An Efficient and Non-Intrusive GPU Scheduling Framework for Deep Learning Training Systems

Shaoqi Wang*, Oscar J. Gonzalez†, Xiaobo Zhou*, Thomas Williams†,
Brian D. Friedman†, Martin Havemann† and Thomas Woo†
*Department of Computer Science, University of Colorado, Colorado Springs, CO, USA
†Nokia Bell Labs, New Providence, NJ, USA

***Abstract*—Efficient GPU scheduling is the key to minimizing the execution time of the Deep Learning (DL) training workloads. DL training system schedulers typically allocate a fixed number of GPUs to each job, which inhibits high resource utilization and often extends the overall training time. The recent introduction of schedulers that can dynamically reallocate GPUs has achieved better cluster efficiency. This dynamic nature, however, introduces additional overhead by terminating and restarting jobs or requires modification to the DL training frameworks.**

We propose and develop an efficient, non-intrusive GPU scheduling framework that employs a combination of an adaptive GPU scheduler and an elastic GPU allocation mechanism to reduce the completion time of DL training workloads and improve resource utilization. Specifically, the adaptive GPU scheduler includes a scheduling algorithm that uses training job progress information to determine the most efficient allocation and reallocation of GPUs for incoming and running jobs at any given time. The elastic GPU allocation mechanism works in concert with the scheduler. It offers a lightweight and non-intrusive method to reallocate GPUs based on a "SideCar" process that temporarily stops and restarts the job's DL training process with a different number of GPUs. We implemented the scheduling framework as plugins in Kubernetes and conducted evaluations on two 16-GPU clusters with multiple training jobs based on TensorFlow. Results show that our proposed scheduling framework reduces the overall execution time and the average job completion time by up to 45% and 63%, respectively, compared to the Kubernetes default scheduler. Compared to a termination-based scheduler, our framework reduces the overall execution time and the average job completion time by up to 20% and 37%, respectively.

***Index Terms*—deep learning, GPU clusters, resource scheduling, container, Kubernetes**

I. Introduction

Deep Learning (DL) [1]–[3] has achieved remarkable success across a wide range of applications, including image recognition, object detection and natural language processing [4]–[6], and this has stimulated significant interest in applying DL to commercial products. Enterprises are now building flexibility into new training systems and tools so that they can efficiently share a cluster of GPUs in both public and private cloud environments to support the concurrent execution of multiple DL training jobs [7], [8].

DL training systems rely on an orchestration platform (e.g., Kubernetes [9]) to manage the lifecycle of the training jobs, and often, to allocate the computational resources needed. The platform uses one or more Docker containers to run a training job and launches the DL training processes in the containers. The training jobs utilize distributed training frameworks (e.g., TensorFlow [10] and PyTorch [11]) to efficiently handle large training datasets by employing multiple GPUs in parallel.

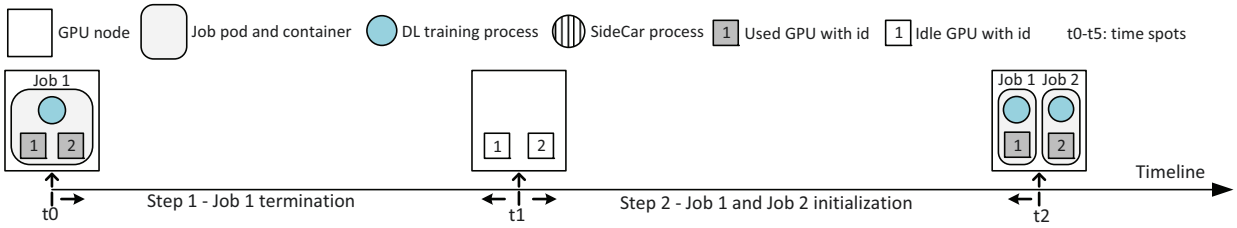
In a DL training system, efficient GPU scheduling is the key to minimize the overall training time (the makespan) of a set of jobs. The default scheduler of one of the most popular orchestration systems, Kubernetes [9], [12], allocates a fixed number of GPUs to each job, based on its specifications. Use of this type of scheduler has two major drawbacks that need to be resolved in order to improve the efficiency of GPU cluster usage: (1) it typically under-utilizes the resources and (2) it typically extends the makespan of the workload.

With regard to the first drawback, such schedulers are not capable of reallocating idle GPUs to other running jobs when a given job completes, and so such idle GPUs continue to sit idle until another training job is submitted. This can cause resource under-utilization. With regard to the second drawback, when running jobs hold all of the GPUs in the cluster, incoming jobs are queued. Thus, the makespan will be longer than that when a subset of currently used GPUs are reallocated to incoming jobs. We refer to the GPU reallocation capability as "reshaping". Specifically, reallocating idle GPUs to running jobs is referred to as "reshaping up" and reallocating used GPUs to incoming jobs is referred to as "reshaping down".

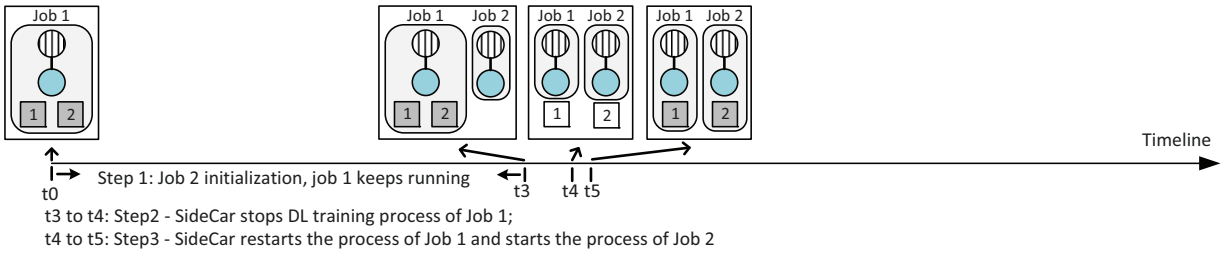
Calculating the gains from reshaping is complicated because the relationship between the training speed of a DL job and the number of GPUs allocated to the job is not linear and is subject to diminishing returns [8], [13], [14]. For example, in training a ResNet model [15], when the number of allocated GPUs increases from one to two, the training speed becomes 1.7x compared to the speed with one GPU (i.e., a 70% improvement). When the number of GPUs increases from two to four, the training speed becomes 2.4x (a 41% improvement compared to the speed with two GPUs). When a running job holds all four GPUs in a cluster and achieves 2.4x training speed, the default scheduler would queue an incoming job. Instead, removing two GPUs from the running job and reallocating them to the incoming job (i.e., "reshaping down") would be a better scheduling strategy that reduces the makespan of the two jobs. In this case, the two jobs both obtain 1.7x training speed.

Reshaping has the potential to significantly improve the efficiency of GPU cluster usage. Recent efforts propose advanced schedulers that use reshaping to dynamically reallocate GPUs

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE



(a) Reshaping based on job termination (without SideCar).



(b) Reshaping with SideCar.

