AutoPipe: A Fast Pipeline Parallelism Approach with Balanced Partitioning and Micro-batch Slicing

Weijie Liu, Zhiquan Lai, Shengwei Li, Yabo Duan, Keshi Ge, Dongsheng Li
National Laboratory for Parallel and Distributed Processing(PDL)
College Of Computer, National University Of Defense Technology
Changsha, China
{liuweijie,zqlai,yaboduan,gekeshi,dsli}@nudt.edu.cn, lucasleesw9@gmail.com

2022 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-6654-9856-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLUSTER51413.2022.00042

***Abstract*—Recently, pipeline parallelism has been widely used in training large DNN models. However, there are still two main challenges for efficient pipeline parallelism: i) a balanced model partition is crucial for pipeline efficiency, whereas prior works lack a sound solution to generate a balanced partition automatically. ii) the startup overhead is inevitable and especially significant for deep pipelines, which is an essential source of pipeline bubbles and severely affects pipeline scalability. We propose *AutoPipe* to solve these two problems, which contains i) a *planner* for automatically and quickly generating a balanced pipeline partition scheme with a fine-grained partitioner. This partitioner groups DNN in the sub-layer granularity and finds the balanced scheme with a heuristic search algorithm; and ii) a *micro-batch slicer* that reduces pipeline startup overhead according to the planner results by splitting the micro-batch evenly. This slicer automatically solves an appropriate number of micro-batches to split. The experimental results show that AutoPipe can accelerate training by up to 1.30x over the state-of-the-art distributed training framework Megatron-LM, with a 50% reduction in startup overhead and an order-of-magnitude reduction in pipeline planning time. Furthermore, AutoPipe Planner improves the partition balance by 2.73x-12.7x compared to DAPPLE Planner and Piper.**

***Index Terms*—artificial neural networks, distributed system**

I. Introduction

Deep neural networks (DNNs) have been successfully advanced in many domains [1]–[3]. In pursuit of higher accuracy, DNN models are getting larger. The parameter size of the state-of-the-art (SOTA) models has exponentially increased since Transformer [2] was presented [4], leading to the surge in model training time. To alleviate this problem, several distributed DNN training schemes are proposed. Data parallelism [5]–[7], a simple and widely used scheme, places multiple model replicas on different accelerators (*e.g.*, GPUs), and can scale the training to up to hundreds of GPUs [8]. However, it fails when recent transformer-based models [9]–[11] are too large to compute by a single GPU, due to the limited GPU memory.

To address the problem of speeding up large model training, pipeline parallelism [12]–[21] has been proposed. The key idea is to partition the model layers into stages and distribute each stage into different GPUs. Pipeline parallelism splits a mini-batch into micro-batches and makes stages process these micro-batches in a pipeline way during training. However, the bubbles (*i.e.*, idle GPU times) in pipeline parallelism downgrade the system's throughput, especially for transformer-based models [12], [22]. We observe two main reasons that result in the bubbles.

Firstly, the imbalanced model partition scheme leads to bubbles. When the workloads among stages are imbalanced, some GPUs are forced to wait for their following pipeline stages to complete, thus reducing the throughput of pipeline parallel training system [22]. DAPPLE Planner [12] tends to partition the model into a two-stage pipeline, and its model partition schemes are imbalanced. Piper [23] proposes a pipeline partition algorithm that automatically determines the pipeline partition scheme within a sampled search space, with a target of minimizing Time-Per-Sample (TPS) [23], [24]. However, it reduces the TPS by partitioning the model into more stages, making the pipeline inefficient. Finding a balanced pipeline partition scheme meets two non-trivial challenges. 1) It is difficult to achieve a balanced scheme with the layer-level partition. In transformer-based models, the size of embedding layers is large but not computationally intensive, unlike other layers. Therefore, the stages containing embedding layers have different workloads compared to others. 2) It is essential but difficult to search for a balanced partition scheme with low time costs. The search space of grouping a DNN with tens of layers is significantly ample. Thus, finding a balanced partition is a time-consuming task.

To address the challenges of balanced partitioning, we design *AutoPipe Planner*, an automatic pipeline planner that efficiently searches a balanced pipeline scheme by a fine-grained partitioner. Specifically, we design a pipeline partitioner that automatically finds a balanced sub-layer partition scheme with a heuristic partition search algorithm. The partitioner splits the layers in a transformer-based model into sub-layer granularity to explore more partition space without additional communication overhead. The pipeline partition scheme is assessed by a specially designed pipeline simulator. This simulator stores the intermediate information and reconstructs the critical path [16] by analyzing the complicated dependencies, thus providing an accurate iteration time. The iteration time and the critical path information are fed back to the partitioner. Then the partitioner evaluates the merit of the partition scheme according to the iteration time and heuristically generates a superior scheme by adjusting the partition scheme based on the critical path.

Secondly, the bubbles in pipeline startup are inevitable and

2168-9253/22/$31.00 ©2022 IEEE
DOI 10.1109/CLUSTER51413.2022.00042

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

especially significant for a deeper pipeline [20], [21]. Each time the pipeline processes the first micro-batch data, every stage except the first has to wait for the output from its previous stage, which leads to the startup overhead. Previous works have proposed some solutions to reduce pipeline startup overhead [16], [20]. For example, Megatron-LM [20] adopts an interleaved pipeline schedule that places multiple model chunks in a single GPU and assigns each GPU with multiple pipeline stages. However, the interleaved schedule damages the pipeline balance and thus harms the system throughput. Reducing the startup overhead with low cost is a challenge.

To tackle this challenge, we design *AutoPipe Slicer*, a micro-batch slicer that can halve the startup overhead without affecting pipeline balance or introducing additional memory consumption. The key idea is to evenly split a micro-batch into an appropriate number of pieces based on the partition scheme from AutoPipe Planner. We propose a slicer algorithm to solve the optimal number of micro-batches to be split.

Eventually, we design and implement *AutoPipe*, which integrates AutoPipe Planner and AutoPipe Slicer to speed up the pipeline parallelism DNN training. The AutoPipe first gets the runtime states of the DNN model and hardware configurations from model configs. Using these statistics, the AutoPipe Planner produces a balanced pipeline partition scheme. After that, the AutoPipe Slicer splits the specific micro-batches for the partition scheme to reduce the startup overhead, and generates the final solution used by distributed DNN training.

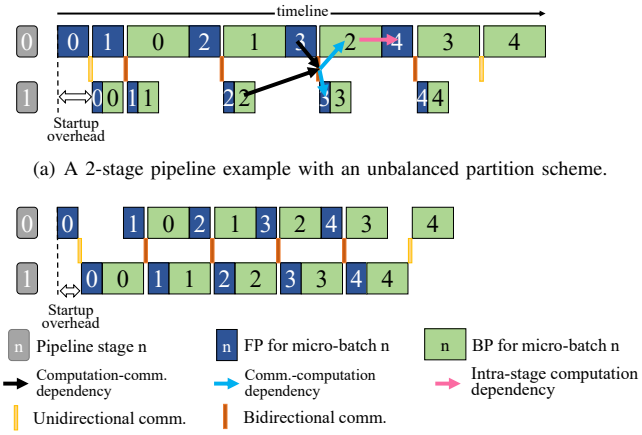
Our main contributions are summarized as follows:

- We designed a pipeline planner for pipeline parallelism of DNN training through an accurate pipeline simulator and fine-grained pipeline partitioner. The simulator can obtain accurate pipeline iteration times and reconstruct the critical path. With the feedback of the simulator, the pipeline partitioner heuristically searches for a balanced partition scheme in the sub-layer granularity.
- We designed a micro-batch slicer to reduce the pipeline startup overhead without extra GPU memory cost. The slicer solves an optimal micro-batch splitting solution and reschedules the Warmup phase that can halve the startup overhead of the pipeline.
- We implement the pipeline planner and micro-batch slicer and integrate them into AutoPipe. We conduct evaluations on a 16 GPUs cluster and compare AutoPipe with 4 SOTA baselines on the representative large models. Experimental results show that AutoPipe achieves up to 1.30x speedup while reducing the startup overhead by half and improves pipeline balance by at least 2.73x.

