EasyScale: Elastic Training with Consistent Accuracy and Improved Utilization on GPUs

Mingzhen Li
Beihang University
Beijing, China
lmzhhh@buaa.edu.cn

Wencong Xiao
Unaffiliated
Hangzhou, China
xiaowencong@gmail.com

Hailong Yang
Beihang University
Beijing, China
hailong.yang@buaa.edu.cn

Biao Sun
Beihang University
Beijing, China
biaosun@buaa.edu.cn

Hanyu Zhao
Unaffiliated
Hangzhou, China
zhaohanyu1994@gmail.com

Shiru Ren
Unaffiliated
Beijing, China
renshiru2000@gmail.com

Zhongzhi Luan
Beihang University
Beijing, China
07680@buaa.edu.cn

Xianyan Jia
Unaffiliated
Hangzhou, China
jiaxianyan@gmail.com

Yi Liu
Beihang University
Beijing, China
yi.liu@buaa.edu.cn

Yong Li
Unaffiliated
Beijing, China
relianceslee@gmail.com

Wei Lin
Unaffiliated
Hangzhou, China
ustcwlin@hotmail.com

Depei Qian
Beihang University
Beijing, China
depeiq@buaa.edu.cn

ABSTRACT

Distributed synchronized GPU training is commonly used for deep learning. The resource constraint of using a fixed number of GPUs makes large-scale training jobs suffer from long queuing time for resource allocation, and lowers the cluster utilization. Adapting to resource elasticity can alleviate this but often introduces inconsistent model accuracy, due to lacking of capability to decouple model training procedure from resource allocation. We propose EasyScale, an elastic training system that achieves consistent model accuracy under resource elasticity for both homogeneous and heterogeneous GPUs. EasyScale preserves the data-parallel training behaviors strictly, traces the consistency-relevant factors carefully, utilizes the deep learning characteristics for EasyScaleThread abstraction and fast context-switching. To utilize heterogeneous cluster, EasyScale dynamically assigns workers based on the intra-/inter-job schedulers, minimizing load imbalance and maximizing aggregated job throughput. Deployed in an online serving cluster, EasyScale powers the training jobs to utilize idle GPUs opportunistically, improving overall cluster utilization by 62.1%.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607054

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Distributed computing methodologies**.

ACM Reference Format:
Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, Wei Lin, and Depei Qian. 2023. EasyScale: Elastic Training with Consistent Accuracy and Improved Utilization on GPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607054

1 INTRODUCTION

Deep learning (DL) is now playing a vital role in supporting a wide range of indispensable applications, such as advertising for online shopping, computer vision for autonomous driving, natural language processing for searching, etc. Recognizing the promising power of DL, large companies have built large-scale shared GPU clusters to expedite the adoption of DL in almost every production scenario. The common practice today often adopts distributed deep learning training (DLT), where each worker typically processes training data in mini-batches and uses synchronized stochastic gradient descent (Sync-SGD) to compute gradients for model update. For example, PyTorch [45] typically allocates a GPU per worker and uses Distributed Data Parallel (DDP) to perform gradient synchronization across mini-batches. However, a DLT job will not start until all resources become available simultaneously due to the gang-scheduling [31, 60]. Besides, the DLT job is executed in a fixed degree of parallelism (DoP), thus can never scale in/out when fluctuating GPU resources become available due to cluster load

change. The fixed DoP prevents the DLT jobs from adapting to the resource elasticity that is common in shared GPU clusters [56, 57].

Recently, a series of researches (e.g., TorchElastic [9], ElasticDL [4], VirtualFlow [44], Pollux [49]) have proposed elastic training frameworks that allow a DLT job to continue its training procedure under resource elasticity. Several cluster management approaches (e.g., Gandiva [59], Optimus [46], KungFu [40]) have also utilized the resource elasticity to maximize cluster utilization or allocate job resources for fast training convergence. Despite the well-known benefit of elasticity, the proposed elastic training frameworks have rarely been used in the industry so far. Because they changes the hyper-parameters and the training procedure according to available resources, and thus introduces the non-determinism during training and inevitably affects model accuracy, e.g., overall accuracy and per-class accuracy. The fundamental obstacle for their adoption is the *inconsistent model accuracy* when training with different resources (§2), which is problematic and may destroy the model usability under elastic training, so that it prevents DL practitioners from embracing elastic training.

When training DL models, the DL developers usually go through two separate stages. *1)* Model designing: the model architecture together with hyper-parameters (e.g., learning-rate / batch-size / optimizer) are determined by developers. *2)* Model training: the model is executed to fit the training dataset over epochs. Given the same output of model designing stage (a.k.a., a certain training job), the training results can be reproduced through model training with the same number of fixed GPUs. However, prior works [4, 9, 44, 49] require DL developers to partially delegate model designing stage to the elastic training frameworks (e.g., allow Pollux to tune learning-rate adaptively and allow TorchElastic to tune batch-size proportionally), which changes the hyper-parameters and the training procedure explicitly or implicitly.

To overcome the issues mentioned above, we propose EasyScale (§3), the first elastic training system that achieves consistent model accuracy under resource elasticity for both homogeneous and heterogeneous GPU resources, thereby improving the overall cluster efficiency by utilizing the idle GPUs at best effort for elastic model training. Compared to other works, EasyScale has two distinct features: *1)* it faithfully preserves all DL developer's intentions in model designing stage, while benefiting DL training jobs with resource elasticity in model training stage; *2)* it avoids introducing extra non-determinism from resource elasticity (for number of GPUs and heterogeneity in GPU types). The goal of EasyScale is to erase the non-determinism in the model training stage of DL training and ensure the accuracy of elastic training consistent with fixed DoP training.

During DL model training, EasyScale *treats rigorous determinism and reproducibility as the first-class goal*. EasyScale explores the possibilities of producing *bitwise-consistent* model regardless of the number and type of GPU resources allocated. The fundamental idea of EasyScale is to decouple the distributed model training procedure from hardware resource allocation. This is done by an abstraction called *EasyScaleThread* (EST in short, §3.2), which encapsulates all the training stages such as data loading, sampling, computation, and communication. The EST abstraction enables the training behaviors under resource elasticity are exactly the same as executed under fixed resources. To minimize the abstraction overhead, EasyScale utilizes the DL characteristics for fast context switching across ESTs, efficient on-demand checkpointing of EST states when the resource scales, and optimized data loading worker sharing along the EST execution. To eliminate the non-determinism in resource elasticity and heterogeneity, EasyScale sources the root causes scattered across the software stack of DL training, and then controls them through embedding the implicit states in EST contexts / checkpoints and fixing others (§3.3). In addition, EasyScale introduces both *intra-job* and *inter-job* schedulers regarding the EST abstraction, to improve the utilization of GPU resources and the aggregated throughput of the entire cluster (§3.4).

EasyScale is implemented by modifying a popular framework, PyTorch, to provide the elastic training capability without compromising model accuracy (§4). In addition, the scheduling policies of EasyScale are implemented on top of Kubernetes scheduler. We evaluate EasyScale on a cluster with 64 heterogeneous GPUs to demonstrate its effectiveness in accuracy-consistent model training using micro-benchmarks on typical workloads (§5.1). We also show the advantage of EasyScale under resource elasticity with production workload trace (§5.2). The trace experiment shows that EasyScale can generate consistent model accuracy for all DLT jobs compared to those using a specific number of GPUs. In addition, EasyScale improves the average job completion time (JCT) by 13.2× and makespan by 2.8× thanks to its intra-job and inter-job schedulers. We have deployed EasyScale in a production cluster equipped with 3,000+ heterogeneous GPUs for online model serving (§5.3). The evaluation result demonstrates that EasyScale improves the GPU allocation ratio by 17.1% and the average GPU utilization by 62.1%. The evaluation result also shows that using EasyScale, the DLT jobs can automatically scale in seconds when co-located with online model serving jobs.

Specifically, the key contributions are as follows:

- We propose the EasyScale framework for elastic distributed model training to achieve consistent model accuracy. It utilizes EST abstraction to preserve consistent training behaviors as PyTorch DDP, and achieves efficient context-switching under resource elasticity.
- We investigate the non-deterministic behaviors of model training in existing elastic training frameworks, and identify factors scattered across the entire DLT software stack that affect the bitwise accuracy of model training.
- We propose the EasyScale scheduler, including intra-job and inter-job schedulers to improve the utilization of heterogeneous GPU resources of the entire cluster, regarding the EST abstraction.
- We deploy EasyScale in production clusters to co-locate elastic training jobs with online model serving jobs. The evaluation results show that EasyScale significantly improves cluster utilization.