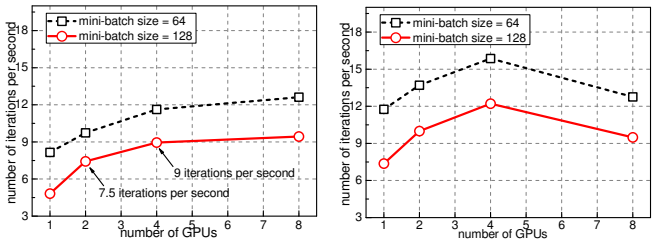
Fig. 1. Steps involved in an "reshaping down" example, in which GPU 2 used by Job 1 is reallocated to Job 2.

in DL training clusters. To minimize the makespan of multiple DL jobs and improve resource utilization, Optimus [14] employs the idea of "reshaping up" and allocates idle GPUs to running jobs one by one. Optimus also quantizes the impact of GPU allocation to the makespan by predicting the job completion time with different numbers of allocated GPUs, thus each allocation aims to achieve makespan minimization. However, Optimus does not support "reshaping down". In each allocation, it has to first terminate a job and then restart it with reallocated GPUs, which incurs additional overhead. In Gandiva [8], when a cluster runs out of idle GPUs, incoming jobs can share GPUs with running jobs. But this "reshaping down" approach is intrusive, requiring source code modifications to the DL training frameworks. The issue with modifying the DL training frameworks is that since such frameworks are updated rather frequently, each time a new version of a framework is released and applied, the old version and its modifications will be lost, and thus, the original modifications would need to be reimplemented in the new version. The scheduler in Gandiva applies this intrusive approach to reduce early feedback latency in DL model training instead of minimizing the makespan.

In order to design an efficient and non-intrusive GPU scheduler, we have to address two primary issues in the orchestration platform that cause reshaping to be inefficient. First, once the orchestration platform instantiates the containers for a given job, reallocation of GPUs requires complete termination and restarting of the job, including pods (basic execution unit in Kubernetes that encapsulates containers of a job), the containers, and the DL training processes within the containers. Second, when GPUs are removed from a running job and reallocated to an incoming job, the orchestration platform cannot proactively initialize the incoming job in advance. The incoming job initialization must wait until the running job is terminated and its GPUs are released. As a result, GPUs are idle during the termination and initialization phases in the "reshaping down". The second issue is due to limitations in the NVIDIA device plugin [16].

In this paper, we propose and develop an efficient, non-intrusive GPU scheduling framework that employs a combination of an adaptive GPU scheduler and an elastic GPU allocation mechanism to reduce makespan and improve resource utilization. The adaptive GPU scheduler includes an algorithm which employs training job progress information to determine the most efficient allocation and reallocation of GPUs for the incoming and running jobs at any given time. The reallocation of GPUs supports both "reshaping up" and "reshaping down". The elastic GPU allocation mechanism works in concert with the scheduler. It disables the NVIDIA device plugin and offers a lightweight method to efficiently reallocate GPUs based on a "SideCar" process.

SideCar is deployed as a process that is co-resident with the DL training process in the same container. The SideCar process performs two main functions. First, it has the ability to stop and restart the co-resident DL training process of a job so that reshaping does not require termination and restarting of the job and its pods and containers. Second, SideCar includes an *early initialization* feature in the "reshaping down" period whereby that the incoming job initialization can be conducted before the DL training processes of the running job release their GPUs. Figure 1 is an example that depicts the steps involved in "reshaping down". In the example, Job 2 is submitted at time *t0*, after which one GPU used by a running job (Job 1) is reallocated to Job 2. Without SideCar, Job 1 termination starts at time *t0* and finishes at *t1*. After termination, Job 1 and Job 2 initializations start at *t1* and finish at *t2*. Thus, the two GPUs remain idle from time *t0* to *t2*. With SideCar, the idle period only ranges from time *t3* to *t5*. Thus, SideCar can significantly reduce the overhead



(a) ResNet-18 model. (b) ResNet-34 model.

Fig. 2. Non-linear relationship between training speed and number of GPUs.

in reshaping. Note that SideCar does not modify the source code of any DL training frameworks, and thus, the scheduling framework is non-intrusive.

In summary, we make the following technical contributions: (1) we propose and develop an adaptive scheduler that uses training job progress information to allocate and reallocate GPUs so as to minimize the makespan of multiple DL jobs. The novelty lies in supporting both "reshaping down" and "reshaping up" in GPU reallocation; (2) we design and develop an elastic GPU allocation mechanism that further improves the efficiency of reshaping. The novelty lies in achieving both lightweight and non-intrusive GPU allocation; (3) the scheduler and elastic mechanism constitute our GPU scheduling framework and we implement the framework as plugins in Kubernetes. We performed evaluations with multiple TensorFlow jobs. Results show that our framework reduces the makespan and the average job completion time by up to 45% and 63%, respectively, compared to the default scheduler. Compared to the termination-based scheduler in Optimus, our framework reduces the makespan and the average job completion time by up to 20% and 37%, respectively.

In the following, Section II provides motivations and a case study. Section III describes the design of the scheduling framework. Section IV provides the implementation details. Section V and Section VI present the experimental setup and evaluation results. Section VII reviews related work. Section VIII concludes the paper.

II. Background and Motivation

TABLE I
TWO DL JOBS IN THE CASE STUDY.

Job id	Submission time	Resources	# of epochs	# of containers
1	0 second	four GPUs	1	1
2	180^{th} second	two GPUs	1	1

A. Distributed DL Training

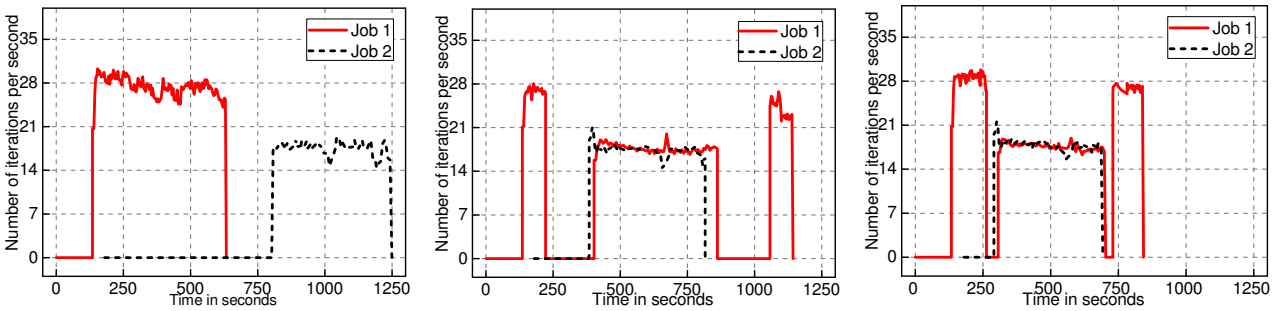
A DL job trains a deep neural network (DNN) model using a training dataset to minimize a loss function. As DNN models become more sophisticated and training datasets become increasingly larger, distributed DL training in GPU clusters is becoming more prevalent. Distributed DL training involves the use of multiple workers to perform the training. Data parallelism is the most common strategy for distributed DL training in popular training frameworks. In data parallelism, the training dataset is divided into equal-sized data batches, and each data batch is further divided into equal-sized mini-batches. The number of mini-batches in one data batch equals the number of GPUs involved in training. In each training iteration, each GPU trains its local DL model and generates local gradients by processing one mini-batch. The synchronous parallel model [17] is widely used in synchronization Alternative models are SSP [18], ASP [19], and A-BSP [20].

In each iteration, each GPU first performs forward and backward computation with one mini-batch, and then synchronizes local gradients with others to update the DNN model. Since the computation load and the communication volume are exactly the same across iterations, the time of one iteration is highly predictable. Moreover, when all mini-batches in the training dataset have been processed once, one training epoch is completed. The number of iterations in one epoch equals the number of data batches, and thus, when the specified number of epochs are completed, the DL job finishes. Therefore, the training time of a distributed DL job is highly predictable when the number of epochs is specified. Note that the number of epochs to train in a distributed DL job can typically be specified as a command-line parameter. When the number of epochs is not specified, a DL job finishes when the validation accuracy (e.g., Top-5 accuracy [21]) reaches a desired level. In this case, previous studies [14], [22] track the training progress on the fly and use online fitting to predict the number of epochs required to achieve the desired accuracy.

