AutoPipe: Automatic Configuration of Pipeline Parallelism in Shared GPU Cluster

Jinbin Hu
Changsha University of Science and Technology
Changsha, China
jinbinhu@csust.edu.cn

Ying Liu
Changsha University of Science and Technology
Changsha, China
yingliu@stu.csust.edu.cn

Hao Wang
Hong Kong University of Science and Technology
Hong Kong, China
hwangdv@connect.ust.hk

Jin Wang*
Hunan University of Science and Technology
Xiangtan, China
jinwang@hnust.edu.cn

ABSTRACT

As training Deep Neural Network (DNN) is time-consuming, people resort to parallelization across multiple accelerators. A plethora of solutions adopt data/model parallelization, but they suffer from frequent weight synchronization overhead or resource under-utilization. Recent work introduces pipeline parallelism to improve the utilization of accelerators, however, most existing pipeline parallelism approaches take a one-shot configuration, while ignoring the fluctuation of available resources, e.g., bandwidth and GPUs. Moreover, the heuristic work partition methods oversimplify the computation and communication process, leading to sub-optimal results. To address this challenge, we present AutoPipe, a self-adaptive pipeline parallelism optimization solution. At its core, AutoPipe introduces a reinforcement learning (RL) based work partitioning model, which takes into account both exact communication procedure and dynamic state switching. To mitigate the stalls on state switching, AutoPipe adopts layer-by-layer computation under switching. We have implemented an AutoPipe prototype and evaluated it via testbed experiments. Our results show that the AutoPipe-enhanced PipeDream can find better work partitioning and benefit from dynamic configuration, outperforming the vanilla solutions by up to 89% for exclusive tasks and 143% in dynamic workloads. Furthermore, we show that AutoPipe can also work well with other pipeline parallelism schemes and achieve considerable performance gains.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; • **Computing methodologies** → *Distributed algorithms.*

KEYWORDS

Distributed Training, Deep Neural Network, Pipeline Parallelism, Reinforcement Learning

*Jin Wang is the corresponding author.

ICPP '24, August 12–15, 2024, Gotland, Sweden

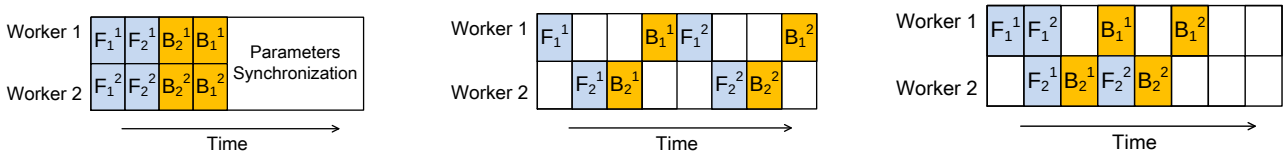
ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673047

ACM Reference Format:
Jinbin Hu, Ying Liu, Hao Wang, and Jin Wang. 2024. AutoPipe: Automatic Configuration of Pipeline Parallelism in Shared GPU Cluster. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673047

1 INTRODUCTION

The sizes of DNN models grow explosively due to the increasingly complex applications [1]. To accelerate the model training, in practice, machine learning (ML) practitioners leverage parallel computing to spread intensive computation across multiple accelerators, e.g., GPUs [2]. Previous solutions [3] often adopt intra-batch parallelism, which focuses on the parallelization within one iteration. The most common intra-batch parallelism approaches are data parallelism and model parallelism. However, data parallelism suffers from high communication cost, recent work shows that the communication overhead can account for 90% over 32 GPUs [4]. For model parallelism, while it has less communication, the computing resources are under-utilized due to the computation dependency among each layer.

To address the issues of high communication overhead and low resource utilization, recent work proposes pipeline parallelism [4, 5]. To reduce the communication volume, pipeline parallelism partitions different layers of the model to different workers, just like the model parallelism. Meanwhile, to avoid the idle of computation resources, it injects multiple batches of different iterations into the model concurrently, therefore the training is inter-batch. The inter-batch introduces weight staleness and inconsistency issues [4]. To address the above issues, recent work GPipe [5] introduces a synchronous pipeline approach. It divides the mini-batch into different micro-batches and pipelines them, the weight updating is only performed when all micro-batches of the mini-batch complete. While GPipe guarantees the consistency, it still does not utilize the computing resources well. To further improve the resource utilization, PipeDream [4] releases the synchronization requirement, it only makes sure that in the same GPU and mini-batch, all forward passes and backward passes have the same weights via snap-shooting different versions of weights. Follow-up works, e.g., DAPPLE [30], Chimera [31], point out that such an asynchronous approach impacts the training accuracy and propose synchronous training framework focusing on large-scale neural networks.



(a) Data Parallelism (b) Model Parallelism (c) Pipeline Parallelism

Figure 1: Different parallelism schemes. Here we show the timeline of training two batches on two workers. "F" refers to forward propagation and "B" refers to backward propagation. The subscript indicates the layer of the model and the superscript indicates mini-batch (in data parallelism) or iteration (in model/pipeline parallelism). We assume the model only has two layers, the forward and backward propagation cost the same time. (a) illustrates the process of data parallelism, which makes full use of computing resources when performing computation. However, it brings the cost for communication of parameters. (b) illustrates the process of model parallelism. We observe the under-utilization of computing resources. (c) illustrates the process of pipeline parallelism. It has better GPU utilization and is free of synchronization cost.

However, we observe that both synchronous and asynchronous pipeline parallelism solutions ignore the fluctuation of available resources during the long time training. Currently, to facilitate DNN training, enterprises or private users typically setup a GPU cluster shared by multi-jobs [6]. In a shared GPU cluster, the available resources, e.g., bandwidth and GPUs, for each job vary over time. Recent work [7] observes three distinct factors can affect cluster utilization, i.e., gang scheduling, locality constraints and failures during training.

Meanwhile, the performance of pipeline parallelism, i.e., utilization of GPUs, is highly related to work partition [4], which determines how to assign the computation of each layer to the participating GPUs. Existing pipeline parallelism solutions establish work partitions based on the available GPUs and bandwidth at the start of training, and these partitions remain fixed. However, the DNN training is notoriously time-consuming, which can take hours or days [7]. During the training, if the resources state changes, e.g., new jobs acquire the shared GPUs or bandwidth, the previous work partition will lead to sub-optimal performance.

A straw-man solution is to apply work partition whenever available resources change. However, this method has drawbacks, particularly in environments with frequent resource fluctuations. Each time a new work partition is applied, workers are assigned new tasks, interrupting the current training and potentially affecting performance. Moreover, calculating a new work partition without considering the previous state overlooks possible improvements. A more refined strategy would involve designing new partitions that take into account the last state, thereby minimizing the impact on active workers by strategically reassigning tasks. Furthermore, we observe that almost all existing solutions [4, 30–32, 36] take heuristic approaches to get the work partition. For example, the classic pipeline parallelism, i.e., PipeDream [4], simplified the modeling of work partition by assuming an all-reduce communication pattern within the workers of the same layer.