2 MOTIVATION

In this section, we first briefly describe the increasing demand for adopting elastic resources when training DL models on large-scale shared GPU clusters. We then analyze the non-determinism of existing elastic training frameworks to motivate EasyScale.

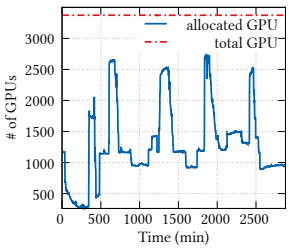


Figure 1: Online serving GPU cluster load variation.

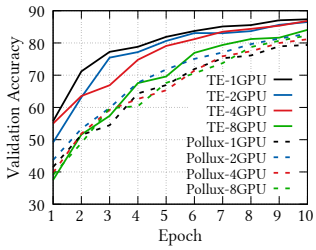


Figure 2: Non-deterministic accuracy curves of ResNet18.

		C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7	C 8	C 9	Total
TorchElastic	1GPU	92.5	96.7	87.1	85.2	93.7	85.7	95.8	95.3	95.3	93.9	92.1
	2GPU	89.1	96.2	91.3	82.5	94.4	82.9	95.3	93.4	93	96.7	91.5
	4GPU	89	93.4	89.3	84.7	94.7	86.8	94.7	92.7	95.6	97.4	91.8
	8GPU	90.8	95.5	85.7	81.6	92.6	90.3	93.3	97.1	95	95.3	91.7
	Δ	3.5	3.3	5.6	3.6	2.1	**7.4**	2.5	4.4	2.6	3.5	**0.6**
Pollux	1GPU	93.9	96.7	87.6	86.1	95	85.7	92.5	94.8	96.7	93.9	92.3
	2GPU	90.2	95.9	87	77.8	95.5	80.1	94.4	96.9	96.2	93.2	90.7
	4GPU	92.6	97.6	75.6	81.7	84.8	89.7	94.9	91.4	96.7	92.3	89.7
	8GPU	91.7	98.3	92.9	84	79	82.7	91.7	93	96.1	85	89.4
	Δ	3.7	2.4	**17.3**	8.3	16.5	9.6	3.2	5.5	0.6	8.9	**2.8**

Figure 3: Non-deterministic per-class accuracy of ResNet18 on CIFAR10 at epoch 100.

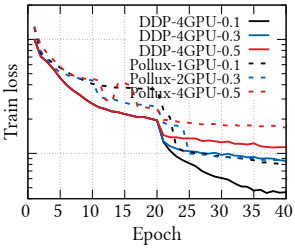


Figure 4: Train loss of ResNet50 with different hyper-parameter gamma.

2.1 Demand for Adapting to Resource Elasticity

Distributed training is widely adopted in production clusters for processing massive data and large models. However, it usually achieves sub-optimal performance on shared clusters for the following reasons. For one reason, large-scale DLT jobs commonly suffer from long queuing time for resource allocation due to gang-scheduling policy [31, 32, 57, 60, 65]. For the other reason, the training can be easily interrupted due to constant resource revocation by high-priority jobs. Our 2-day statistic in a GPU production cluster of CompanyA shows that jobs requesting more than 8 GPUs account for 61.7% of training failures due to resource revocation. Whereas the jobs requesting one GPU only account for 5.3% of training failures due to resource revocation. This discrepancy can be attributed to the inherent Sync-SGD training behavior, where terminating any worker stops the entire model training.

To avoid frequent failure when training DL models on large-scale shared cluster, it is important to establish the ability to adapt the training procedure to resource elasticity (a.k.a., elastic training). Another benefit of elastic training is that a multi-GPU job can start the training immediately with the required number of GPUs gradually allocated, and thus eliminate the long queuing time due to gang scheduling. Moreover, the elastic training also reveals more opportunities to utilize the idle resources of the online model serving cluster for training DLT jobs. Our 2-day GPU allocation statistics of an online model serving cluster (Figure 1) show that the difference of required number of GPUs between idle and peak hours can be up to 2,000. Ideally, the idle GPUs can be shared by both model training and online serving to improve GPU utilization, similar to the big-data workloads [56]. Exploiting such an opportunity also demands elastic training to meet the SLAs of online serving jobs.

2.2 Non-determinism over Elastic Training

To address the drawbacks described above, a series of research works [4, 9, 44, 49] have proposed elastic training frameworks that enable adapting to dynamically scaling resources at runtime. So that training jobs can start executing with available resources as soon as possible, thus eliminating the mandatory queuing time and avoiding the frequent failure, which improves cluster utilization and reduces job completion time. Existing elastic training frameworks usually adopt optimized synchronization methods, such as gradient accumulation [44, 49]), hyper-parameter tuning [40], and batch size adjustment [9]), to eventually reach similar model accuracy compared to the training with static resources. However, they still have rarely been used in industry due to the following drawbacks.

Inconsistent Model Accuracy – The multiple runs of model training with elastic training frameworks fail to generate consistent model accuracy when using different amounts of resources. Figure 2 illustrates the validation accuracy of training ResNet18 model on CIFAR10 dataset, with varying numbers of V100 GPUs. We keep all hyper-parameters and random seeds as default except for using different allocated GPUs. TorchElastic (TE) [9] is configured with linear scaling rule for adjusting learning rates [24], and Pollux [49] can automatically decide the learning rate and batch size accordingly. It is clear that resource elasticity leads to different training behaviors compared to model training on a fixed number of GPUs. The result also shows that Pollux introduces less accuracy variance compared to TorchElastic, however the difference is still non-negligible (e.g., up to 5.8% at epoch 10).

To better understand the inconsistent model accuracy with elastic training, we extend the training using TorchElastic and Pollux to 100 epochs, and report the overall and per-class (10 classes in total) accuracy in Figure 3. The overall accuracy variance for TorchElastic and Pollux is still notable, with 0.6% and 2.8%, respectively. Note that even the latest elastic training framework VirtualFlow [44] also suffers from 0.4% accuracy degradation on ResNet50 according to its experiments. However, we cannot provide a direct comparison with VirtualFlow since its implementation is not publicly available. The per-class accuracy variance for TorchElastic and Pollux is even larger, reaching up to 7.4% and 17.3% maximally, whereas 3.9% and 7.4% on average. The per-class accuracy variance can be detrimental to the model usability, in scenarios such as life-critical pedestrian detection for self-driving cars [23] and profit-critical recommendation/advertising systems [64]. Moreover, the upper bound of the variance/inconsistency for model accuracy remains unknown, which further increases the hesitation of DL practitioners in adopting elastic training.

Difficult to Understand Hyper-parameter Effect – Model developers conduct model training to seek better hyper-parameters and model structure, and reason their effectiveness through deterministic reproducibility on fixed GPUs. Figure 4 shows the experiment of ResNet50 on CIFAR10 by comparing the elastic training on 1/2/4 GPUs using Pollux [49] to the non-elastic training on fixed 4 GPUs using PyTorch DDP. The configurations remain the same except the hyper-parameter of learning rate (gamma), which decides the learning rate decay factor after certain training epochs (e.g., 20

epochs in this experiment). The PyTorch DDP is conducted on 4 GPUs with the gamma value set to 0.1, 0.3, and 0.5 for each run. The Pollux is conducted on 1/2/4 GPUs with gamma of 0.1/0.3/0.5, respectively. As illustrated in Figure 4, when using PyTorch DDP, it is clear to identify the trend of how the hyper-parameter gamma affects the training loss during training. However, when training with different numbers of GPUs, the loss curves of Pollux have many unexpected oscillations and thus reveal no clear trend for DL developers, which invalidates the existing knowledge of hyper-parameter tuning, and thus hinders the productivity of model designing stage.

To summarize, we believe that the non-determinism over elastic training leads to inconsistent model accuracy and complicates the hyper-parameter tuning. The fundamental reason can be attributed to that, the existing elastic training frameworks [3, 4, 9, 40, 44, 49] lack the capability to decouple the resource allocation from the model training procedure (§3.3), and thus fail to provide consistent model accuracy during resource elasticity. In order to effectively utilize the elastic resources on shared GPU clusters without compromising the model accuracy, we argue that a new elastic training framework should be proposed to address the non-determinism across the DLT software stack for achieving consistent model accuracy. The new elastic training framework should also provide further opportunities to improve the throughput of individual DLT jobs as well as the utilization of the entire GPU cluster.

