Online Evolutionary Batch Size Orchestration for Scheduling Deep Learning Workloads in GPU Clusters

Zhengda Bian[1], Shenggui Li[1], Wei Wang[2], Yang You[1]
National University of Singapore[1], ByteDance[2]
Singapore

ABSTRACT

Efficient GPU resource scheduling is essential to maximize resource utilization and save training costs for the increasing amount of deep learning workloads in shared GPU clusters. Existing GPU schedulers largely rely on static policies to leverage the performance characteristics of deep learning jobs. However, they can hardly reach optimal efficiency due to the lack of elasticity. To address the problem, we propose ONES, an *ONline Evolutionary Scheduler* for elastic batch size orchestration. ONES automatically manages the elasticity of each job based on the training batch size, so as to maximize GPU utilization and improve scheduling efficiency. It determines the batch size for each job through an online evolutionary search that can continuously optimize the scheduling decisions. We evaluate the effectiveness of ONES with 64 GPUs on TACC's Longhorn supercomputers. The results show that ONES can outperform the prior deep learning schedulers with a significantly shorter average job completion time.

KEYWORDS

Deep learning, resource scheduling, evolutionary search, distributed training

ACM Reference Format:
Zhengda Bian[1], Shenggui Li[1], Wei Wang[2], Yang You[1]. 2021. Online Evolutionary Batch Size Orchestration for Scheduling Deep Learning Workloads in GPU Clusters. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3480859

1 INTRODUCTION

Deep Learning (DL) has gained significant success in the last decade. Many efficient DL methods have been proposed, continuously renewing state-of-the-art results of machine learning (ML) tasks, such as computer vision [13, 31, 33], natural language processing [7, 27], and speech recognition [2], by taking advantage of the rising amount of data and growing scale of deep neural models. However, training large-scale neural models can be extremely compute-intensive and time-consuming. Therefore, people from both academia and industry are interested in accelerating DL jobs with distributed training on powerful GPU devices. Recently, data parallelism techniques [9, 36–38] have become dominant approaches to distributed training on huge datasets with large-scale GPU resources. Data parallel training can fully utilize available GPUs by dividing a large minibatch to multiple devices, each of which executes an identical replica of the model and synchronizes with each other after each training step. To meet the demand for GPU resources, an increasing number of GPU clusters have been built, where users can share the expensive GPU resources. In the meantime, resource management and job scheduling have become new challenges of training DL jobs in shared GPU clusters.

Scheduling distributed DL jobs in a GPU cluster can be intractable because of the specific performance of DL jobs w.r.t. resource allocation. Due to the communication overhead required by model synchronization and the affect of large minibatches on training convergence [14, 16], increasing the number of devices is not able to linearly accelerate a DL job. Moreover, DL jobs may have severe performance interference between each other, especially when sharing the same GPU device, on which they contend for the GPU cores and memory [15]. However, existing DL schedulers [3, 12, 25, 35, 39] simply focus on the number of GPUs for the jobs, while they pay no attention to whether their batch sizes are appropriate. Besides, these schedulers rely on greedy strategies to make decisions within a limited range of solutions. As a result, the training efficiency of DL jobs are limited.

Another issue arises from scheduling online workloads, which involve online job arrivals and completion. DL jobs are iterative, so that it is hard to know the completion time in advance until they converge. However, we can only make schedule decisions based on the information that already exists, while workloads in the cluster keep changing in real time. As a result, we usually make stale and inefficient decisions for DL jobs. A common solution of existing DL schedulers [3, 12, 25, 39] is to periodically update job status and reschedule them at a certain time interval. Such schedulers are not able to make timely response to online workloads, as some jobs may be waiting between each rescheduling round, even when there already exist available resources.

To address the above limitations in training and scheduling efficiency, we propose ONES, a new scheduler for DL workloads, which is based on a key idea that an elastic batch size orchestration for each DL job is essential to improve scheduling performance. The batch size of each job is *elastic* because we allow the scheduler to dynamically adjust it and find the optimal value to maximize training efficiency. The first challenge to implement this idea is to determine the batch size for each job. We design an evolutionary search algorithm instead of greedy strategies to dynamically explore optimal decisions in an online fashion. The algorithm maintains a group of solutions (i.e. population) that continuously evolve

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3480859

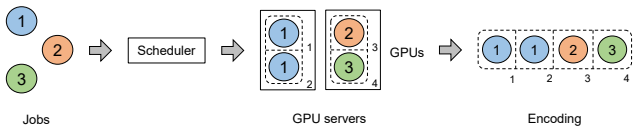


Figure 1: An example of scheduling GPUs for DL jobs.

towards higher performance. Hence compared with existing scheduling strategies, evolutionary search has the following advantages to obtain more efficient decisions. First, the algorithm works with a population that helps the scheduler have more chance to make high-quality decisions. Second, the algorithm is efficient to deal with online workloads because it continuously updates its solutions according to real-time job status. Especially, ONES builds an online prediction model to obtain possible distributions of job lengths throughout training, from which the algorithm can sample solutions with shorter estimated average job completion time. Besides, as shown in Figure 1, scheduling GPUs can be easily encoded into genomes (i.e. the basic form of solutions in evolutionary search) so that the algorithm can be directly applied to the scheduling problem. The second challenge is to execute scheduling decisions in an efficient way. We design an elastic batch size scaling mechanism, which minimizes the cost to execute changes in the batch size and number of GPUs of DL jobs that are induced by new scheduling decisions. The mechanism uses a centralized scheduler to make scheduling decisions and manage batch sizes, and sends messages to specific nodes to execute scaling at only necessary workers. Unlike common scaling approaches based on model checkpointing, ONES can automatically re-configure the batch size as well as number of GPUs through NCCL communication [23] at the end of a training step without stopping training. Thus, its cost is almost invisible to the original job. Besides, for the purpose of efficient utilization of limited GPU resources, ONES follows a set of scaling policies to guarantee the training performance of each job as well as the scheduling efficiency.

In summary, we make the following contributions.

- We propose ONES, an *online evolutionary scheduler* that is the first scheduler to schedule batch sizes for DL jobs to maximize resource utilization.
- We design an online scheduling algorithm based on evolutionary search. Different to greedy strategies, our algorithm dynamically selects a superior population of solutions according to jobs' online performance prediction derived from their real-time status.
- We design elastic batch size scaling to execute adjustment of resource allocation with almost invisible cost, which does not need to stop the original training process.
- ONES is efficient in minimizing job completion time (JCT). The evaluations on a testbed of TACC's Longhorn GPU nodes [4] and trace-driven workloads show significant improvements over state-of-the-art DL schedulers.

2 BACKGROUND AND MOTIVATION

2.1 Existing schedulers for deep learning

Existing DL schedulers have proposed a variety of strategies that optimize specific performance of DL jobs based on greedy heuristics. For example, SLAQ [39], Optimus [25] and OASiS [3] aim to find the optimal job prioritization via performance analysis. Tiresias [12] maintains a Multi-Level Feedback Queue based on the Least Attained Service policy to reduce queuing time. Some other schedulers [15, 35] focus on optimizing job locality and placement to minimize the affect of communication overhead. However, the efficiency of existing scheduling algorithms still confronts the following limitations.