B. GPU Scheduling

Static GPU allocation in default scheduler. In a GPU cluster with multiple DL training jobs submitted over time, efficient GPU scheduling is the key to minimize the makespan of the jobs. The popular orchestration platform Kubernetes runs a DL job using one or more Docker containers and launches the DL training processes inside them. Its default scheduler allocates a fixed number of GPUs to each job, based on its specifications. The number of allocated GPUs remains static during training.

Non-linear performance gain. The number of GPUs allocated to a job significantly influences the training speed and hence the training completion time. Recent research shows that, when the number of allocated GPUs increases, the communication overhead increases [13], [14], [23]. Thus, there is no linear relationship between the training speed of a job and the number of allocated GPUs. Figure 2 shows the training speed with different numbers of GPUs using the AllReduce architecture when we train ResNet-18 and ResNet-34 models, two of the state-of-the-art DNNs for image classifcation, on the ImageNet dataset [24] with different mini-batch sizes. We see that increasing GPUs does not lead to linear training speed improvement and can even slow down model training.



(a) Default static GPU allocation. (b) GPU allocation based on job termination. (c) The elastic GPU allocation mechanism.

Fig. 3. Microscopic views of the execution of two jobs.

C. A Case Study

We created a 4-GPU cluster to demonstrate the drawbacks of the default scheduler. Specifically, two DL jobs are submitted at different times by a user. Table I gives the job details. The performance metrics include makespan and the average job completion time. Three approaches are examined: (1) the default static GPU allocation, (2) an advanced GPU allocation approach with reshaping that was used in Optimus [14] and the reshaping is performed based on job termination, and (3) the proposed elastic GPU allocation mechanism based on SideCar.

Figure 3 shows the microscopic views of the execution of two jobs using the three approaches. Figure 3(a) shows the number of iterations per second in the two jobs due to the default approach. Specifically, when Job 1 is submitted at time 0, Kubernetes initializes the job and allocates four GPUs to start training. When Job 2 is submitted at the 180^{th} second, all cluster GPUs are used by Job 1 and so Job 2 is queued. At the 633^{rd} second, Job 1 finishes all of its epochs and releases all GPUs. After that, Kubernetes initializes Job 2 and allocates two GPUs to the job to start training. At the 807^{th} second, Job 2 starts running with two GPUs. At the 1249^{th} second, Job 2 finishes all of its epochs.

Figure 3(b) shows the execution process using the approach that is based on job termination. When Job 2 is submitted at the 180^{th} second, this approach removes two GPUs from Job 1 by first terminating the job and then restarting it with only two GPUs. After termination, Kubernetes initializes Job 2 and allocates the removed two GPUs to the job. At the 384^{th} second, Job 2 starts running. At the 815^{th} second, Job 2 finishes all of its epochs. To utilize the two GPUs released by Job 2, this approach allocates them to Job 1 based on job termination. At the 1057^{th} second, Job 1 starts running with four GPUs. At the 1143^{rd} second, Job 1 finishes all of its epochs.

Figure 3(c) shows the execution process using the proposed elastic GPU allocation mechanism. According to the early initialization feature, Job 2 is initialized after it is submitted at the 180^{th} second. After initialization, SideCar in Job 1 stops and restarts its DL process with only two GPUs. Then, SideCar in Job 2 starts its DL process at the 290^{th} second immediately after two GPUs become available. At the 692^{nd} second, Job 2 finishes all of its epochs and releases its two GPUs. After that, SideCar in Job 1 stops and restarts its DL process. At the 729^{th} second, Job 1 starts running with four GPUs, At the 845^{th} second, Job 1 finishes all of its epochs.

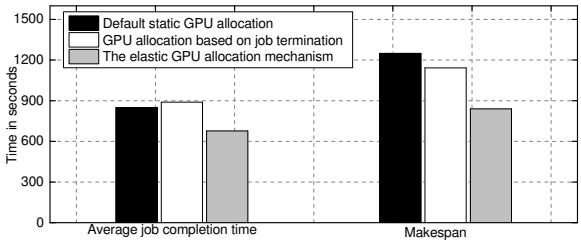


Fig. 4. Performance due to the three approaches.

Figure 4 depicts the performance using the three different approaches. Overall, the default approach leads to the longest makespan. The reasons are twofold. First, the other two approaches conduct "reshaping down" such that the two GPUs used by Job 1 are reallocated to Job 2. The "reshaping down" can reduce makespan because of the non-linear performance gain. Second, the default approach cannot reallocate two idle GPUs to Job 2 while it is running, leading to resource under-utilization. In contrast, the other two approaches conduct "reshaping up" to improve the utilization. The approach based on job termination introduces in non-trivial reshaping overhead. In the case study, this results in the largest average job completion time because the two jobs are relatively small and thus the overhead becomes more severe. Our mechanism achieves the shortest makespan and average job completion time due to its elasticity. It spends 27 and 37 seconds on the "reshaping down" and "reshaping up", respectively, which is much shorter than the time (204 and 242 seconds) spent in the termination-based approach.

III. Scheduling Framework Design

A. Architecture and Workflow

Figure 5 illustrates the architecture of the scheduling framework whose components are shown in grey. Specifically, the elastic GPU allocation mechanism consists of four out of the five scheduling framework components: an In-memory Database, a Job Launcher, a GPU Coordinator, and SideCar

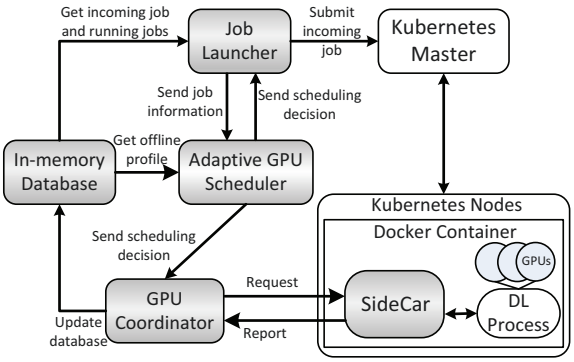


Fig. 5. The architecture of the scheduling framework (in grey).

processes. Note that one SideCar process is deployed in each Docker container. The SideCar process receives a request from the GPU Coordinator and sends back a corresponding report. The SideCar process is also able to restart the DL training process according to the received request.

Placing a SideCar process in the same container as the DL training process enables resource allocation to be conducted with fine granularity. In particular, without SideCar, GPU allocation is conducted at the job level, and thus requires restarting the entire job, including its associated containers and all of the processes inside them. With SideCar, GPU allocation is conducted at the process level, and thus it only requires restarting the DL training processes inside the containers of a given job.

Workflow for incoming job. When a job is submitted, the framework allocates GPUs to the job. The workflow contains two phases: early initialization and GPU allocation.

(1) *Early initialization.* The Job Launcher first gets the information of the incoming job and all running jobs from the In-memory Database. The information is sent to the Adaptive GPU Scheduler which also retrieves offline profile from the In-memory Database. Based on this information, the scheduler then determines the number of GPUs to be allocated to the incoming job. The allocated GPUs include idle GPUs and GPUs to be removed from a running job. The scheduler sends its scheduling decision to both the Job Launcher and the GPU Coordinator. Based on the decision, the Job Launcher submits the incoming job to Kubernetes so as to initialize it. Finally, after the incoming job is initialized, the SideCar in the job reports to the GPU Coordinator.

