2022 IEEE 15th International Conference on Cloud Computing (CLOUD) | 978-1-6654-8137-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLOUD55607.2022.00068

A Data-Loader Tunable Knob to Shorten GPU Idleness for Distributed Deep Learning

Danlin Jia*, Geng Yuan*, Xue Lin* and Ningfang Mi*
* Northeastern University, Boston, USA
Email: {jia.da, yuan.geng, xue.lin}@northeastern.edu, ningfang@ece.neu.edu

***Abstract*—Deep Neural Network (DNN) has been applied as an effective machine learning algorithm to tackle problems in different domains. However, training a sophisticated DNN model takes days to weeks and becomes a challenge in constructing research on large-scale DNN models. Distributed Deep Learning (DDL) contributes to accelerating DNN training by distributing training workloads across multiple computation accelerators (e.g., GPUs). Although a surge of research works has been devoted to optimizing DDL training, the impact of data-loading on GPU usage and training performance has been relatively under-explored. It is non-trivial to optimize data-loading in DDL applications that need intensive CPU and I/O resources to process enormous training data. When multiple DDL applications are deployed on a system (e.g., Cloud and HPC), the lack of a practical and efficient technique for data-loader allocation incurs GPU idleness and degrades the training throughput. Therefore, our work first focuses on investigating the impact of data-loading on the global training throughput. We then propose a throughput prediction model to predict the maximum throughput for an individual DDL training application. By leveraging the predicted results, A-Dloader is designed to dynamically allocate CPU and I/O resources to concurrently running DDL applications and use the data-loader allocation as a knob to reduce GPU idle intervals and thus improve the overall training throughput. We implement and evaluate A-Dloader in a DDL framework for a series of DDL applications arriving and completing across the runtime. Our experimental results show that A-Dloader can achieve a 23.5% throughput improvement and a 10% makespan improvement, compared to allocating resources evenly across applications.**

I. Introduction

Accelerating the training procedure of large-scale Deep Neural Networks (DNNs) has attracted concerns from both industry and academia. A remarkable number of solutions have been proposed in recent years to shrink the training time for DNNs, including distributed deep learning (DDL) [1], computation accelerator optimization [2], and communication overhead reduction [3]. However, data-loading is still one of the major components that dominate the training time of DDL applications [4] because GPUs are idle while training kernels are waiting for loading and transferring training data from storage drives. Data-loading involves loading data from storage to memory and pre-processing data, i.e., data augmentation and normalization. One possible solution to accelerating data-loading is to cache data in DRAM. However, modern training data sets range from hundreds of GB to tens of TB [5], making it impractical to buffer the whole data set in DRAM.

In such a way, data dependency exists between data loading and GPU kernel training. GPU kernels can be idle to wait for the preprocessed data before performing training. Intuitively, users can mitigate this data-loading cost by assigning more data-loading workers to DDL applications. As data-loading workers read and preprocess data simultaneously, increasing the number of workers can efficiently boost the data-loading speed. However, resource contention might occur when the number of workers increases to a certain extent. It thus becomes challenging to allocate workers to simultaneously running DDL applications. Users and system managers need to be aware of resource availability and application characteristics. A heuristic method is to construct experiments on different combinations of workers for a set of applications and run part of the applications because DDL applications are iterative in nature. However, this method is very time-consuming and cannot guarantee generality.

In this work, we design an **A**utomatic **Data-loader** allocation mechanism (named A-Dloader) for DDL frameworks, which allocates CPU and I/O resources among DDL applications practically and efficiently. Our design targets are to (1) make effective use of available data-loading workers based on different DDL model characteristics and resource demands and (2) shorten GPU idle intervals and thus optimize the overall training throughput. To achieve this goal, we confront three challenges: 1) estimate the resource demands (i.e., worker number) to achieve the maximum throughput for each application, 2) design an efficient worker allocation algorithm for overall throughput optimization, and 3) implement the worker reallocation mechanism programmatically and efficiently. The major contributions of this work are summarized as follows.

- **Model individual training throughput.** We first investigate the DDL training pipeline and derive a throughput prediction model (TPM) to model both data-loading and training phases. We use regression algorithms to learn our models and achieve a high prediction accuracy (above 90%). We also capture CPU and I/O contentions in TPM to avoid resource over-provisioning.
- **Design a run-time worker allocation mechanism.** We propose an efficient run-time worker allocation algorithm to optimize the global throughput. The reallocation mechanism is supported by an event message communication system for orchestrating each component. We also develop an application master to perform intra-application management.
- **Implement and evaluate A-Dloader on a real system.** We implement A-Dloader on top of PyTorch and evaluate A-Dloader in a real testbed by constructing experiments on both static and dynamic workloads. In all experiments, we consider different applications with various workload characteristics. The experimental results show that A-Dloader can improve up to 23.5% overall throughput and

[1]This work was partially supported by National Science Foundation Award CNS-2008072.

2159-6190/22/$31.00 ©2022 IEEE
DOI 10.1109/CLOUD55607.2022.00068

10% makespan (i.e., end-to-end execution time).

In the remainder of this paper, we introduce the background of distributed deep learning and our motivation in Sec. II. Sec. III proposes TPM and introduce our implementation of A-Dloader. In Sec. IV, we show the evaluation of A-Dloader on static and dynamic workloads in a real testbed. Sec. VI discusses the related work. The conclusion and future work is shown in Sec. VII.

II. Background and Motivation

A. Deep Neural Network

Deep Neural Networks (DNNs) are generally constructed of multiple stacked layers. According to the types of layers used, the DNN can be divided into different categories, such as the multilayer perceptron (MLP), convolutional neural network (CNN), recurrent neural network (RNN). In this work, our study focuses on the training of CNNs that are commonly used in image classification applications. Besides the types of DNNs, the network structures of the same type of DNNs can also vary a lot, leading to a significant difference in training a neural network. VGG [6] and Resnet [7] are two common CNNs in a range of computer vision tasks. The former is a "plain" network with simply stacked convolutional layers, while the latter introduces a residual learning structure that mitigates the problem of vanishing/exploding gradients, making the large-scale network trained quickly.