Lack of elasticity: Elastic job size can help the scheduler improve the cluster efficiency, as it gives the opportunity to increase resource utilization and reduce fragmentation. Unfortunately, most existing schedulers are not aware of the elasticity and execute DL jobs with fixed amount of resources that are requested by users. However, a fixed resource configuration can be inefficient in many aspects. On one hand, allocating resources to DL training jobs has to follow the gang scheduling policy [35], which leads to longer queuing delay. In gang scheduling [24], all the workers of a job need to become active together, so that as long as their resource requirements cannot be fulfilled simultaneously, the allocation will fail. Therefore, with aggressive configurations, many jobs need to wait for the available resources in the cluster until their requirements can be fulfilled, even though they are able to run at a smaller scale. On the other hand, using conservative configurations results in low resource utilization and intolerable training time. For example, it is reported [9] that training ResNet-50 [13] on the ImageNet dataset [18] takes 29 hours to reach the state-of-the-art accuracy, with a batch size of 256 images on 8 Tesla P100 GPUs. However, with a larger batch size of 8192, using 256 GPUs, the training time can be reduced to less than one hour. Unfortunately, users often submit inappropriate configurations when they are not familiar with the low-level system, which leads to an obvious resource underutilization.

There are DL schedulers trying to improve the elasticity by dynamically adjusting the the number of GPUs for each job [19, 25, 35, 39]. However, the degree of elasticity in these schedulers is not efficient enough. They only focus on scheduling the number of GPUs and treat DL jobs as black-boxes, but are not aware to take a step forward to schedule the size of minibatches at the same time. Consequently, inappropriate combinations of the number of GPU and batch size reduce the efficiency of distributed training, as increasing the number of workers leads to increase in the communication overhead, so that the utilization of each single GPU will be decreased.

Suboptimal scheduling quality: Existing resource schedulers focus on optimizing resource utilization through static greedy algorithms, such as Shortest Job First (SJF) policy [10], Least Attained Service (LAS) policy [12], or time-sharing-based slicing strategy [35]. Unfortunately, such scheduling strategies still confront efficiency issues. First, resource scheduling is an NP-hard optimization problem, so that the greedy solutions may often lead to local minima that is far from the optimal performance. Second, existing schedulers consider DL jobs as black-boxes, not aware of the significant

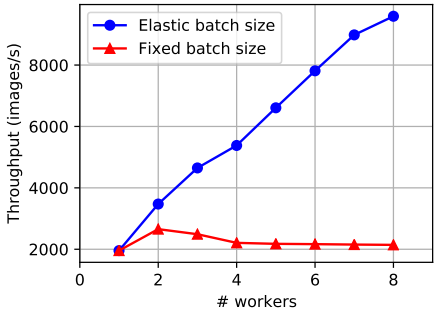


Figure 2: Training speed of training ResNet50 on the CIFAR10 dataset.

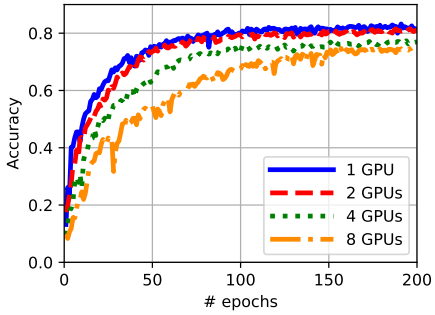


Figure 3: Training accuracy with fixed local batch size of 256 and different number of GPUs.

role of batch sizes in improving training efficiency. Consequently, regardless of how the schedulers allocate available resources, the scheduling performance is largely limited.

Besides, some DL schedulers [25, 39] minimize average completion time by predicting the remaining length of each job. However, predicting accurate job length can be difficult in practice and not scalable for all kinds of DL workloads. On one hand, many DL jobs do not have smooth loss curves or validation accuracy curves (e.g. a reward function in RL and a loss with cyclical learning rate [29]). On the other hand , not all DL jobs can end normally, as some jobs are manually killed, some are early-stopped, some crashed due to errors, etc.

Inefficient online scheduling: To make online scheduling decisions, In order to make online scheduling decisions, most of existing schedulers (e.g. SLAQ [39], Optimus [25], Tiresias [12], and OASiS [3]) periodically update job status and reschedule them based on the analysis of current job performance. They basically use fixed scheduling intervals, so that they may not be able to make immediate response to changes in the workloads. For example, Optimus's interval between 2 rounds of rescheduling is 10 minutes, and that of OASiS is 1 hour. At each round, all the jobs will be paused to wait for the next scheduling decisions. As a result, there will also be a nontrivial waste of resources because the GPUs will become idle during rescheduling.

Besides, there is a new trend to use deep reinforcement learning (DRL) techniques to learn dynamic scheduling policies [8, 21, 22]. The DRL schedulers can automatically train the models based on offline traces, and dynamically fine-tune them in the online decision-making phase. However, these DRL schedulers are not able to allow job preemption so far due to the limited action space size. Therefore, it is questionable whether they are efficient to schedule DL jobs that are usually time-consuming.

2.2 Advantages of the elasticity

The scheduler can make the best use of GPU resources to the advantage of the elastic batch sizes in different ways. Through cautious batch size orchestration, DL jobs can find appropriate batch sizes to achieve higher GPU utilization, faster training speed and shorter job completion time. For example, Figure 2 compares the speed of training ResNet50 [13] on the CIFAR10 dataset [17] between using an elastic batch size scaled from 256 to 2048 and a fixed batch size of 256. However, with fixed global batch size, the utilization of each single worker will be reduced as the number of worker increases. In the meantime, the throughput will be penalized by the increasing communication overhead. As a result, when the number of workers exceeds 2, the throughput no longer increases and starts to drop. Thus, to efficiently scale a job, the global batch size should increase as the number of workers increases, so that the throughput can keep increasing as well. An existing common approach is to run workers with fixed local batch size on each GPU. However, people use very large batch sizes with caution [14, 16]. With fixed local batch size, the global batch size will grow with the number of GPUs. As shown in Figure 3, simply increasing the number of GPUs with a fixed local batch size can significantly affect the training convergence. Using more GPUs leads to a slower convergence because the global batch size becomes larger, especially when the number of GPUs is greater than 2. In order to guarantee training performance, selecting a proper learning rate will also be necessary when scaling the global batch size [30, 36, 37].

Variable	Description
J	the set of jobs
C	the set of GPU devices
c_j	the number of GPUs allocated to j
B_j, b_j	the global & local batch size of j
T_j	remaining time of j
Y_j	remaining workload of j
X_j	throughput of j
R_j	batch size limit of j

Table 1: Summary of major variables.

Besides, using elastic batch sizes to train DL jobs can intuitively avoid fragmentation problem, where the number of idle GPUs is smaller than the required size of any pending job so that the GPUs are wasted. In contrast, we allow the scheduler to execute some job with a smaller size first. As a result, we can saturate the cluster to fully utilize the GPU resources, and in the meantime reduce waiting time of the jobs.

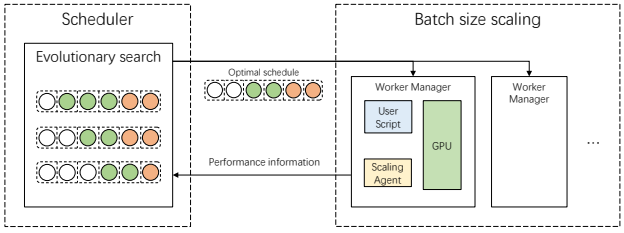


Figure 4: Overview of the scheduler architecture.

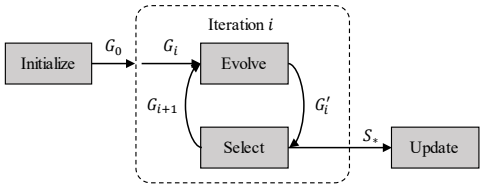


Figure 5: The online evolutionary scheduling algorithm. G_0: the initial population; G_i the current population; G_i': new solutions derived from G_i; G_{i+1}: solutions with higher scores selected from G_i'; S_*: the best solution.