(2) *GPU allocation.* After the incoming job is initialized, the GPU Coordinator sends a request to the SideCar in the running job based on the scheduling decision. The request specifies the GPUs to be removed. After the SideCar stops and restarts the DL training processes of the running job, the GPU Coordinator sends a request to the SideCar in the incoming job. The SideCar starts the DL training processes with the allocated GPUs including idle GPUs and the GPUs removed from the running job. Finally, the GPU Coordinator updates the In-memory Database accordingly. Note that, if the scheduling decision does not remove GPUs from a running job, the SideCar in the incoming job starts its processes immediately after initialization.

Workflow for running job. When there are idle GPUs but no incoming jobs, the framework adds idle GPUs to running jobs. This process is performed in three steps. First, the Job Launcher sends the information of all running jobs to the Adaptive GPU Scheduler. The scheduler then selects a running job to which idle GPUs can be added. Finally, the GPU Coordinator updates the In-memory Database and sends a request to the SideCar in the running job. The request specifies the GPUs to be added.

B. Elastic GPU Allocation Mechanism

TABLE II
AN EXAMPLE OF GPU INFORMATION IN THE IN-MEMORY DATABASE.

GPU id	State	Job id	Worker id
0	in use	0	0
1	in use	0	1
2	in use	1	0
3	reserved	2	null

TABLE III
AN EXAMPLE OF JOB INFORMATION IN THE IN-MEMORY DATABASE.

Job id	State	number of GPUs	GPU ids	Job type	optional numbers of GPUs
0	running	2	0, 1	0	1,2,4
1	adjusting	1	2	1	1,2,4
2	initialized	1	3	2	1,2,4
3	submitted	0	null	3	1,2,4,8
4	completed	0	null	4	1,2,4,8

In our elastic GPU allocation mechanism, SideCar is the key component that enables resource allocation to be conducted with fine granularity. Other components manage jobs and GPUs in the cluster and work in concert with SideCar. In particular, the In-memory Database stores GPU cluster information and the offline job profile that is used by the Adaptive GPU Scheduler. The Job Launcher continuously monitors the job queue for pending (as yet unscheduled) and newly submitted jobs. After obtaining a scheduling decision from the Adaptive GPU Scheduler, it uses manifest files to launch job containers in Kubernetes. The GPU Coordinator is responsible for communicating with the SideCar process itself.

1) In-memory Database: The In-memory Database contains a global view of all GPUs and jobs in a Kubernetes cluster. To accomplish this, we use two tables to store the global information on GPUs and jobs. Tables II and Table III show an example of job information. A job contains one or more workers. When a GPU is allocated to a job, it can only be used by one worker in the job. Note that each job can be configured with several optional numbers of GPUs, specifying the number of GPUs that can be allocated to the job. For example, when the optional numbers of GPUs of a job are 1, 2, and 4, the job can run successfully with 1, 2, and 4 GPUs, respectively.

The database also stores an offline job profile that includes a training speed profile and a training accuracy profile. The

training speed profile is used to predict the job completion time for a certain number of epochs. If the number is not specified, the training accuracy profile is used to initialize an online fitting model that predicts the number of epochs. Specifically, we classify all jobs into different types of jobs based on DNN model, training dataset, and hyperparameters. For each type, the training speed profile contains the training time of one iteration with different numbers and types of GPUs, and the training accuracy profile contains Top-5 accuracies with different numbers of iterations. Tables IV and V show examples of the two types of profiles. Note that profiling is inexpensive because it only requires running each type for a small number of iterations. The offline profile also includes job termination and initialization time in Kubernetes, training process termination and initialization time in DL training framework, and model checkpoint time.

TABLE IV
AN EXAMPLE OF TRAINING SPEED PROFILE.

Job type	GPU type	# of GPUs	Time of one iteration
0	1080Ti	1	0.31 second
0	1080Ti	2	0.23 second
0	P100	1	0.25 second
0	P100	2	0.19 second
0	V100	1	0.18 second
0	V100	2	0.14 second

TABLE V
AN EXAMPLE OF TRAINING ACCURACY PROFILE.

Job type	# of iterations	Top-5 accuracy
0	1	0.008
0	5	0.019
0	10	0.025
0	20	0.032

2) Job Launcher: The Job Launcher periodically checks the states of all GPUs and jobs by querying tables in the In-memory Database. When there is an incoming job, the Job Launcher triggers GPU scheduling by sending information about all GPUs and jobs to the scheduler. After receiving the scheduling decision from the scheduler, the Job Launcher submits the incoming job to Kubernetes. In order to do this, the Job Launcher is responsible for building the Docker image of the job including its training code, dataset, training framework, SideCar, and other dependencies. It also triggers scheduling when there are idle GPUs but no incoming jobs.

3) GPU Coordinator: The GPU Coordinator provides three main functions. 1) It receives the scheduling decision from the scheduler and forwards the decision to SideCar to ensure that the scheduling decision is applied. 2) It sends the scheduling decision at the correct moment to reduce GPU idle time and avoid conflict by preventing multiple jobs from running simultaneously on the same GPU. In particular, when the scheduler decides to remove GPUs from a running job and allocate them to an incoming job, the GPU Coordinator requests the running job to remove GPUs after the incoming job has been initialized. After removing the GPUs, it informs the incoming job to start running. In this case, the removed GPUs are still utilized by the running job during the initialization of the incoming job, and the removed GPUs become idle just before the incoming job starts its DL training processes. 3) Once the GPU Coordinator receives a report from SideCar, it updates the In-memory Database. The goal is to maintain the consistency between the states of GPUs and jobs in Kubernetes cluster and the corresponding information in the In-memory Database.

4) SideCar: Each job's Docker image includes the SideCar code, which is invoked when the container is instantiated by Kubernetes at the direction of the Job Launcher. SideCar receives requests from the GPU Coordinator and sends updates to it. There are two types of requests. The first type of request is sent to the SideCar in an incoming job that has been initialized, after which the SideCar starts its DL training processes. The second type of request is sent to the SideCar in a running job. The request specifies the number of GPUs that should be removed from or added to the job. After receiving the request, the SideCar stops and restarts the DL training processes with the adjusted number of GPUs.

There are two types of updates. The first type of update is the acknowledgment to confirm that the request from the coordinator has been executed successfully. The second type of update contains the utilization information of the associated GPUs. Specifically, SideCar continues to monitor the usage of GPUs and the progress of the DL training processes. It periodically reports the collected information to the GPU Coordinator, which, in turn, updates the In-memory Database.

Algorithm 1 Incoming job scheduling with "reshaping down"

```
/* select running jobs whose GPUs are removable */
The incoming job: J_new;
Set S_opt_J_new: the optional numbers of GPUs of J_new;
The number of idle GPUs: N_idle;
Initialize a hash table: hashmap;
for job J_i in running jobs
    Set S_opt_J_i: the optional numbers of GPUs of J_i; The maximal number is N_max and the minimal number is N_min;
    N_current: the number of GPUs used by J_i;
    for N_remove ranges from N_min to N_max:
        if N_current - N_remove is in S_opt_J_i:
            add N_remove and J_i in hashmap;
/* decide the number of GPUs allocated to J_new */
Initialize T = [0, null, 0] and MS_min = infinity;
for optional number of GPUs N_opt in set S_opt_J_new:
    if N_idle ⩾ N_opt
        Predict completion time of J_new with N_opt GPUs based on profile;
        Predict makespan MS_predict;
        if MS_predict < MS_min
            MS_min = MS_predict;
            T = [N_opt, null, 0];
    else
        N_remove = N_opt − N_idle;
        Get set S_job_remove from hashmap;
        for job J_i in set S_job_remove
            Predict remaining running time of J_i based on profile;
            Predict completion time of J_new with N_opt GPUs based on profile;
            Predict makespan MS_predict;
            if MS_predict < MS_min
                MS_min = MS_predict;
                T = [N_opt, J_i, N_remove];
return T;
```