II. Background

A. DNN Training

Due to its universality and excellent performance in various tasks, the DNNs, especially transformer-based models [9]–[11], [25], [26], have dramatically developed in recent years. Training such a model requires a large dataset and a loss function [27]. The dataset is divided into mini-batches and input



(a) A 2-stage pipeline example with an unbalanced partition scheme.

(b) A 2-stage pipeline example with a balanced partition scheme.

Fig. 1. Balanced and unbalanced pipelines with the same load, the balanced scheme significantly reduces the GPU idling time and requires less training time. And three kinds of dependencies exist in the pipeline, as shown in (a), which are complicated but vital to ensure the consistency between distributed pipeline running and single machine running. Furthermore, the startup overhead of pipelines is inevitable at the beginning of every mini-batch.

iteratively into the model to reduce loss value until the model converges. There are three phases in each training iteration: (1) forward propagation (FP): a mini-batch is passed through the model layer by layer to produce a loss. (2) backward propagation (BP): gradients of parameters are computed in the reverse order of FP. (3) parameter update: gradients are applied to model parameters by an optimizer, such as Adam [28].

B. Synchronous Pipeline Parallelism

Synchronous pipeline parallelism [12], [16], [17] is a common approach to training large models, which divides the mini-batch into micro-batches and feeds accelerators with micro-batches in a pipeline way so that multiple accelerators work simultaneously. Moreover, synchronous pipeline parallelism does not affect model convergence [21]. Three kinds of dependencies exist in pipeline parallelism: intra-stage computation, communication-computation and computation-communication. We illustrate these dependencies in Fig. 1 (a). These dependencies are complicated but vital to ensure the consistency between distributed pipeline running and single machine running. Fig. 1 also shows pipeline parallelism's two kinds of communication: unidirectional and bidirectional. Since accelerators can send and receive simultaneously, and the communication volume between pipeline stages is too tiny to saturate the network bandwidth, bidirectional communication (bidirectional comm.) costs do not increase with the doubled volumes compared to unidirectional communication (unidirectional comm.). The startup overhead affects the scalability of pipeline parallelism significantly. As shown in Fig. 1 (a), the *startup* time is inevitable at the beginning of every mini-batch, where each stage processes the first micro-batch. The startup overhead of pipeline parallelism is the time spend by the last pipeline stage for receiving activations of the

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

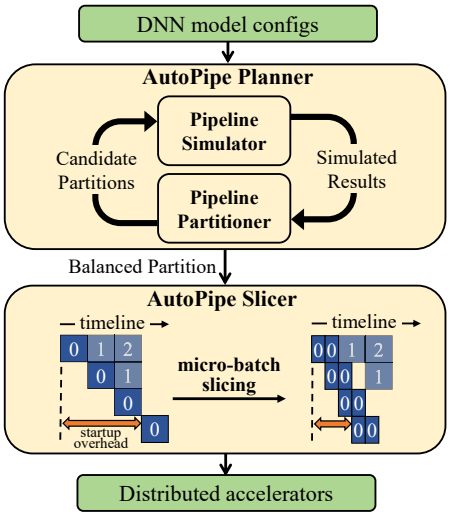


Fig. 2. AutoPipe overview. AutoPipe consists of a Planner and a Slicer. The Planner uses model configs as input and generates a superior pipeline partition scheme. Then the Slicer reschedules the Warmup phase using micro-batch slicing to halve the startup overhead. Results generated by the Planner and Slicer are used for distributed training.

first micro-batch and is proportional to the number of pipeline stages, making a deeper pipeline inefficient.

We demonstrate an example of the balanced and imbalanced pipeline in Fig. 1. Compared with the imbalanced partition scheme (Fig. 1 (a)), the balanced scheme (Fig. 1 (b)) significantly reduces the GPU idling time, and requires less training time. This motivates us to speed up pipeline training by searching for a balanced pipeline partition scheme.

C. Activation Checkpoint

Activation checkpoint [14], [29]–[31] is a commonly adopted technique to trade the training speed for memory footprint by large model training. This technique dramatically reduces the memory footprint during training by stashing only a tiny part of activations after FP, as the entire set of intermediate activation is much larger. When using the activation checkpoint, FP will be executed for the second time before BP, and whole activations will be restored according to the activations stashed earlier to ensure the correctness of the whole process. We use this technique to reduce memory consumption since the whole intermediate activation of transformer-based models requires significant memory.

III. The AutoPipe Approach

A. Overview

Fig. 2 shows the overview of AutoPipe, mainly including a Planner and a Slicer. Model configs contain both configurations and runtime statistics of a given DNN model, which can be collected offline within several minutes. AutoPipe Planner uses the model configs as input and consists of a pipeline simulator and a pipeline partitioner. The pipeline simulator can estimate the iteration time of a pipeline partition scheme according to the input data and obtain relevant information about the pipeline. And the pipeline partitioner can generate potential pipeline partition schemes heuristically according to the simulator's output and current partition scheme until finding the optimal partition scheme. Then AutoPipe Slicer halves the startup overhead by slicing specific micro-batches and rescheduling their FPs according to the Planner output. Finally, results generated by the Planner and the Slicer are used by distributed accelerators (*e.g.*, GPUs) for synchronous pipeline training.

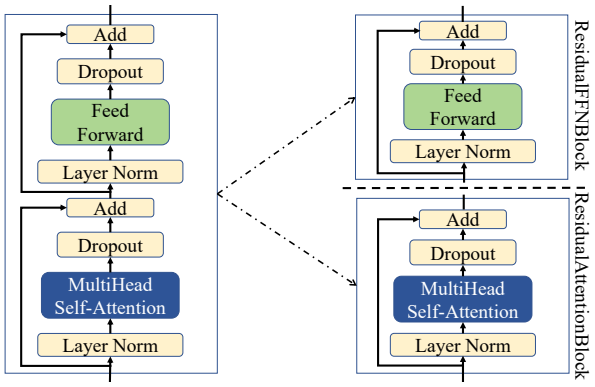


Fig. 3. Sub-layer granularity of transformer layers. By analyzing the structure of the transformer layer, we split it into ResidualFFNBlock and ResidualAttentionBlock, refining the granularity of pipeline planning without increasing the communication overhead between pipeline stages and optimizing the balance of potential partition schemes while increasing the search space of pipeline planning.

B. AutoPipe Planner

Searching for a balanced pipeline partition scheme in layer granularity is an NP-hard problem [32] due to its ample search space. AutoPipe Planner searches with sub-layer granularity to find a balanced partition scheme, which doubles the search space. To reduce the search time, the Planner uses the critical path. Specifically, AutoPipe Planner consists of a simulator and a partitioner. The simulator gets the iteration time and critical path from the partition scheme. With the help of critical path, the partitioner generates potential partition schemes efficiently using a heuristic approach. The Planner generates a balanced pipeline partition scheme by calling the simulator and partitioner iteratively.