To improve pipeline parallelism in a shared GPU cluster, we introduce AutoPipe, an automatic configuration framework, which is self-adaptive to the available resources. At its core, AutoPipe consists of three key techniques: 1) communication-aware pipeline: AutoPipe profiles the integrated communication process, which takes the bandwidth variation into consideration. 2) RL-based automatic configuration: previous solutions apply Dynamic Programming (DP) to work partition problem. However, when considering the real communication, the problem becomes complicated and time-consuming. Reinforcement Learning (RL) is proved to solve such dynamic optimization problems quickly and effectively [12, 14]. 3) fine-grained state switching: simply stopping the pipeline whenever we reassign each worker's tasks can lead to resource wastage. Therefore, AutoPipe applies a layer-by-layer switching to maintain the pipeline while changing the task assignment.

We have implemented an AutoPipe prototype to support DNN training based on PipeDream [4]. The key components include: 1) a resource changing detector, which is used to monitor the available bandwidth and GPUs; 2) the training of the RL model for work partition; 3) a scheduler for state switching and executors in the worker side. We further integrate AutoPipe into PyTorch [8], a well-known ML framework.

We evaluate AutoPipe on our testbed with 10 NVIDIA P100 GPUs and 10/25/40/100Gbps networks. We test AutoPipe with different DNNs and also provide a deep dive into the effectiveness of each design component. The experimental results show that AutoPipe outperforms PipeDream and vanilla framework by up to 1.89× and 2.77×, respectively. We also observe that our RL-based solution can further improve the overall training performance when AutoPipe is deployed on multiple jobs. Finally, we apply the idea of AutoPipe to other latest pipeline parallelism solutions [30–32]. The evaluation shows that AutoPipe can further improve these pipeline parallelism optimization.

In summary, this paper makes the following contributions:

- We are among the first to identify the sub-optimal work partition of current pipeline parallelism in a shared GPU cluster.
- We apply reinforcement learning to find the optimal partition, which is efficient and obtains considerable performance gains.
- We fully implement our algorithm, integrate it into PyTorch, and evaluate it on a real testbed with multi-GPUs.

2 BACKGROUND

2.1 Pipeline Parallelism

To accelerate DNN training, distributed training is widely used. There are mainly three types. We illustrate and compare them in Figure 1, here we focus on the pipeline parallelism. Different from previous intra-batch parallelism, pipeline parallelism spans multiple iterations, which is so-called inter-batch parallelism [4]. Inter-batch

parallelism can achieve better performance due to less communication and the overlapping of communication and computation. Pipeline parallelism leverages pipelining optimization based on model parallelism. As we can see in figure 1 (c), worker 1 injects two mini-batches into the pipeline and performs the FP of the first layer. Once the FP of the first mini-batch is completed, worker 1 transmits the output to worker 2, while the first layer FP of the second mini-batch is simultaneously started. Worker 2 starts the BP on a mini-batch once the FP completes. For the BP, worker 2 asynchronously sends the gradient to worker 1 while performing the computation of the second mini-batch. The literatures present two kinds of pipeline parallelism according to the synchronization requirement [4, 5, 30–32, 34–36], i.e., synchronous pipeline parallelism and asynchronous pipeline parallelism.

Synchronous pipeline parallelism. All inputs from the last flush use the same weights. The timeline of pipeline can be divided into FP pass and BP pass. GPipe [5] can be categorized as a typical synchronous pipeline parallelism. GPipe is the improved version of naive model parallelism which overcomes the low computation resource utilization and memory limitation of GPU. In GPipe, each mini-batch is divided into multiple smaller micro-batches. The micro-batches are trained in a pipelined manner, therefore, multiple GPUs can train different parts of the model at the same time. Considering the memory cost, GPipe recomputes the FP and makes sure the micro-batches of the same mini-batch pass all GPUs sequentially. Therefore, it sacrifices the performance and still has room for pipeline throughput improvement. Here we list other recent work on synchronous pipeline parallelism.

- DAPPLE [30] combines data parallelism with pipeline parallelism, addressing critical issues such as improving computing efficiency, ensuring model convergence, and reducing memory usage without adding computing costs. It designs a planner based on dynamic programming to solve the partition and placement of the hybrid parallelism.
- Megatron-LM [34] is a large language model training framework that effectively combines tensor and pipeline model parallelism for training large-scale transformer-based networks. It specifically addresses the challenges of training models with a massive number of parameters, optimizing both computing efficiency and memory usage.
- Chimera [31] effectively maps stages of the model in linear and opposite orders across two pipelines, optimizing the execution of micro-batches. This unique configuration significantly reduces idle times (bubbles) in both forward and backward passes. Chimera's design is adaptable to a varying number of micro-batches, ensuring high utilization of resources even with constraints for model convergence.

Asynchronous pipeline parallelism. To fill up the pipeline, the weights of each worker may be stale and delayed. These approaches [4, 32, 35, 36] are proposed to improve the utilization of GPU, they allow asynchronous (thus faster) weight to update as long as enough gradients are accumulated. PipeDream [4] is one of the representatives. PipeDream renders multiple GPUs concurrent work by simultaneously injecting multiple mini-batches into the pipeline. To mitigate the weight inconsistency issue, PipeDream snapshots the weights of each active mini-batch. However, asynchronous pipeline faces inconsistent weight and staleness issues, because the mini-batches are cross-trained. Besides PipeDream, we list three recent works on asynchronous pipeline parallelism.

- PipeMare [36] addresses the hardware efficiency and memory cost issues commonly associated with pipeline parallel training. It introduces a robust training method that tolerates asynchronous updates during pipeline parallel execution without sacrificing utilization or memory.
- Kosson, Atli, et al. [35] explore the use of small batch fine-grained pipelined backpropagation to eliminate the fill and drain overhead, by updating weights without the need to first drain the pipeline. To mitigate the accuracy downgradation caused by the asynchronicity, they introduce Spike Compensation and Linear Weight Prediction.
- PipeDream-2BW [32] focuses on memory-efficient pipeline parallelism for training large DNN models with billions of parameters. It uses a novel pipelining and weight gradient coalescing strategy along with double buffering of weights to ensure high throughput, low memory footprint, and weight update semantics similar to data parallelism.

Work partition. In pipeline parallelism, the partitioning of the model, i.e. work partition and the choice of pipeline depth, has a huge impact on the GPU utilization and the training speed. Existing pipeline parallelism schemes for work partitioning primarily fall into three categories. The first category involves evenly splitting of structurally uniform models like Transformer, exemplified by Megatron-LM [34], PipeDream-2BW [32], and Chimera [31]. The second category has no generic work partition strategy, it only offers manually designed work partition schemes for specific models used in experiments, such as PipeMare [36]. The third category designs a universal work partition algorithm, utilizing Dynamic Programming (DP) to find optimal work partition strategies, seen in PipeDream [4] and DAPPLE [30]. The first two methods are relatively simple and lack versatility, so we focus on the third category. Since the work partition algorithms of DAPPLE and PipeDream are similar, with the former targeting synchronous and the latter asynchronous scenarios, we will primarily discuss PipeDream. For the work partition in PipeDream, it first performs the profiling, which records three numbers for each layer, i.e., the total computation time, the size of the output activations and the size of weight parameters. Then, based on the profiling results, it leverages DP to compute: 1) a partitioning of layers with the form of stages; 2) number of workers for each stage; 3) optimal number of on-the-fly mini-batches to fill the pipeline. The model of DP simplifies the communication, it assumes that the network topology is hierarchical and each level has the same bandwidth.