The backpropagation (BP) algorithm [8] is widely used to train a CNN model, which propagates the total loss back into the neural network in iterations. Generally, each iteration of BP consists of three steps, i.e., forward propagation, backward propagation, and model updates. The training latency of a neural network largely depends on the size of the neural network (e.g., the total number of parameters/weights), the computation costs (FLOPs), and the number of training samples in each iteration.

B. Distributed Deep Learning

To train DNNs efficiently on an extensive training data set, distributed deep learning (DDL) frameworks apply the concept of data parallelism to distribute the workload across computation accelerators (e.g., GPUs) and coordinate the training kernels globally [1]. Fig. 1 shows how modern DDL frameworks (e.g., PyTorch) read training data sets from storage drives and perform the DNN training on multiple GPUs. Specifically, the training data set is first partitioned into batches based on the predefined batch size that specifies the number of the training data samples (e.g., images or frames in videos) in a batch. DDL frameworks train a DNN model in iterations, where each iteration performs the same operations on a batch for training. Each batch is further divided into mini-batches that spread across GPUs evenly. Each GPU saves a replica of the DNN model and performs the same computation on different mini-batches. At the end of iterations, the outputs from each GPU are aggregated into a global result for updating the model.

As shown in Fig. 1, DDL frameworks typically launch multiple master (CPU) processes per GPU to distribute mini-batches across GPUs. Each master process spawns worker processes to load training data from the storage drive simultaneously. A worker process involves loading and preprocessing mini-batches before sending data to GPUs. A master process maintains a FIFO (First-In First-Out) queue to transfer the preprocessed mini-batches to its attached GPU training kernels. Workers enqueue the preprocessed mini-batches into the FIFO queue and immediately load the next mini-batch. The training kernel then fetches enqueued preprocessed mini-batches at the beginning of each iteration.

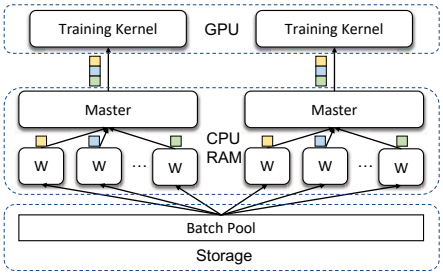

Fig. 1. Distributed Deep Learning framework. "W" denotes worker process.

C. DDL Training Pipeline

Fig. 2 shows the DDL training pipeline with two workers for a single GPU training kernel. As mentioned above, a master process first spawns multiple workers for data-loading (i.e., loading and preprocessing mini-batches). These workers then start to simultaneously read (I/O related) and preprocess (CPU related) mini-batches. At the same time, the GPU training kernel is idle until the preprocessed mini-batch is ready. Once the master process transfers a preprocessed mini-batch to the GPU training kernel, the training kernel starts to run the BP algorithm [1]. We denote an execution of BP as a step. The time between the ends of two consecutive steps is called the latency of an *iteration*. The DDL training continues iteratively until the last mini-batch is consumed. We call the procedure of training all batches as an *epoch*. The DDL training repeats epochs until the accuracy that the model achieves is satisfied. Therefore, the overall training time depends on the number of epochs of an application and an epoch's latency, where the former mainly relies on the model accuracy while the latter is mainly determined by the iteration latency.

A surge of research works have been conducted to shrink the training pipeline by reducing the time cost of steps [1] [2] [3]. However, the impact of data-loading in the DDL training pipeline has been relatively under-explored. At each iteration, a training kernel might be idle and waiting for the preprocessed mini-batch. We can derive the idle time at GPU kernels by excluding the step time from the iteration duration, see Fig. 2. Meanwhile, the data-loading and data transfer of iteration *i* can

[1]The execution flow of BP algorithm varies across different DDL architectures, e.g., Parameter Server, All-Reduce, and Ring-Reduce.To simplify our throughput model, we consider that the forward propagation, backward propagation, and model updates are executed sequentially in PyTorch.

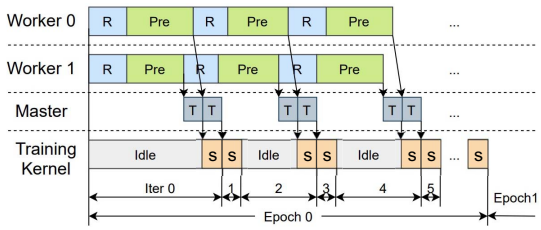

Fig. 2. DDL training pipeline. "R" denotes reading data from local/remote storage. "Pre" refers to preprocessing data. "T" is transferring data from CPU to GPU. "S" represents the training step, i.e., the execution of the backpropagation (BP) algorithm on a batch/mini-batch.

overlap with the step of iteration $i-1$ at GPU training kernels because the data-loading is independent of the outputs of step. For example, as shown in Fig. 2, as there are two workers loading mini-batches, the second iteration can immediately start the next step. We notice that idle intervals (especially long ones) at GPU kernels are undesirable because GPU resources have not been fully utilized for training. If one can shorten or remove idle intervals, then the DDL training time can be reduced, and the throughput[2] of DDL training can be increased. *Therefore, the goal of this work is to maximize resource utilization on GPU kernels and obtain high training throughput by avoiding the idle time at each training kernel.*

D. Motivation: Throughput Investigation

One practical way to avoid idle time at training kernels is to reduce the data-loading time at workers so that GPU kernels can continuously run steps for preprocessed mini-batches without any waiting, such as iteration 1 in Fig. 2. It is intuitive to allocate as many workers as possible to a DDL application to decrease the data-loading time and thus reduce the idle time. However, the resources (i.e., CPU and I/O bandwidth) for workers in the system are limited. When multiple DDL training applications are launched simultaneously, assigning too many workers to each application can inevitably cause severe resource contention at the master and thus contrarily degrade data loading performance. Therefore, *this paper tackles the critical problem for the design of a resource (worker) allocation scheme among multiple applications, aiming to minimize GPU idle time and maximize the overall training throughput.* It is a non-trivial problem because the allocation can be affected by many factors, including resource capability and availability, application workload characteristics, and system running status.

We conduct preliminary experiments in real systems to investigate the impact of worker allocation on the performance regarding training throughput. In these experiments, we launch multiple applications simultaneously, with various numbers of data-loading worker processes allocated to each application.