3 DESIGN

3.1 Overview

The design principle of EasyScale has been inspired by the big data analysis systems [16, 18, 30, 36, 61], which guarantee the consistent output regardless of the allocated resources. Similarly, the DL frameworks that train neural network models by analyzing data samples, can be viewed as specialized data analysis systems for artificial intelligence, should also produce consistent model accuracy regardless of allocated resources. Previous elastic training frameworks fail to maintain the consistent model accuracy due to the changed behaviors of training procedure upon elastic GPU allocation.

Different from existing approaches, we think elastic training should generate *bitwise identical* model parameters compared to the non-elastic DDP training over a fixed number of GPUs. As a result, the model accuracy is also identical and consistent. Figure 5 shows an example of elastic training by scaling in from four GPUs (5(a)) to two GPUs (5(b)). To preserve the training behaviors, ideally we would like the four training workers to be executed in parallel on two GPUs. However, by multiplexing multiple training workers on a GPU, the concurrent memory usage increases in the forward pass, which can easily lead to either out-of-memory (OOM) exceptions [35] or significant overhead in memory swapping [12, 60]. Besides, the aggregated memory usage of CUDA contexts (including that of training framework and CUDA itself) is also considerable. For example, 16 workers on a 16GB V100 GPU costs 12GB GPU memory for CUDA contexts (around 750MB per context).

As illustrated in Figure 5(c), the key challenges in achieving accuracy-consistent elastic training is to preserve the training behaviors (e.g., number of workers) as well as sharing GPU resources efficiently. To address the above challenges in the design of EasyScale framework, we introduce the abstraction of EasyScaleThread (EST) to decouple resource allocation from the training procedure (§3.2). We further split the states of a EST into a stateful context and a stateless part, and minimize the size of the context that needs to be saved and optimize the context switching overhead. Then, we identify the sources of non-determinism across the training software stack and present our approach to eliminate non-determinism (§3.3). We also propose EasyScale scheduler to better utilize homogeneous / heterogeneous GPUs regarding ESTs to improve cluster utilization (§3.4).

3.2 EasyScaleThread

We introduce EasyScaleThread (EST) as a key abstraction in EasyScale. EST is inspired by the classical *thread* concept in operating systems and the "single program multiple data" (SPMD) model adopted commonly in DL [13], which can separate the training procedure from underlying resource allocation, and is flexible to enable resource sharing through context switching. As shown in Figure 5(c), each EasyScale worker is launched on one GPU. The execution of original DDP training workers (e.g., PyTorch workers) is treated as that of ESTs, and any EST (i.e., a thread) can be dynamically allocated to a EasyScale worker (i.e., a process) during training. In a EasyScale worker, multiple ESTs take turns to occupy the GPU for training (e.g., model forward and backward passes) in the time-slicing manner. EasyScale hooks the key steps of model training, such as data loading, model backward, and model updating through users' annotations, therefore ESTs can perform efficient context switching at mini-batch boundaries. The user-defined model training semantics, including model structure, data augmentation, batch size, etc., remain as usual. Under the EST abstraction, users only need to consider the number of logical training workers for tuning the hyper-parameters (e.g., batch size and learning rate), which is the same as their experiences of using DDP on a fixed number of GPUs. However, with EST they can benefit from elastic training automatically without concerns about inconsistent accuracy.

The execution of EST is similar to that of the worker of DDP training. For each training step, training samples are processed by conducting forward-backward computation over the current model to generate gradients as output. After gradient synchronization, the model is updated. Figure 6 illustrates the case when training with four ESTs, the available resource scales from two GPUs to one GPU. To enable efficient sharing among ESTs, each EasyScale worker maintains one one CUDA context to share among ESTs, thus it does not consume multiple times of GPU memory. For each global step of data-parallel training, the input data is split across all ESTs. During runtime, each EasyScale worker schedules one EST at a time, executes it in the EasyScale worker by occupying one GPU (i.e., a local step), and all belonging ESTs gets executed in the time-slicing manner. The global step is completed after all ESTs finish execution of the local steps and all produced gradients are aggregated to update the model parameters.

Context switching – When context switching between ESTs, the training states of the ESTs need to be saved and swapped out from GPU to CPU in order to avoid over-subscribing GPU memory, which can be costly to deteriorate training throughput. The key to enabling lightweight context switching is to reduce the states to be saved. EasyScale leverages the unique characteristics of DL jobs to

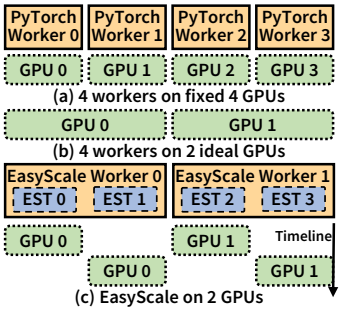


Figure 5: Idea of EasyScale.

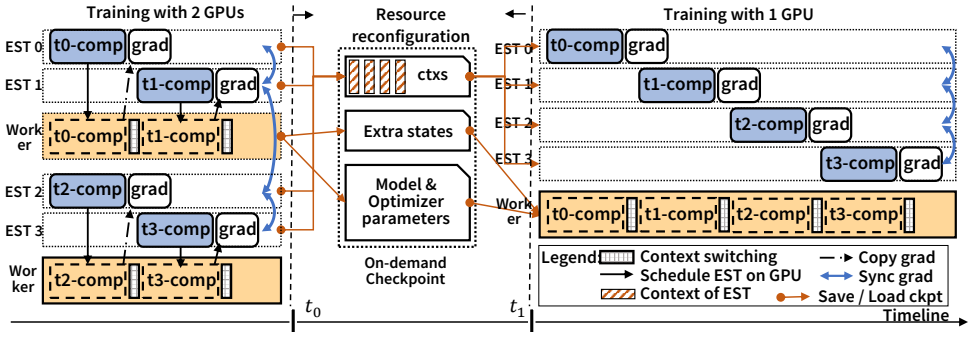


Figure 6: Execution flow of EasyScaleThread (EST).

allow most states to be shared and reused among different ESTs. In addition, EasyScale chooses to perform context switching of ESTs after the forward-backward computation at the boundaries of each mini-batch, which further reduces GPU-CPU data traffic.

Specifically, we propose the following two approaches to reduce the states to be saved during context switching: *1)* locating the sources of non-determinism that affect the model accuracy and record only the necessary states (e.g., the states of random number generators (RNGs)), and *2)* leveraging the data-parallel behavior of DDP training to minimize the working set for data swapping. The working set resided in GPU memory of an EST can be classified into three categories, including temporal tensors and activations, model parameters and optimization states, and gradients [44]. During context switching, we handle each category differently to reduce the working set to be swapped to CPU side effectively. Firstly, temporal tensors and activations are created in the forward pass and destroyed in backward pass after the gradient generation [59, 60]. Their working set is automatically freed up at the end of mini-batches, which do not need to be swapped to CPU. Therefore, we constraint the minimal time slice of an EST's local step to one mini-batch. Secondly, for the model parameters and optimizer states, a replica is maintained by each EasyScale worker during training, and only updated at the end of global steps. Therefore, they remain the same for all ESTs till all ESTs are finished, thus they can be reused among ESTs, with no need to be swapped to CPU. Finally, the gradients are calculated based on the different input data across ESTs, and cannot be shared nor reused. Therefore, only the gradients need to be swapped to CPU during EST context switching. Fortunately, the gradients are only used in distributed gradient synchronization at the end of a global step. To mitigate the cost of saving gradient working set, we overlap the gradient swapping with the backward computation of current EST and the forward computation of next EST to be switched in. In such a way, each EasyScale worker executes the ESTs alternately until all ESTs finish local steps. After that, the distributed gradient synchronization is triggered with model parameters updated.