C. Adaptive GPU Scheduler

The Adaptive GPU Scheduler provides efficient GPU scheduling via two scheduling modes: *incoming job scheduling*

Algorithm 2 Running job scheduling with "reshaping up"

```
1: /* select running jobs who are able to add GPUs */
2: The number of idle GPUs: N_idle;
3: Initialize a hash table: hashmap;
4: for job J_i in running jobs
5:     Set S_opt_J_i: the optional numbers of GPUs of J_i;
6:     N_current: the number of GPUs used by J_i;
7:     for N_add ranges from 1 to N_idle:
8:         if N_current + N_add is in S_opt_J_i:
9:             Add N_add and J_i in hashmap;
10: /* select the job to add GPUs */
11: Initialize T = [null, 0] and MS_min = infinity;
12:     for N_add ranges from 1 to N_idle:
13:         Get set S_job_add from hashmap;
14:         for job J_i in set S_job_add
15:             Predict remaining time of J_i based on profile;
16:             Predict makespan MS_predict;
17:             if MS_predict < MS_min
18:                 MS_min = MS_predict;
19:                 T = [J_i, N_add];
20: return T;
```

and *running job scheduling*. The goal of both is to minimize the makespan. Incoming job scheduling is triggered when a new job is submitted. The scheduling not only allocates idle GPUs to the new job but also supports "reshaping down". Running job scheduling is triggered when there are idle GPUs but no incoming jobs, and supports "reshaping up" to improve resource utilization. The novelty of the Adaptive GPU Scheduler lies in supporting both "reshaping down" and "reshaping up" in GPU scheduling for makespan minimization.

1) Incoming Job Scheduling: The scheduling policy for an incoming job is shown in Algorithm 1. The algorithm first checks the number of idle GPUs in the cluster (lines 1 to 4). It then initializes a hash table, in which the key is the number of GPUs N_remove and the value is the set of jobs from which N_remove GPUs can be removed (line 5). Note that when removing GPUs from a job, the number of remaining GPUs is equal to one of the job's optional numbers of GPUs (lines 6 to 11). The GPU allocation scheme is notated as a triple T (line 13), in which the first element is the number of GPUs allocated to the incoming job, the second is the running job from which GPUs are to be removed, and the third specifies the number of GPUs to be removed. The algorithm also maintains a global minimal makespan MS_min (line 13).

To obtain the scheme with the minimal makespan, the algorithm iterates through every optional number of GPUs specified in the resource requirements of the incoming job (line 14). For an optional number N_opt, the algorithm predicts the job completion time with N_opt GPUs (line 16 and line 26). If the number of idle GPUs is smaller than N_opt (line 23), it calculates the number of GPUs to be removed (N_remove) and iterates through every running job from which N_remove GPUs can be removed (lines 22 to 24). For a running job, the algorithm predicts its remaining running time when N_remove GPUs are removed. Finally, the algorithm predicts the makespan $MS\_predict$ based on the predicted remaining running time of the running job and the job completion time of the incoming job. If $MS\_predict$ is smaller than the global minimal makespan, both MS_min and T are updated (lines 19 to 20 and lines 29 to 30).

The predicted completion time of a new job is the sum of job initialization time in Kubernetes, training process initialization time in the DL training framework, model training time for a certain number of training epochs, model checkpoint time before termination, training process termination time in the DL training framework, and job termination time in Kubernetes. Similarly, the predicted remaining running time of a running job is the sum of model checkpoint time before restarting training process, training process termination and initialization time in the DL training framework, model training time for a certain number of remaining training epochs, model checkpoint time before termination, training process termination time in the DL training framework, and job termination time in Kubernetes.

When the number of training epochs is not specified, the scheduler employs the online fitting method of Optimus [14] to predict the number of remaining training epochs required to achieve a desired validation accuracy. In particular, the scheduler first obtains training accuracy data points from the training accuracy profile and then collects more data points during the training. Each data point is a pair of training accuracy and the number of iterations. Based on the points collected so far, the scheduler uses a non-negative least squares solver [14] to build a training accuracy curve that fits the points. Finally, the scheduler uses the accuracy curve to predict the number of epochs. Since the scheduler can collect more and more data points as the job progresses, the prediction accuracy improves continuously.

Note that the algorithm considers that each "reshaping down" involves one running job and one incoming job. Based on this consideration, the algorithm enumerates all possible values for the three elements in the triple T and chooses the values that achieve the minimum makespan. Thus, the algorithm derives the optimal GPU allocation. The time complexity is O(n*m*k), in which n is the number of optional numbers of GPUs in Table III, m is the number of running jobs, and k is the average number of GPUs used by running jobs.

2) Running Job Scheduling: This scheduling policy is shown in Algorithm 2. The algorithm checks the number of idle GPUs in the cluster and initializes a hash table where the key is the number of GPUs N_add and the value is the set of jobs that are able to add N_add GPUs (lines 2 to 3). Note that when adding GPUs to a job, the total number of allocated GPUs is equal to one of the job's optional numbers of GPUs (lines 4 to 9).

The GPU allocation scheme is notated as a tuple T (line 11), in which the first element is the running job to which GPUs will be added and the second specifies the number of GPUs to be added. The algorithm also maintains a global minimal makespan MS_min (line 11). To determine the scheme with the minimal makespan, the algorithm first iterates through the number of idle GPUs (N_add) from one to N_idle (line 12). Then, it iterates through every running job to which N_add GPUs can be added (line 14). For a running job, it predicts its remaining running time when N_add GPUs are added. Finally, the algorithm predicts makespan $MS\_predict$ based on the predicted remaining running time. If $MS\_predict$ is

smaller than the global minimal makespan, both MS_min and T are updated (lines 18 to 19).

Predicting the remaining time of a running job in running job scheduling is similar to that in incoming job scheduling. The only difference is that when the GPUs to be added are not in the same host machine as the containers of the job, the overhead of the terminating and restarting a job in Kubernetes is considered in the time prediction.

Note that the algorithm considers that each "reshaping up" involves one running job. Based on this consideration, the algorithm derives the optimal "reshaping up". The time complexity is O(*m***c*), in which *m* is the number of running jobs and *c* is the number of idle GPUs.

IV. Implementation

We implement the scheduling framework in Python as plugins in Kubernetes. We use MongoDB [25] as the In-memory Database. When a new job is submitted by a user, a new record is added to the job information table in the database. In the new record, the job status is set to 'submitted'.

In the Job Launcher, there are two steps to submit a job to Kubernetes. 1) The job is encapsulated as one or more Docker containers. 2) A job manifest is generated to launch the Docker containers in the Kubernetes cluster. Ansible Playbook [26] is used to ensure that the two steps are performed sequentially.

A Python Tornado server [27] is installed in both the GPU Coordinator and SideCar so that they can communicate with each other. SideCar controls (e.g., stops, restarts or checkpoints) the co-resident DL training process by sending Unix signals. SideCar also modifies the environment variables NVIDIA_VISIBLE_DEVICES and CUDA_VISIBLE_DEVICES to control the GPUs that are visible to the process.

TABLE VI
Seven types of jobs in Job Set One.

Job type	ResNet model	Batch size	Training dataset
0	ResNet-18	64	ImageNet
1	ResNet-18	128	ImageNet
2	ResNet-18	256	ImageNet
3	ResNet-34	64	ImageNet
4	ResNet-34	128	ImageNet
5	ResNet-50	64	ImageNet
6	ResNet-8	1024	Cifar10

TABLE VII
Seven types of jobs in Job Set Two.