2.2 Automatic Configuration

Our approach strives to achieve an automatic configuration of pipeline parallelism. Here we introduce two typical scenarios of automatic configuration.

Flow scheduling. Flow scheduling is a classical problem in datacenter networks [11, 17]. Previous works make the big-switch assumption, i.e., the network has full bisection bandwidth, and simplify the flow scheduling problem to the issues of deciding the

flow sending order [33]. A widely used flow scheduling methodology is strict priority queueing with preemption and it is often implemented as a multi-level feedback queuing with K priority queues. The challenge is to determine the demotion thresholds of each queue. Previous solutions [10] apply a static threshold configuration. However, the flow distribution and available bandwidth vary in the datacenter, and they affect the choice of optimal thresholds. Recent work AuTO [12] proposes an end-to-end automatic traffic optimizations system, which applies deep reinforcement learning (DRL) to handle the thresholds choosing problem.

Communication scheduling. DNN model has a layer-wise structure, which makes it possible to overlap the communication and computation among different layers. Further, recent works [13, 14] propose communication scheduling strategies. ByteScheduler [13] performs tensor partition to enable the preemptive scheduling. There are several hyper-parameters which determine the efficiency of communication scheduling in ByteScheduler. The dynamic changing resources, i.e., available bandwidth and GPUs, also impact the choosing of these hyper-parameters. To perform automatic configuration for the communication scheduling, AutoByte [14] enhances the ByteScheduler with a meta-network, which takes the system's runtime statistics as its input and output predictions for speedup under specific configurations.

3 MOTIVATION

3.1 Pipeline Utilization in Pipeline Parallelism

In this section, we provide three key observations on the blemishes of existing pipeline parallelism solutions, which limit the utilization of computation resource.

Observation 1: Existing configurations are all one-shot. The existing configuration is determined before the training starts and stays the same during the training. Therefore, the fluctuation of bandwidth or computation resource may cause the previous optimal configuration to become stale. A recent measurement of Microsoft [7] shows that more than half of the training jobs complete in tens minutes. However, about 30% jobs may suspend during the training. Therefore, during the lifetime of a training job, other shared GPU jobs may start, complete or suspend, which causes the fluctuation of GPU resources. The fluctuation of bandwidth is more common, since non-DNN jobs can also impact the available bandwidth.

To find a pipeline parallelism solution for a dynamic environment, a straw-man approach is to perform work partition whenever the available resource state changes. However, two issues exist for this straightforward approach. 1) Calculation cost: PipeDream needs to re-profile whenever each resource changes. However, the profiler of PipeDream needs to monitor 1000 mini-batches. In addition, the calculation for new work partition also costs time, up to 8 seconds in the experiments of PipeDream [4]. 2) State switching cost: after updating the work partition, the task assigned to each worker will be changed. Therefore, we need to pause the training. Meanwhile, the flow line requires a restart phase, i.e., startup starts in Figure 2, which brings bubbles and reduces the pipeline utilization.

To minimize the cost of switching during pipeline parallelism, it is important to consider the state of the current work partition.

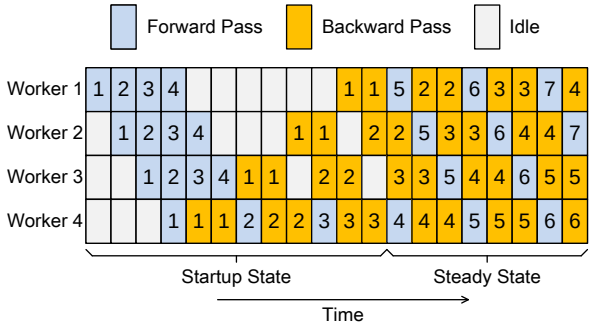


Figure 2: An ideal case of filling the pipeline in PipeDream. Note that the ideal case still needs time for the startup of the pipeline.

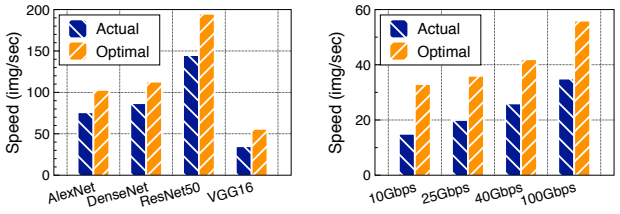
Typically, fluctuations in bandwidth and computing resources are localized, affecting only a few GPUs or links at any given time. This insight allows for the maintenance of most workers' current tasks while computing a new work partition. Additionally, to avoid the issue of stale parameters, a method like PipeDream's weight stashing can be employed. This involves maintaining multiple versions of parameters in each worker, enabling fine-grained state switching of different weights to preserve pipeline continuity.

Observation 2: Existing models oversimplify the training. The efficiency of a pipeline largely hinges on two elements: the assignment of tasks to workers (work partition of the model) and the number of mini-batches in flight. To identify the optimal configuration, PipeDream utilizes Dynamic Programming (DP) to calculate these factors. It starts by profiling three crucial values for each layer: the computation time, the size of the output activations and input gradients, and the size of weight parameters required for communication.

However, PipeDream's modeling has two major drawbacks. 1) PipeDream only measures the computation speed of one exclusively used GPU. However, there may be multiple types of GPUs in the shared GPU cluster, e.g., P100, V100, A100. Meanwhile, when multiple jobs share the GPU, the actual training speed may change significantly. 2) PipeDream makes too many assumptions on the modeling of communication between GPUs. It assumes a hierarchical network topology in which the bandwidths within the same level are identical. PipeDream assumes all_reduce collective communication for the workers of the same layer, the actual communication may use other approach, e.g., parameter server.

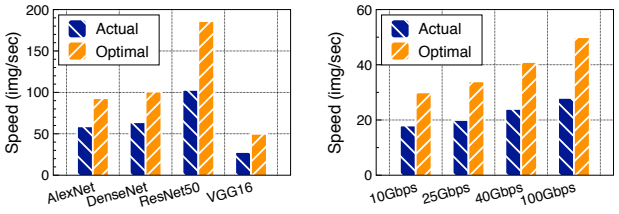
A naive improvement solution would be to patch the current model of PipeDream to solve the above problems. However, two issues exist. First, to fetch the real-time and more precise computation and communication speed, we need to perform profiling periodically and record more detailed data, i.e., the computation speed of each layer in each worker, the communication volume and speed among each worker. Second, the execution of DP will be more time-consuming. PipeDream shows that the running time cost of its simplified work partition is in the order of seconds. The complicated model will significantly increase the solution time. Our validation shows that the complicated model takes tens of minutes, which may be even longer than the overall training time.

Observation 3: Existing methods fail to fill the pipeline in practice. In traditional model parallelism, GPU resource utilization



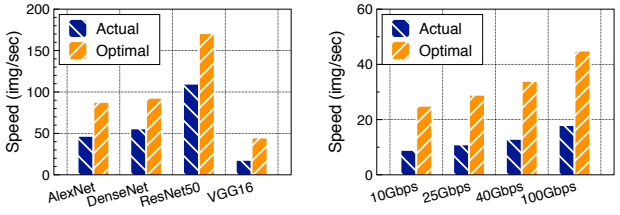
(a) Model influence (b) Network speed influence

Figure 3: Impact of dynamic changing bandwidth on PipeDream.