[2]The training throughput is defined as the number of training samples that are processed on training kernels in unit time. Since multiple GPUs are training on different mini-batches from the same batch simultaneously, the throughput of a DDL application is calculated by dividing the batch size on the average iteration latency.

We first equally distribute data-loading workers among applications, i.e., each application receives the same number of workers regardless of the variance of their training workloads caused by different model sizes, batch sizes, and other hyper-parameters. This is considered as the baseline, referred to as *balance*. We also design the other two experiments, where the number of data-loading workers is proportional (resp. inversely-proportional) to the model size of each application. We refer to the "proportional" experiment as *proportion* and the "inversely-proportional" experiment as *inverse-proportion*.

Specifically, we run these experiments on the testbed shown in Sec. IV. In each experiment, we submit three applications to simultaneously train three DNNs, including Resnet18 (11.7 million weight parameters) and two small synthetic models (i.e., mdResnet12 and smResnet20 have 1.8 and 0.9, respectively, million weight parameters). We use ImageNet as our training data set, and set the same hyper-parameters for all models, e.g., the batch size is set to 128 images, the learning rate is 0.1, the momentum is 0.9, and the weight decay is 0.0001. Each application is distributed across two GPUs (one GPU is shared across two applications), training the model with 500 iterations. The performance of applications is expected to be consistent across iterations.

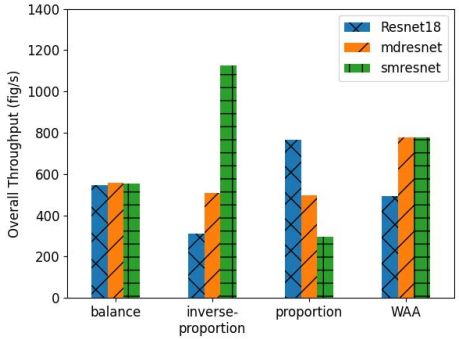

Fig. 3. Training throughput of three DDL applications. Balance, Inverse-proportion, and Proportion gives Resnet18, mdResnet12, and smResnet, (6,6,6), (2,4,12), and (12,4,2) workers, respectively. WAA (our new scheme) allocates workers dynamically.

Fig. 3 shows the training throughput of each application under three manual allocation approaches. We will discuss our dynamic allocation scheme (WAA) later in Sec. IV-B. We also calculate the overall training throughput under each approach by summing the individual throughput of the three applications in each experiment. We observe that all applications, as expected, have a similar throughput under the balance, as applications are allocated with the same number of workers, and the overall throughput is $1,660$ images/sec. However, when we proportionally allocate more workers to applications with larger models (a more intuitive way), the overall throughput, in contrast, drops by 7% (i.e., $1,554$ images/sec). We notice that Resnet18 is a relatively large model that requires more computation (i.e., long step time) at GPU training kernels. Allocating too many workers to Resnet18 can help reduce idle time but not significantly because step is the bottleneck for training Resnet18. Mean-

while, the training throughput of the other two applications with smaller models is dramatically degraded because fewer workers incur much longer idle intervals. Therefore, under the *inverse-proportion* setup, we counter-intuitively give more workers to applications with smaller models. We found that although we sacrifice Resnet18's throughput, the other two applications (i.e., mdResnet12 and smResnet20) obtain the throughput boosts that make up the throughput degradation in Resnet18. The overall training throughput under the *inverse-proportion* is thus increased by 14% $(1,928$ images/sec), compared to the balance.

The above throughput improvement under the *inverse-proportion* indeed comes from the fact that the *inverse-proportion* increases GPU utilization by shortening the GPU idle time. We remark that such an improvement can significantly reduce the overall training time. Considering a general DNN application training on the ImageNet dataset with 150 epochs, the *inverse-proportion* can save 10+ hours of training time in total. This motivates that optimizing data-loading has significant potential in accelerating DNN training. Furthermore, from the above observations obtained in Fig. 3, we conclude that manually setting the number of workers for applications with different DNN models cannot fully utilize resources in the system. This thus motivates us to design a framework that can automatically and dynamically allocate workers among running applications considering resource availability and DNN workload features.

III. A-DLOADER DESIGN

A. Problem Formulation and Challenges

Our design focuses on the optimization of the overall throughput for multiple DDL training applications running simultaneously on a multi-GPU node by automatically adjusting the CPU and I/O resources allocation (i.e., workers) at runtime. We formulate this problem as a maximization optimization problem shown as formula 1, where $T_i(d_i)$ represents a throughput function of application i with the number of workers d_i allocated to the application as a variable.

$$\max_{\{d_1,d_2,\ldots,d_N\}} \sum_{i=1}^{N} T_i(d_i) \tag{1}$$

We have the following challenges in solving the problem:

- The first challenge is to formulate the throughput function T_i with respect to d_i. The throughput of a DDL training application is affected by multi-factors, including system specifications (e.g., I/O bandwidth, CPU processing rate, and GPU number and processing rate) and workload characteristics (e.g., batch size, model size, and epoch number). We need to consider these factors in our formulation for a precise model.
- The second challenge is how to apply our throughput prediction model to instruct worker allocation to improve system-wide throughput. We need a global worker allocation scheme to distribute limited CPU and I/O resources across all applications for throughput optimization and consider the contentions caused by potential over-provisioning, i.e., worker processes compete for resources.
- The third challenge is that, as we know, there are no dynamic worker allocation mechanisms in PyTorch or other DDL frameworks, nor existing research works on it. Once the system receives a new application submitted by users or an application completes, our design needs to reallocate workers for applications at run-time with a negligible impact on the system performance.

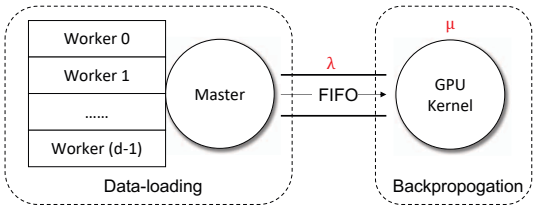

Fig. 4. Data-loading and processing rates.