Sub-layer granularity. Previous research on transformer-based model training was planning a pipeline on layer granularity. Such planning granularity can make the pipeline balanced when only considering transformer layers. However, once other layers are added for consideration, it is hard to divide the model evenly across pipeline stages. In order to improve the balance of pipeline partition schemes, sub-layer granularity is necessary. However, there are two drawbacks when using sub-layer granularity: additional communication overhead and multiply search spaces of pipeline partition schemes. Additional communication overhead makes the pipeline execution complex as the communication volume

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

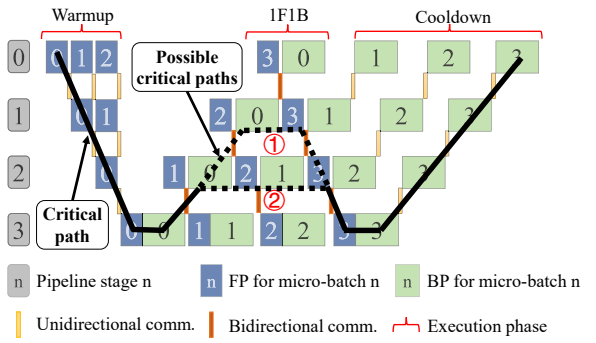


Fig. 4. Example for multiple critical paths. There are multiple critical paths if we just borrow the critical path definition from the AOE network. For example, the entire critical path of the pipeline can be the combination of the determinate part (solid line) and one of the possible parts (dotted lines).

between pipeline stages varies in finer granularity, and multiply search spaces make finding a balanced pipeline partition scheme unacceptable. By analyzing the structure of the transformer layer, we split it into two parts. Fig. 3 shows the sub-layer granularity we use, which divides transformer layers into ResidualAttentionBlocks and ResidualFFNBlocks. Although these two parts are different in computation time and memory consumption, this granularity can improve the balance of the pipeline partition schemes while keeping the communication volume between pipeline stages the same as layer granularity, thus improving pipeline throughputs. Besides, the sub-layer granularity will not affect the model convergence since it does not modify the model's overall structure.

Critical path and master stage. We can construct an AOE network [33] for operations in a pipeline by considering the FP and BP as edges with a weight value of execution time. Then we can define the critical path of the pipeline as the longest path starting from the first FP to the end of the last BP. However, multiple critical paths can exist in this way. As illustrated in Fig. 4, there are two possible critical paths that can be combined with the exclusive path to become a full critical path. This situation occurs because there are paths of equal length that pass through different pipeline stages in the 1F1B phase. To ensure the uniqueness of the critical path, we define the critical path as the one closest to the last pipeline stage in the 1F1B phase among all the longest paths. Therefore, the critical path in Fig. 4 should be the path made of the solid line and the possible critical path two. We then define the stage that the critical path passes in 1F1B phase as the master stage since it has the heaviest load and dominates the pipeline in 1F1B phase through its FP and BP operations. For example, as shown in Fig. 4, the FP for micro-batch 2 in the last stage is driven by the corresponding FP in stage 2, causing a bubble in the last stage.

1) Pipeline Simulator: The simulator simulates the pipeline execution according to the partition scheme and model configs and gets the simulated iteration time and critical path by analyzing the complicated pipeline dependencies.

Fig. 5 shows a balanced three-stage pipeline with six

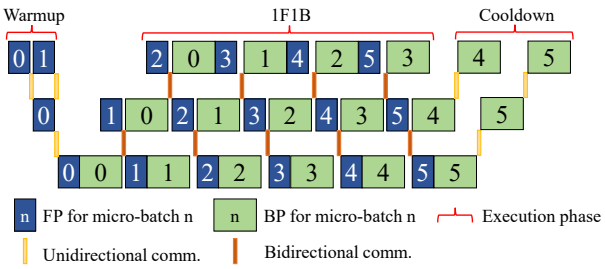


Fig. 5. Three phases of pipeline execution per iteration. Only FP or BP operation is performed with unidirectional communication during the Warmup and Cooldown phases, while FP and BP operations are executed with bidirectional communication alternatively in the 1F1B phase.

micro-batches. Pipeline execution can be divided into three phases: Warmup, 1F1B, and Cooldown. The dependencies in the pipeline are intricate. For example, in Fig. 5, both computation-communication dependency and communication-computation dependency exist between the FP operations of each micro-batch on different stages, as well as the BP operations of each micro-batch, and intra-stage computation dependency takes effect between adjacent operations on the same stage. These dependencies guarantee consistency in distributed pipeline running and single machine running.

Dependencies in the Warmup phase are straightforward and will not affect the overall pipeline time, thus we estimate the Warmup phase overhead with the total FP time of one micro-batch. However, processes in the 1F1B and Cooldown phases are more unstable, we need to consider these phases specifically. By renumbering the micro-batches of the 1F1B and Cooldown phases, the simulator can unify the complex pipeline dependencies, which is convenient for us to simulate the pipeline execution. Fig. 6 shows the 1F1B and Cooldown phases in Fig. 5 after renumbering. For the 1F1B phase in Fig. 6, we treat adjacent FP and BP operations on each accelerator as a block and assign the same number to operations within a block. The number of blocks is determined by the sequence number of pipeline stages and the total number of micro-batches per iteration. For example, for a n-stage pipeline that processes m micro-batches per iteration, the number of blocks owned by the accelerator at stage k is $max(0, m-n+k+1)$. Further, we define $t(x,y,z)$ as the start time of the z-th operation of block y in stage x. There are two values for z: 0 and 1, representing FP and BP, respectively. Besides, we define f_x, b_x as the duration of FP, BP operation for stage x, and $Comm$ as the costs of a single communication operation since both unidirectional and bidirectional communication have almost the same costs in our experimental environment. We can express pipeline dependencies in the 1F1B phase as:

$$t(x,y,0) = max(t(x-1,y-1,0)+f_{x-1}, t(x,y-1,1)+b_x)$$
$$t(x,y,0) = t(x,y,0)+Comm, \quad if\ x \neq 0$$

$$t(x,y,1) = max(t(x+1,y,1)+b_{x+1}, t(x,y,0)+f_x)$$
$$t(x,y,1) = t(x,y,1)+Comm, \quad if\ x \neq n-1$$

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

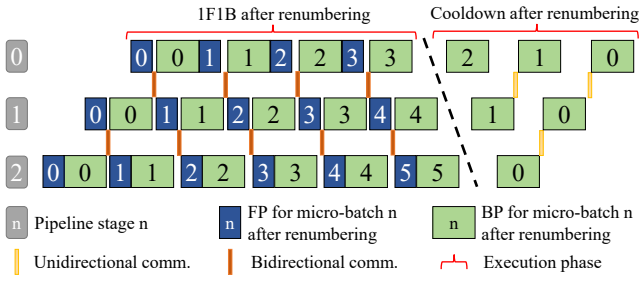


Fig. 6. Renumbering of the 1F1B and Cooldown phases. We regard adjacent FP and BP operations in the 1F1B phase as a block, numbering these blocks from 0 and assigning the same number to the operations inside a block. For the Cooldown phase, we treat each BP operation as a block and number them in the reverse order.