Job type	DL model	Batch size	Training dataset
0	ResNet-18	256	ImageNet
1	ResNet-34	128	ImageNet
2	InceptionV3	128	ImageNet
3	VGG16	64	ImageNet
4	GoogLeNet	512	ImageNet
5	AlexNet	256	ImageNet
6	LeNet	1024	MNIST

V. Evaluation Setup

A. Testbed and Workloads

We built two clusters to evaluate the performance of our proposed scheduling framework. **Physical Cluster** is a 16-GPU cluster that consists of two GPU servers. One server has eight NVIDIA V100 GPUs and the other has eight NVIDIA P100 GPUs. **AWS Cluster** is a 16-GPU cluster that consists of two AWS GPU instances. One instance is p3.16xlarge with eight V100 GPUs and the other is p2.8xlarge with eight NVIDIA K80 GPUs. The servers and instances run CentOS 7 and employ Kubernetes 1.14.

Two job sets were built for the evaluation. **Job Set One** only includes the representative ResNet model [15] and **Job Set Two** consists of various popular DL models such as ResNet, Inception [28], LeNet [29], VGG [30], GoogLeNet [31], and AlexNet [32]. In each job set, jobs are classified into seven types based on DL model type, batch size, and training dataset (i.e., ImageNet [24], Cifar10 [33], or MNIST [34]). Each job is encapsulated as one container and employs a synchronous parallel model based on the MirroredStrategy in TensorFlow. The details are shown in Tables VI and VII.

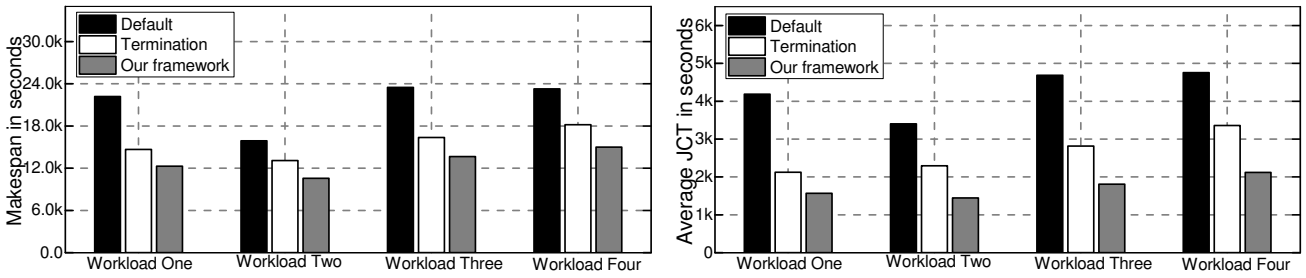
In **Physical Cluster**, we ran two workloads based on **Job Set One**. They contain 40 jobs (Workload One) and 20 jobs (Workload Two), respectively. In **AWS Cluster**, we ran two workloads (Workload Three and Workload Four) based on **Job Set Two**. The two workloads both contain 40 jobs. The only difference between them is that the jobs in Workload Four do not specify the number of epochs, so that these jobs finish when the validation accuracy reaches a desired level. In each workload, job arrival times are calculated using an exponential distribution (as that in Optimus) between [0, 11000] seconds. Upon an arrival event, we randomly choose a job from the corresponding job set.

In Workload One, Two, and Three, the number of training epochs is set to 1, 10, and 10 for jobs that use ImageNet, Cifar-10, and MNIST, respectively. For each job in Workload Four, the desired validation accuracy is set to the Top-5 accuracy reached by the same job in Workload Three.

B. Metrics and Compared Approaches

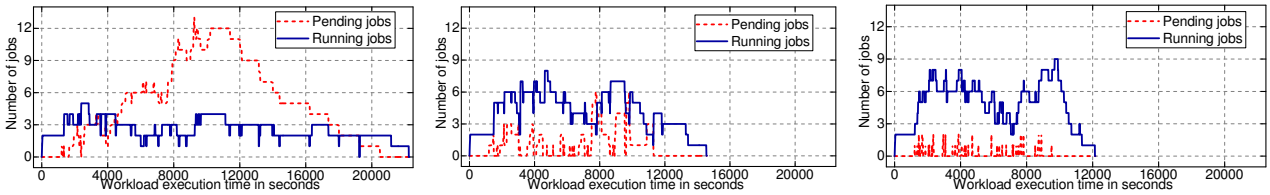
The performance metrics include the average job completion time (JCT) and makespan (as that in Optimus [14]). We evaluate the performance of three scheduling approaches: our proposed scheduling framework (our framework), a termination-based scheduling approach (termination), and the default scheduler in Kubernetes (default). In the default scheduler, the number of GPUs allocated to each job is randomly chosen from the optional numbers of GPUs.

Note that the second approach adopts the scheduling algorithm proposed in Optimus. That is, it only supports "reshaping up". More specifically, when there are idle GPUs in the cluster, the approach first allocates one GPU (the minimal number of required GPUs) to an incoming job. It then adds idle GPUs to the job one by one based on job termination. When there are no idle GPUs in the cluster, the incoming job is queued.



(a) Makespan. (b) Average job completion time.

Fig. 6. The makespan and average job completion time due to the three approaches.



(a) Default. (b) Termination. (c) Our framework.

Fig. 7. The numbers of pending and running jobs in Workload One due to the three approaches.

VI. Evaluation

A. Makespan

Figure 6(a) shows the makespan of the four workloads based on the three approaches. Compared to the default scheduler, our framework reduces the makespan by 45%, 34%, 42%, and 36% for the four workloads, respectively.

The reasons are twofold. First, when the cluster GPUs are not fully used by the running jobs and there are no new jobs submitted yet, the default scheduler cannot utilize the idle GPUs. In contrast, our framework can efficiently conduct "reshaping up" due to its adaptive scheduling and elastic GPU allocation so as to utilize idle GPUs and reduce the makespan.

Second, in the default scheduler, when all cluster GPUs are allocated to running jobs, a new job is pending in a queue until one or more running jobs finish training and release their GPUs. If each job runs with its minimal number of required GPUs, the pending job does not affect the makespan because there is no space for "reshaping down". Otherwise, "reshaping down" can reduce the makespan because of the non-linear performance gain. Note that the performance improvement in Workload Four is lower than that in Workload Three due to the error in the prediction of the number of training epochs.

Compared to the termination approach, our framework reduces the makespan by 17%, 20%, 17%, and 18% for the four workloads, respectively. The reasons are twofold. First, the termination approach incurs significant overhead in "reshaping up". Second, the termination approach does not support "reshaping down".

Figure 7 plots the number of pending and running jobs in Workload One. Figure 8 further shows the number of pending jobs when each job is submitted in Workload One. Our framework can offer GPUs to a new job by removing a subset of GPUs from a running job. Thus, it achieves the lowest number of pending jobs among the three approaches. When the last job is submitted, the number of pending jobs due to our framework is zero, which is much smaller than the number (12 and 3) due to default and termination approaches.

In the default and termination approaches, a new job remains pending while all cluster GPUs are allocated to running jobs. However, the number of pending jobs due to the termination approach is much lower than that due to the default scheduler. The reason is that the number of times this pending scenario occurs due to the termination approach is much lower than that due to the default scheduler. In particular, the termination approach only allocates one GPU to an incoming job during the submission and adds idle GPUs to the job later. Thus, after the initial allocation, the number of idle GPUs due to the termination approach is higher than that due to the default scheduler.

In the default scheduler, jobs that require a large number of GPUs could be negatively affected by out-of-order scheduling. For example, consider a job that requires 8 GPUs. While this job is waiting for such configuration, if a new job that requests 2 GPUs is submitted, the new job is scheduled on a machine before the 8-GPU job when two GPUs become available. The 8-GPU job can only be scheduled when eight GPUs become available and no job requests a smaller number of GPUs.