3 ONLINE EVOLUTIONARY SCHEDULER

In this section, we introduce ONES for scheduling DL training jobs. The goal of ONES is to schedule the batch size of each job instead of the number of GPUs, and try to reach an appropriate balance between throughput and training performance by finding the optimal combination of batch size and number of GPUs. More specifically, as shown in Table 1, let J and C denote the set of current jobs and GPU resources in the cluster respectively, and let c_j denote the number of GPUs allocated to each job j. ONES maintains dynamic schedule decisions, which can be represented as the following mapping from jobs to resources:

$$S : J \times C \rightarrow \{b_j^i\}, \quad b_j^i \in \mathbb{N}, \ \forall i \in C, \ \forall j \in J. \tag{1}$$

where b_j^i denotes the batch size for the worker of job j on GPU_i. We can also derive the global batch size and number of allocated GPUs as

$$B_j = \sum_{i \in C} b_j^i; \quad c_j = \sum_{i \in C} \min(1, b_j^i). \tag{2}$$

If a job has $B_j = c_j = 0$, it cannot run and needs to wait for the next schedule. Otherwise, the job will be running.

There are two core problems to be addressed: (1) how to determine the optimal batch size for each job, and (2) how to scale the batch size of a job with minimum cost. We present our solutions in the rest of this section.

3.1 Architecture

ONES consists of two core functionalities as illustrated in Figure 4. First, there is a central scheduler that dynamically collects the real-time status of the cluster and jobs, and determines the batch sizes of all the existing jobs. The scheduler optimizes the solution based on the evolutionary search algorithm, in order to continuously reach high performance for online workloads. Then the scheduler allocates available GPUs based on the solution, and invokes corresponding workers to execute the decision on the GPU nodes.

Next, to execute the new solution, we may need to re-configure some jobs with new batch sizes. However, the common practice is to stop the training and restart them with new configurations, which can result in significant overheads. We present a new approach to execute batch size scaling that is almost invisible to the original training. More specifically, a worker manager is bound to each GPU device, which receives the new configuration from the scheduler, and invokes a scaling agent to automatically adjust the execution configurations of its worker in the background. Thus, the batch size scaling no longer needs to stop the training.

Besides, each worker uploads its training progress (e.g. number of processed samples, training loss and validation accuracy) to the central scheduler at the end of each training epoch. Each job ends when its model converges. Then the scheduler will release its workers and clear its resource allocation.

3.2 Online Evolutionary Search

To determine appropriate batch sizes for the DL jobs that continuously arrive and compete for a limited amount of GPU resources, we design an online scheduler based on evolutionary search to minimize job completion time (JCT).

We present the workflow of the algorithm as illustrated in Figure 5. The algorithm optimizes a population that consists of a group of candidate solutions with an iterative evolution process. It starts with an initial population G_0. In each iteration i, new candidates are derived from the current population G_i, and those with higher scores will be selected to form the new population G_{i+1}. Finally, the schedule will be updated by deploying the optimal candidate S_*.

The evolutionary search can be more suitable than existing scheduling strategies that make greedy decisions to schedule DL workloads in the following aspects. First, the algorithm continuously optimizes a population of solutions that reduce the chance to be trapped in local minima. Thus it can achieve higher-quality solutions than greedy strategies that optimize a single solution. Second, the population of solutions keeps evolving according to real-time job status, which helps ONES make immediately response to online workloads, especially when GPU resources become available after job completion, or very short jobs arrive and provisionally taking over some GPUs from long jobs can be beneficial. Besides, compared with other approximate search algorithm such as Simulated Annealing (AS), Tabu Search (TS), Nearest Neighbor Search (NNS) and Ant Colony Optimization (ACO), the evolutionary search is the most intuitively suitable for the scheduling problem as the allocation of GPUs can be directly encode into genomes (as illustrated in Figure 1). In contrast, it is not very intuitive to define the "neighborhood" for NNS or the "path" for ACO. In the meantime, the evolutionary search has relatively fast iterative speed.

3.2.1 *What Candidate Wins?*

The goal of ONES scheduling algorithm is to minimize the average JCT. Preferentially, serving the job with the *shortest remaining processing time* (SRPT) is the solution to this problem. [5, 10, 11].

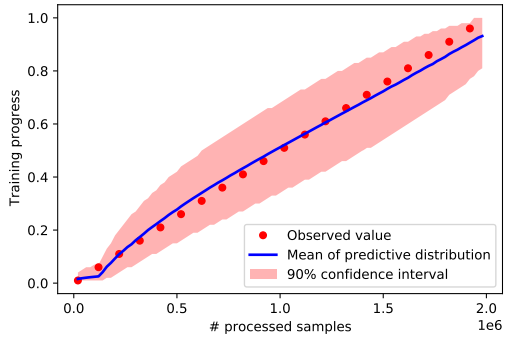


Figure 6: An example of online prediction progress with uncertainty.

Sadly, this approach may lead to a potential consequence that the scheduler allocates as many GPUs as possible to each job. This is apparently not sensible because the training speed of distributed DL jobs does not scale linearly with the number of GPUs due to a considerable communication overhead in parameter synchronization. Therefore, we extend SRPT and define *smallest remaining utilization first* (SRUF) as our goal. Let J_r denote the running jobs, and T_j denote the remaining processing time of job $j \in J_r$. Note that T_j is supposed to be a function of the batch size B_j. We can represent the utilization of job j w.r.t both time and space as $T_j(B_j) \times c_j$. Then, we solve the optimization problem to determine the batch size for each job by minimizing the overall utilization as follows:

$$\text{minimize} \sum_{j \in J_r} T_j(B_j) \times c_j, \tag{3}$$

$$\text{subject to} \sum_{j \in J_r} \min(1, b_j^i) = 1, \forall i \in C. \tag{4}$$

Note that we do not allow multiple jobs to share a single GPU device due to the severe interference caused by GPU sharing [15], i.e. each worker of the jobs will be exclusively executed on a single GPU (Equation 4).

However, it is difficult to obtain the remaining time of each job because we can hardly know how many samples to train until a DL job is completed. Let X denote the throughput (i.e. training speed) and Y denotes the remaining workload (# samples to process until convergence), we compute the remaining time as

$$T = \frac{Y}{X}, \tag{5}$$

We can easily profile real-time throughput by measuring the throughput on each GPU during training, and use the mean value of collected measures in computation. To obtain Y, we take advantage of the cluster history and try to excavate useful information. The key insight is that, despite the difficulty to know the exact remaining workload size, we may be able to model probability distribution of it based on historical jobs that have similar training performance.

To obtain the distributions for online workloads, we conduct the following online prediction process. The exact numbers of workload of DL jobs have a wide range, so that it is not appropriate to directly predict the workload. Instead, we predict the training progress (i.e. percentage of completion) that can be limited in $(0, 1)$. Then,

Algorithm 1: Probability sampling

Input : $S_1, ..., S_K; Be_1, ..., Be_J$
Output: The selected sample S_*

```
for j = 1 to J do
    sample ρ_j from Be_j
end
for i = 1 to K do
    score_i ← Σ_{j∈J_r} (Y_processed_j c_j / X_j) (1/ρ_j − 1)
end
S_* ← S_i if score_i ≤ score_k, ∀k ∈ K
```

we use Beta distributions to model the uncertainty, because Beta distributions can produce flexible shapes between 0 and 1. Moreover, the shape of a Beta distribution $Be(\alpha, \beta)$ is unimodal when $\alpha, \beta > 1$, which fits the law of large numbers. Let ρ denote the progress, we can build the following regression model to predict its distribution. For any job with input features x [1] , we assume its training progress is an independent random variable that

$$\begin{aligned} \rho &\sim Be(\alpha, \beta) \\ \text{where } \alpha &= Y_{processed}/\|D\|, \\ \beta &= \max(\mathbf{A}x + b, 1), \end{aligned} \tag{6}$$

which approximate the number of processed epochs and epochs to process respectively, defined by a pair of parameters A and b. We apply a threshold function to both α and β to guarantee $\alpha, \beta \geq 1$.