For the Cooldown phase, we renumber the micro-batches in the opposite order from the original as shown on the right side of Fig. 6. Here we define $t(x, y)$ as the start time of renumbered y-th BP on stage x, while the definitions of $Comm$ and b_x are the same as before. Then we can represent the pipeline dependencies in the Cooldown phase as:

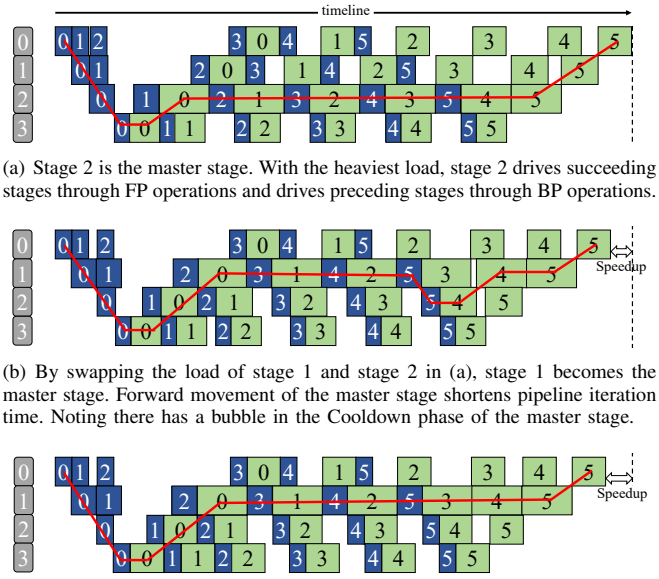
$$t(x,y) = max(t(x,y+1)+b_x, t(x+1,y)+b_{x+1})+Comm$$

Processing of the first micro-batch in the pipeline is hardly choked due to the balanced partition scheme generated by the partitioner, and we can quickly obtain the first block's start time in the 1F1B phase with the Warmup phase overhead. Combining these initial conditions with the formulas above, we can get the start time of all FP and BP operations. Furthermore, we reconstruct the critical path of the partition scheme and get the master stage by analyzing the results.

2) Pipeline Partitioner: The partitioner splits models with sub-layer granularity, attempting to improve the balance of pipeline partition schemes with the expanded search space.

Heuristic pipeline planning algorithm. Using sub-layer granularity implies doubling the search space of pipeline planning. To reduce the search time of pipeline planning with expanded search space, we propose a heuristic pipeline planning algorithm that finds potential partition schemes by moving the master stage. The range of the master stage is the pipeline depth, which is relatively small compared to the number of devices used for training. *e.g.*, Megatron-LM [20] uses 3072 accelerators to train a GPT model with one trillion parameters, but its pipeline depth is only 64. Therefore, the search time for pipeline planning can be acceptable by combining sub-layer granularity and the master stage. We introduce Algorithm 1 first since it takes effect in the heuristic algorithm. Using the model blocks array $model$ and the pipeline depth p as input, Algorithm 1 generates a relative balance partition scheme for $model$ with p stages through dynamic programming.

The heuristic rules of the heuristic algorithm are based on two findings when we move the master stage: 1) By comparing Fig. 7(a) and Fig. 7(b), we see that the iteration time of the pipeline decreases when the master stage moves forward (towards a lower stage). 2) By comparing Fig. 7(b) and Fig. 7(c),



(a) Stage 2 is the master stage. With the heaviest load, stage 2 drives succeeding stages through FP operations and drives preceding stages through BP operations.

(b) By swapping the load of stage 1 and stage 2 in (a), stage 1 becomes the master stage. Forward movement of the master stage shortens pipeline iteration time. Noting there has a bubble in the Cooldown phase of the master stage.

(c) By redistributing the load of stage 2 and stage 3 in (b), the bubble in the Cooldown phase of the master stage vanishes and pipeline iteration time is further decreased.

Fig. 7. The impact of the master stage and the partition scheme of subsequent stages after the master stage on pipeline iteration time. Though the overall load is the same, the load distribution differs, resulting in the variation in pipeline iteration time.

we find that when the master stage is the same, assuming it is the i-th stage, the workload distribution among pipeline stages after stage i can affect the pipeline iteration time. With the same inputs as Algorithm 1, the heuristic algorithm outputs a balanced partition scheme with p stages. Assuming stage i is the master stage for the processing partition scheme, and array $model$ is represented by ResidualFFNBlocks and ResidualAttentionBlocks, the specific process of the heuristic algorithm is shown as follows:

1) An initial relatively balanced partition scheme for the whole model is obtained from Algorithm 1. The pipeline simulator obtains the master stage and iteration time for the partition scheme.
2) To eliminate pipeline bubbles in the Cooldown phase, we adjust the partition scheme after stage i such that for any s greater than i, we have:

$$\sum_{j=i+1}^{s} f_j + \sum_{j=i+1}^{s} b_j \leq (s-i)b_i \tag{1}$$

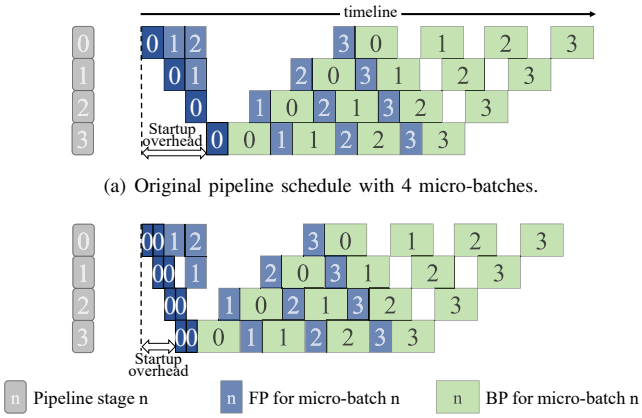
   If the master stage changes after the adjustment, stop the adjustment and go to 3) with the adjusted partition scheme.
3) If i is equal to 0, stop the heuristic algorithm for the processing partition scheme. Otherwise, shifting the master stage forward by moving the first block of stage i to stage $i-1$ or moving the last block of stage i to stage $i+1$. Specifically, the partitioner applies Algorithm 1 to the first $i-1$ stages when moving the first

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Algorithm to get a relatively balanced partition scheme

Input: Array $Model$, pipeline depth p
Output: Array $partition$

1 Get forward time f_i and backward time b_i for block i in array $Model$
2 Set n as the number of blocks in the array $Model$, $Comm$ as a single communication overhead between stages
3 Create array $prefix\_sum[n+1]$ and array $time[n+1][min(p,n)+1]$
4 $prefix\_sum[0] = 0$
5 **for** $i \leftarrow 1$ **to** n **do**
6 $\quad prefix\_sum[i] \leftarrow prefix\_sum[i-1] + f_i + b_i$
7 **end**
8 **for** $i \leftarrow 1$ **to** n **do**
9 $\quad$ **for** $j \leftarrow 1$ **to** $min(p,i)$ **do**
10 $\quad\quad$ **for** $k \leftarrow 0$ **to** $i-1$ **do**
11 $\quad\quad\quad time[i][j] = min(time[i][j], max(time[k][j-1], prefix\_sum[i] - prefix\_sum[k]))$
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 **end**
15 Reconstruct the result $partition$ array based on $time$



(a) Original pipeline schedule with 4 micro-batches.

(b) Pipeline schedule with 4 micro-batches after slicing.