B. Throughput Prediction Model

In a typical DDL application, the training data (i.e., a batch) is distributed across multiple training kernels (i.e., GPUs), and a single training kernel repeats iterations of processing mini-batches as shown in Fig. 2. Because all training kernels need to synchronize the updated model at the end of an iteration, the (average) iteration time of all training kernels is the same. Therefore, the throughput of an application is defined as how many training samples are processed per second, as calculated by formula 2, where B is batch size and $\bar{t}_{iter}$ represents the average iteration time. We can further break $\bar{t}_{iter}$ to average idle time ($\bar{t}_{idle}$) and average backpropagation (BP) time ($\bar{t}_{bp}$).

$$T = \frac{B}{\bar{t}_{iter}} = \frac{B}{\bar{t}_{idle} + \bar{t}_{bp}} \tag{2}$$

To derive $\bar{t}_{idle}$ and $\bar{t}_{bp}$, we investigate the data-loading and training rate at workers and GPU kernels, respectively. Fig. 4 shows the procedure that a master process sends pre-processed mini-batches to training kernels via a FIFO queue. We use λ to represent the data-loading rate defined as the number of mini-batches pre-processed per second. We denote the training rate (i.e., batch-consuming rate) as μ, indicating the number of mini-batches consumed by the training kernel per second. Therefore, we derive $\bar{t}_{bp}$ from formula 3, and $\bar{t}_{idle}$ from formula 4. In formula 4, $\mu < \lambda$ means training kernel consumes mini-batches slower than data-loading, and the training kernel can (almost) always fetch a mini-batch from the FIFO queue. Thus the average idle time is approximately zero. Otherwise, the average idle time is determined by the difference between the time taken for data-loading and backpropagation.

$$\bar{t}_{bp} = \frac{1}{\mu} \tag{3}$$

$$\bar{t}_{idle} = \begin{cases} \theta_1 \approx 0 & \text{if } \mu < \lambda \\ \frac{1}{\lambda} - \frac{1}{\mu} & \text{if } \mu \geq \lambda \end{cases} \tag{4}$$

As λ and μ are determined by multiple factors, including system performance and workload characteristics, we extend

λ and μ to functions $F_\lambda(d)$ and $F_\mu(\omega)$, where ω represents parameters representing the application's characteristics (such as batch size, model size and FLOPs), and d is the worker number. Therefore, we have a relationship between the throughput and the worker number, bridged by the formulation of λ and μ, as shown in formula 5.

$$T = \begin{cases} B * F_\mu(\omega) & \text{if } \mu < \lambda \\ B * F_\lambda(d) & \text{if } \mu \geq \lambda \end{cases} \quad (5)$$

Training Rate. The training rate (F_μ) depends on the average time cost to complete a backpropagation algorithm. For each iteration, the average BP time consists of the time cost of doing forward propagation (i.e., calculating loss), backward propagation (i.e., calculating local gradients), communications between GPUs, local gradients aggregation, and model updates, as introduced in Sec. II-C.

Forward propagation executes the same operators on a mini-batch of samples, as the neural network is predefined. Assume the mini-batch size is b, and the number of floating-point operators is f, the forward propagation time cost is proportional to b and f, formulated as $\alpha_0(b*f)$. Backward propagation has the same computation characteristics as forward propagation but has two times the computation expense of the forward propagation [9]. Therefore, we formulate backward propagation as $2\alpha_0(b*f)$. The communications involve gathering local gradients from all GPUs to one and then passing the updated model to all other GPUs. We denote the communication bandwidth as S and assume the application uses n GPUs, then each GPU obtains $S/(n-1)$ bandwidth. As the model has the same size as the gradient, the time cost of communications is formulated as $\alpha_1 \frac{2*M}{S/(n-1)}$, where M represents the model size. Once gradients derived for all samples are collected on one GPU, the gradient aggregation happens with a time cost proportional to the model size and the mini-batch size. Therefore, the aggregation cost is formulated as $\alpha_2(b*M)$. The last part is the model update, which is only related to the model size and can be formulated as $\alpha_3 M$.

We put S into α_1 and all constant numbers into other α's as a part of coefficients to learn. We also merge the expression of forward and backward propagation. Therefore, the training rate can be modeled as formula 6. It is worth noticing that F_μ is irrelevant to the number of workers, as the worker number only affects CPU and I/O resources allocated to an application. Therefore, given the hyper-parameters of an application, we can predict the training rate before the application runs.

$$F_\mu = (\alpha_0(b*f)+\alpha_1(M*n)+\alpha_2(b*M)+\alpha_3 M+\alpha_4)^{-1} \quad (6)$$

We use different regression models to learn α's by collecting samples of (b, f, M, n, F_μ). We run each training application on ImageNet for 500 iterations. Note that due to the iterative nature of DDL applications, hundreds of iterations should be enough to collect reliable samples. We try as many as possible combinations of mini-batch size and number of GPUs (i.e., *b* and *n*) for different DDL model architectures. For each DDL model architecture, we use THOP [10] to get floating-point operations and the model size (i.e., f and *M*). Each run takes about tens to hundreds of seconds. We filter out samples where the BP time takes 95% of the iteration time to make sure these samples satisfy $\mu < \lambda$, i.e., no idle time. We generate 100+ samples and use 60% percent of samples for training and the other samples for validation. Table I summarizes the used regression models and their corresponding prediction accuracy. We choose the F_μ model with the highest accuracy to infer the throughput prediction.

TABLE I
REGRESSION MODEL ACCURACY.

Model	Training Rate	Data-loading
Linear Regression	0.93	0.98
Polynomial Regression	0.94	0.98
K-Nearest Neighbor	0.84	0.97
Random Forest Regressor	0.87	0.97

Data-loading Rate. Data-loading of a mini-batch involves two sequential procedures i.e., reading samples from storage drives and executing pre-processing on samples. Therefore, the data-loading rate (F_λ) can be formulated as $(\bar{t}_{read} + \bar{t}_{pre})^{-1}$. We denote R as I/O bandwidth and U as CPU processing rate. For a single data-loading worker, without consideration of contentions, the data-loading rate is $(b/R + b/U)^{-1}$, where R and U are constant for a given worker. As we have d workers for each GPU, the data-loading rate can be modeled in formula 7.

$$F_\lambda = \frac{d}{b} * \frac{1}{R^{-1} + U^{-1}} \quad (7)$$

However, we found that contentions of CPU and I/O resources are unavoidable as the number of workers increases system-widely. To learn the impact of contentions on F_λ, we construct experiments on a training application with a small DDL model, which ensures that $\mu \geq \lambda$, i.e., throughput is determined by arrival rate, i.e., F_λ. Because the worker processes of applications are scheduled by the OS and are homogeneous regarding the impact of contentions, we can use one application to investigate the degradation of the data-loading rate caused by resource contentions. Fig. 5 shows the data-loading rate as a function of the number of workers. We observe that the increasing rate of F_λ drops at 11 workers, which means it reaches the point where resource contention starts to degrade the data-loading rate. Finally, F_λ saturates at around 24 workers as our testbed can only support 24 CPU threads.

