Elastic Averaging for Efficient Pipelined DNN Training

Zihao Chen
zhchen@stu.ecnu.edu.cn
East China Normal University†

Chen Xu*
cxu@dase.ecnu.edu.cn
East China Normal University†

Weining Qian
wnqian@dase.ecnu.edu.cn
East China Normal University†

Aoying Zhou
ayzhou@dase.ecnu.edu.cn
East China Normal University†

Abstract

Nowadays, the size of DNN models has grown rapidly. To train a large model, pipeline parallelism-based frameworks partition the model across GPUs and slice each batch of data into multiple micro-batches. However, pipeline parallelism suffers from a bubble issue and low peak utilization of GPUs. Recent work tries to address the two issues, but fails to exploit the benefit of vanilla pipeline parallelism, i.e., overlapping communication with computation. In this work, we employ an *elastic averaging-based framework* which explores elastic averaging to add multiple parallel pipelines. To help the framework exploit the advantage of pipeline parallelism while reducing the memory footprints, we propose a schedule, *advance forward propagation*. Moreover, since the numbers of parallel pipelines and micro-batches are essential to the framework performance, we propose a *profiling-based tuning method* to automatically determine the settings. We integrate those techniques into a prototype system, namely *AvgPipe*, based on PyTorch. Our experiments show that AvgPipe achieves a 1.7x speedups over state-of-the-art solutions of pipeline parallelism on average.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**.

Keywords: deep learning system, pipeline parallelism, elastic averaging

*Chen Xu is the corresponding author
†Shanghai Engineering Research Center of Big Data Management

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577484

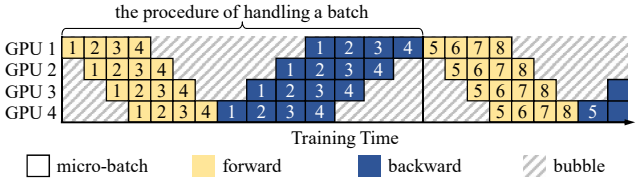


Figure 1. Vanilla Pipeline Parallelism

1 Introduction

With the increase of data volume and hardware evolution, deep neural networks (DNNs) saw great progress in the last decade. To train models across accelerators, solutions including TensorFlow [7], PyTorch [3], Caffe [15], and MXNet [1], were proposed. Typically, they provide interfaces to train DNN models in data parallelism, which replicates model weights on each GPU, and thus cannot support large model training. However, recently, DNN models have grown rapidly in size to achieve higher quality outputs across a range of applications, especially for natural language processing (NLP) [8, 29]. Hence, a model-parallel training fashion, called pipeline parallelism [14], has emerged. Pipeline parallelism partitions successive model layers across GPUs, slices each batch of data into micro-batches, and pipeline them through GPUs, as illustrated in Figure 1. Moreover, pipeline parallelism avoids synchronizing model weights [23] and naturally overlaps the communication across GPUs with computation, potentially outperforming data parallelism.

Nonetheless, pipeline parallelism suffers from underutilized GPUs. DNN training involves backward propagation, blocking the training pipeline and causing a bubble issue [14]. For example, in Figure 1, after the forward propagation of micro-batch 4, GPU 1 has to wait for the downstream GPUs to send back the backward results. A simple solution is to add more micro-batches into a batch, reducing the proportion of the bubble time. This, in turn, increases the batch size and hinders the statistical efficiency [16, 21, 34], i.e., requiring more epochs to achieve the target quality of the model.

To address the bubble issue without changing the batch size, PipeDream [23] versions model weights, and fills in the bubbles with the multi-version pipeline. However, the memory footprint of PipeDream is linearly related to the GPU number due to the multiple versions, meaning one cannot

scale out the cluster to train larger models [24]. To conserve memory, PipeDream-2BW [24] reduces the version number via bounded-stale training. In addition, PipeDream-2BW and Dapple [11] present an one-forward-one-backward (1F1B) schedule that alternates between the forward and backward propagation of micro-batches to avoid stashing activation. Chimera [19] also employs bidirectional pipelines to eliminate bubbles. Nonetheless, all of them are strict to communication efficiency, and may have even poorer performance than vanilla pipeline in distributed environments. This is because, when interleaving forward and backward propagation, the system cannot follow the vanilla pipeline to overlap communication with computation.

In this paper, we propose AvgPipe, a distributed training system employing an *elastic averaging-based framework*, to mitigate the bubble issue for pipeline parallelism while maintaining the statistical efficiency. Basically, the elastic averaging technique scales the batch number fed to training processes per iteration. AvgPipe employs multiple parallel pipelines, where each pipeline handles a batch of data per iteration and trains a parallel model via elastic averaging. Via processing more batches, AvgPipe is able to slice each batch into more micro-batches and reduce bubbles. In addition, instead of creating a new optimizer like recent work on elastic averaging [18], we explore a new framework so that AvgPipe is compatible with various optimizers (e.g., Adam [17]).