Fig. 8. The AutoPipe Slicer halves the startup overhead by slicing the optimal number of micro-batches. Here we show an example of halving the startup overhead of a 4-stage pipeline by slicing micro-batch 0.

block of stage i to stage $i-1$, and applies Algorithm 1 to the first i stages when moving the last block of stage i to stage $i+1$. By combining partition schemes with and without the application of Algorithm 1, the partitioner generates potential partition schemes and puts them into the pipeline simulator to get the master stage and iteration time. Partition schemes with a master stage less than or equal to i will be processed by 2).

4) Return the partition scheme with minimum iteration time.

C. AutoPipe Slicer

The AutoPipe Slicer reschedules the Warmup phase using micro-batch slicing to halve the startup overhead. Fig. 8(b) shows an example of a 4-stage pipeline after rescheduling. By applying micro-batch slicing to micro-batch 0, the Slicer reduces the startup overhead by half, and the iteration time decreases due to the reduction of pipeline bubbles. However, micro-batch slicing doubles the FP operations' communication numbers and increases the possibility of pipeline blockage. For example, in Fig. 8(b), once micro-batch 1 is sliced, the communication of the first half will be blocked at stage 2 since stage 3 is busy at that time, delaying the computation of second half and increasing the iteration time. The blockage occurs at the Warmup phase's last FP operation over all stages. To address the issue of these micro-batches, we cancel the communication of first half and aggregate it with the communication of second half. Moreover, applying micro-batch slicing to all micro-batches in the Warmup phase is unnecessary since most of them are inoperative for startup overhead reduction.

Since the optimal number of micro-batches to be split is related to the partition scheme that contains pipeline depth and load distribution across stages, we design Algorithm 2 to solve it automatically. Using the partition scheme output by AutoPipe Planner as input, Algorithm 2 shows the detail of the finding process. Array $startt$ records the start time of the first FP operation in 1F1B phase for each stage, and array $endt$ records the end time of FP operations for split micro-batches of each stage. We set the same number to split micro-batches as shown in Fig. 8 (b) and use $endt[i][0]$ and $endt[i][1]$ represent the end time of the first half and the second half of split micro-batches for stage i. The algorithm first initializes array $startt$ (lines 4-15), then it calculates the end time of the split micro-batch according to pipeline dependencies (lines 17-28) and the start time of the unbroken micro-batch on stage 0 by using the $startt$ array (lines 29-33), once the start time of the unbroken micro-batch is greater than or equal to the end time of second half of the split micro-batch, the algorithm returns the number of split micro-batches (lines 34-36). Otherwise, the algorithm splits the next micro-batch (line 37).

IV. Evaluation

A. Experimental Setup

Implementation. AutoPipe is implemented as a pipeline front-end based on PyTorch [34]. We develop the Planner and Slicer algorithms using C++ and integrate them into AutoPipe. We use the runtime of Megatron-LM as the pipeline back-end, while adopting NCCL [35] for inter-device communication.

Benchmarks. Table I summarizes the four representative DNN models that we use as benchmarks. The training datasets are Wikipedia, BookCorpus [36], [37], and OpenWebText [38]. For all experiments, we use the activation checkpoint technique to avoid Out-of-Memory (OOM) errors. And we do not conduct experiments on convergence because the method used by AutoPipe does not affect the model convergence.

Experimental platform. All the experiments are conducted on a 4-node cluster. Each node has 4 NVIDIA 3090 GPUs (24GB memory), 2 Intel Xeon CPUs (2.40GHz), and 96GB DDR4 host memory. The nodes are connected via 100Gbps InfiniBand [39]. All nodes in the cluster run 64-bits Ubuntu18.04

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: The slicing algorithm of AutoPipe Slicer

Input: Array $partition$
Output: Number of sliced micro-batches mb

Get forward time f_i and backward time b_i for block i in $partition$
Set p as the number of blocks in $partition$, $Comm$ as a single communication overhead between stages
Create array $startt[p]$, array $endt[p+1][2]$, and variable $tempt, mb$
Initial elements in $startt, endt$ to 0
$tempt = 0, mb = 1$
for $i \leftarrow 0$ **to** $p-2$ **do**
    $tempt = tempt + f_i/2 + Comm/2$
end
$tempt = tempt + f_{p-1}/2$
for $i \leftarrow p-1$ **to** 1 **do**
    $tempt = tempt + b_i + Comm$
    $startt[p-1-i] = tempt$
end
$tempt = tempt + b_0$
$startt[p-1] = tempt$
while $true$ **do**
    for $i \leftarrow 0$ **to** $p - mb$ **do**
        for $j \leftarrow 0$ **to** 1 **do**
            $endt[i][j] = endt[i][(j+1)\%2] + f_i/2$
            if $i > 0$ **then**
                $endt[i][j] = max(endt[i][j], endt[i-1][j] + f_{i-1}/2)$
            end
            if $i \neq p-1$ **then**
                $endt[i][j] = endt[i][j] + Comm/2$
            end
            $endt[i][j] = max(endt[i][j], endt[i+1][(j+1)\%2]$
        end
    end
    $tempt = startt[mb-1]$
    for $i \leftarrow p-1-mb$ **to** 1 **do**
        $tempt = tempt - f_i - Comm$
    end
    $tempt = tempt - f_0$
    if $tempt \leq endt[0][1]$ **then**
        $return\ mb$
    end
    $mb = mb + 1$
end

TABLE I
BENCHMARK MODELS.

Model	# layers	Hidden size	# params (millions)
GPT-2 345M [26]	24	1024	345
GPT-2 762M	36	1280	762
GPT-2 1.3B	24	2048	1314
BERT-large [25]	24	1024	340

with CUDA 11.3 [40], cuDNN 8.2.1 [41], NCCL 2.10.3, and PyTorch 1.10.0.

B. Overall Performance

To evaluate the overall performance of AutoPipe, we compare pipeline iteration time with **Megatron-LM**, the SOTA distributed training framework, which evenly divides transformer layers into each pipeline stage. The costs of both the Warmup and Cooldown phases are counted in the iteration time. As shown in Fig. 9 and Fig. 10, AutoPipe achieves 1.02x-1.30x speedups over Megatron-LM.

TABLE II
PIPELINE PLANNING OF THE GPT-2 345M MODEL.

Partition ID	Number of layers in pipeline stages			
	stage 0	*stage 1*	*stage 2*	*stage 3*
1	5	7	6	6
2	6	6.5	6.5	5
3	6	7	6	5
4	6.5	6.5	6.5	4.5
5	6.5	6.5	6	5
6	7	5.5	6	5.5
7	7	6.5	5.5	5

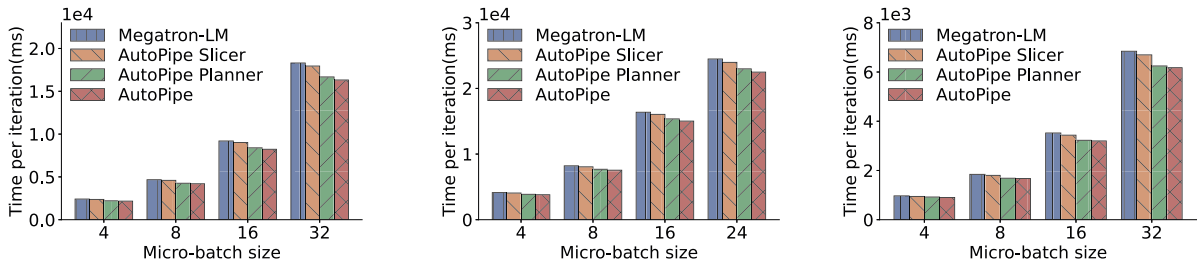
Fig. 9 shows the results under different micro-batch sizes when we fix the number of pipeline stages to 4 and run 8 micro-batches per iteration. Since GPT-2 762M has an OOM error when the micro-batch size is 32, the maximum micro-batch size we use for GPT-2 762M is 24. AutoPipe gains 1.07x-1.12x overall speedups over Megatron-LM. Slicer gains reasonable speedups up to 1.03x. By redistributing the load across pipeline stages, AutoPipe Planner gains 1.05x-1.10x speedups compare to Megatron-LM. AutoPipe outperforms Megatron-LM with 1.12x speedup by combining its Planner and Slicer. In addition, for all models, the speedup of AutoPipe becomes more significant as the micro-batch size gets larger.