To reflect the above impact of resource contentions, we use regression models to learn values of R^{-1} and U^{-1} instead of using constant values. Specifically, we divide the function of F into three stages (i.e., as shwon in Fig. 5) and learn a pair of $\{R, U\}$ for each stage. The accuracy of different regression models is shown in Table I, where the accuracy is as high as 0.98. It is worth noticing that our TPM training cost includes the overhead of collecting samples and training the model. The former can be finished by running only a few hundred of iterations of applications because DDL applications repeat the same computation across iterations. The latter only takes

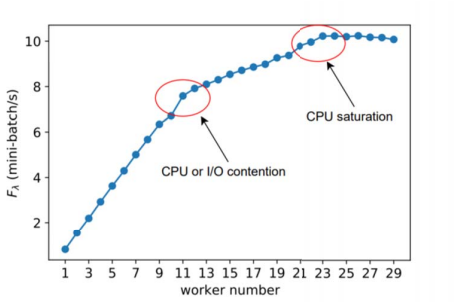

Fig. 5. Data-loading rates under resource contentions. Contention-free stage: workers range from 1 to 11. Contention stage: workers range from 11 to 24. Saturation stage: workers larger than 24.

tens of seconds (e.g., 12 seconds in our experiments) because regression models are less complex.

C. A-Dloader Implementation

As discussed in Sec. III-A, no dynamic worker allocation mechanisms have presently been supported in existing DDL frameworks. Because most recently emerging works on data-loading optimization are implemented on PyTorch (a widely used DDL framework), and as reported by Facebook [11], a workload study shows that more than 60% of GPU hours during 05/11/20 to 06/05/20 were spent on calling PyTorch packages. Therefore, one of our main contributions is the implementation of A-Dloader on top of PyTorch to enable worker reallocation at run-time.

1) System Overview: Fig. 6 shows the architecture of A-Dloader . Specifically, 1 the user can submit DDL training applications via the *user interface* to the scheduler. 2 The *scheduler* is responsible for managing and scheduling I/O and CPU resources for applications at run-time by dynamically altering the number of workers for applications. 3 The *life-circle monitor* tracks the status of applications, and 4 reports to the *worker allocator*. The worker allocator also maintains the information of on-the-fly applications, including both user specifications and run-time status. Then the worker allocator makes reallocation decisions by using the throughput prediction model. Finally, 5 the worker allocator instructs the scheduler to reallocate workers among applications at run-time.

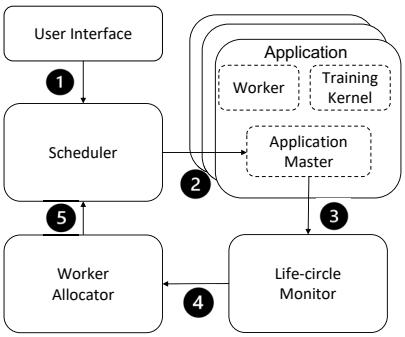

Fig. 6. A-Dloader Architecture.

2) Worker Allocator: The worker allocator module receives the run-time status of applications from the life-circle monitor. It also obtains the application's information about hyper-parameters passed by users in submission. Then the worker allocator uses our throughput prediction model (TPM), see Sec. III-B, to estimate the training rate and data-loading rate for each application. With the combination of formulas 5, 6, and 7, we derive the maximum throughput that each application can achieve (e.g., $T_{max} = B * F_\mu$ when F_μ equals F_λ). Considering the limited worker processes a node can support, we design a worker allocation algorithm to maximize the overall throughput of all running applications, as shown in Algorithm 1.

The Worker Allocation Algorithm (WAA) takes the hyper-parameters of all applications, the number of applications, and the maximum number of CPU threads as input arguments and returns the number of workers per GPU (master) for each application to maximize the overall throughput. The WAA first makes sure each GPU at least has one worker, see lines 1-2. Then, for each application, the WAA calculates the training rate by passing the hyper-parameters to formula 6 and calculates $d_i^{'}$ that can maximize the throughput, i.e., lines 4-5. However, $d_i^{'}$ is a float number, and WAA needs to return the number of workers per GPU, which is an integer. Therefore, we denote d_i to be the ceiling value of $d_i^{'}$ (i.e., line 6), and the gap between $d_i^{'}$ and d_i is the number of over-provisioned workers. To avoid over-provisioning, the WAA sorts d_i by the over-provisioned workers descendingly in line 8. Finally, the WAA decreases d_i by 1 from the most over-provisioned application until the overall workers are equal to the maximum CPU threads, i.e., line 10 to 15. We note that the WAA ensures each application at least obtains one worker per GPU in lines 13-14.

Algorithm 1: Worker Allocation Algorithm

Input: The hyper-parameter set $W = \{\omega_i\}$, maximum CPU threads C, and application number N
Result: The worker number set $D = \{d_i\}$ per GPU

```
1  if C ≤ N then
2  |   return d_i = 1,  ∀d_i ∈ D
3  for i ← 1 to N do
4  |   μ ← F_μ(ω_i)
5  |   d_i' ← the worker number under μ = F_λ(d_i')
6  |   d_i ← the ceiling value of d_i'
7  end
8  D ← sort D by d_i − d_i' descendingly
9  while sum of D > C do
10 |   for i ← 1 to N do
11 |   |   if sum of D < C then
12 |   |   |   return D
13 |   |   else if d_i > 1 then
14 |   |   |   d_i = d_i − 1
15 |   end
16 end
```

3) Scheduler: The scheduler dynamically reschedules workers at points where the submission or completion of applications happens by pausing and retrieving applications with new worker allocation. In order to trigger worker reallocation at run-time, we design an event message system for communications among the scheduler, application masters, and the user

interface. When a new batch of applications is submitted to the user interface, the user interface sends an "Arrival" signal to the scheduler with the applications' information. The scheduler first wakes up the worker allocator that takes the information of the currently running applications and the latest submitted ones to calculate the new d_i. The scheduler then sends "Pause" signals to application masters that need to reallocate workers, i.e., the new d_i is different from the previous one. The life-circle monitor sends a "Paused" signal back to the scheduler if applications are paused successfully. Then the scheduler launches the submitted applications and resumes the paused applications with new worker allocations.