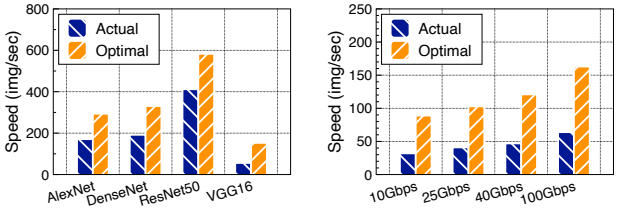
(a) Model influence (b) Network speed influence

Figure 4: Impact of dynamic changing computation resource (GPU) on PipeDream.



(a) Model influence (b) Network speed influence

Figure 5: Impact of concurrent distributed training jobs on PipeDream.



(a) Model influence (b) Network speed influence

Figure 6: Impact of the finishing of old distributed training jobs.

is significantly low, with only one GPU computing at any given moment. The primary advantage of pipeline parallelism is its ability to utilize all GPUs through effective pipelining. PipeDream tries to fill the pipeline. However, we find that it is difficult for PipeDream to fill the pipeline in practice. Figure 2 shows an ideal case of PipeDream filling the pipeline. In this case, after the startup state, PipeDream can achieve 100% utilization of the pipeline. However, several assumptions need to be satisfied for this case to be established, which are nearly impossible in practice. 1) The communication across four workers, i.e., Worker 1, Worker 2, Worker 3 and Worker 4, is negligible; 2) The computation time of each layer is the same; 3) The forward passes take exactly half time of the backward pass (e.g., the forward pass of batch 1 occupies 4 blue time units, while the backward pass of batch 1 occupies 8 yellow time units).

3.2 Resource sharing in GPU cluster

To verify the above analysis, we perform three groups of experiments on our testbed to compare the actual training speed of PipeDream (use the original work partition configuration) with that of the optimal (re-execute the work partition) under the dynamic environment. Considering the generality, for each group, we test different models and speeds of the network.

Dynamic changing of bandwidth. We only consider the dynamic changing of bandwidth in the first experiments. Other communication intensive tasks, e.g. uploading/downloading data and contention of other jobs, may cause the fluctuation of available bandwidth of current jobs. To show the influence of the bandwidth on PipeDream, we assume the current job exclusively occupies the bandwidth initially. When the measurement starts, the available bandwidth is halved, we measure the training speed of PipeDream at this moment. As a comparison, we perform work partition again according to the halved environment and measure the training speed . Figure 3 shows that for all models and different network speeds, the training speed of PipeDream decreases when the bandwidth changes, especially for the communication intensive models, e.g., VGG16. Meanwhile, we find that the performance degradation is more severe on the lower speed network, with a larger communication ratio. For example, the training speed under 10Gbps is decreased by up to 55% compared to the optimal result.

Dynamic changing of GPU resource. For the second group, we only consider the dynamic changing of computation resource, i.e., GPU. Such scenario includes the starting or completion of GPU-intensive tasks, e.g., image or video processing and single-GPU job. We assume the exclusive occupation of GPU for current job initially. When the measurement starts, we add an extra job on each GPU to emulate the contention of GPU-intensive tasks. In our experiment, the additional job is to train ResNet50 on the ImageNet. Similarly, the baseline is that we still use PipeDream but perform work partition again according to the new environment. Figure 4 shows the impact of GPU resources changing. We find that the changing of GPU resources has a significant impact on PipeDream across four different models. In addition, with a higher network speed, the performance degradation is more severe, due to a relatively higher proportion of calculations. When the link speed increases from 10Gbps to 100Gbps, the performance degradation of training speed is increased from 39% to 45% compared with the optimal speed.

The impact of multiple distributed training jobs. In a shared GPU cluster, it is common and frequent for new distributed training jobs joining and old jobs finishing or suspending [1]. Such cases change both available bandwidth and GPU resource of current jobs, which may make the current configuration of PipeDream sub-optimal. Similar to the above two experiments, we measure the training speed of the two different configurations before or after the

starting of the new job. Figure 5 shows that the joining of a new job also causes significant performance degradation of PipeDream under four training models and various network speeds. The training speeds are decreased by 36% and 60% under ResNet50 and 100Gbps, respectively. All the above experiments (Figure 3 - Figure 5) show the advantage of re-configuration when the resources are reduced. In fact, we have the same conclusion with increased environment resources. Figure 6 shows the reversed process of Figure 5, i.e., an old distributed training job finishes. We can find that re-executing the work partition (optimal) still be ahead of keeping the original configuration (actual).

4 AUTOPIPE DESIGN

4.1 Overview

Design goals. The design of AutoPipe has three objectives. 1) Optimal modeling: AutoPipe's model aims to accurately reflect the actual training process, including details like the current computation and communication speed of each worker; 2) Low time cost: AutoPipe should respond to the dynamic resource changing in time and providing work partition solutions that adapt to dynamic resource shifts promptly; 3) Fast switching: When transitioning to a new work partition, AutoPipe prioritizes maintaining continuous pipeline operation, aiming to avoid suspension or interruption of the ongoing processes.

Challenges. Three challenges prevent us from achieving the design goals. 1) The current method, which employs Dynamic Programming (DP) instead of exhaustive enumeration, has already optimized the computation time for work partition to a great extent. This makes any further optimization based on traditional mathematical modeling challenging. 2) Altering the work partition inevitably brings additional overhead. This becomes particularly problematic in environments with frequent resource changes, requiring a strategic balance between reaction sensitivity and environmental fluctuations. 3) Switching work partitions necessitates pausing the current batch's training, as it alters each worker's task. This interruption unavoidably leads to blocking the pipeline, posing a significant hurdle to maintaining continuous workflow.

AutoPipe overview. The core idea of AutoPipe is to apply RL to gradually fine tune the worker partition solution. We chose RL for its quick adaptation to complex settings, outperforming traditional methods in speed and flexibility [12, 14]. Considering the state switching cost, each worker partition reallocation only involves two workers. We first apply meta-learning to fetch the mapping of worker partition solution with the training speed. Then, without the need for profiling, we can quickly find the optimal worker partition of the next step. Due to the switching cost, we do not always perform state switching, instead, we apply an RL model to make decisions. In the following, we will introduce our pipeline model to predict the actual training speed with a meta-network and the RL model for determining whether to perform the new worker partition. Then we describe the offline training and online adapting for the two models. Finally, we show how to perform the fine granularity state switching without suspending the pipeline.

Table 1: Profiling Metrics of AutoPipe

Symbol	Shape	Specification
L	1	Total number of layers
N	1	Total number of workers
O_i	$L*1$	The size of output activations in layer i
G_i	$L*1$	The size of input gradients in layer i
P_i	$L*1$	The size of weight parameters in layer i
B_i	$N*1$	The available bandwidth of worker i
$FP_{i,j}$	$N*L$	FP computation time of layer j in worker i
$BP_{i,j}$	$N*L$	BP computation time of layer j in worker i

4.2 Integrated Pipeline Model

Given that the previous modeling in PipeDream ignores the dynamics of resources and oversimplifies the communication process, we re-model the pipeline parallelism. We monitor more data compared to the model of PipeDream. Meanwhile, considering the time cost, we set a new goal for our model, predicting actual training speed according to the worker partition. Instead of taking a heuristic approach to solve the model, we use a learning-based approach.