To verify the effectiveness of AutoPipe, we make the Planner search for partition schemes for a specific number of stages and fix the number of micro-batches to twice the pipeline depth for experiments in Fig. 10. We set the micro-batch size to 4 for GPT-2 345M and GPT-2 762M, while 16 for BERT-large. As Megatron-LM requires the pipeline depth to be a factor of model layer number, GPT-2 762M uses the 9-stage pipeline instead of the 8-stage pipeline used in other models. AutoPipe gains 1.02x-1.30x overall speedups over Megatron-LM. The Slicer increases the iteration time when pipeline depth is 2, indicating that micro-batch slicing is unsuitable for a shallow pipeline. It achieves a 1.04x speedup with a deeper pipeline compared to Megatron-LM. AutoPipe Planner reduces iteration time in all cases and achieves 1.25x speedups compared to Megatron-LM when training BERT-large with a 12-stage pipeline. Moreover, the performance improvement of AutoPipe is more evident as the pipeline stage increases.

C. Pipeline Simulator of AutoPipe Planner

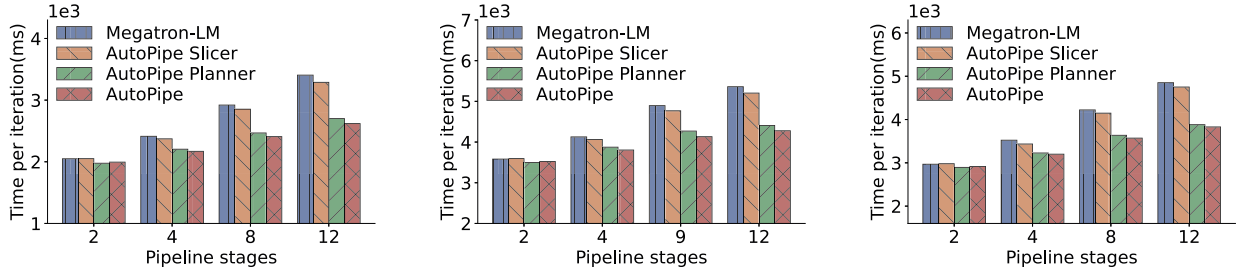
We evaluate the performance of the pipeline simulator using a 4-stage pipeline. Table II shows the seven different partition schemes for the GPT-2-345M model. Depending on the partition scheme, the decimal part of data in the table may represent a ResidualFFNBlock or a ResidualAttentionBlock. For schemes in Table II, we compare the execution time per micro-batch obtained from the pipeline simulator and from the actual case.

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.



(a) GPT-2 345M, 1.11x-1.12x (b) GPT-2 762M, 1.08x-1.09x (c) BERT-large, 1.07x-1.11x

Fig. 9. Iteration time for different configurations with different micro-batch sizes. The lower the better.



(a) GPT-2 345M, 1.03x-1.30x (b) GPT-2 762M, 1.02x-1.25x (c) BERT-large, 1.02x-1.27x

Fig. 10. Iteration time for different configurations at different pipeline stages. The lower the better.

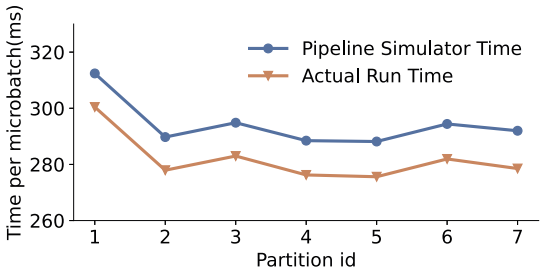


Fig. 11. The performance of the pipeline simulator under 7 different model partition schemes in Table II. Though there is somewhat bias between the two fold lines, the trend of both lines is the same and the gap between them is relatively stable.

TABLE III
PLANNER COMPARISON WITH LOW MEMORY DEMAND.

Model	Mbs	# of GPUs	Alg.	Time per iteration(ms)		
				Gbs=128	Gbs=256	Gbs=512
GPT-2 345M	4	4	D	11091	19925	39815
			P	6755.6	**12842**	**25421**
			A	**6510.5**	12852	25737
		16	D	-	-	-
			P	2081.1	4066.0	**7216.0**
			A	**2077.6**	**4063.7**	7236.5

[1] Mbs represents micro-batch size; Gbs represents global batch size; Alg. represents different planning algorithms.
[2] D represents DAPPLE Planner; P represents Piper; A represents AutoPipe Planner.
[3] - represents the runtime error.

As shown in Fig. 11, the simulator results are somewhat biased compared to the actual run results. However, for different partition schemes, the trend of simulator results and actual run results is the same, and the gap between the two results is also relatively stable. This indicates that using the simulator results for pipeline planning is reasonable.

D. Planner Comparison with DAPPLE and Piper

DAPPLE Planner and Piper are SOTA works of pipeline planning. This section compares them with AutoPipe Planner under a specific number of GPUs. We evaluate DAPPLE Planner and Piper by applying corresponding algorithms' results to Megatron-LM. AutoPipe automates the pipeline planning, and its data-parallel size is the number of GPUs over the pipeline stages. Besides, AutoPipe combines data and pipeline parallelism in the way Megatron-LM uses.

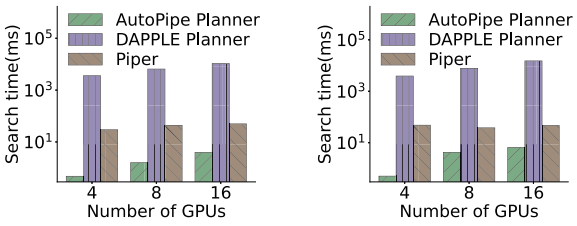
Table III shows the comparison of iteration time for DAPPLE Planner, Piper, and AutoPipe Planner for GPT-2 345M with micro-batch size 4. As the memory demand is low, both Piper and AutoPipe Planner use complete data parallelism, making the results similar. However, DAPPLE Planner uses a 2-stage pipeline, making the result worse than others with 4 GPUs. Since DAPPLE allows different pipeline stages to use different data parallelism sizes and prefers to use larger data parallelism sizes in the second pipeline stage. When using

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
PLANNER COMPARISON WITH HIGH MEMORY DEMAND.

Model	Mbs	# of GPUs	Alg.	Time per iteration(ms)		
				Gbs=512	Gbs=1024	Gbs=2048
GPT-2 345M	32	4	D	27477.3	53539.9	105635
			P	30285.9	56399.5	108694
			A	**27020.9**	**50969.7**	**99110.9**
		8	D	15828.6	30853.7	60898.3
			P	18143.6	31901.0	60397.7
			A	**15445.3**	**27147.5**	**51217.4**
GPT-2 1.3B	16	4	D	OOM	OOM	OOM
			P	67087.2	128831	253052
			A	**62885.0**	**119788**	**234781**
		8	D	OOM	OOM	OOM
			P	36783.1	70017.3	137135
			A	**34512.7**	**62869.1**	**120732**

[1] Mbs represents micro-batch size; Gbs represents global batch size; Alg. represents different planning algorithms.
[2] D represents DAPPLE Planner; P represents Piper; A represents AutoPipe Planner.