However, the techniques of elastic averaging cannot directly work with pipeline parallelism, since the replica (parallel) models increase memory footprints. To conserve memory, AvgPipe follows PipeDream-2BW and Dapple to employ the 1F1B schedule. Furthermore, given that 1F1B cannot fully overlap communication with computation, we propose *advance forward propagation*. It schedules partial forward propagation in advance, so that AvgPipe can exploit bubble time and overlap communication with computation with lower memory footprints.

Moreover, the numbers of parallel pipelines and micro-batches, termed as parallelism degrees, are essential to the performance and memory footprints. For example, slicing a batch into more micro-batches may cause lower peak utilization of GPUs, and adding more parallel pipelines may lead to higher memory footprints. To ease the setting of parallelism degrees, we propose a *profiling-based tuning method*. The method predicts the performance under different settings based on a short profiling run, and chooses the setting of the best performance under the given memory constraint. Finally, our experimental results show that, AvgPipe is able to achieve 4.7x and 1.7x speedups over state-of-the-art data parallelism and pipeline parallelism, respectively.

In the rest, we highlight the motivation for elastic averaging in Section 2, and make the following contributions.

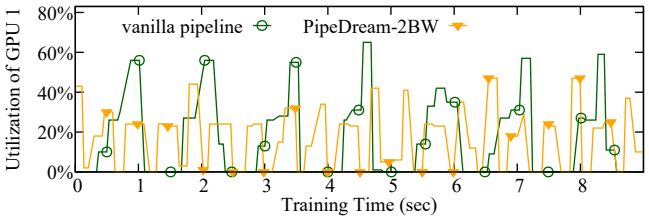


Figure 2. Underutilized GPU in the Example of BERT

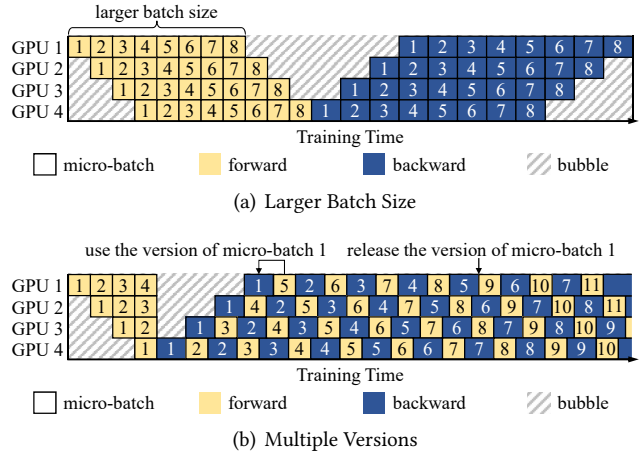


Figure 3. Approaches to Reduce Bubbles

- We propose an *elastic averaging-based framework* in Section 3, which improves the performance of pipeline parallelism and decouples from optimizers.
- We propose *advance forward propagation* in Section 3, which conserves memory while preventing performance degradation from communication overhead.
- We propose a *profiling-based tuning method* in Section 5, which tunes parallelism degrees with negligible profiling time, to exploit the performance of AvgPipe.
- We discuss the implementation of AvgPipe in Section 6, and demonstrate its performance in Section 7.

In addition, we introduce related work in Section 8 and summarize our work in Section 9.

2 Motivating Elastic Averaging

In pipeline parallelism, the bubbles and low peak utilization of GPUs motivate us to employ elastic averaging.

Bubbles. Due to backward propagation, the GPUs in pipeline parallelism would idle periodically, causing bubbles. Figure 2 illustrates the bubbles of training Bidirectional Encoder Representations from Transformers (BERT) [9]. To reduce the bubble portion, one can increase the batch size, as depicted in Figure 3(a). However, a larger batch size reduces statistical efficiency and causes more training epochs in turn [18, 34].