ONES continuously updates the regression model for online prediction. Each time when a job is completed, we train the model by maximizing the log marginal likelihood to fit the data collected from historical job information. Note that we maintain a limited size of training dataset where the data points are uniformly sampled from training logs of completed jobs. By doing so, we can control a reasonable training time and prevent overfitting.

Then, the regression model is used to predict the distribution of $\rho_j \sim Be_j$, for online workloads $\forall j \in J$. Figure 6 shows an example of the prediction where the line illustrates the mean values of the output distributions, while the shadowed area represents the 90% confidence intervals of the distributions. Next, we can derive the remaining workload Y_j as

$$Y_j = Y_{processed_j}(\frac{1}{\rho} - 1), \tag{7}$$

where $Y_{processed_j}$ denotes the number of samples that are already processed by job j. Applying Equation 7 to Equation 5, we can derive the overall utilization of each solution as

$$\sum_{j \in J_r} T_j \times c_j = \sum_{j \in J_r} \frac{Y_{processed_j} c_j}{X_j} (\frac{1}{\rho_j} - 1) \tag{8}$$

[1]We use $x = \{\|D\|, \mathcal{L}_{initial}, Y_{processed}, r_{\mathcal{L}}, \mathcal{A}\}$ that can be captured from the training history of completed jobs as input features of the GPR predictor, where $\|D\|, \mathcal{L}_{initial}$ represent the epoch size and initial loss (before training) of the job respectively, $Y_{processed}$ represents the number of already processed samples, $r_{\mathcal{L}}$ represents the loss improvement ratio, computed as $r_{loss} = 1 - \text{current loss/initial loss}$, and $\mathcal{A}$ represents the validation accuracy.

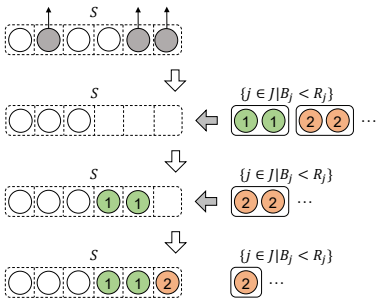


Figure 7: An example of refreshing the cluster and filling the idle GPUs. Job 1 is firstly selected by probability sampling and there is still an idle GPU left; then Job 2 is selected so that the cluster is full and the *refresh* operation end.

Finally, we can select an expected optimal solution as illustrated in algorithm 1. The sampling algorithm draws samples from the training progress distributions of each job (line 2) to compute the score of each solution (line 5). Then the solution with the smallest score is selected (line 7).

3.2.2 Search Process.

We present the details of the evolutionary search process in Figure 5 as follows.

Initialization: We generate an initial population G_0 with K simple solutions. We suggest to use the same size for the population as the cluster. Thus, the initial solutions can be easily generated by running a random job on each GPU respectively.

Evolution: The algorithm then generates new candidate solutions by randomly adjusting the batch sizes of the jobs. We design four random operations to evolve candidates from the current population G_i, including the *refresh*, *uniform crossover*, *uniform mutation* and *reorder* operations. Note that we set a limit to the batch size as R_j of each job $j \in J$ to avoid convergence issue. R_j is dynamically updated according to the real-time training performance. The details will be described in §3.3.2.

The evolution operations are executed as follows.

- *Refresh*: This operation is designed to update the schedules in G_i with the real-time job status. Thus, in this step, ONES (1) cleans up the GPU resources where the jobs have been completed (2) scales down any job $j \in J_r$ by $c_j - \lfloor \frac{R_j \cdot c_j}{B_j} \rfloor$ GPUs if $R_j < B_j$, and (3) allocates N new jobs with N available GPUs.
  When the number of available GPUs in a schedule is smaller than N after (1) and (2), the scheduler will take GPUs from the jobs with the largest $T_{processed}$ until N new jobs are fully allocated. We can thereby avoid starvation of new jobs by the preferential allocation.
  Furthermore, when there are still idle GPUs after (1) (2) and (3), we will need to fill them by resuming waiting jobs or scaling up running jobs. We compare the jobs by computing the utilization gain of each job $j \in J$ after increasing its batch size from B_j to R_j with $\lfloor \frac{R_j \cdot c_j}{B_j} \rfloor - c_j$ more GPUs. Then, as illustrated in Figure 7, we can continuously select one job by probability sampling from $\{\Delta\varphi_j Y_j\}$ using algorithm 1 until there is no idle GPU.
- *Uniform crossover*: This operation is designed to combine two parent candidates to generate two child candidates. More specifically, it scans over each GPU $i \in C$ identically and independently. On each GPU, the job from the first parent is inherited by a random child, and the job from the second parent is inherited by the other child, as shown in Figure 8. In each iteration, K pairs of parents are randomly selected from the original population, and K pairs of children will be generated.
- *Uniform mutation*: This operation is designed to exchange the GPU resources of some jobs in a candidate schedule with other jobs. More specifically, K candidates will be randomly selected from the original population, where each job in each candidate S will be randomly preempted with a mutation rate θ. The GPU resources of these preempted jobs will be filled with waiting jobs or other running jobs selected as shown in Figure 9.
- *Reorder*: The operations above may generate candidates with poor placement because workers of the same job are not guaranteed to be closely located. Therefore, we design the *reorder* operation to pack the workers of the same job in the order of the occurrence of each job, as illustrated in Figure 10.

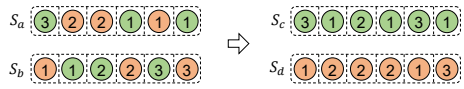


Figure 8: An example of the uniform crossover operation. The left side: parent candidates; the right side: child candidates.

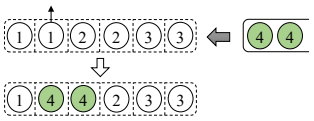


Figure 9: An example of the uniform mutation operation. Two original workers are replaced by Job 4.

Figure 10: An example of the reorder operation. The left side: before reorder; the right side: after reorder.

Selection: In the set of new candidate schedules G_i' generated by the evolution operations, the score of each candidate is derived using Eq. 8, based on the predicted distribution of training progress $Be_1, ..., Be_J$. Then the best K candidates are selected from G_i' to form the new population G_{i+1} by probability sampling that is similar to Alg. 1 but will draw K samples with the top scores.

Update: The solution with the highest score S_* will be executed in the cluster. However, too frequent update may reduce the scheduling performance to some degree due to the overhead of adjusting

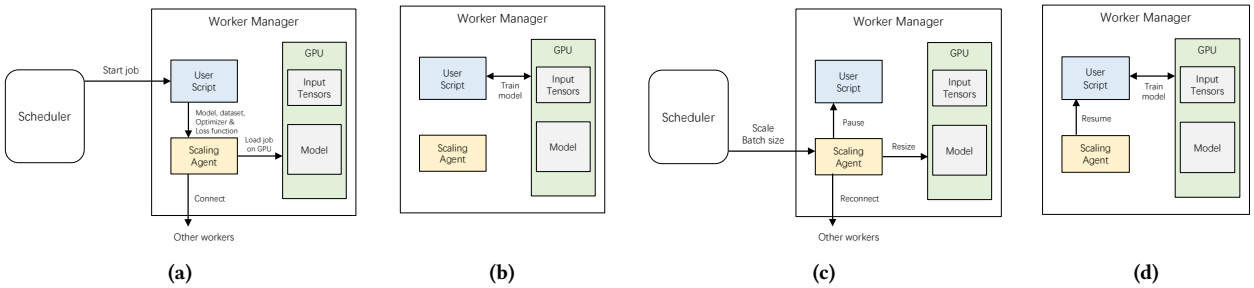


Figure 11: Scaling the batch size in 4 steps.