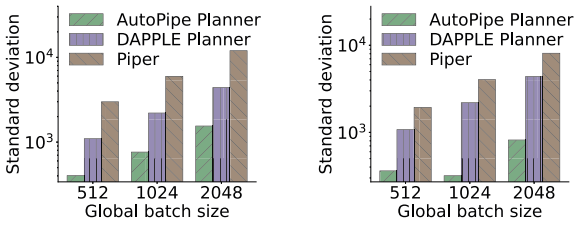


(a) GPT-2 345M. (b) GPT-2 1.3B.

Fig. 12. Search time of different algorithms on different models.

16 GPUs, DAPPLE uses a data parallelism size of 15 in the second stage, causing the number of devices in the second pipeline stage to exceed the micro-batch size, which leads to runtime errors.

Table IV shows the results of planner comparison with high memory demand, where all planning algorithms adopt pipeline parallelism to run the model. AutoPipe outperforms DAPPLE Planner and Piper for all cases. For GPT-2 345M with micro-batch size 32, both DAPPLE Planner and AutoPipe Planner use a 2-stage pipeline. However, DAPPLE Planner tends to assign more load to stage 2, unbalancing the pipeline. The results show that AutoPipe Planner achieves 1.19x speedups compared to DAPPLE Planner. Since Piper adopts a pipeline with more than 2 stages and distributes the load unbalanced, the performance is relatively worse than others. AutoPipe Planner achieves 1.18x speedups over Piper. For GPT-2 1.3B with micro-batch size 16, DAPPLE Planner occurs an out-of-memory error as it adopts a 2-stage pipeline. When using 4 GPUs, both Piper and AutoPipe Planner use a 4-stage pipeline. However, the load distributions generated by Piper are unbalanced, making the result worse than AutoPipe Planner. Moreover, Piper adopts a 5-stage pipeline when using 8 GPUs. Since AutoPipe Planner uses a 4-stage pipeline and distributes the load more balance than Piper, AutoPipe Planner outperforms Piper with 1.07x-1.14x speedups.



(a) 4 GPUs. (b) 8 GPUs.

Fig. 13. Balance comparisons based on GPT-2 345M with micro-batch size 32.

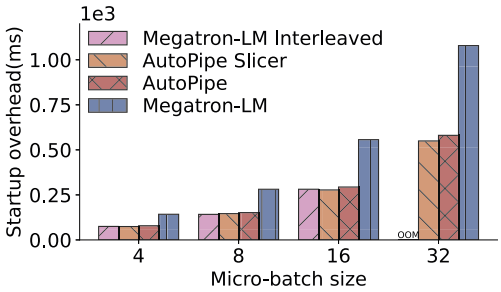
Fig. 12 shows the comparison of search time for different algorithms. The time cost is obvious since DAPPLE's planner is implemented in Python, and its search space is the largest due to the search for device placement. Since we use a homogeneous environment and the speed for intra-device and inter-device communication is almost identical, device assignment is unnecessary. Both Piper and AutoPipe are implemented in C++. However, Piper increases its search space by considering different pipeline stages using different data parallelism sizes, making it necessary to search for the number of devices in the data parallelism dimension. AutoPipe, on the one hand, omits the search in the data parallelism dimension by using the same data parallelism size for each pipeline stage; on the other hand, it reduces the search space significantly by applying heuristic adjustment to the master stage. The experimental results show that the AutoPipe search time can be an order of magnitude faster than Piper.

E. Ablation Study

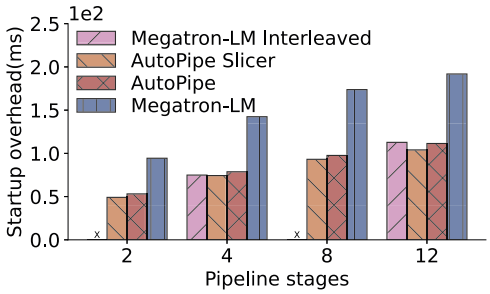
We conduct ablation experiments on AutoPipe by testing the balance improvement of AutoPipe Planner and the effect of AutoPipe Slicer on reducing the startup overhead, respectively.

1) Balance Analysis: We perform balance comparisons based on the results of GPT-2 345M in Table IV, results are shown in Fig. 13. To show the effect of AutoPipe Planner, we use the standard deviation among the running time of each pipeline stage as a criterion for judging the balance of a partition scheme and compare the balance of DAPPLE Planner, Piper, and AutoPipe Planner. Compared to DAPPLE Planner and Piper, AutoPipe Planner achieves a 2.73x-12.7x improvement in balance. Both DAPPLE Planner and AutoPipe Planner use a 2-stage pipeline. However, DAPPLE Planner tends to put more load into stage 2, while AutoPipe distributes the load evenly. For example, when using 4 GPUs, DAPPLE Planner assigns 17 layers to stage 2 for a 24-layer GPT-2 345M, while AutoPipe places 11 layers to stage 2. Thus AutoPipe Planner achieves a 2.73x-6.89x improvement in balance compared to DAPPLE Planner. Since Piper tends to use pipelines with more stages (e.g., 4 stages for 4 GPUs and 6 stages for 8 GPUs), the balance is worst among the three planning algorithms. AutoPipe Planner achieves a 5.35x-12.7x improvement in balance over Piper.

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.



(a) Startup overhead on micro-batch size. The OOM indicates that the method represented by the column will have an out-of-memory error when running with the corresponding configuration.



(b) Startup overhead on pipeline stages. The X represents that the method represented by the column cannot run the model properly with the corresponding configuration.

Fig. 14. Startup overhead comparison.

2) Startup Overhead Reduction: We compare the startup overhead based on the GPT-2 345M. And We use a 4-stage pipeline to compare the startup overhead in Fig. 14 (a) and fix the micro-batch size to 4 in Fig. 14 (b). Megatron-LM reduces the startup overhead by using the interleaved schedule that places multiple model chunks in a single GPU and assigns each GPU with multiple pipeline stages. We find that both AutoPipe Slicer and the interleaved schedule halve the startup overhead compared to Megatron-LM. However, as the interleaved schedule demands more GPU memory to store activations than Megatron-LM, it gets the out-of-memory error with a large micro-batch size. Moreover, the interleaved schedule requires an even number of model blocks per pipeline stage, making it unable to work properly with some pipeline depths for a specific model. Compared to the interleaved schedule, the Slicer and AutoPipe work well under all configurations. The startup overhead of AutoPipe is slightly larger than the Slicer because AutoPipe moves the load of the last pipeline stage forward to balance the pipeline.

V. Related Work

The outstanding performance of large models on many tasks has led to many concerns about large model training. In this section, we discuss some techniques related to large model training.

Data Parallelism. Previous works [8], [42]–[45] have proposed various approaches to reduce the communication overhead of data parallelism. Gradient accumulation [46] is widely adopted to decrease the communication ratio in the training process. Another orthogonal approach is the overlap of computation and communication [5], [34], [47], [48]. To train large models with data parallelism, sharded data parallelism [49], [50], which reduces the memory consumption by sharding optimizer states, is introduced. ZeRO [4], [51] expands this idea by sharding weights and gradients as well.