Similarly, when an application is completed, the application master sends a "Finish" signal to the scheduler via the life-circle monitor. Then the scheduler wakes up the worker allocator to check whether it is necessary to reallocate the spare workers released by the finished application to the other applications. If the throughput of specific running applications can benefit from more workers, the scheduler will pause and retrieve those applications with a new worker number. Otherwise, the scheduler will keep those release workers idle until new applications come.

4) Application Master: The application master is responsible for launching data-loader workers and training kernels. The application master will be launched first when the scheduler launches an application. Then the application master spawns a set of worker processes and training kernels on GPUs. To communicate between the master and training kernels to track the application's information (e.g., heartbeat) at run-time, we also design an intra-application event message system. Once the application master receives a "Pause" signal from the scheduler, the master sends "Pause" signals to all training kernels and waits for all kernels to exit. Since all training kernels have the same model at the end of each iteration, the training kernel with index 0 saves the current model into a storage drive for future retrieval. When all training kernels exit, the application master sends a "Paused" signal with the model path to the life-circle monitor that further notifies the scheduler.

5) User Interface and Life-circle Monitor: Any submission or completion of an application can trigger the pause/retrieve procedure in the scheduler of A-Dloader. High overhead (e.g., paused time) may be caused by high trigger frequency. We thus design two buffers in the user interface and the life-circle monitor to mitigate this impact. For example, we set a time window (a configurable parameter) in the user interface for buffering submitted applications. Suppose multiple applications arrive within the same time window. In that case, they are submitted in a batch to the scheduler at the end of the time window. Similarly, we design a completion buffer at life-circle monitor to control the reallocation frequency caused by finishing application. We set the time window of 30 seconds for both buffers in all of our experiments. Users can adjust this parameter value based on their workload intensity.

IV. Evaluation

A. Methodology

Testbed. We evaluate our design on a GPU server with 4 Geforce RTX 2080ti GPUs (11GB memory per GPU), 128GB RAM, 24 Intel i9-9920X CPUs (3.50GHz), and a Samsung 970 EVO 2TB SSD. Our design is on Ubuntu18.04 with kernel 5.4.0-90-generic. A-Dloader is implemented on top of PyTorch 1.7.1. We apply NCCL [12] (NVIDIA Collective Communications Library) as the protocol for GPU communications. We only use 18 CPU cores to avoid the interference of other background applications. The ImageNet [13] dataset is used for evaluation with an input image size of 224×224.

Workload Generator. We design a workload generator to emulate the user behavior submitting applications to A-Dloader. The workload generator generates a list of application submission requests. Each request includes the arrival time, hyper-parameters, GPU index to use, and the number of iterations to finish. As our goal is to maximize the overall throughput and the change of worker number does not affect prediction accuracy, we discard the consideration of accuracy in our work. We use the VGG and Resnet family models as the representative DNNs in our experiments. Besides the commonly used model configurations, such as Resnet18, Resnet34, VGG13, and VGG16, we also generate a set of synthetic DNN models to extend our workload variety for learning TPM and an extensive evaluation. The synthetic models include the Resnet and the VGG models with the customized number of convolutional layers (e.g., VGG7, Resnet28) and the corresponding $4\times$ and $2\times$ narrower versions, which are named smResnet, smVGG, mdResnet, and mdVGG, respectively.

Evaluation Metrics. We use throughput, defined as the number of training samples processed by GPUs per second, as our first metric to represent the utilization of GPUs in the system. We use makespan, defined as the duration between the start of the first application and the end of the last application, to investigate the end-to-end latency under different worker allocation schemes.

Baselines. We have three baselines of worker allocation schemes. The proportion baseline allocates more workers to larger models, while the inverse-proportion baseline allocates more workers to smaller models. The balance baseline gives the same number of workers to all applications. Whereas, in a scenario where users submit applications at random time intervals, our baseline distributes available workers evenly across applications without exceeding the maximum worker numbers.

B. Static Workload

We first construct experiments on a static workload, where all applications are submitted at the same time, to evaluate the accuracy of our throughput prediction model (TPM) and the effectiveness of our worker allocation algorithm (WAA). Under a static workload, the WAA is only triggered at the beginning of the experiment and the completion of each application. The same DDL applications (i.e., Resnet18,

mdResnet12, and smResnet20) as used in Sec.II-D are considered in this set of experiments. Fig. 3 shows the throughput of each application under three baselines and our WAA. Table II also shows the performance comparisons in terms of GPU idle time, GPU utilization, and execution time under different schemes.

As shown in Fig. 3, compared to the balance baseline, our WAA achieves the throughput improvement by 23.5%, which is slightly better than the inverse-proportion and does not degrade the throughput of Resnet18 dramatically. We further observe that by optimizing data-loading, our WAA can reduce the GPU idle intervals and thus increase the average GPU utilization, see Table II. As a result, both total training throughput and average training latency are improved under WAA. But, we also notice that WAA obtains a slightly longer makespan than the other two baselines (i.e., balance and inverse-proportion). This is because the latency of Resnet18 is increased, which causes a longer makespan, although the latencies of mdResnet12 and smResnet20 decrease. We remark that when DDL applications use more epochs to train, our WAA is able to reduce the makespan significantly, see the results in Sec. IV-C.