Adapting to elasticity – When available resource (e.g., number of GPUs) of a training job changes (a.k.a., resource elasticity), EasyScale adopts on-demand checkpointing to preserve necessary states, as shown in Figure 6. The checkpoint contains the contexts of all ESTs, the extra states (including the training progress and other states for achieving accuracy-consistency illustrated in §3.3), and parameters (e.g., model, optimizer, and learning rate scheduler). Unlike the EST contexts, only one replica of the extra states and parameters is required, because they can be shared across ESTs within each global training step. Note that after continuing the model training on the reconfigured resources, EasyScale manages each worker loads a copy of extra states and parameters, as well as the corresponding contexts of re-distributed ESTs, so that all ESTs can resume from the last saved states.

Optimizing data pre-processing – Since the data pre-processing is becoming more resource-intensive, existing DL frameworks such as PyTorch commonly use standalone data workers on CPUs to accelerate the model training on GPUs. Specifically, the data workers run asynchronously to the training workers for data supply, load samples and perform data augmentations to build training batches. To further optimize the training efficiency with ESTs, we need to consider the data supply along with EST execution. The number of data workers is usually configured for each training worker by users to ensure the GPU training is not being blocked by data supply. Naively scaling the number of data workers regarding the number of ESTs can easily lead to massive CPU processes, thus overwhelming the training system. For example, if the users configure 8 data workers per training worker, for a case with 16 ESTs sharing a GPU, the total data workers on one machine can be 128. In EasyScale, we can avoid the above problem by sharing data workers among all ESTs, because only one EST is executing within a EasyScale worker at any time. Therefore, the data consuming rate is similar to that of executing one training worker.

To enable data worker sharing, EasyScale employs a distributed data sampler that jointly considers the global indices of ESTs and the time-slicing manner to generate data indices in a queue. The data indices are then processed by data workers in order. Figure 7 shows the case of sharing three data workers between two ESTs, where the total number of ESTs is four (i.e., training with two GPUs shown in Figure 6). The training batches of *EST*0 and *EST*1 are *b0* and *b1* for mini-batch 0, and *b4* and *b5* for mini-batch 1. The state of data worker *j* to process data indices for EST *i* on a dedicated GPU is denoted as *Ri-j*, shown in Figure 7. Note that due to the asynchronous execution of data workers, the progress of data workers (e.g., mini-batch index) is usually ahead of the training progress. To maintain the consistent state for elastic training, a queuing buffer is introduced to record the necessary states (e.g., the state of RNG) for mini-batches that ESTs do not consume. To

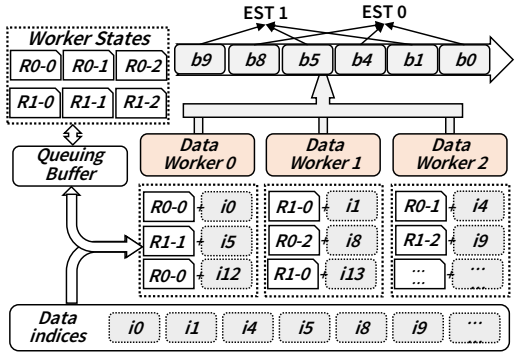


Figure 7: Data worker sharing.

balance the load, data workers in EasyScale take turns to obtain the corresponding state (e.g., *Ri-j*) of given data indices from the queuing buffer for pre-processing, and the state is committed back to the queuing buffer once pre-processing is finished. Since the worker states are dequeued from the queuing buffer according to training progress, they are categorized as an extra state during the on-demand checkpointing.

3.3 Eliminating Non-determinism

While ESTs can decouple the resource allocation from the training procedure and facilitate efficient resource sharing through EST context switching, using ESTs still may result in inconsistent accuracy compared to the DDP training. Besides, utilizing elastic training to make use of heterogeneous GPUs can also introduce non-trivial, non-deterministic, and previously unstudied behaviors. To tackle non-determinism during training, we use a top-down approach comparing the tensors of EasyScale and DDP. We conduct experiments using the same number of workers but with different configurations to identify the factors that impact training accuracy in bitwise. Surprisingly, we find that the root causes are scattered across almost the entire software stack used for training.

For the *implicit framework state*, existing training frameworks commonly maintain several implicit states, which must be consistent throughout the training for determinism. Although they organize operators (e.g., convolution, batch normalization) in a computation graph [10], several operators implicitly rely on additional states beyond their predecessors' outputs. For instance, the `Dropout` operator depends on the random number generator (RNG) states, whereas the `BatchNorm` operator tracks its running states. In addition, the data loader and data augmentation also depend on RNG states from Python, NumPy, PyTorch, etc.

For the *communication mechanism*, the gradient synchronization via all-reduce used in DDP is non-deterministic when resource elasticity is involved. During synchronization, the gradients are gathered into gradient buckets to achieve higher throughput and lower latency [34]. The mapping of gradients to buckets firstly follows a static reversed topological order of the computation graph, and then is reconstructed at the end of the first mini-batch based on the order of when gradient tensors are derived. However, when resource scales in/out, the training workers will restart and rebuild the communication channels, which changes the mapping and eventually disrupts the gradient aggregation order, leading to non-determinism together with the all-reduce implementation [5, 6].

For the *Operator implementation*, existing training frameworks may select different implementations for the same operator during training, which can result in subtle differences in training accuracy. There are two reasons why different operator implementations are selected. Firstly, profiling-based optimizations adopted by frameworks [8], compilers [55], or vendor libraries [1] can apply various kernel implementations on GPUs to optimize operator performance based on profiling results across mini-batches. Secondly, the kernel implementation may be hardware-specific, such as implementations designed for a specific number of SM units and low-bit components, which makes it unsuitable for all types of GPUs.

Solutions for different levels of determinism – To address the non-determinism across the software stack, EasyScale defines different levels of determinism for elastic training, and applies solutions to guarantee consistent accuracy for each level.

D0: Static determinism – Multiple training runs with a fixed number of GPUs should always result in identical model accuracy. To achieve *D0*, consistent framework states and operator implementations are required. As for consistent framework states, we fix the random seeds of RNGs at the beginning of training and record the RNG states of both the data workers and ESTs in the extra states and EST contexts of the on-demand checkpoint. As for operator implementations, we disable profiling-based optimizations and select deterministic kernel implementations (e.g., without atomic instructions). DL frameworks such as PyTorch and TensorFlow also recommend this approach to improve model reproducibility [6, 7].

D1: Elastic determinism – Multiple training runs with different numbers of GPUs should always result in identical model accuracy. *D1* requires necessitates resolving the non-deterministic aspects of the communication mechanism beyond *D0*. To achieve this, we assign a constant virtual communication rank to each EST and store the indices that make up the gradient buckets in the checkpoint. When resource scales in/out, the training recommences using the checkpoint and reconstructs the gradient buckets by initially reinstating recorded indices of gradient-bucket mapping. Subsequently, reconstruction of the communication channel is disabled.

D2: Heterogeneous determinism – Multiple training runs with different types of GPUs should always result in identical model accuracy. To achieve *D2*, we develop hardware-agnostic operator implementations on GPU, involving two main aspects: *1)* we modify operator implementations (e.g., `reduce`, `dropout` in PyTorch) by selecting a specific number of SMs and threads that can run on any type of GPU, and *2)* we deterministically choose the same operator implementations (e.g., `convolution` in cuDNN, and `gemm`, `gemv` in cuBLAS) by fixing the algorithm identifier (*algo_id*) in library calls.

In EasyScale, *D0* and *D1* are enabled by default due to their negligible overhead (§5.1.3). However, enabling *D2* may result in noticeable overhead for certain types of operators such as convolution, because they cannot utilize vendor-optimized kernels on GPU. To address this issue, EasyScale automatically analyzes a DL model by scanning the PyTorch `nn.Module` to identify whether it relies on hardware-specific kernel optimizations. If not, *D2* is enabled and elastic training can use heterogeneous GPUs. Otherwise, EasyScale restricts its use to homogeneous GPUs.

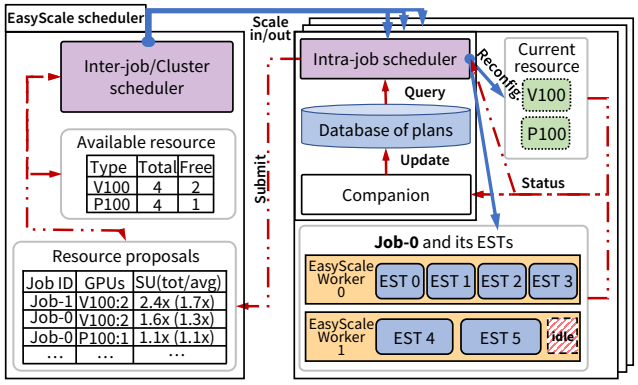


Figure 8: Workflow of the EasyScale scheduler.