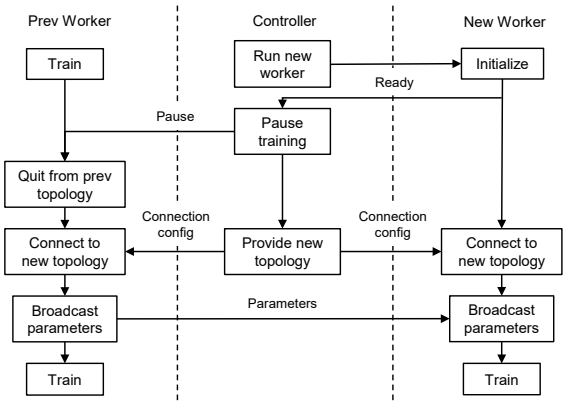


Figure 12: Workflow of adding a new worker.

resource allocation and job placement. At least, we have to ensure that each job can finish one epoch so as to completely scan its training dataset. Therefore, ONES updates the schedule with the optimal candidate S_* after all the running jobs finish at least one epoch.

3.3 Elastic Batch Size Scaling

To execute the scheduling solutions with minimum cost and without affect on the training performance, we present elastic batch size scaling that is based on the analysis of common DL training practices.

3.3.1 Batch Size Scaling Mechanism.

The batch size scaling re-configures a job with a new batch size. The re-configuration is generally not performed in a direct way, because the change in batch size involves not only itself but also the distributed model replicas, the input tensors and the input data batches. Today the dominant practice is to save and stop the training, and then restart it with the new configuration, which usually takes tens of seconds to complete. Gu *et al* [12] provide the detailed numbers of the overhead of migrating Tensorflow [1] jobs with checkpoints, which are basically around tens of seconds. Our scaling mechanism avoids stopping the training and completes the scaling in background in the following four steps, as illustrated in Figure 11.

Initially, the scheduler binds several worker managers to a job, and start the job on them. Each worker manager starts the user script to initialize the job, and invokes a scaling agent to load the modules to execute on the GPU device. The scaling agent also connects with other workers of the same job for distributed communication.

Secondly, after the initialization, the user script is allowed to start the training.

Next, when the job needs scaling, the scheduler inform its worker managers of the new configuration. Then the scaling agent pauses the user script at the end of a training step, resizes the modules in the GPU devices, and reconnects other workers. If there are new workers added to the job, we will need to share the current model parameters with them. To avoid unnecessary waiting time, we can start the new workers first and overlap their initialization with previous training. As illustrated in Figure 12, the new workers first start with initialization. When they are ready, the workers will notify all the previous workers via the controller. After the notified previous workers complete a training step, they quit from the previous topology. Then all the workers connect to the new topology together, and broadcast the current parameters together from one of the previous workers.

Finally, after the scaling, the scaling agent resumes the user script to continue the training with the new configuration.

3.3.2 Training Performance Control.

To implement elastic batch size scaling, we still need to address several underlying performance problems so as not to affect the original training convergence.

Empirically, the batch size is carefully selected to reach high accuracy, so that adjusting the batch size may affect the training performance. According to the analysis of prior studies [14, 16], with a very large batch size, a job will take more epochs to converge and even reach a worse accuracy. However, some recent works [9, 30] observe that by carefully adjusting the learning rate, DL jobs that use gradient descent optimizers can still reach equivalent accuracy after the same number of training epochs. They suggest to linearly scale the learning rate with the batch size, because for example by increasing the batch size from B to kB, the number of steps will be reduced by k, so that the number of updates will be reduced by k; in the meantime the learning rate is increased from η to $k\eta$, which will also enlarge each update by k. Therefore, the training can still achieve an equivalent progress.

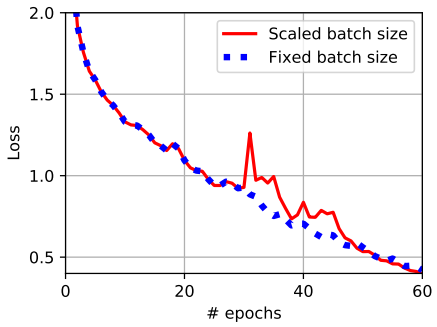


Figure 13: Scaling batch size from 256 to 4k at epoch 30 (training ResNet50 on the CIFAR10 dataset).

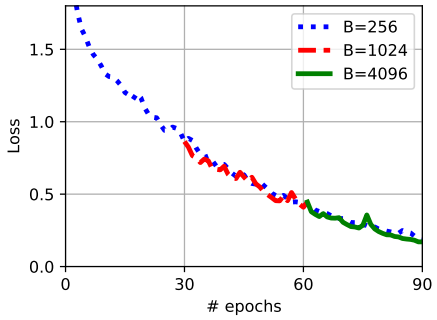


Figure 14: Training loss of scaling batch size from 256 to 4096 gradually. $B = 256$ in the first 30 epochs; $B = 1024$ in the next 30 epochs; $B = 4096$ in the last 30 epochs.

Another problem that affects the training performance occurs when there is a too large gap of batch size before and after scaling. As shown in Figure 13, such scaling results in a sudden increase in the training loss. This is caused by the noise in gradient and momentum states when the batch size explodes [20]. After the increase, the training will need many epochs to return the equivalent level of normal performance. In contrast, if the batch size grows gradually into a larger value, as shown in Figure 14, the training performance can be basically guaranteed. Therefore, we only allow the batch size to be scaled within a limited range at each time.

In general, ONES jointly manages the batch size and learning rate of each job according to their initial values based on linear scaling, so that an equivalent training performance can be presented as expected.

To control the training performance, we set a dynamic limit to the batch size of each job, denoted as R so that the scheduler will not allow the job to exceed the limit. As the training proceeds, R will be adjusted according to the following scaling policies that depend on different situations.

- *Start:* Upon arrival, a job will have to limit its batch size to be accommodated in a single GPU until completing a few warm-up steps.
- *Resume:* A job that is waiting for service can request the batch size limit R that does not exceed the amount before being preempted, but once it is rejected (i.e. the job will still be waiting in the next schedule), R will be halved. The purpose is to reduce the queuing time and prevent starvation.
- *Scale-up:* It is necessary to gradually scale up each job. Therefore, we allow a running job to request to scale up its batch size after each training epoch, by doubling the batch size limit as $R' = 2R$.
- *Scale-down:* A running job with long elapsed time will need to be scaled down to prevent the Convoy Effect. We penalize the job with

$$R' = \lceil \frac{2R}{\lceil \sigma \cdot T_{processed} + 1 \rceil} \rceil,$$

where $T_{processed}$ denotes the executed time of the job and σ denotes a fixed factor. We suggest $\sigma = \lambda$ where λ denotes the average job arrival rate, in order to penalize jobs that are longer than the average arrival time interval $1/\lambda$.

4 EVALUATION

4.1 Setups

Testbed: We evaluate ONES on TACC's Longhorn supercomputers [4]. Our testbed cluster consists of 16 GPU servers connected by Mellanox EDR Infiniband network. Each server has 2 20-core IBM Power 9 CPUs, 256GB memory and 4 NVIDIA V100 GPUs, so there are 64 GPUs in total. We store datasets, model checkpoints and training logs in a Hadoop Distributed File System (HDFS) via 1Gbps Ethernet.