TABLE II
WORKER ALLOCATION SCHEME COMPARISON.

	balance	propotion	inverse-propotion	WAA
ave. GPU idle (s)	92	125	107	84
ave. GPU util. (%)	28	18	27	30
ave. latency (s)	130	153	147	119
makespan (s)	132	222	163	160

To elaborate on how WAA works, we further show WAA's run-time events for allocating workers in Fig. 7. Recall that the total number of workers available is 18 as we only use 18 CPU cores for data loading in our testbed, see Sec. IV-A. At the beginning of the execution, we observe that WAA allocates 2, 8, and 8 workers to Resnet18, smResnet20, and mdResnet12, respectively. At around 97 seconds, smResnet20 and mdResnet12 are finished and stored in the completion buffer. After the scheduler receives the completion buffer, the worker allocation mechanism is triggered to reallocate the just-released workers to the remaining application, i.e., Renset18. Note that Renset18 only needs eight workers to ensure the data-loading rate equals the training rate. Thus, after pausing and retrieving, Renset18 is assigned with eight workers to achieve its maximum throughput, finishing at around 160 seconds. Lastly, we measure the overhead of one round of pausing and retrieving, i.e., about 16 seconds, which is relatively low and ignorable when performing a general DNN training that usually takes tens and even hundreds of hours. We contribute the throughput improvement to WAA's two designs. First, WAA relies on our TPM to accurately predict the number of workers that should be allocated to running applications to maximize the overall training throughput. Consequently, WAA distributes workers inversely proportionally to the model sizes, which is consistent with the inverse-proportion method. In addition, WAA ensures no over-provisioning of CPU and I/O resources to any applications. For

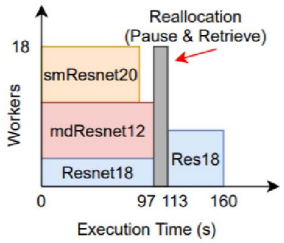

Fig. 7. WAA's run-time events for worker allocation under static workload.

example, eight workers are sufficient for Resnet18 to reach the optimal throughput point. Under this consideration, WAA only allocates eight workers (as shown in Fig. 7) to Resnet18. This design can avoid reallocation if a new application that requires less than eight workers comes after 113 seconds in Fig. 7. Second, our WAA automatically reallocates workers if spare workers are detected. This design can better utilize CPU and I/O resources and further improve the overall throughput compared to the inverse-proportion method.

C. Dynamic Workload

We further construct experiments on dynamic workloads that have applications arriving and completing at random points to evaluate our design systematically. Table III summarizes the information of a dynamic workload that we use in these experiments, where we have totally eight applications arriving within 800 seconds, and these applications have different model sizes and workload hyperparameters, such as batch size, iteration number. We also design this dynamic workload by considering the resource limit of CPUs and GPUs in our testbed (see Sec. IV-A for details) to avoid GPU's out of memory (OOM) and CPU's overloading. For example, we set a small batch size (e.g., 64) for a large model (e.g., VGG13) to fit its training procedure in GPU's memory. Additionally, a resource scheduler (such as Yarn [14] and Mesos [15]) commonly lets a new application wait in a queue if the required resource (e.g., virtual CPUs) is more than the available one. We thus configure the workload with the total number of workers allocated simultaneously no more than the limit of CPU cores (e.g., 18) to ensure all applications start their executions once they are submitted.

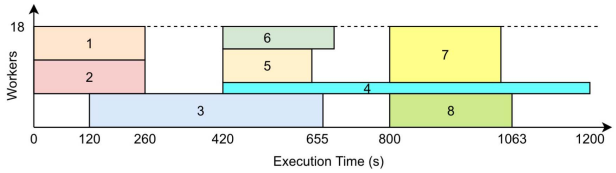

Fig. 8. Run-time events for the dynamic workload without worker reallocation. The makespan is 1,200 seconds.

First, we run an experiment on the dynamic workload where worker reallocation is disabled in WAA during the run-time, i.e., each application is exactly allocated with the required workers when they arrive (see "workers" column in Table III). We use the result of this experiment as the baseline to evaluate the dynamic worker reallocation module

TABLE III
DYNAMIC WORKLOAD SUMMARY.

Index	Model	FLOPs	Model Size	Batch Size	Iterations	Workers	Arrival Time
app1	mdResnet20	0.5 B	3.4 M	128	1000	6	0
app2	smResnet22	0.2 B	1.2 M	128	1000	6	0
app3	Resnet32	3.4 B	21.5 M	128	1500	6	120
app4	VGG13	11 B	9.9 M	64	1500	2	420
app5	mdResnet12	0.3 B	1.8 M	128	500	6	420
app6	smResnet32	0.2 B	1.4 M	128	500	4	420
app7	Resnet28	3.0 B	19 M	64	1000	6	800
app8	smVGG7	1.4 B	2.4 M	64	1500	4	800

in A-Dloader. Fig. 8 shows the run-time events for worker allocations in this experiment. Each rectangle represents the execution of an application, where the height indicates the worker number allocated to the application, and the length indicates the duration of the application. We also mark important timestamps on the x-axis across the execution time. The y-axis indicates the worker allocation among running applications. Without worker reallocation, intervals with idle workers can be frequently observed during the execution time. For example, only 2 workers are in use to run *app4* even after other applications finish and release their workers, e.g., at the time of 655 and 1063 seconds. This inevitably under-utilizes CPU and I/O resources and incurs an extremely long execution time of *app4*. Consequently, the makespan (i.e., the end-to-end execution time of the whole workload) is increased as well, e.g., $1,200$ seconds in Fig. 8. Then, we run another

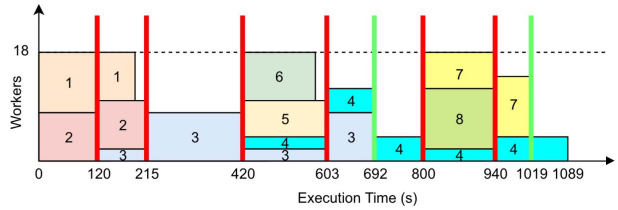

Fig. 9. Run-time events for the dynamic workload with worker reallocation. The makespan is 1,089 seconds.

experiment on the same dynamic workload but enable the workload reallocation mechanism in WAA. The corresponding run-time events of worker allocation are shown in Fig. 9. Each red line represents a worker reallocation triggered by the arrival or the completion of an application. In addition, we use a green line to mark the moment that the arrival or the completion of an application does not trigger the pause and retrieve procedure. Compared to the baseline in Fig. 8, WAA's worker reallocation mechanism significantly reduces the number of idle workers and thus improves data loading parallelism and resource utilization during the execution.