EasyScale does not rewrite operators, it only selects deterministic operator implementations or specifies allowed SMs/threads for existing operators. Currently, EasyScale has already supported diverse models (CV/NLP models in Table 1), popular structures (e.g., `convolution`, `transformer`), and fundamental operators (e.g., `gemm`, `reduction`). In the future, we will extend EasyScale to support more deterministic operators, therefore less effort is needed from the users to achieve *D2* determinism. Notably, to achieve operator determinism, EasyScale is analogical to prior works (e.g., Rammer [38], REEF [26]) that accelerate training process through novel operator designs. Currently, hardware-specific kernels can be supported in EasyScale with *D1* determinism. For future work, we will allow the users to customize *D2* kernels via Cutlass to enable the exploration of more hardware features for heterogeneous GPUs.

3.4 EasyScale Scheduler

In this section, we describe the EasyScale scheduler, which is co-designed with the EasyScale framework, to improve the training job throughput and the cluster utilization, guaranteeing of consistent accuracy at the same time. It is capable of scheduling the EasyScale jobs to available GPUs of the cluster. For each job, it can schedule each job to a set of heterogeneous GPUs, utilizing the EasyScale framework's capability of decoupling training procedure from underlying resources. The EasyScale scheduler has a hierarchical architecture, as illustrated in Figure 8. The **inter-job (cluster) scheduler** is responsible for allocating resources among jobs at the cluster scale. Additionally, each training job has an **intra-job scheduler** that coordinates ESTs and currently allocated GPUs.

In EasyScale, the scheduling decision is dispatched from the inter-job (cluster) scheduler to the intra-job scheduler. Then, the intra-job schedulers of current jobs reconfigure (i.e., scale in/out) the occupied GPUs resources respecting the scheduling decision, query the database to get the best plan, and consequently reschedule the ESTs to current resources. In contrast, the scheduling decision comes from the resource proposals submitted by the intra-job schedulers, and is approved by the inter-job scheduler. Specifically, each intra-job scheduler leverages **a standalone companion module** to maintain the database of plans. And the intra-job scheduler tries to scale out with incremental homogeneous GPUs (e.g., 2 V100), thereby selecting top-K plans according to estimated throughput improvement as the resource proposals to submit to the cluster scheduler. The companion module leverages the job information reported by EasyScale framework to initialize the database and actively update the database once it has monitored significant biases between the estimated throughput and the reported throughput.

Intra-job scheduler – Its main responsibility is to generate the EST-to-GPU mapping configurations with the help of the companion module and the database of scheduling plans. There are three roles for the intra-job scheduler. *Role-1:* Under current available GPU resources, it queries the database and applies the top-1 configuration with the highest estimated throughput. *Role-2:* Supposing the job is allowed to scale out for higher training throughput, it queries the database to explore new configurations, calculates the incremental GPU number and the estimated speedup, and then forms the resource proposals submitted to the inter-job scheduler. *Role-3:* Once it receives any scheduling plan, it scales in/out the GPU resources accordingly and intermediately. Then it reschedules the ESTs to current GPUs (*Role-1*) and generates the resource proposals later (*Role-2*). Additionally, if it has observed the slowdown with incremental resources, it falls back to using previous resources and releases the newly allocated ones.

Companion module – The intra-job scheduler's companion module keeps a database of scheduling plans that consider the available GPU types and the maximum number of ESTs (*maxP*) specified by DL developers during model designing stage. Each plan includes the quantity of GPU resources, EST-to-GPU mapping configuration, and estimated throughput based on job details and performance models. When a job runs for the first time, the companion module initializes the database using historical data [57].

The companion module aims to achieve load balance among different GPUs by generating plans that allocate proper ESTs, based on the quantum property of EST allocation (i.e., integer number) and the consecutiveness of GPUs' computing capability. To estimate the throughput of EST-to-GPU mapping configurations, it uses an analytical performance model. Specifically, it introduces a new metric called *waste* to indicate the quantity of wasted computing capability due to load imbalance, which can represent two scenarios: *1)* allocated ESTs cannot fully utilize GPUs' computing capability (Equation 1b–1c), and *2)* the EST allocation is over-provisioned to satisfy the *#of total ESTs* $\geq$ *maxP* constraint (Equation 1a). The number of available GPUs is denoted as N_i, where i represents GPU type. The workload-related computing capability C_i is estimated as mini-batches per second, and the maximum number of assigned ESTs for GPU type i is denoted as A_i. Additionally, an overload factor ($f_{overload}$) represents the maximum overload for requested GPU types. If one GPU type undertakes too many ESTs, it becomes a bottleneck and slows down other GPUs due to Sync-SGD. Finally, we derive estimated throughput by subtracting *waste* from our calculations (Equation 1d).

$$nEST = \sum_i N_i \times A_i, \quad nEST \geq maxP \tag{1a}$$

$$f_{overload} = \max_{i,N_i>0} A_i/C_i \tag{1b}$$

$$waste = \sum_{i,N_i>0} (C_i - A_i/f_{overload}) + (nEST - maxP)/f_{overload} \tag{1c}$$

$$throughput = (\sum_i N_i \times C_i) - waste \tag{1d}$$

Inter-job cluster scheduler – It evaluates the submitted proposals by considering available resources and proposal priorities. To improve the aggregated job throughput and cluster utilization, it adopts a heuristic (greedy policy in practice) that tends to accept the proposals with a higher speedup per GPU. If multiple proposals offer the same speedup, it prioritizes the one with more GPUs. By synchronizing fluctuating free resources to the table of available resources, it supports co-locating EasyScale jobs with other non-elastic jobs (such as online serving jobs), making optimal use of temporarily available idle resources. Notably, the inter-job scheduler reserves flexible interfaces for users to experiment with other scheduling policies. If needing more contexts for scheduling, the intra-job scheduler can also be easily extended to report more framework/resource information to inter-job scheduler.

4 IMPLEMENTATION

The EasyScale framework is built on PyTorch 1.8 LTS [45] and requires approximately 1,200 lines of Python code and 2,000 lines of C++ modifications to PyTorch. The C++ implementation includes a distributed data-parallel communication library called `ElasticDDP`, which supports communication among multiple ESTs for all-reducing gradients and building communication buckets consistently during resource elasticity. Execution control flow and context switching are implemented as an add-on PyTorch module.

A prototype cluster scheduler of the EasyScale scheduler is implemented on Kubernetes [15] for evaluation. We implement AIMaster, which includes the intra-job scheduler and the companion module, with around 2,000 lines of Python code. AIMaster performs three main functions: collecting performance profiling reported by EasyScale runtime through an RPC library; submitting resource proposals; monitoring resource allocation timeout through a Kubernetes Python informer; and containing a policy controller to calculate and submit incremental resource requests to the cluster scheduler. We adopt on-demand checkpoint [59] to record DL model, epoch, and necessary states mentioned in §3.3 to support continuous job training when resource elasticity occurs.

The EasyScale jobs run in Docker containers with EasyScale installed within our internal GPU cluster scheduler that is optimized from Kubernetes version. Currently, we have deployed EasyScale on two internal GPU production clusters used for serving DL development (i.e., Jupyter Notebook) and model inference respectively. One deployed cluster consists of more than 10K GPUs.

5 EVALUATION

In this section, we present the evaluation of EasyScale. Firstly, we demonstrate its accuracy-consistency and efficiency through micro-benchmarks. Secondly, we evaluate it using real workloads on a small cluster with 64 GPUs to show its scheduling efficiency. Lastly, we evaluate it on a production cluster equipped with thousands of GPUs to highlight the improvement in cluster utilization.

The micro-benchmark experiments and trace experiments are conducted on a cloud GPU cluster with 16 servers, specially, 4 servers each with 8 V100, 8 servers each with 2 P100, and 4 servers each with 4 T4. Each server runs CentOS 7.8, and their GPUs are powered by Nvidia driver 450.102.04, CUDA 10.1, and CuDNN 7. As for the workloads, eight state-of-the-art DL models are selected from Github, together with open datasets, as summarized in Table 1. They are implemented based on PyTorch 1.8 LTS, and are ported to EasyScale with a few lines of code changing.