Trace-driven workload: We generate our custom traces with typical DL tasks that are popularly used in the experiments of prior works [12, 25, 35], as listed in Table 2. These tasks include both computer vision (CV) and natural language processing (NLP) models and datasets, including AlexNet [6], ResNet [13], VGG [28], GoogleNet [31], Inception [32] and BERT [7]. We implement the tasks with distributed PyTorch. To generate workloads with sufficient diversity, we train the jobs with different dataset sizes instead of the original sizes (e.g. using a subset of 10k and 20k images from the original ImageNet dataset). Consequently, the training time of some long jobs is reduced, and all jobs can basically finish within 2 hours. As listed in Table 2, the number of different workloads is $4 \times 6 + 3 \times 5 + 4 + 1 + 6 = 50$. Normally ONES ends a job when there is no increase in its validation accuracy for several consecutive epochs. In our implementation and experiments, we set a target validation accuracy for each job in Table 2, and stop the training when there are already 10 consecutive epochs that exceed the target accuracy. The reason is to make sure that, in comparison between ONES and baseline schedulers, different schedulers can provide similar convergence for each job.

Baselines: We compare the performance of ONES against the following state-of-the-art DL schedulers that also aim to minimize JCT.

- Deep Reinforcement Learning (*DRL*): We adopt the basic scheduler design in [8] but modify its action space because we use the All-reduce architecture for distributed training instead of parameter servers. The scheduler trains its scheduling policy based on DRL for purpose of minimizing JCT.

Task	Dataset	Model	Dataset Size	# Classes
CV	ImageNet	AlexNet	10k, 12k, 14k, ..., 20k	10, 12, 14, ..., 20
		ResNet50		
		VGG16		
		InceptionV3		
	CIFAR10	ResNet18	20k, 25k, 30k, 35k, 40k	10
		VGG16		
		GoogleNet		
NLP	COLA	pre-trained BERT	5k, 6k, 7k, 8k	2
	MRPC		3.6k	2
	SST-2		10k, 12k, 14k, ..., 20k	2

Table 2: Workloads in our evaluation trace.

Scheduler	Greedy/Dynamic Strategy	Allow Preemption	Elastic Job Size	Elastic Batch Size
ONES	Dynamic	Y	Y	Y
DRL	Dynamic	N	Y	N
Tiresias	Greedy	Y	N	N
Optimus	Greedy	Y	Y	N

Table 3: Comparison of ONES and the state-of-the-art DL schedulers.

It can dynamically determine the size of each job. Only one job can be rescheduled at each time.

- *Tiresias* [12]: This scheduler maintains the waiting jobs in a multi-level priority queue according their attained service to reduce JCT. It only allows fixed job size but does not depend on any prediction.
- *Optimus* [25]: This scheduler can dynamically adjust the size of each job, and uses a greedy strategy to allocate resources based on the predicted remaining processing time of each job so as to reduce average JCT.

As shown in Table 3, Optimus and Tiresias adopt greedy scheduling strategies, while ONES and DRL are dynamic because they are based on evolutionary search and deep reinforcement learning respectively. Different to the baseline schedulers, ONES is the only one that is able to allow elastic batch sizes. In contrast, Tiresias even does not allow elastic job size so that the number of GPUs of each job is fixed after submission. DRL cannot allow preemption so that it does not stop any job during its training until it is completed, which to some extent affects the scheduling efficiency because DL jobs are usually long, and appropriate job preemption can be beneficial to reduce average JCT.

Metrics: We use average job completion time (JCT) as the foremost metric of scheduling performance. The JCT of a job is the time period between its submission and completion. To clearly present the performance of ONES, we also provide box plots and cumulative frequency (CF) curves to illustrate the distribution over the JCT of all the jobs.

	p value (two-sided test)	p value (one-sided negative test)
vs. DRL	5.17e-4	0.99974
vs. Tiresias	4.53e-8	0.99999
vs. Optimus	7.55e-10	0.99999

Table 4: Significance tests using Wilcoxon test

4.2 Performance

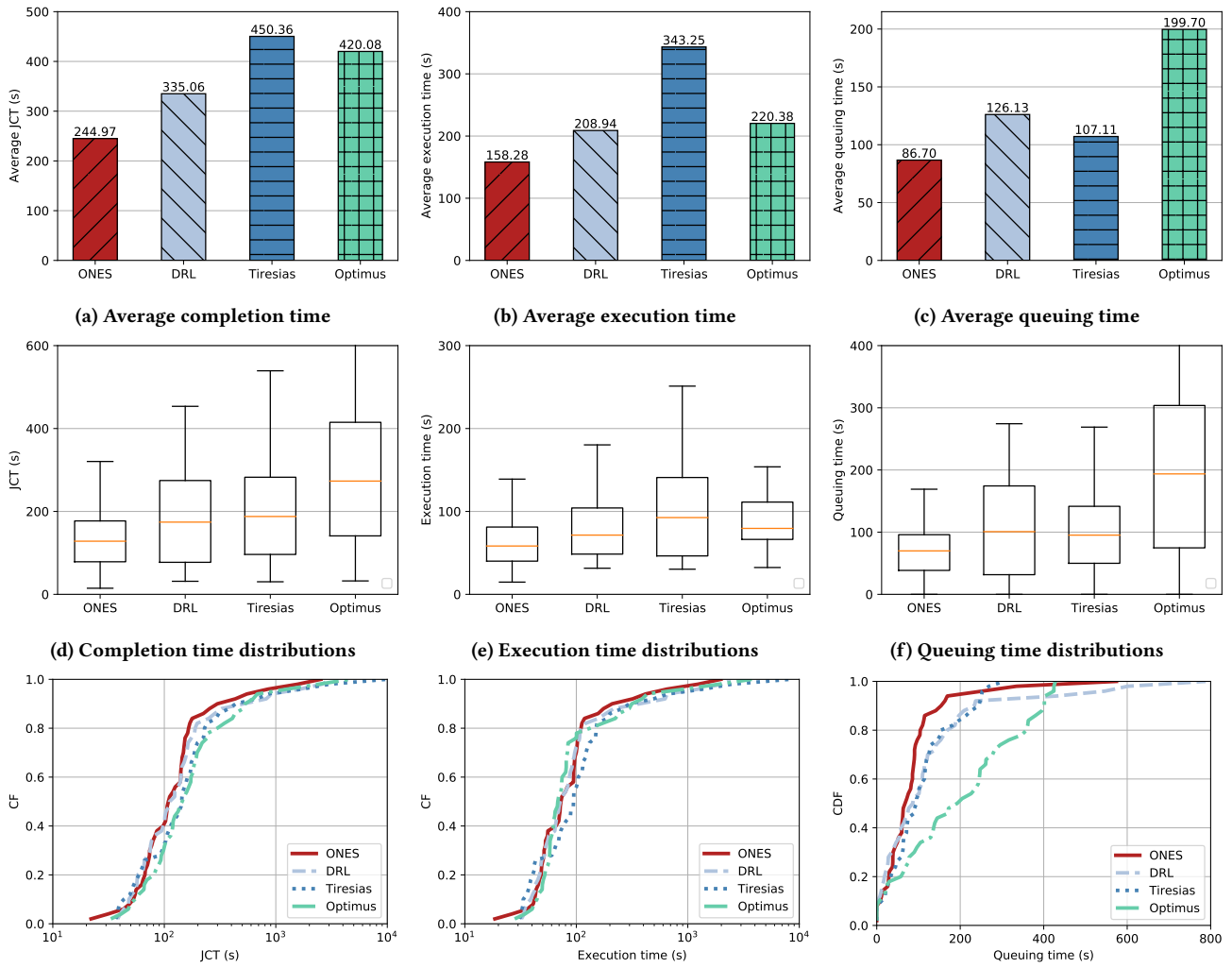
JCT comparison: Figure 15a shows the results of average JCT. We see that ONES can reduce the average JCT by 26.9%, 45.6% and 41.7%, compared to DRL, Tiresias [12] and Optimus [25], respectively. ONES can achieve the smallest average JCT of 244.97s. To see more details, Figure 15d and 15g shows the JCT distributions. We see that ONES is effective to reduce JCT for especially long jobs, as the fraction of jobs completed within 200s is 86%, while the baseline schedulers can only achieve about 60-80%. From the distribution, we observe that ONES is especially effective in reducing the completion time of slow jobs, which greatly contributes to the reduction in average completion time. Table 4 shows comparisons of ONES against baseline schedulers based on significance tests. We use non-parametric Wilcoxon tests [34] to present statistical analysis of each job's JCT and evaluate whether ONES outperforms the baselines. The two-sided tests evaluate a hypothesis that the results of ONES and the other scheduler are equivalent. Since the tests give p values (i.e. confidence) that are much smaller 0.05, we can reject their hypotheses. Next, The one-sided negative tests evaluate a hypothesis that the results of ONES are smaller than the other scheduler. We can accept the hypotheses and consider that ONES outperforms the baseline schedulers, because the tests give very high p values.