B. Average Job Completion Time

Figure 6(b) shows the average job completion time of the four workloads. Compared to the default scheduler, our framework reduces the average job completion time by 63%, 58%, 61%, and 55% for the four workloads, respectively. Compared to the termination approach, our framework reduces the average job completion time by 27%, 37%, 35%, and 36% for the four workloads, respectively.

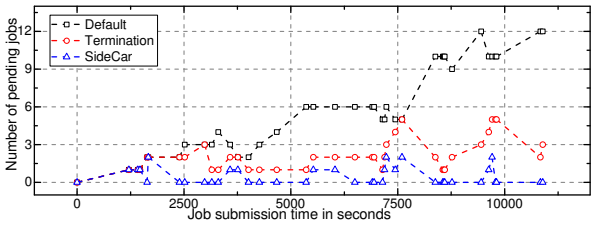


Fig. 8. The number of pending jobs when each job is submitted.

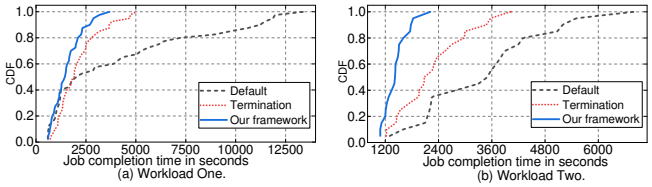


Fig. 9. CDF of the job completion time due to the three approaches.

Figure 9 also plots the job completion time distribution of Workload One and Workload Two. The results show that the job completion time of the default scheduler has a severe heavy tail distribution since the jobs suffer from long pending times. Compared to the default scheduler, the heavy tail phenomena due to the termination approach is less severe because the jobs incur a shorter average job pending time.

The job completion time using our scheduling framework is more evenly distributed than that due to the termination approach. The reasons are twofold. First, our framework achieves a shorter average job pending time. Second, the termination approach gradually searches for the optimal number of GPUs that minimizes the makespan by adding GPUs to a running job one by one. In contrast, the Adaptive GPU Scheduler in our framework can find the optimal number directly, based on its scheduling algorithm. The search process in the termination approach prolongs the job completion time.

C. GPU Usage

Figure 10 depicts the number of allocated GPUs in Workload One based on the three approaches. At the beginning of the workload, two jobs are submitted and there are no incoming jobs until the *t2* second. In this period (from the beginning to the t2 second), eight GPUs are idle due to the default scheduler since it cannot allocate idle GPUs to the running jobs. The idle GPUs can only be allocated when there are new jobs submitted at the *t2* second. The termination approach gradually adds GPUs to running jobs so that all GPUs are allocated at the *t1* second. In our framework, all GPUs are allocated to running jobs at the beginning, according to the scheduling decision from the Adaptive GPU Scheduler.

In the default scheduler, when the number of GPUs in use drops to 12 around the $20,000^{th}$ second, there is a pending job in the cluster. The number of GPUs required by that job is eight. However, the default scheduler cannot allocate the four idle GPUs to the job due to its inelasticity.

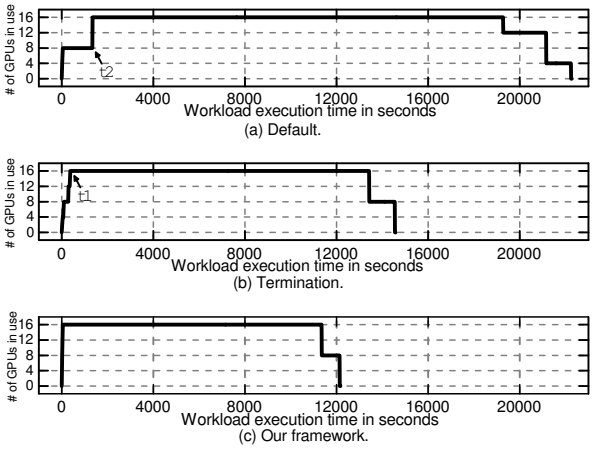


Fig. 10. GPU usage in Workload One due to the three approaches.

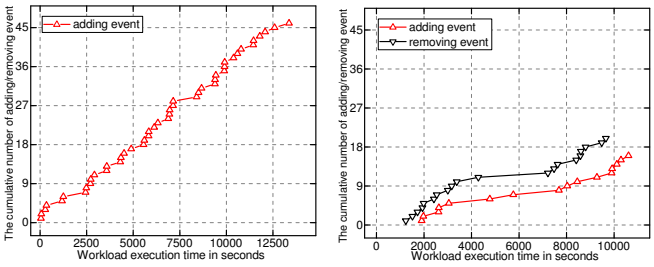


Fig. 11. The adding and removing events in Workload One.

D. Overhead Analysis

Table VIII presents the overhead of the four workloads based on our framework and the termination approach. The overhead is measured as the ratio of the total time spent on removing and adding GPUs to the total runtime of all jobs. The overhead due to our framework is small compared to its performance improvement. The termination approach incurs much higher overhead compared to our framework.

Taking a closer look at the overhead analysis, Figure 11 plots the numbers of removing and adding events in Workload One based on the two approaches. The results show that the number of adding events (46) due to the termination approach is much higher than that (16) of our framework because the termination approach only allocates one GPU to a job during the job submission, and gradually adds GPUs to the job. There is no removing event in the termination approach because it does not support "reshaping down".

TABLE VIII
THE OVERHEAD OF OUR FRAMEWORK AND TERMINATION APPROACH.

	Workload One	Workload Two	Workload Three	Workload Four	Average
Termination	29.3%	21.9%	27.1%	28.6%	26.7%
Our framework	8.9%	6.7%	7.6%	8.3%	7.9%

E. Discussions

Compatibility. Our scheduling framework considers that jobs are running in Docker containers. Thus, it can be extended as plugins in other orchestration platforms, such as Apache Yarn [35]. It can also be applied to other types of jobs (e.g., batch jobs) that run in Docker containers.

Straggler. Our framework considers that there are no stragglers [36], [37] in job training because a given GPU is not shared by multiple jobs, and thus there is no interference between jobs that are running simultaneously.

Synchronous parallel model. TensorFlow employs two strategies (MirroredStrategy and MultiWorkerMirroredStrategy) to support AllReduce-based synchronous training. Our framework is compatible with both strategies and it launches appropriate number of SideCar processes for each of them.

Communication pattern. Job completion time prediction in our framework is based on per-iteration execution time and is agnostic to the communication pattern in training. Thus, our framework works for both AllReduce and Parameter Server architectures.

SideCar limitation. In performing running job scheduling, SideCar cannot reallocate GPUs for a given job when the GPUs to be added are not in the same host machine as the containers of the job. In this scenario, our framework must terminate and restart the job to add GPUs. Note that the overhead of the terminating and restarting a job is considered in the scheduling algorithm.

Model parallel. In DL training frameworks, there are two strategies for distributed training: data parallel and model parallel. Since SideCar is non-intrusive to DL training frameworks and is agnostic to the training strategy used in the DL training process, our proposed framework can also be applied to DL jobs that employ model parallel distributed training.

VII. Related Work

Improving the efficiency of distributed DL training has become the focus of recent works. The techniques in our scheduling framework are related to the following research.