Pipeline Parallelism. Pipeline parallelism [52], [53] is a common technique used to train DNN models. The most common approach is synchronous pipeline parallelism. GPipe [14] attempts a pipeline approach to train large models with limited device memory. TeraPipe [54] uses token-level granularity for pipeline parallelism. DAPPLE [12] and Hippie [13] explore a combination of pipeline and data parallelism on a set of devices. PipeTransformer [55] flexibly adjusts the degree of pipeline and data parallelism by scaling the layers to be trained through freezing weights. Pipeline parallelism can be implemented in asynchronous too. PipeDream [21] combines data and pipeline parallelism for asynchronous training. PipeDream-2BW [19] makes further optimization in memory consumption on top of PipeDream. PipeMare [18] explores asynchronous pipeline parallelism for DNN training.

Automatic partition. Recent works [24], [56]–[58] explore automatic partition across devices with the help of cost models. However, all of these do not consider pipeline parallelism. Learning-based approaches [59]–[63] use reinforcement learning to find the optimal pipeline partition scheme. FlexFlow [64] uses a heuristic MCMC-based approach that requires extra knowledge to generate partition schemes. PipeDream [21] and DAPPLE [12] use dynamic programming to partition the model automatically, while Piper [23] uses the heuristic algorithm for pipeline planning. Due to the vast search space, the algorithms are unsatisfactory in efficiency.

VI. Conclusion

In this paper, we propose AutoPipe, which achieves pipeline planning based on the critical path of pipeline with fine-grained model slicing. Moreover, we reduce the pipeline startup overhead by slicing micro-batches in the Warmup phase, and we implement an algorithm to automatically solve the number of micro-batches that needs to be sliced for the current partition scheme. Experiments show AutoPipe Planner outperforms DAPPLE Planner and Piper by 1.19x and 1.18x, respectively. AutoPipe Slicer halves the startup overhead without bringing additional memory requirements. And we achieve 1.30x speedups on GPT-2 345M by combining the Planner and Slicer.

VII. Acknowledgment

The work was partially supported by the National Key R&D Program of China (No. 2021YFB0301200) and National Natural Science Foundation of China (No. 61806216). Dongsheng Li is the corresponding author.

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008.

[3] A. Cully, J. Clune, D. Tarapore, and J. Mouret, "Robots that can adapt like animals," *Nat.*, vol. 521, no. 7553, pp. 503–507, 2015.

[4] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

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

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.

[7] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

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

[9] C. Rosset, "Turing-nlg: A 17-billion-parameter language model by microsoft," *Microsoft Blog*, vol. 1, no. 2, 2020.

[10] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2020.

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[12] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.

[13] X. Ye, Z. Lai, S. Li, L. Cai, D. Sun, L. Qiao, and D. Li, "Hippie: A data-paralleled pipeline approach to improve memory-efficiency and scalability for large dnn training," in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[14] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[15] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Koster, "Pipelined backpropagation at scale: training large models without batches," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 479–501, 2021.

[16] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[17] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[18] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "Pipemare: Asynchronous pipeline parallel dnn training," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 269–296, 2021.

[19] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.

[20] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[21] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[22] S. Eliad, I. Hakimi, A. De Jagger, M. Silberstein, and A. Schuster, "Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 381–396.

[23] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for dnn parallelization," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[24] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino, "Efficient algorithms for device placement of dnn graph operators," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 451–15 463, 2020.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[29] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[30] A. Griewank and A. Walther, "Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 1, pp. 19–45, 2000.

[31] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica, "Checkmate: Breaking the memory wall with optimal tensor rematerialization," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 497–511, 2020.

[32] Z. Luo, X. Yi, G. Long, S. Fan, C. Wu, J. Yang, and W. Lin, "Efficient pipeline planning for expedited distributed dnn training," *arXiv preprint arXiv:2204.10562*, 2022.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[35] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, vol. 2, 2017.

[36] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[37] R. Kiros, Y. Zhu, R. Salakhutdinov, R. S. Zemel, A. Torralba, R. Urtasun, and S. Fidler, "Skip-thought vectors," *arXiv preprint arXiv:1506.06726*, 2015.

[38] A. Gokaslan and V. Cohen, "Openwebtext corpus," http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[39] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 102, 2001.

[40] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[41] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[42] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[43] N. Arivazhagan, A. Bapna, O. Firat, D. Lepikhin, M. Johnson, M. Krikun, M. X. Chen, Y. Cao, G. Foster, C. Cherry *et al.*, "Massively multilingual neural machine translation in the wild: Findings and challenges," *arXiv preprint arXiv:1907.05019*, 2019.

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.

[44] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[45] Y. You, J. Hseu, C. Ying, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large-batch training for lstm and beyond," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–16.

[46] T. D. Le, T. Sekiyama, Y. Negishi, H. Imai, and K. Kawachiya, "Involving cpus into multi-gpu deep learning," in *Proceedings of the 2018 ACM/SPEC international conference on performance engineering*, 2018, pp. 56–67.

[47] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 132–145, 2019.

[48] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 181–193.

[49] Y. Xu, H. Lee, D. Chen, H. Choi, B. Hechtman, and S. Wang, "Automatic cross-replica sharding of weight update in data-parallel training," *arXiv preprint arXiv:2004.13336*, 2020.

[50] S. Kumar, V. Bitorff, D. Chen, C. Chou, B. Hechtman, H. Lee, N. Kumar, P. Mattson, S. Wang, T. Wang *et al.*, "Scale mlperf-0.6 models on google tpu-v3 pods," *arXiv preprint arXiv:1909.09756*, 2019.

[51] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: memory optimization towards training a trillion parameter models. arxiv e-prints arxiv: 11910.02054 (2019)," 1910.

[52] J. Zhan and J. Zhang, "Pipe-torch: Pipeline-based distributed deep learning in a gpu cluster with heterogeneous networking," in *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 2019, pp. 55–60.

[53] J. Geng, D. Li, and S. Wang, "Elasticpipe: An efficient and dynamic model-parallel solution to dnn training," in *Proceedings of the 10th Workshop on Scientific Cloud Computing*, 2019, pp. 5–9.

[54] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6543–6552.

[55] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "Pipetransformer: Automated elastic pipelining for distributed training of large-scale models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 4150–4159.

[56] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," *Advances in neural information processing systems*, vol. 31, 2018.

[57] Z. Cai, X. Yan, K. Ma, Y. Wu, Y. Huang, J. Cheng, T. Su, and F. Yu, "Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1967–1981, 2021.

[58] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks." in *ICML*, 2018, pp. 2279–2288.

[59] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2017, pp. 2430–2439.

[60] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *International Conference on Learning Representations*, 2018.

[61] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. C. Ma, Q. Xu, M. Zhong, H. Liu, A. Goldie, A. Mirhoseini *et al.*, "Gdp: Generalized device placement for dataflow graphs," *arXiv preprint arXiv:1910.01578*, 2019.

[62] S. Bojja Venkatakrishnan, S. Gupta, H. Mao, M. Alizadeh *et al.*, "Learning generalizable device placement algorithms for distributed machine learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[63] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1676–1684.

[64] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks." *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

Authorized licensed use limited to: University of Tokyo. Downloaded on August 24,2023 at 08:26:21 UTC from IEEE Xplore. Restrictions apply.