For example, all 18 workers (instead of 12 in Fig. 8) are allocated to two applications (i.e., 10 workers to *app1* and 8 workers to *app2*) at the beginning and then reallocate to the three applications inversely proportionally to their model sizes when *app3* arrives. As a result, the worker utilization is maximized and the latency of *app1* and *app2* drops from 260 seconds in the baseline to 210 seconds in A-Dloader. when three new applications (i.e., *app4*, *app5*, and *app6*) arrive at 420 seconds, A-Dloader reallocate all 18 workers to the four applications, where *app5* and *app6* that train smaller models receive more workers, i.e., six and eight workers, respectively. In this way, *app5* and *app6* can finish faster and then release workers for reallocation.

Another example, when *app8* finishes at time 940 seconds, WAA reallocates some of the released workers to both *app4* and *app7*, which thus shortens the latency of these two applications. Consequently, the makespan (i.e., 1,089 seconds) of A-Dloader drops by around 10%, compared with the baseline (i.e., 1,200 seconds). Moreover, we found that WAA does not trigger worker reallocation when *app3* and *app8* finish. This is because WAA detects that *app4* has already received enough workers (i.e., 4) to achieve the maximum throughput. As discussed before, this design can avoid the extra overhead for pausing and retrieving applications.

D. Impact of Resource Contentions.

As discussed in Sec. III-B, the contention of I/O and CPU resources happens when the number of workers increases to some saturating point. our Throughput Prediction Model (TPM) thus strives to capture the impact of contentions on the data-loading rate in the model. To evaluate the accuracy of our TPM under contentions, we conduct an experiment on a simple workload that has two applications (i.e., Resnet32 and VGG16) under A-Dloader with and without considering CPU and I/O contentions in the model prediction. Specifically, when our TPM captures the contention impact in the data loading rate, A-Dloader allocates eight and ten workers to Resnet32 and VGG16, respectively and obtains the overall training throughput as $1,654$ images/sec. Whereas, when the TPM ignores the contention in the model, Resnet32 gets six workers and VGG16 gets four workers. The overall throughput is reduced to $1,391$ images/sec. The reason is that due to resource contention, applications need more workers to achieve the maximum throughput. Our TPM captures this impact and thus predicts the accurate number of workers allocated to these applications. We finally remark that it is more critical to consider the contention impact in the prediction model when the workload has more applications with larger models.

V. DISCUSSION

Method Generality. To generalize our method to a new GPU cluster, the user only needs extra efforts to run DDL applications in hundreds of iterations to collect samples and train the throughput prediction model. The characteristics of

heterogeneous devices (i.e., GPUs and CPUs) and environments (i.e., CUDA version and OS version) are captured by our throughput formulation.

Scalability. Our method targets to optimize training throughput by reducing GPU idleness. Although we only evaluate A-Dloader on a multi-device node, A-Dloader can also be extended to multi-node clusters. We notice that network communication may cause extra latency for transferring data among nodes in a multi-node cluster.

Model Diversity. Our work currently focuses on the training of convolutional neural networks (CNNs) because CNNs are the commonly used DNNs in the deep learning domain. We notice that other types of DNNs, such as the multilayer perceptron (MLP) and Transformer, have different features. The extension of our solutions to different types of DNNs will be our new research direction in the future.

Fairness Issue. The target of this work is to maximize the overall throughput system-widely. The worker allocation algorithm proposed in this work prioritizes applications with smaller DNN models. It is thus possible that applications with relatively larger DNNs models are sacrificed with increasing latency. In addition, DDL applications with similar resource requirements can hardly benefit from A-Dloader. A sophisticated algorithm can be further designed on top of our A-Dloader to address the fairness and starvation.

VI. Related Work

A surge of research efforts has been devoted to accelerating the training process of DNNs by optimizing data-loading. In [16], several optimizations are adopted to alleviate the bottleneck of data loading, including using multiprocessing to add more workers to overlap the loading of different batches and utilizing multi-threading to parallelize sample preprocessing. The work [17] proposes DeepIO, an entropy-aware I/O pipelining framework, which utilizes a temporal in-memory storage system to avoid redundant data loading from the backend storage system. DeepIO designs a storage buffer supported by RDMA to efficiently load training data inter-computation nodes. Works in [16] and [18] focus on storage stack optimization for DDL training via cache and pre-fetch, but lack the analysis and optimization of preprocessing that becomes the bottleneck in modern DNN training.

In [5], NVIDIA Data Loading Library (DALI) is applied to optimize the data-loading by migrating data-preprocessing to GPUs. DALI attempts to share tasks of preprocessing data between CPUs and GPUs for releasing the CPU intensity of data-loading. The prior work [19] designs a DS-Analyzer to analyze and mitigate data stall in DNN training. The DS-Analyzer measures the latency of each stage of data-loading to predict the latency of data-loading. However, these existing works are proposed to optimize a single application without considering resource contentions and potential resource competition in multi-application scenarios.

VII. Conclusion and Future Work

We present a throughput prediction model for DDL applications with consideration of CPU and I/O contentions. We propose a lightweight training data-loader allocation mechanism (A-Dloader) to dynamically reallocate data-loading workers at run-time, which therefore reduces GPU idleness. We evaluate A-Dloader on a real GPU server by constructing experiments on static and dynamic workloads. The experimental results show that our design can achieve 23.5% throughput improvement compared to intuitive worker allocation mechanisms. In the future, we will extend our approach to other types of DNN models and multi-node GPU clusters.

References

[1] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[2] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 776–789.

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

[4] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," *IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019.

[5] M. Zolnouri, X. Li, and V. P. Nia, "Importance of data loading pipeline in training deep neural networks," *arXiv preprint arXiv:2005.02130*, 2020.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[8] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, pp. 541–551, 1989.

[9] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," in *International Conference on Machine Learning*. PMLR, 2020.

[10] "THOP: PyTorch-OpCounter, https://pypi.org/project/thop/."

[11] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[12] "NCCL: NVIDIA Collective Communications Library,."

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011.

[16] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019.

[17] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-aware i/o pipelining for large-scale deep learning on hpc systems," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018.

[18] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018.

[19] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," *arXiv preprint arXiv:2007.06775*, 2020.