Distributed DL training. Based on a Parameter Server (PS) architecture [38], [39], a number of distributed DL frameworks (e.g., TensorFlow [10] and MXNet [40]) have been developed. BigDL [41] is proposed as a distributed DL library for Spark [17]. With BigDL, users can write their DL applications as standard Spark programs. Poseidon [13] uses wait-free backpropagation that overlaps the backward propagation computation with the gradient communication. iBatch [42] is a novel communication approach that batches parameter communication and forward computation in PS to overlap them with each other. P3 [43], TicTac [44], and ByteScheduler [6] change the transmission order of different DNN layers in order to reduce the communication overhead in a PS architecture. PHub [45] is a multi-tenant and rack-scale PS design for cloud-based distributed deep neural network training. Parallax [46] integrates PS with an AllReduce architecture to optimize the amount of data transmission. LAG [47] proposes a lazily aggregated gradient that adaptively skips the gradient calculations to reduce communication and computation in PS. PyTorch [11] supports distributed DL training using AllReduce primitives. Horovod [48] is designed for scalable distributed deep learning using TensorFlow. It implements an AllReduce operation using a ring-based algorithm and MPI for communication.

The above approaches optimize the training of a single distributed DL job, whereas our scheduling framework optimizes the GPU scheduling of multiple distributed DL jobs.

Resource scheduling in multi-job DL clusters. Tiresias [5] is a GPU cluster resource manager that minimizes the job completion time based on the characteristic study of the Microsoft cluster, Philly [4]. SLAQ [22] uses a dedicated cluster to run distributed ML jobs in the early feedback phase. The goal is to maximize the average model accuracy in that phase. It dynamically allocates CPU resources at runtime based on the job resource demand and intermediate model accuracy. Gandiva [8] accelerates DL model training in feedback-driven exploration by dynamically changing GPU usage modes of distributed deep learning jobs at runtime. It can reduce early feedback latency and improve the scheduling efficiency in GPU clusters. However, this approach is intrusive to DL training frameworks. Optimus [14] minimizes makespan and average job completion time of multiple jobs running in a PS architecture. It designs a novel resource-performance model and proposes a scheduling scheme to dynamically adjust the number of allocated workers and servers. However, the adjustment does not support "reshaping down" and is implemented based on job termination, which results in significant overhead.

Our proposed scheduling framework is not intrusive to DL training frameworks and supports both "reshaping down" and "reshaping up". It also includes an elastic GPU allocation mechanism to further reduce the overhead in reshaping. Thus, it is efficient and non-intrusive in GPU utilization.

VIII. Conclusion

This paper presents an efficient, non-intrusive GPU scheduling framework that employs a combination of an Adaptive GPU Scheduler and an elastic GPU allocation mechanism to reduce makespan and improve resource utilization. The Adaptive GPU Scheduler uses training job progress information to determine the most efficient GPU allocation and reshaping. The elastic GPU allocation mechanism further reduces the reshaping overhead by using a SideCar process that is able to control the DL training processes of a job. We implemented the scheduling framework as plugins in Kubernetes and conducted evaluations on two 16-GPU clusters with multiple training workloads based on TensorFlow. The evaluation results show that our proposed scheduling framework reduces the overall execution time and the average job completion time by up to 45% and 63%, respectively, compared to the default scheduler.

IX. Acknowledgements

This research was initiated as a summer internship of Shaoqi Wang at Nokia Bell Labs in 2019, and then supported in part by U.S. NSF grant SHF-1816850.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook *et al.*, "Cosmoflow: Using deep learning to learn the universe at scale," in *Proc. of SC*, 2018.

[3] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on simd architectures," in *Proc. of SC*, 2018.

[4] 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 *Proc. of USENIX ATC*, 2019.

[5] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *Proc. of USENIX NSDI*, 2019.

[6] 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 *Proc. of ACM SOSP*, 2019.

[7] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," in *Proc. of ACM ASPLOS*, 2017.

[8] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. of USENIX OSDI*, 2018.

[9] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. of EuroSys*, 2015.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Proc. of USENIX OSDI*, 2016.

[11] V. Subramanian, *Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch*. Packt Publishing Ltd, 2018.

[12] K. Hightower, B. Burns, and J. Beda, *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[13] 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 *Proc. of USENIX ATC*, 2017.

[14] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proc. of EuroSys*, 2018.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of IEEE CVPR*, 2016.

[16] "Nvidia device plugin for kubernetes," https://github.com/NVIDIA/k8s-device-plugin.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of USENIX NSDI*, 2012.

[18] 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," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[19] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Proc. of NIPS*, 2012.

[20] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ml jobs on clusters," in *Proc. of ACM/IFIP Middleware*, 2018.

[21] S. Ravuri and O. Vinyals, "Classification accuracy score for conditional generative models," in *Proc. of NIPS*, 2019.

[22] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *Proc. of ACM SoCC*, 2017.

[23] 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 *Proc. of EuroSys*, 2016.

[24] "Imagenet," http://www.image-net.org/.

[25] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[26] L. Hochstein and R. Moser, *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. " O'Reilly Media, Inc.", 2017.

[27] M. Dory, A. Parrish, and B. Berg, *Introduction to Tornado: Modern Web Applications with Python*. " O'Reilly Media, Inc.", 2012.

[28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. of IEEE CVPR*, 2016.

[29] 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.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of IEEE CVPR*, 2015.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. of NIPS*, 2012.

[33] "Cifar dataset," cs.toronto.edu/~kriz/cifar.html.

[34] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[35] "Apache yarn," https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[36] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml," in *Proc. of ACM SoCC*, 2016.

[37] S. Wang, W. Chen, X. Zhou, and M. Ji, "Addressing skewness in iterative ml jobs with parameter partition," in *Proc. of IEEE INFOCOM*, 2019.

[38] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *Proc. of USENIX OSDI*, 2014.

[39] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameter server consistency models." in *Proc. of AAAI*, 2015.

[40] 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," *arXiv preprint arXiv:1512.01274*, 2015.

[41] Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, Y. Wan, Z. Li, J. Wang, S. Huang *et al.*, "Bigdl: A distributed deep learning framework for big data," *arXiv preprint arXiv:1804.05839*, 2018.

[42] S. Wang, A. Pi, and X. Zhou, "Scalable distributed dl training: Batching communication and computation," in *Proc. of AAAI*, 2019.

[43] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *Proc. of SysML*, 2019.

[44] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," in *Proc. of SysML*, 2019.

[45] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: a rack-scale parameter server for distributed deep neural network training," in *Proc. of ACM SoCC*, 2018.

[46] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun, "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proc. of EuroSys*, 2019.

[47] T. Chen, G. Giannakis, T. Sun, and W. Yin, "Lag: Lazily aggregated gradient for communication-efficient distributed learning," in *Proc. of NeurIPS*, 2018.

[48] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

1. Testbed: we built a 16-GPU cluster that consists of two GPU servers. One server has eight NVIDIA Tesla V100 GPUs and the other one has eight NVIDIA Tesla P100 GPUs. The two servers in the cluster run CentOS 7 and share a file system. The Kubernetes version is 1.14 and the Docker version is 18.09.

2. Open-source software used in our framework: we use MongoDB as In-memory Database. Job Launcher submits a job to Kubernetes based on Ansible Playbook. GPU Coordinator and each SideCar process include a Python Tornado server so that they can communicate with each other.

3. Workloads: we built two workloads based on representative ResNet model and training datasets ImageNet and Cifar-10. The two workloads contain 40 jobs and 20 jobs using TensorFlow 1.12, respectively. Job arrival times are calculated using the exponential distribution. To submit a job, a user inserts a new record in the job information table in the In-memory Database. The record specifies the location of the dataset and training code in the shared file system.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: NVIDIA Tesla V100 and P100 GPUs

Operating systems and versions: CentOS 7

Compilers and versions: Python 3.6.8

Applications and versions: Kubernetes 1.14, TensorFlow 1.12,

Libraries and versions: Docker 18.09

Key algorithms: traversal

Input datasets and versions: ImageNet, Cifar-10