Table 1: Deep learning workloads in experiments.

Model	Task	Dataset
ShuffleNetv2 [39]	Image Classification	ImageNet [19]
ResNet50 [28]	Image Classification	ImageNet [19]
VGG19 [54]	Image Classification	ImageNet [19]
YOLOv3 [52]	Object Detection	PASCAL [21]
NeuMF [29]	Recommendation	MovieLens [27]
Bert [20]	Question Answering	SQuAD [51]
Electra [17]	Question Answering	SQuAD [51]
SwinTransformer [37]	Image Classification	ImageNet [19]

5.1 Micro-benchmarks

5.1.1 Ensuring accuracy-consistency.

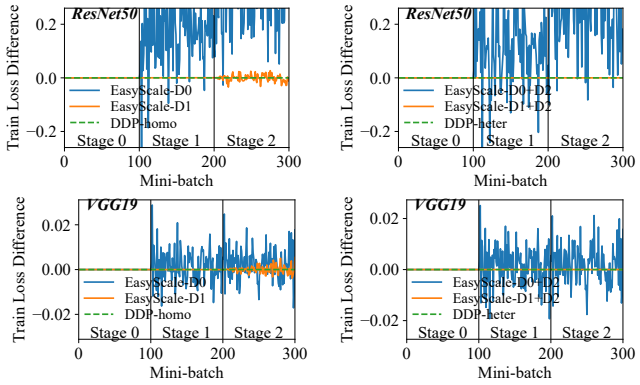
To demonstrate the accuracy-consistency of EasyScale, which guarantees to produce bitwise-identical DL models under an elastic number of heterogeneous GPUs, we use EasyScale to train the DL workloads of Table 1 in three different stages with different GPU configurations: *stage 0* with 4 V100 GPUs, *stage 1* with 2 V100 GPUs, and *stage 2* with 1 V100 and 2 P100 GPUs. Changing from from *stage 0* to *stage 1* represents the *resource elasticity*, and changing from from *stage 1* to *stage 2* represents the *resource heterogeneity*. In each stage, the workloads are trained for 100 mini-batches.

We use PyTorch DDP with 4 V100 GPUs as the baselines. Both EasyScale and DDP have 4 workers in total (i.e., 4 ESTs for EasyScale). EasyScale is configured with four determinism configurations: two homogeneous determinism configurations (*D0* and *D1*) and two heterogeneous ones (*D0+D2* and *D1+D2*). DDP has two corresponding configurations, *DDP-homo* with fixed random seeds and deterministic algorithms to ensure the reproducibility, and *DDP-heter* with additional selection of heterogeneous deterministic kernels (originally belong to *D2*). Figure 9 shows the loss curve differences of the last worker on `ResNet50` and `VGG19`. The train loss of *D1* is identical to that of *DDP-homo* in *stage 0* and *stage 1*, and the train loss of *D1+D2* is identical to that of *DDP-heter* in all stages, demonstrating how EasyScale can preserve consistent accuracy.

By comparing the curves of *D0* with *D1*, and also *D0+D2* with *D1+D2*, we can highlight the elasticity determinism of EasyScale. We have observed that both *D0* and *D0+D2* start experiencing loss differences since *stage 1* after checkpointing and restarting. This is because *D0* ignores the states of gradient-to-bucket mapping in the checkpoint, which results in losing these states after restarting. In contrast, *D1*/*D1+D2* records these states in the checkpoint and thus have identical loss curve to *DDP-homo*/*DDP-heter*.

Furthermore, by comparing *D1* with *D1+D2*, we can highlight the heterogeneous determinism. Specifically, in *D1*, loss differences begin to emerge from *stage 2*, due to automatic selection of different low-level kernel implementations on heterogeneous GPUs. However, enabling *D2* to fix the kernel selection for EasyScale and DDP eliminates loss difference in *stage 2*. The results of the other models

are similar and have been omitted due to space constraints. In summary, EasyScale with *D1+D2*, can ensure the accuracy-consistency with DDP after any number of training iterations.



(a) Under homogeneous determinism configurations (b) Under heterogeneous determinism configurations

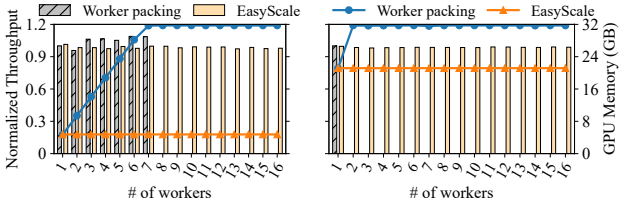
Figure 9: Loss curve difference of EasyScale and DDP.

5.1.2 *Efficient GPU resource sharing.*

To demonstrate EasyScale's ability to efficiently run multiple workers (ESTs) on a single GPU, we compared it with worker packing proposed by Gandiva [59]. Worker packing involves multiplexing the same GPU across multiple workers and is another potential method for achieving accuracy-consistency with resource elasticity. We trained two typical models, `ResNet50` and `ShuffleNetV2`, on a V100 GPU. The batch size of `ResNet50` is set to 32 as it is commonly used in benchmarks. The batch size of `ShuffleNetV2` is set to 512 to fully utilize the 32GB V100 memory using one worker, which is typically how DL researchers utilize a GPU's capability. EasyScale is configured as *EasyScale-D1*, and worker packing is implemented with *DDP-homo* to ensure the reproducibility.

We conducted 10 runs of EasyScale and worker packing with varying numbers of workers. Figure 10 shows the training throughput (batch size divided by average mini-batch time) and peak GPU memory usage. All throughput values are normalized to one worker under worker packing. As expected, when running only one worker, both methods have similar throughput and memory usage. However, as the number of workers increases, the GPU memory usage for EasyScale remains constant while worker packing experiences a gradual increase in GPU memory usage. Worker packing suffers from out-of-memory (OOM) exceptions after 8 workers for `ResNet50` and 2 workers for `ShuffleNetV2`. In contrast, EasyScale carefully reuses the DL components across ESTs such as model parameters and optimizer states while minimizing EST context. As a result, its GPU memory usage remains almost constant regardless of the number of workers. Furthermore, EasyScale has an almost constant training throughput regardless of the worker number. But worker packing grows at the beginning and reaches 1.11× compared to EasyScale, resulting from higher GPU utilization due to the concurrent execution of multiple kernels, but at a cost of higher memory usage as shown above.

To demonstrate the lightweight context switching, we run different workloads using one EST per GPU, with and without the context



(a) ResNet50 (b) ShuffleNetv2

Figure 10: The peak GPU memory usage (curves) and throughput (bars) of EasyScale and worker packing when executing multiple ESTs/workers on a V100.

switching. Note that EasyScale cannot generate accuracy-consistent results same as DDP when there is no context switching. Figure 11 illustrates that in most cases, the overhead is negligible, with a maximum of 1.9% for `Electra`, because EasyScale meticulously identifies non-determinism and only records determinism-critical states instead of large model parameters in contexts.

We further evaluate the data worker sharing optimization among above workloads with 8 ESTs. Enabling this optimization results in an average decrease of 67.1% in training time for the first mini-batch. This is because data worker sharing significantly reduces the number of required data workers (e.g., reduced from 32 to 4), thereby reducing their launch time when responding to elasticity.

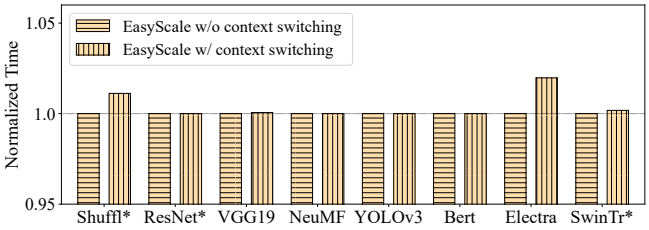


Figure 11: The time of lightweight context switching.

5.1.3 *Overhead of ensuring accuracy-consistency.*

We examine the overhead of ensuring accuracy-consistency by measuring the training time of typical DL workloads. EasyScale is configured with two configurations: *a) D1*, which ensures accuracy consistency when using an elastic number of homogeneous GPUs, and *b) D1+D2*, which ensures consistency when using an elastic number of heterogeneous GPUs. The baseline is set as the official version of PyTorch. The experiment is conducted on V100, P100, and T4 GPUs. Figure 12 presents the per-iteration time normalized to the baseline for each type of GPU.