Profiling the training. Compared to the PipeDream's profiler, our profiling mechanism considers the volatility and variability of the resource. As shown in Table 1, AutoPipe first records the model level metrics before training, i.e., the size of output activations, input gradients and weight parameters in each layer, these quantities are constant during the training. AutoPipe also monitors three variables, i.e., the available bandwidth of each worker, the FP&BP computation time of each layer in each worker. Our profiler works on the idea of not interfering with training. For the available bandwidth of each worker, we measure it from the communication speed of the last iteration. We observe that the ratio of the computation time of each layer is almost constant. Therefore, we do not need to record all $FP_{i,j}$ and $BP_{i,j}$. We measure the ratios before training, and obtain the speed of the certain layer (e.g., layer j) of the worker (e.g., worker i) from the last iteration. Then we calculate the $FP_{i,j}$ and $BP_{i,j}$ for worker i based on the speed of layer j and the ratios.

Predicting the training time. PipeDream gives the partition solution directly by simplifying the modeling. A dilemma here is that over-simplistic modeling leads to sub-optimal solution, and close to realistic modeling leads to too long calculation time. Actually, we do not need to know the training details. Therefore, we apply an end-to-end learning-based methods for this task. Note that the training can be in different environments, to summarize the generic knowledge from various environments, we apply meta-learning [15]. Meta-learning, known as learning to learn, is methodically observing how various ML algorithms perform on different learning tasks and then learning from observations.

Figure 7 shows the meta network proposed by AutoPipe. There are four components. 1) Static metrics: these metrics are constant during the training, i.e., the first five metrics in Table 1. 2) Dynamic changing metrics: AutoPipe monitors these metrics, i.e., the last three metrics in Table 1, every iteration. 3) Worker partition solution: it is described in the form of an array with size N, each element in the array represents the assigned layers of each worker. 4) The meta network: we use a long short-term memory (LSTM)

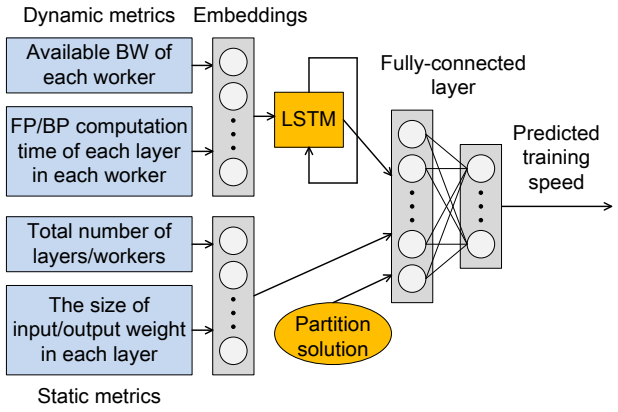


Figure 7: The meta-network of AutoPipe predicts the actual training speed according to the worker partition solution.

block to learn the dynamic environment, then together with the static inputs and partition solution, we apply the fully connected layers. Finally, we predict the training speed.

New worker partition. Now we can predict the training speed of each partition solution. Therefore, we can find the optimal by enumerating possible solutions without the real deployment and measurement. However, the enumeration is time consuming. In this case, we limit the new partition solution to only change the two workers' tasks in comparison to the old one (we use PipeDream's solution to do the initialization). There are two benefits: 1) The enumeration space is reduced, and the time complexity is only $O(L^2)$; 2) The change involving just two workers can be done without interrupting the pipeline. In addition, we can gradually migrate to the optimal by adjusting multiple times.

4.3 RL-based Automatic Configuration

We presented a new worker partition method. However, due to the switching overhead, we do not need to conduct a state switch for each iteration. Here, we use RL [24] to automatically determine whether to transmit to the new partition solution.

The arbiter model architecture. A typical RL model consists of three parts. The input and output of the model, internal structure of the model and a reward function to score the decision, i.e., the output. The input of our RL model consists of three parts, the environment metrics described in Table 1, the current partition solution and the new partition. The output is simply a boolean value that determines whether or not to switch. We use a fully connected neural network as the structure of our RL model. Our practical test shows that two hidden layers with 32 and 16 neurons are enough for the good performance. The reward function is the training speed of one iteration. We consider the normalized switching cost in this case. To calculate the switching cost, we apply a similar meta-network as the speed prediction model.

Offline training and online adapting. One major concern is that DNN is a data-driven solution that cannot address the problem of out-of-distribution. It is not practicable to obtain a perfect distributed training dataset because the distributed deep-learning system environment may change in both hardware (GPU and network) and software (model and DL architecture) parameters. One option is to conduct online training. On the target distributed deep learning task, we can train and update the meta-network and RL model online. However, this creates system overhead because we must repeatedly try exhaustive parameter sets to match a variety of system situations. This strategy goes against our original goal of increasing training speed. To address the problem, we apply an offline training, online adaptation techniques. The central idea is to employ transfer learning to swiftly adjust the meta-network and RL model to the current environment while minimizing system overhead. The following are three advantages of this strategy. 1) It does not result in a significant increase in system overhead; 2) The meta network optimizer can swiftly adapt to the present situation thanks to transfer learning; 3) This approach has a greater speed prediction accuracy than the offline trained variant.

4.4 Fine-Grained State Switching

To improve pipeline utilization, AutoPipe does not stop the pipeline when switching to a new partition solution. To achieve this, we refer to the pipelined context switching in PipeSwitch [16].

Layer-by-layer computation. PipeSwitch observes that DNN models have the layered structure. To pack multiple training jobs to the same GPU, PipeSwitch borrows the idea of fine-grained CPU time-sharing. Pipelining on per-layer granularity is the most basic method, in which the system sends the layers to the GPU memory one by one, with the calculation for each layer halted before it is sent. Layer-by-layer pipelining introduces additional sources of system overhead, such as the cost of making numerous PCIe calls to send the data. The transmission overhead is dominated by the data size for a big amount of data, for example, aggregating the entire model to a large tensor to broadcast together.