To explain the performance improvement of ONES, we then present further evaluations by breaking down the completion time of each job into execution time and queuing time.

Training faster: By dynamically scaling the batch size and GPU resources of each job, ONES makes its best effort to improve resource utilization and reduce training time. Figure 15b, 15e and Figure 15h present the results of ONES and show its efficiency in reducing execution time, compared to the baselines. While the Tiresias scheduler is not able to adjust the resource size for each job, ONES can sufficiently utilize the entire cluster and reduce 53.9% execution time. Besides, ONES achieves smaller execution time than both DRL and Optimus, because ONES can eliminate the performance bottleneck between the increasing number of GPUs and the decreasing training speed per GPU. As a result, for jobs with a relatively larger batch size, ONES can achieve better performance than the DRL and Optimus.

The distribution show that for fast jobs ONES has similar performance to the others, whereas ONES can obviously shorten the execution time of slow jobs. Basically all the jobs is completed under 2k seconds. However other schedulers all have jobs with longer execution time up 10k seconds.

Waiting less: Figure 15c, 15e and 15i show the queuing time performance of ONES. The online evolutionary algorithm (§3.2) can



(a) Average completion time (b) Average execution time (c) Average queuing time

(d) Completion time distributions (e) Execution time distributions (f) Queuing time distributions

(g) Completion time cumulative frequency curves (h) Execution time cumulative frequency curves (i) Queuing time cumulative frequency curves.

Figure 15: Comparisons of the scheduling performance.

help our scheduler achieve the smallest queuing time, compared to the strategies used by the baselines. We find that ONES can reach the smallest average queuing time, though its maximum queuing time may be longer. In overall, ONES improves the DRL scheduler by 31.3%, Tiresias by 19.1%, and Optimus by 56.6%. However, the queuing time performance of Optimus is a special case. The Optimus scheduler runs in a periodical pattern. Suppose jobs arrive at completely random time, the expected average queuing time is supposed to be half of the scheduling interval. When smaller scheduling interval leads to smaller average queuing time, it also results in higher overhead due to frequent rescheduling. In our experiments, we adopt the same scheduling interval of 10 minutes as in the Optimus paper [25].

4.3 Overhead Analysis

The major overhead of ONES comes from job size scaling. In this section, we compare the performance between our elastic batch size scaling and checkpoint-based migration for training different models. Checkpoint-based migration is a common approach used by DL jobs today. It first stops the training and saves a job status into checkpoint files, and then resumes the job from the checkpoint with new configurations. Figure 16 illustrates the result of the scaling overheads of different models implemented in PyTorch. The result shows that the overhead of elastic batch size scaling is basically around 1 second, while that of checkpoint-based migration can be greater than 20 seconds. Checkpoint-based approach mainly wastes time on preparing data and loading model to GPU devices. In contrast, they are not necessary in our approach. Therefore, our approach is able to significantly reduce the overhead.

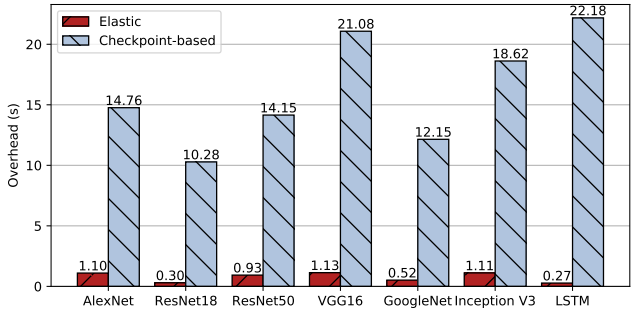


Figure 16: Comparison between the overheads of elastic batch size scaling and checkpoint-based migration.

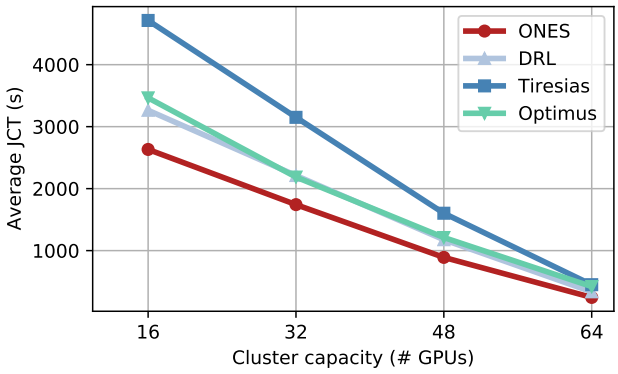


Figure 17: Comparison of scheduling scalability.

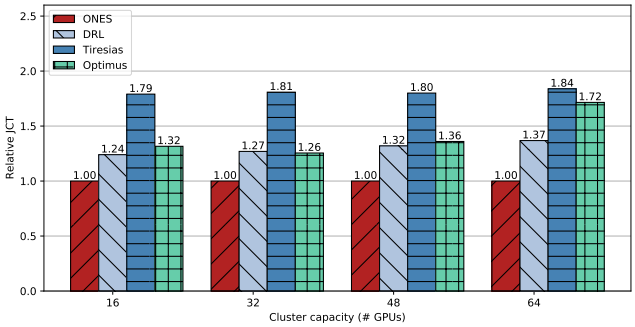


Figure 18: Scalability of improvements in JCT.

4.4 Scalability Analysis

Lastly, we examine the scheduling scalability by evaluating the average JCT performance under different cluster capacities. In this evaluation, we scale the cluster capacity from 16 GPUs to 64 GPUs.

The results are presented in Figure 17 and Figure 18. We see that all the schedulers are supposed to show similar trends. By increasing the number of GPUs, the average JCT will be almost linearly reduced. Especially the average queuing time will decrease linearly. Besides, we notice that by increasing the number of GPUs from 16 to 64, the improvement of ONES also increases from 19%/44%/24% to 27%/46%/42%, compared to DRL, Tiresias and Optimus respectively. with more GPUs (for example, with 64 GPUs in the figures), ONES may have a larger improvement. In other words, compared with the baselines, ONES can make the most efficient use of free GPU resources.

5 RELATED WORKS

Many prior works develop a variety of analytic performance models to optimize their resource scheduling algorithms based on different objectives. For example, SLAQ [39] adopts an online fitting model to estimate the training quality for future steps of each job by collecting the quality and resource usage information from concurrent jobs. Accordingly, it schedules the jobs to machines for maximizing the overall improvement in the training quality of all the jobs. Similarly, to minimize average JCT, Optimus [25] periodically adjusts the amount of resources for the DDL jobs with PS architecture. For each schedule, it builds a resource-speed model for each job, and then continuously adds a worker or a parameter server to the job with the maximum decrease in estimated JCT until the cluster is full. It allocates at least one worker and one parameter server to each job for fairness, whereas such policy may result in inefficiency when the cluster is over-subscribed. OASiS [3] gives the analysis of the performance of asynchronous training. It also presents a scheduler based on the greedy policies for solving an integer programming problem that aims to maximize the overall system utilization.

