available: https://doi.org/10.1145/2901318.2901323
[47] Tensorflow. (2019) tf cnn benchmark. [Online]. Available: https://github.com/tensorflow/benchmarks/tree/master/scripts/tf-cnn-benchmarks
[48] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, 2013, pp. 351–364. [Online]. Available: https://doi.org/10.1145/2465351.2465386
[49] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, 2015, pp. 18:1–18:17. [Online]. Available: https://doi.org/10.1145/2741948.2741964
[50] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," in *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, 2001, pp. 601–608. [Online]. Available: http://papers.nips.cc/paper/2070-latent-dirichlet-allocation
[51] Baidu. (2019) paddlepaddle. [Online]. Available: https://www.paddlepaddle.org.cn/
[52] A. Financial. (2019) Elasticdl. [Online]. Available: https://github.com/sql-machine-learning/elasticdl
[53] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, "DL2: A deep learning-driven scheduler for deep learning clusters," *CoRR*, vol. abs/1909.06040, 2019. [Online]. Available: http://arxiv.org/abs/1909.06040
[54] A. Or, H. Zhang, and M. J. Freedman, "Resource elasticity in distributed deep learning," in *Proceedings of the Third Conference on Machine Learning and Systems, MLSys2020. Austin, USA, March 2-4, 2020*.
[55] Amazon. (2019) Amazon ec2 spot instance. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html
[56] G. Cloud. (2019) Google pre-emptible vm instances. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html
[57] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: agile ML elasticity through tiered reliability in dynamic resource markets," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, 2017, pp. 589–604. [Online]. Available: https://doi.org/10.1145/3064176.3064182
[58] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons, "Tributary: spot-dancing for elastic services with latency slos," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/harlap
[59] P. Sharma, T. Guo, X. He, D. E. Irwin, and P. J. Shenoy, "Flint: batch-interactive data-intensive processing on transient servers," in *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, 2016, pp. 6:1–6:15. [Online]. Available: https://doi.org/10.1145/2901318.2901319
[60] Y. Yang, G. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B. Chun, "Pado: A data processing engine for harnessing transient resources in datacenters," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, 2017, pp. 575–588. [Online]. Available: https://doi.org/10.1145/3064176.3064181
[61] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "Tr-spark: Transient computing for big data analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, 2016, pp. 484–496. [Online]. Available: https://doi.org/10.1145/2987550.2987576

Yidi Wu is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include machine learning systems, distributed data processing systems, and cluster scheduling systems.

Kaihao Ma is currently a Ph.D. student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include deep learning and graph neural networks.

Xiao Yan is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include large-scale similarity search, distributed machine learning, federated learning and graph neural networks.

Zhi Liu is currently a Ph.D. student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include graph databases, stream processing systems and cluster management.

Zhenkun Cai is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include distributed deep learning, graph neural network systems and GPU cluster scheduling.

Yuzhen Huang is a Research Scientist at Facebook. He obtained his Ph.D. degree from the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include distributed deep learning systems, large-scale machine learning and distributed data analytic systems.

James Cheng is currently an associate professor in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include graph databases, distributed computing systems, cluster management and job scheduling, graph neural networks, and large-scale similarity search.

Han Yuan received her Ph.D. degree from Harbin Institute of Technology. She is currently a researcher in the Theory of Computation Lab of Huawei. Her research interests include deep learning and combinatorial optimization.

Fan Yu received his Ph.D. degree from University of Science and Technology of China. He has been working in Huawei for 10 years and is currently the leading architect of Huawei's MindSpore. He has led the development of Huawei's MindSpore, Huawei Cloud cluster management system and large scale SDN routing architecture. He has obtained over 30 patents.