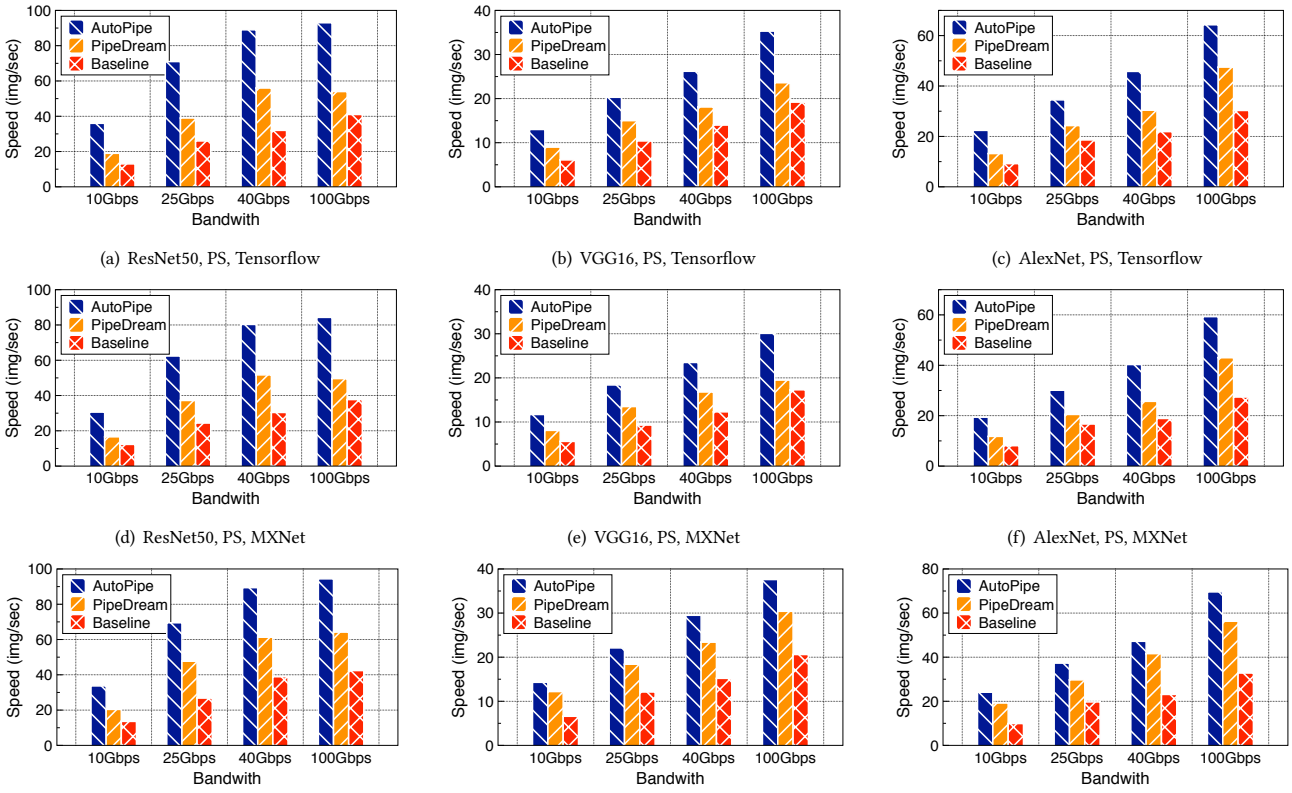
State switching in AutoPipe. Note that when switching the partition solution, AutoPipe will only affect the tasks of two workers. We perform the layer-by-layer computation for the two workers. However, the pipeline will still be blocked due to the stagnation of the corresponding layers of the two workers. To address the issue, we note that PipeDream applies a technique called weight stashing, which keeps numerous weight copies, one for each active mini-batch. By migrating the weight copy of later active mini-batch first, AutoPipe avoids the pipeline stall caused by the two workers.

5 EVALUATION

We conduct testbed experiments to evaluate AutoPipe[1]. The following are some of the highlights:

- In the testbed experiments, we evaluate AutoPipe across different DNN models, ML frameworks and synchronization paradigms (§5.2). Compared to the state-of-the-art work PipeDream [4], AutoPipe achieves up to 89% training speedup without decreasing the accuracy.
- In the deep dive, we measure each module of AutoPipe separately and evaluate AutoPipe under different dynamic environment including changing bandwidth and the available GPUs. We also preliminarily implement the AutoPipe-enhanced DAPPLE, Chimera, PipeDream-2BW and show the improvement.

[1]For convenience, AutoPipe represents the AutoPipe-enhanced PipeDream.



(a) ResNet50, PS, Tensorflow (b) VGG16, PS, Tensorflow (c) AlexNet, PS, Tensorflow

(d) ResNet50, PS, MXNet (e) VGG16, PS, MXNet (f) AlexNet, PS, MXNet

(g) ResNet50, Ring, PyTorch (h) VGG16, Ring, PyTorch (i) AlexNet, Ring, PyTorch

Figure 8: Training three widely-used DNN models, i.e., ResNet50, VGG16 and AlexNet under different communication patterns, i.e., Parameter Server and Ring All-reduce, with different machine learning frameworks, i.e., Tensorflow, MXNet and PyTorch. We set different link bandwidth from 10Gbps to 100Gbps.

- Experiments show that while improving the training speed, AutoPipe also guarantees the same convergence accuracy and does not introduce much additional CPU overhead (less than 1%).

5.1 Testbed Setup

Testbed topology. Our testbed has 5 physical GPU servers, each with 2 NVIDIA P100 GPUs, 40 CPU cores, 128GB memory, 1 Mellanox ConnectX5 100Gbps dual ports NIC, and 1 Mellanox SN2100 switch, which builds a single switch topology. Our operating system is Ubuntu 18.04 with Linux kernel version 4.15.0-55-generic. The Mellanox driver version is 5.1-0.6.6.0.

Models and baselines. In our experiments, we employ three models and one dataset. Our models have three image classification tasks: VGG16 [18], ResNet50 [19] and AlexNet [20] training on the synthetic data as the format of ImageNet [21]. We use two common parameter synchronization schemes: PS and Ring All-reduce for the data parallelism part. For the mini-batch size, we set 64 for VGG-16, 128 for ResNet-50 and 256 for AlexNet. We mainly compare AutoPipe with the baseline (vanilla ML frameworks) and PipeDream [4]. For the training speed metric, we calculate how many images are processed per second. For the convergence accuracy metric, we use the same metric in PipeDream [4], i.e., Top-1 accuracy. For the training speed metric, we chose the number of images processed per second.

5.2 Static Resource Allocation

Note that to emulate the scenarios of shared GPU cluster, we run three identical jobs in every experiment. Figure 8 compares the training speed of the baseline and PipeDream with AutoPipe on three models with network bandwidth ranging from 10Gbps to 100Gbps (10Gbps, 25Gbps, 40Gbps, 100Gbps) and two different communication schemes (PS and Ring All-reduce) implemented by TensorFlow [22], MXNet [23], and PyTorch [8], respectively. Figure 8 (a)-(c) show the results under PS and Tensorflow. We observe that AutoPipe can outperform baseline / PipeDream by up to 177% / 89% for ResNet50, 113% / 44% for VGG16, 143% / 70% for AlexNet. Figure 8 (d)-(f) show the results under PS and MXNet. We observe that AutoPipe can outperform baseline / PipeDream by up to 171% / 82% for ResNet50, 104% / 41% for VGG16, 124% / 58% for AlexNet. Figure 8 (g)-(i) show the results under Ring All-reduce and PyTorch. We observe that AutoPipe can outperform baseline / PipeDream by up to 148% / 65% for ResNet50, 117% / 17% for VGG16, 143% / 26% for AlexNet. The observations are the following: 1) AutoPipe outperforms PipeDream in all cases, and AutoPipe even obtains more speedup based on PipeDream in several cases. 2) The speedup of PS is greater than Ring All-reduce. The reason is that the worker partition model of PipeDream assumes the data parallelism part is Ring All-reduce. Therefore, it will be inaccurate under PS. 3) AutoPipe shows more speedup in ResNet50. The reason

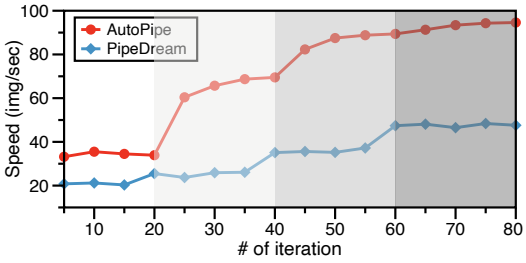


Figure 9: Training DNN under dynamic bandwidth.