The models can be classified into two categories based on their overhead. The first category includes models such as `NeuMF`, `Bert`, `Electra`, and `SwinTransformer`. For these models, ensuring accuracy consistency (including both *D1* and *D1+D2*) results in less than 1% overhead. Therefore, we can train them using elastic and heterogeneous GPU resources with negligible overhead. The second category includes models such as `ShuffleNetV2`, `ResNet50`, `VGG19`, and `YOLOv3`. Ensuring consistency on homogeneous GPUs for these models also brings negligible overhead. However, ensuring

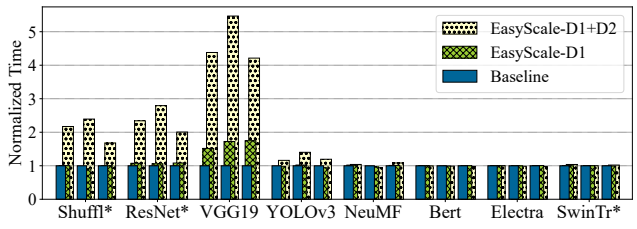


Figure 12: The overhead of ensuring accuracy-consistency. Each bar reports the time normalized to the baseline of the specific GPU. For each model, the three bars indicate the results in V100, P100, and T4, from left to right.

consistency on heterogeneous GPUs will introduce considerable overhead (i.e., 236% on average). This is because EasyScale-*D2* turns off vendor-optimized convolution kernels in these workloads for determinism. Nevertheless, EasyScale can automatically identify the training jobs that do not rely on such kernels and allow them to use elastic and heterogeneous GPU resources while using homogeneous GPU resources for other jobs instead.

We further measure the gradient copy and synchronization overhead that the EST abstraction might introduce. EasyScale is configured to execute 8 ESTs on 1 GPU, while DDP runs on 8 GPUs. To ensure accurate results, we skip the first 10 mini-batches for warm-up and recorded the average execution time of each EST. EST 0-6 asynchronously copies the generated gradients through D2H operations, and EST 7 performs gradient synchronization similar to DDP. Surprisingly, as shown in Figure 13, EasyScale achieves superior or competitive performance compared to DDP. For EST 0-6, this is because of the overlapping between gradient copy and the backward computation as well as the forward computation of next EST. For EST 7, this is because when EST 7 starts gradient synchronization, the other replicas of gradients (EST 0-6) are already ready for synchronization. In contrast with our findings in EasyScale, it is difficult to ensure simultaneous production of gradients among all workers in DDP, which could lead to potential delays. Besides, when only one EST resides on each GPU, the gradient copy is not needed, and EasyScale shows competitive performance to DDP.

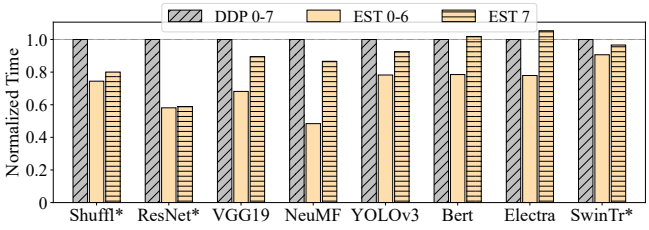


Figure 13: The overhead of gradients copy and sync.

5.2 Trace Experiment

To demonstrate the improved resource utilization and job throughput of EasyScale, we conducted trace experiments on a cloud cluster consisting of 32 V100 GPUs, 16 P100 GPUs, and 16 T4 GPUs.

Workloads – The training jobs are configured as the DL workloads in Table 1. The job arrival time of the trace is configured according to Microsoft [59], and the job runtime distribution is down-sampled from our production training jobs. All jobs are considered to be submitted to the same tenant with the same priority.

Settings – We compare EasyScale to Apache YARN's capacity scheduler (YARN-CS), a production GPU cluster scheduler used in Microsoft Philly [31]. YARN-CS enforces FIFO mode for processing jobs to ensure inter-job fairness. In the experiment, all jobs use gang-scheduling to allocate GPU resources, and the minimal number of GPUs was set to 0 in EasyScale. YARN-CS allocates the same type of GPUs (e.g., all V100 GPUs) for a job based on its requirement. And EasyScale has two configurations: *1)* $\text{EasyScale}_{\text{homo}}$, where a job can only use homogeneous GPUs by constraining scheduling plans of intra-job scheduler to homogeneous GPUs; and *2)* $\text{EasyScale}_{\text{heter}}$, where a job can use heterogeneous GPUs.

Results – Figure 14 shows the average job completion time (JCT) and the makespan of different schedulers when scheduling the same job trace. $\text{EasyScale}_{\text{homo}}$ and $\text{EasyScale}_{\text{heter}}$ is compared to the capacity scheduler. The results show that $\text{EasyScale}_{\text{homo}}$ improves average JCT by 8.3× and makespan by 2.5×, while $\text{EasyScale}_{\text{heter}}$ improves by 13.2× and 2.8×. Enabling elasticity eliminates the gang-scheduling requirement for training jobs, which significantly enhances performance through incremental utilization of idle GPUs. This results in a speedup as shown in $\text{EasyScale}_{\text{homo}}$. With the ability to utilize heterogeneous GPU resources, $\text{EasyScale}_{\text{heter}}$ can utilize more available GPU resources. During execution, the allocated GPUs of $\text{EasyScale}_{\text{heter}}$ are generally higher than those of $\text{EasyScale}_{\text{homo}}$. By ensuring consistent accuracy using heterogeneous GPU resources for elastic training, EasyScale jobs can further utilize available GPUs of other types to achieve better throughput.

5.3 Cluster Experiment

We have deployed EasyScale in a shared GPU cluster with more than 3,000 GPUs. This production cluster used to dedicate for online GPU serving or development (e.g., Jupyter Notebook). Similar to Borg [56], which classifies jobs as production jobs (i.e., high-priority) and non-production batch jobs, we treat inference serving jobs as production jobs with guaranteed quota and treat EasyScale jobs as non-production jobs to utilize the idle GPUs.

To illustrate the cluster efficiency improvement from EasyScale, one-day statistic is collected in Dec. 2021, right after EasyScale is fully deployed in this cluster. As shown in Figure 16, the first 1,440 minutes (day-1) indicate the statistic collected before the deployment of EasyScale, while the last 1,440 minutes (day-2) show how EasyScale utilize the idle GPUs. On day-2, EasyScale jobs are submitted to this cluster according to the business patterns in real-world applications. These jobs contain different training workloads (CV/NLP) with different hyper-parameter settings. On average, EasyScale improves the GPU allocation ratio by 17.1% and improves the GPU SM utilization by 62.1%. During the one-day statistic, the elastic EasyScale jobs use 459 temporally idle GPUs on average that can quickly scale in to release GPUs for high-priority online serving jobs in seconds. After the leaving of those inference jobs, EasyScale jobs full up the idle GPUs within 5 minutes. Our

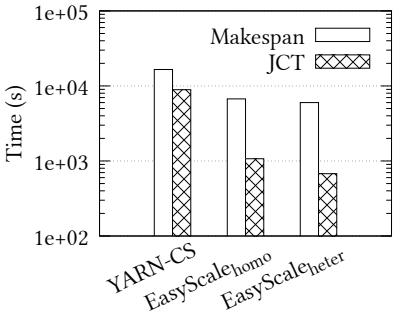


Figure 14: Comparison of YARN-CS, EasyScale$_{homo}$, and EasyScale$_{heter}$.

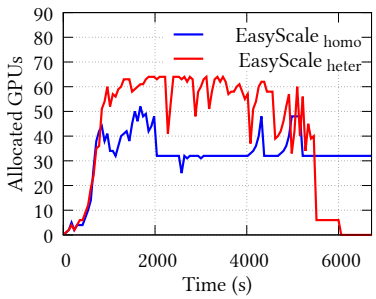


Figure 15: Allocated GPUs of EasyScale$_{homo}$ and EasyScale$_{heter}$.