Some other DL schedulers put many efforts in developing job prioritization and placement policies. Gandiva [35] uses a time-slicing mechanism on GPU resources across multiple jobs to reduce the latency and improve cluster utilization. It works in an introspective way that continuously packs and migrates job placement to optimize the locality performance. Tiresias [12] attempts to reduce average JCT under the assumption that job lengths are hard to be measured in advance, which applies to our work as well. It proposes discretized priority queues for scheduling, based on attained service time of each job. But it cannot dynamically adjust the number of GPUs allocated to each job. Philly [15] is designed based on the analysis of the trace from Microsoft DL cluster. The analysis indicates the affect of the locality of DL jobs on queuing delay and GPU utilization, and then gives implications that schedulers should consider prioritizing locality, mitigating interference, as well as improving failure handling.

Apart from greedy scheduling algorithms, there is a recent trend in learning scheduling policy based on deep reinforcement learning [8, 22, 26]. On one hand, the performance of such schedulers heavily relies on the training traces because they assume that the distribution of scheduling policy will not change over time. In practice, whether the policy distribution can always work should be questionable. On the other hand, the DRL agent of such schedulers produces one action at each time, while each action usually works with only one job, so as to avoid oversized action space. Besides, the iteration speed of DRL schedulers is apparently slower than evolutionary search.

6 CONCLUSION

The focus of this paper is to study the resource scheduling problem for deep learning workloads in a shared GPU cluster. We aim to optimize the overall training performance by allocating an appropriate batch size to each job in real-time. In this paper, we propose ONES, which is the first to schedule the elastic batch sizes instead of the number of GPUs for DL jobs. The scheduler dynamically manages the batch sizes through an online evolutionary search algorithm and efficient batch size scaling mechanism. As a result, ONES is able to reduce the average JCT by up to 45.6%, compared to state-of-the-art methods.

7 ACKNOWLEDGMENTS

We thank CSCS (Swiss National Supercomputing Centre) for supporting our project to get access to the Piz Daint supercomputer. We thank TACC (Texas Advanced Computing Center) for supporting our project to get access to the Longhorn supercomputer and the Frontera supercomputer. We thank LuxProvide (Luxembourg national supercomputer HPC organization) for supporting our project to get access to the MeluXina supercomputer.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*. 173–182.

[3] Y. Bao, Y. Peng, C. Wu, and Z. Li. 2018. Online Job Scheduling in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 495–503. https://doi.org/10.1109/INFOCOM.2018.8486422

[4] Texas Advanced Computing Center. 2021. LONGHORN - TEXAS ADVANCED COMPUTING CENTER. https://www.tacc.utexas.edu/systems/longhorn.

[5] Mark E Crovella, Robert Frangioso, and Mor Harchol-Balter. 1999. *Connection scheduling in web servers*. Technical Report. Boston University Computer Science Department.

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Yifan Gong, Baochun Li, Ben Liang, and Zheng Zhan. 2019. Chic: experience-driven scheduling in machine learning clusters. In *Proceedings of the International Symposium on Quality of Service*. 1–10.

[9] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[10] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466. https://doi.org/10.1145/2740070.2626334

[11] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. 2018. SRPT for multiserver systems. *Performance Evaluation* 127 (2018), 154–175.

[12] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.

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

[14] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*. 1731–1741.

[15] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*.

[16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).

[17] Alex Krizhevsky. 2009. CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html.

[18] Stanford Vision Lab. 2016. ImageNet. http://www.image-net.org/.

[19] Chan-Yi Lin, Ting-An Yeh, and Jerry Chou. 2019. DRAGON: A Dynamic Scheduling and Scaling Controller for Managing Distributed Deep Learning Jobs in Kubernetes Cluster. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science, CLOSER 2019, Heraklion, Crete, Greece, May 2-4, 2019*. 569–577. https://doi.org/10.5220/0007707605690577

[20] Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. 2019. Dynamic mini-batch SGD for elastic distributed training: learning in the limbo of resources. *arXiv preprint arXiv:1904.12043* (2019).

[21] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 50–56.

[22] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.

[23] NVIDIA. 2020. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[24] Zafeirios C Papazachos and Helen D Karatza. 2010. Performance evaluation of bag of gangs scheduling in a heterogeneous distributed system. *Journal of systems and software* 83, 8 (2010), 1346–1354.

[25] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 3.

[26] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. 2019. DL2: A Deep Learning-driven Scheduler for Deep Learning Clusters. *arXiv preprint arXiv:1909.06040* (2019).

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

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

[29] Leslie N Smith. 2017. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 464–472.

[30] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

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

[32] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[33] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[34] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[35] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[36] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888* (2017).

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

[38] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet Training in Minutes. arXiv:1709.05011 [cs.CV]

[39] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. 2017. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 390–404.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We implemented our experimental scheduling system in Python codes. The system consists of a central controller and managers that are distributed on each node. The components communicate with each other by remote calls of the RPyC library.

ONES can be deployed as an independent resource manager for GPU resources, or on the top of common cluster resource managers such as Kubernetes with GPU support. The scheduler exposes an API for users to submit jobs with training scripts, initial batch sizes, and corresponding learning rates. When deployed on the top of some cluster resource manager such as Kubernetes, the scheduler acts as a broker between users and the resource manager by determining resource allocation and then running each worker of the jobs on a specified GPU.

In our experiments, we used our custom job traces that were generated by simulating the characteristics of public production traces from Microsoft. To generate a similar distribution of GPU utilization (#GPUs $\times$ execution time) to the Microsoft traces, we uniformly sampled 400 jobs from the traces and replace each with one of our jobs that has the closest GPU time. Besides, job arrivals follow a Poisson process with an average arrival time interval of 30 seconds. To capture steady state, we do not consider the first 5% jobs in the traces when collecting the results. We report not only the average values but also the distributions over all the jobs.

We implemented the jobs in Pytorch, with very few lines of changes in common training codes to enable the elastic batch size scaling. Some example codes are shown as follows.

```
1  import scaling
2
3  sa = scaling.ScalingAgent(ip_addr, port)
4  model, optimizer, loss_fn, dataloader = \
5      sa.load(user_model, user_optimizer,
6              user_loss_fn, user_dataset)
7
8  while not converge:
9      for inputs, targets in dataloader:
10             ### forward & backward
11             ...
12             ###
13             if sa.scaling_flag:
14                     model, dataloader = sa.scale()
15                     break
```

With the `scaling` package, in line 3-6, the user needs to create a scaling agent with the corresponding IP address and port of the worker, then load the job execution information such as the model, optimizer, loss function, and dataset into the scaling agent. Then in line 13-15 in the training loop, the user needs to check `sa.scaling_flag` and call `sa.scale()`. The method will invoke the scaling agent to automatically communicate with the central scheduler to execute batch size scaling in the background, as described in Section 3.3.1.

Author-Created or Modified Artifacts:

```
Persistent ID: https://doi.org/10.5281/zenodo.5205668
Artifact name: ONES scheduler for SC'21 AD/AE
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: We ran all the experiments on a platform of 16 GPU servers and HDFS. Each server has 2 20-core IBM Power 9 CPUs, 256GB memory and 4 NVIDIA Tesla V100 GPUs.

Operating systems and versions: Ubuntu 18.04 LTS.

Compilers and versions: Python v3.7.7

Applications and versions: AlexNet, ResNet, VGG16, GoogleNet, Inception V3, pre-trained BERT

Libraries and versions: Pytorch v1.4.0, Torchvision v0.5.0, NumPy v1.17.4, SciPy v1.3.1, CUDA v10.2, cuDNN v7.6.5, NCCL v2.5.6, RPyC v5.0.1.

Key algorithms: N/A

Input datasets and versions: CIFAR-10, ImageNet, COLA, MRPC, SST-2