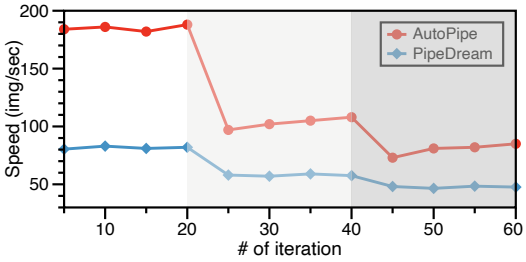


Figure 10: Training DNN under dynamic GPUs.

is that ResNet50 contains more layers than the other two models. Therefore, AutoPipe obtains more benefit from the more accurate modeling and fine granularity switching.

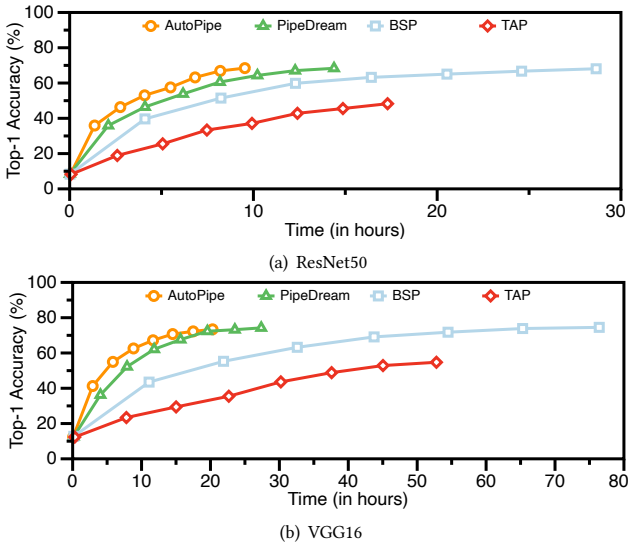
5.3 Dynamic Resource Allocation

Speedup under dynamic environment: The above experiments are under a fixed configuration with three identical jobs. With a dynamic environment, we found that AutoPipe may further speed up the training. We choose to train ResNet50 with Ring and PyTorch.

To evaluate AutoPipe under dynamic bandwidth first. We set the initial bandwidth to 10Gbps, we change the bandwidth to 25Gbps at the 20^{th} iteration, 40Gbps at the 40^{th} iteration, 100Gbps at the 60^{th} iteration. We compare AutoPipe with PipeDream, which has a fixed worker partition setting at the beginning. Figure 9 shows that AutoPipe outperforms the PipeDream all the time, and the speedup increases with the bandwidth growing. The result is in line with our expectation.

In the second experiment, we simulate the change of computation resources (GPU) by adding new local training jobs. At the beginning, there are no additional jobs. We add one more training job at the 20^{th} and 40^{th} iterations. From Figure 10, we can see that AutoPipe still keeps the leading all the time, and shows more performance gain with more additional jobs. Meanwhile, we observe that the factor of computation resources has more impact on the training speed, and AutoPipe obtains more speedup with the dynamic computation resource.

Impact on the model convergence: To prevent GPUs from being idle, AutoPipe applies asynchronous pipeline parallelism. Within a minibatch, after finishing the forward pass, each stage transfers the output to the next stage asynchronously. Here, we measure the impact of such synchronization on the training convergence accuracy. We compare the top-1 accuracy of AutoPipe with PipeDream, BSP (Bulk Synchronous Parallelism) and TAP (Total Asynchronous Parallelism) on two widely-used DNN models, ResNet50 and VGG16. As shown in Figure 11, AutoPipe can achieve the same top-1 accuracy



(a) ResNet50

(b) VGG16

Figure 11: Accuracy vs. time for ResNet50 and VGG16

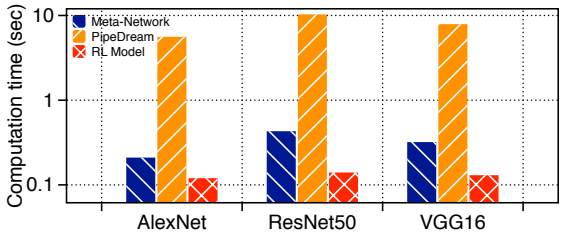


Figure 12: Computation time of worker partition modeling.

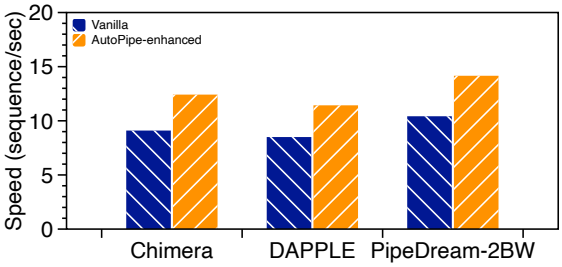


Figure 13: Improvement of AutoPipe-enhanced solutions.

as PipeDream and BSP under convergence, and 1.42 × /1.35× compare to TAP on ResNet50 / VGG16. Meanwhile, AutoPipe achieves the fastest convergence speed, which is 1.53×, 3.13× and 1.95× on ResNet50, and 1.35×, 3.25× and 2.31× on VGG16 compared to PipeDream, BSP and TAP, respectively.

Model computation overhead: To make AutoPipe ready deployable, we need to ensure the computation time of the meta-network and RL model is short enough. Here we measure both the meta-network and RL model computation time, and compare them with the DP algorithm used in PipeDream. We first find that their CPU utilization is quite low, less than 1%. Figure 12 shows the computation time under three different models. We find that the time cost of our meta-network and RL model is further less than the DP of PipeDream. The total time cost of the worker partition calculation of AutoPipe is less than one second in all cases.

Improvement on other pipeline parallelisms: Although our design is heavily based on PipeDream, the idea of AutoPipe is naturally applicable to improve other pipeline parallelism variants. Here, we implement and compare the AutoPipe-enhanced version of three recent works, i.e., DAPPLE [30], Chimera [31] and PipeDream-2BW [32]. Since these solutions focus on large-scale DNNs, we train Bert-48 on Wikipedia dataset, the mini-batch size is 256, other settings are the same as the testbed experiment. As shown in Figure 13, all the AutoPipe-enhanced versions outperform the vanillas.

6 RELATED WORK

Automated DNN parallelization: To explore efficient parallelization strategies, many automated frameworks have been proposed in the past few years. ColocRL [24] leverages reinforcement learning to automatically learn efficient device assignments for DNN model parallelizing across multiple GPUs, but it only performs parallelism in the device dimension. FlexFlow [3] uses the deep learning engine to automatically find efficient parallelization strategies in the search space. However, the above frameworks optimize distributed DNN training assuming a fixed computation graph. TASO [25] is the first work to focus on optimizing DNN computation graph, which automatically generates graph substitutions.