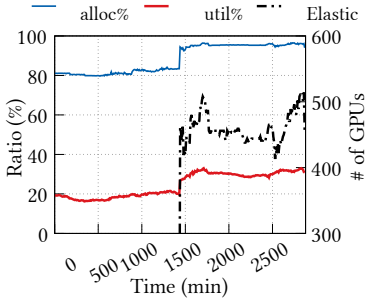


Figure 16: One-day statistic on a production cluster of CompanyA.

cluster statistic records a total number of 362 preemptions on that day and no EasyScale job fails.

6 RELATED WORK

Determinism and reproducibility – Determinism, reproducibility, and ablation study are important for DL researches [41, 47, 48]. DL frameworks [7] and NVIDIA [6] have studied deterministic model training using single GPU. However, it is a hard problem due to the floating-point dominating execution, complicated software stack, and hardware optimized implementations. The design of EasyScale derives from the understanding of non-determinism, considers bitwise identical in every step of model training, and extends the determinism to elastic training over heterogeneous GPUs. In Ampere GPU (e.g., A100), cuBLAS supports only internal heuristics approach without public interface to select low-level kernel implementation, which can hardly produce accuracy-consistent results compared to that of using previous generation of GPUs [2].

Elastic deep learning – TorchElastic [9], ElasticDL [4], and Horovod Elastic [3] support elastic training and fault tolerance, however, they introduce non-determinism in model accuracy. KungFu [40] and Pollux [49] support adjusting training algorithms, including both adaptive batch sizes and learning rates, allowing both customized and build-in adaptation policies for efficient scaling. VirtualFlow [44] and Varuna [11] leverage the gradient accumulation approach to achieve elasticity. Those works cannot guarantee the trained model with consistent accuracy among different runs. As our parallel works, AutoPS [25], Singularity [53], and Pathways[13] also explore elastic training in different ways, including the model aggregations in parameter server architecture, CUDA calls analytics, and heterogeneous interconnects. EasyScale utilizes the DL characteristics to achieve efficient and accuracy-consistent elastic training. EasyScale currently focuses on data parallel, however, new parallel strategies are proposed for large model training [22, 33, 42, 50], and we consider supporting them as future works.

Cluster scheduling – Resource management for DL jobs has been studied to improve utilization [57, 59, 60] and fairness [43]. SLAQ [62] and Pollux [49] prioritize resources by considering model convergence. To improve cluster utilization, ONES [14] tunes batch size of training jobs. Optimus [46] and EDL [58] adjust the number of parameter-servers and workers. PipeSwitch [12] overlaps computation with layered model loading. Retiarii [63] dynamically allocates resources among AutoML jobs and applies cross-job optimization. Gandiva [59] and AntMan [60] utilize the unique DL characteristic to optimize scheduling at mini-batch boundaries.

7 CONCLUSION

Through EasyScale, we demonstrate the success of decoupling DL training process from underlying resource allocation for achieving accuracy-consistent model training under elasticity. Specifically, EasyScale presents several innovations to address non-determinism during elastic training by *1)* introducing the EST abstraction to preserve the training behaviors over elasticity and heterogeneity, *2)* sourcing the non-deterministic behaviors scattered in the DLT software stack and solving them, and *3)* developing intra-job and inter-job schedulers utilizing the heterogeneous GPU cluster. Going forward, we hope EasyScale can draw attention to the deterministic computation of DL, and we should not always trade determinism for performance when designing DL systems.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Ziheng Wu, Zhen Zheng for the valuable comments on the early version of this paper. This work is supported by National Natural Science Foundation of China (No. 62322201, 62072018 and U22A2028), and the Fundamental Research Funds for the Central Universities. Hailong Yang is the corresponding author.

REFERENCES

[1] 2023. Auto kernel selection of NVIDIA cuDNN. https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnGetConvolutionBackwardDataAlgorithm_v7.
[2] 2023. Auto selection of cuBLAS in Ampere GPUs. https://docs.nvidia.com/cuda/cublas/index.html#cublas-GemmEx.
[3] 2023. Elastic Horovod. https://github.com/horovod/horovod/blob/master/docs/elastic.rst.
[4] 2023. ElasticDL, https://github.com/sql-machine-learning/elasticdl/.
[5] 2023. NCCL deterministic. https://github.com/NVIDIA/nccl/issues/157.
[6] 2023. NVIDIA Framework-determinism, https://github.com/NVIDIA/framework-determinism.
[7] 2023. PyTorch Reproducibility. https://pytorch.org/docs/stable/notes/randomness.html.
[8] 2023. torch.backends.cudnn.benchmark. https://pytorch.org/docs/stable/backends.html.
[9] 2023. TorchElastic. https://pytorch.org/docs/stable/distributed.elastic.html.

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[11] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 472–487. https://doi.org/10.1145/3492321.3519584

[12] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 499–514.

[13] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 430–449.

[14] Zhengda Bian, Shenggui Li, Wei Wang, and Yang You. 2021. Online Evolutionary Batch Size Orchestration for Scheduling Deep Learning Workloads in GPU Clusters *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 100, 15 pages. https://doi.org/10.1145/3458817.3480859

[15] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *ACM Queue* 14 (2016), 70–93.

[16] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[17] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=r1xMH1BtvB

[18] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, Miami, FL, USA, 248–255.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, USA, 4171–4186.

[21] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. 2010. The pascal visual object classes (voc) challenge. *International journal of computer vision* 88, 2 (2010), 303–338.

[22] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 431–445. https://doi.org/10.1145/3437801.3441593

[23] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. 2020. Importance-driven deep learning system testing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, Seoul, Korea (South), 702–713.

[24] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017). arXiv:1706.02677 http://arxiv.org/abs/1706.02677

[25] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, and Aditya Akella. 2022. Elastic Model Aggregation with Parameter Service. *CoRR* abs/2204.03211 (2022). arXiv:2204.03211 https://arxiv.org/abs/2204.03211

[26] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[27] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)* 5, 4 (2015), 1–19.

[28] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

[29] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 173–182. https://doi.org/10.1145/3038912.3052569

[30] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. Association for Computing Machinery, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[31] 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 (USENIX ATC 19)*. 947–960.

[32] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *Tech. Rep.* (2018).

[33] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 673–688.

[34] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[35] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU Memory Sharing for Concurrent DNN Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 161–175.

[36] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 439–453.

[37] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. *CoRR* abs/2103.14030 (2021). arXiv:2103.14030 https://arxiv.org/abs/2103.14030

[38] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[39] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*. IEEE, Munich, Germany, 116–131.

[40] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 937–954.

[41] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. 2018. Deterministic implementations for reproducibility in deep reinforcement learning. *arXiv preprint arXiv:1809.05676* (2018).

[42] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[43] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 481–498.

[44] Andrew Or, Haoyu Zhang, and Michael None Freedman. 2022. VirtualFlow: Decoupling Deep Learning Models from the Underlying Hardware. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 126–140.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.

[46] 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 European Conference on Computer Systems*. ACM, New York, NY, USA.

[47] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2021. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 771–783. https://doi.org/10.1145/3324884.3416545

[48] Shangshu Qian, Hung Pham, Thibaud Lutellier, Zeou Hu, Jungwon Kim, Lin Tan, Yaoliang Yu, Jiahao Chen, and Sameena Shah. 2021. Are My Deep Learning Systems Fair? An Empirical Study of Fixed-Seed Training. *Advances in Neural Information Processing Systems* 34 (2021).

[49] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18.

[50] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Atlanta, GA, USA, 1–16.

[51] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*. The Association for Computational Linguistics, 2383–2392. https://doi.org/10.18653/v1/d16-1264

[52] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *CoRR* abs/1804.02767 (2018). arXiv:1804.02767 http://arxiv.org/abs/1804.02767

[53] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. 2022. Singularity: Planet-Scale, Preemptible, Elastic Scheduling of AI Workloads. *CoRR* abs/2202.07848 (2022).

[54] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1409.1556

[55] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 909–923. https://doi.org/10.1145/3297858.3304072

[56] 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*. ACM, New York, NY, USA, 18.

[57] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960.

[58] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2022. Elastic Deep Learning in Multi-Tenant GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2022), 144–158. https://doi.org/10.1109/TPDS.2021.3064966

[59] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[60] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 533–548.

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX, 1 pages.

[62] 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, New York, NY, USA, 390–404.

[63] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. 2020. Retiarii: A Deep Learning Exploratory-Training Framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 919–936.

[64] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–38.

[65] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. 2020. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 515–532.