Communication optimization of DNN training: There are lots of approaches to optimize communication for DNN training. These include, but are not limited to: 1) overlapping communication with computation through various synchronous parallel strategies to alleviate communication bottleneck among workers [9, 26, 27]; 2) adjusting the mini-batch size to reduce communication rounds and speedup convergence for DNN training [21]; 3) using the programmable switch data plane to aggregate the model parameters from multiple rack switches to share the switch resources across simultaneously running jobs [28]; 4) using gradient compression technique to reduce the traffic volume in each iteration and make a good tradeoff between the communication bandwidth and the convergence time [29].

7 CONCLUSION

This paper presents AutoPipe, a highly efficient and self-adapted pipeline parallelism for a shared GPU cluster. AutoPipe contains three key innovations, it leverages meta-learning to predict the training speed of certain worker partition solution. With the predicted training speed, AutoPipe chooses the optimal partition solution and applies reinforcement learning to determine whether to perform the switching. Our evaluation results show that AutoPipe achieves the best performance under different models, communication schemes and frameworks with little system overhead. AutoPipe can optimize the system configuration when resource availability changes, reducing training time by up to 143% in dynamic network environment.

ACKNOWLEDGMENTS

This work was sponsored in part by National Natural Science Foundation of China under Grant (No.62072056, No.62102046), and the Natural Science Foundation of Hunan Province under Grant (No.2024JJ3017, No.2022JJ30618).

REFERENCES

[1] H. Wang, H. Tian, J. Chen, X. Wan, J. Xia, G. Zeng, W. Bai, J. Jiang, Y. Wang, and K. Chen. Towards Domain-specific Network Transport for Distributed DNN Training. In Proc. USENIX NSDI, 2024.
[2] M. Li, D. G. Andersen, J. W. Park, et al. Scaling Distributed Machine Learning with the Parameter Server. In Proc. USENIX OSDI, 2014.
[3] Z. Jia, M. Zaharia, and A. Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In Proc. MLSys, 2019.
[4] D. Narayanan, A. Harlap, A. Phanishayee, et al. PipeDream: Generalized Pipeline Parallelism for DNN Training. In Proc. ACM SOSP, 2019.
[5] Y. Huang, Y. Cheng, A. Bapna, et al. Gpipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In Proc. NIPS, 2019.
[6] W. Xiao, R. Bhardwaj, R. Ramjee, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In Proc. USENIX OSDI, 2018.
[7] M. Jeon, S. Venkataraman, A. Phanishayee, et al. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In Proc. USENIX ATC, 2019.
[8] A. Paszke, S. Gross, S. Chintala, et al. Automatic Differentiation in Pytorch. In Proc. NIPS, 2017.
[9] K. Hsieh, A. Harlap, N. Vijaykumar, et al. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In Proc. USENIX NSDI, 2017.
[10] W. Bai, L. Chen, K. Chen, et al. Information-Agnostic Flow Scheduling for Commodity Data Centers. In Proc. USENIX NSDI, 2015.
[11] J. Hu, C. Zeng, Z. Wang, J. Zhang, K. Guo, H. Xu, J. Huang, K. Chen. Load Balancing with Multi-level Signals for Lossless Datacenter Networks. IEEE/ACM Transactions on Networking, 2024.
[12] L. Chen, J. Lingys, K. Chen, et al. Auto: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In Proc. ACM SIGCOMM, 2018.
[13] Y. Peng, Y. Zhu, Y. Chen, et al. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In Proc. ACM SOSP, 2019.
[14] Y. Ma, H. Wang, Y. Zhang, et al. AutoByte: Automatic Configuration for Optimal Communication Scheduling in DNN Training. In Proc. IEEE INFOCOM, 2022.
[15] J. Vanschoren. Meta-learning: A survey. arXiv preprint:1810.03548, 2018.
[16] Z. Bai, Z. Zhang, Y. Zhu and X. Ji. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In Proc. USENIX OSDI, 2020.
[17] J. Hu, Y. He, W. Luo, J. Huang , J. Wang. Enhancing Load Balancing with In-network Recirculation to Prevent Packet Reordering in Lossless Data Centers. IEEE/ACM Transactions on Networking, 2024.
[18] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. arXiv preprint:1409.1556, 2014.
[19] K. He, X. Zhang, S. Ren, et al. Deep Residual Learning for Image Recognition. In Proc. IEEE CVPR, 2016.
[20] K. Alex, I. Sutskever and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Proc. NIPS, 2012.
[21] P. Goyal, P. Dollár, R. Girshick, et al. Accurate, Large Minibatch SGD: Training Imagenet in 1 Hour. arXiv preprint:1706.02677, 2017.
[22] M. Abadi, P. Barham, J. Chen, et al. Tensorflow: A System for Large-Scale Machine Learning. In Proc. USENIX OSDI, 2016.
[23] T. Chen, M. Li, Y. Li, et al. Mxnet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint:1512.01274, 2015.
[24] A. Mirhoseini, H. Pham, Q. V. Le, et al. Device Placement Optimization with Reinforcement Learning. In Proc. ICML, 2017, pp. 2430-2439.
[25] Z. Jia, O. Padon, J. Thomas, et al. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In Proc. ACM SOSP, 2019.
[26] C. Chen, W. Wang and B. Li. Round-robin Synchronization: Mitigating Communication Bottlenecks in Parameter Servers. In Proc. IEEE INFOCOM, 2019.
[27] J. Xu, S. Huang, L. Song and T. Lan. Live Gradient Compensation for Evading Stragglers in Distributed Learning. In Proc. IEEE INFOCOM, 2021.
[28] C. Lao, Y. Le, K. Mahajan, et al. ATP: In-network Aggregation for Multi-tenant Learning. In Proc. USENIX NSDI, 2021.
[29] D. Alistarh, D. Grubic, J. Li, et al. QSGD: Communication-efficient SGD via Gradient Quantization and Encoding. In Proc. NIPS, 2017.
[30] S. Fan, Y. Rong, C. Meng, et al. DAPPLE: A pipelined data parallel approach for training large models. In Proc. ACM SIGPLAN, 2021.
[31] S. Li and T. Hoefler. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In Proc. ACM SCw, 2021.
[32] D. Narayanan, A. Phanishayee, K. Shi, et al. Memory-Efficient Pipeline-Parallel DNN Training. In Proc. PMLR, 2021.
[33] J. Hu, J. Huang, J. Wang, J. Wang.A Transmission Control Mechanism for Lossless Datacenter Network Based on Direct Congestion Notification. ACTA ELECTONICA SINICA, 51(9): 2355-2365, 2023.
[34] Narayanan, Deepak, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021.
[35] Kosson, Atli, et al. Pipelined backpropagation at scale: training large models without batches. Proceedings of Machine Learning and Systems, 3: 479-501, 2021.
[36] Yang, Bowen, et al. Pipemare: Asynchronous pipeline parallel dnn training. Proceedings of Machine Learning and Systems, 3: 269-296, 2021.