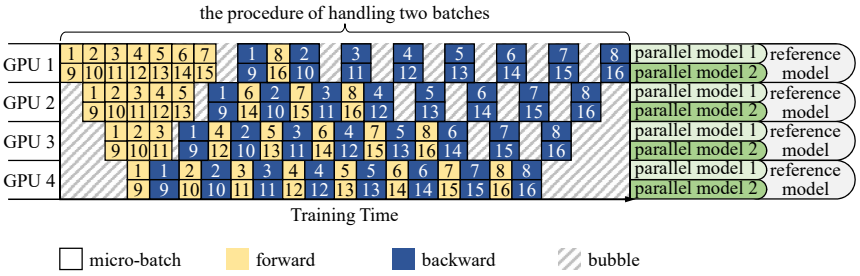


Figure 4. An Overview of Pipeline Parallelism with Elastic Averaging

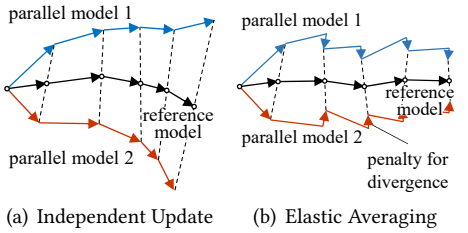


Figure 5. The Rationale of Elastic Averaging

Alternatively, PipeDream [23] uses multi-version pipelines to fill bubbles. As illustrated in Figure 3(b), PipeDream generates a version of the model after the backward propagation of micro-batch 1 in GPU 1. Then, based on this version, PipeDream immediately performs the forward propagation of micro-batch 5, to eliminate bubbles. Unfortunately, PipeDream cannot release the version of micro-batch 1 until finishing micro-batch 5. That means, PipeDream has to maintain four (equal to the number of GPUs) versions during the training, which is not memory-efficient [24, 33]. To conserve memory, PipeDream-2BW [24] allows stale training and maintains only two versions. Nonetheless, as shown in Figure 2, PipeDream-2BW still has GPU idled periodically. The reason behind is the communication overhead among GPUs, which will be further discussed in Section 4.

Low Peak Utilization of GPUs. In pipeline parallelism, each GPU handles one micro-batch at a time. However, due to the small size of a micro-batch, the arithmetic intensity is not high enough so that the training cannot exploit the computing power of GPUs. As depicted in Figure 2, despite the idle and communication time, the peak utilization of GPU 1 is around only 60% in pipeline parallelism.

Overall, we learnt two lessons. First, although a large micro-batch number addresses bubbles, it also reduces statistical efficiency. Second, the small size of micro-batches results in low arithmetic intensity and thus underutilized GPU kernels. Hence, the challenge here is to increase the micro-batch number and arithmetic intensity, while remaining the original statistical efficiency. To mitigate this, we explore a new framework based on elastic averaging [35]. As depicted in Figure 4, the framework slices a batch into more micro-batches and handles multiple batches in parallel, to overcome the bubbles and low peak utilization of GPUs.

3 Elastic Averaging in Pipeline Parallelism

To improve the performance of pipeline parallelism, we employ the idea of elastic averaging. We start with the pros and cons of elastic averaging in Section 3.1. Then, we propose an elastic averaging-based framework in Sections 3.2, to overcome the defect of existing elastic averaging techniques.

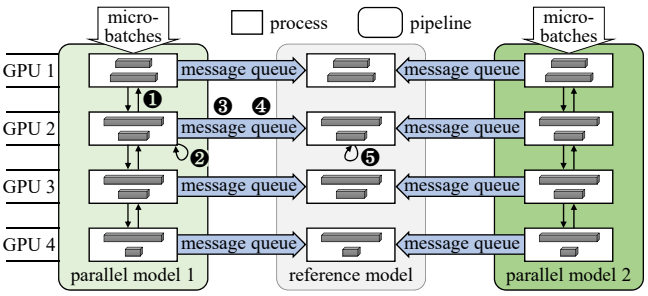


Figure 6. The Elastic Averaging-based Framework

3.1 Pros and Cons of Elastic Averaging

The main idea of elastic averaging is to increase parallelism while remaining statistical efficiency. In practice, the use of small batch sizes has been observed to improve generalization performance and optimization convergence [16, 18, 32]. Typically, the gradient norm trained from a larger batch is smaller, which may result in lower statistical efficiency, since the gradient is normalized by the batch size. To increase parallelism and preserve statistical efficiency, we can add multiple parallel models. Each model is trained with a small batch size independently [30], as depicted in Figure 5(a). Nonetheless, this may lead to the divergence of parallel models [20]. Hence, we use an elastic mechanism to pull the parallel models towards the center position, as depicted in Figure 5(b). This offers us the flexibility to adjust parallelism degrees (e.g., increase the number of micro-batches or parallel models).

However, one of the defect of existing elastic averaging techniques [18, 35] is, they serve as extended SGD optimizers coupling with SGD. Given that a range of optimizers [10, 17, 27] have shown better statistical efficiency than the vanilla SGD optimizer on different models, the extended SGD optimizers have limited usage in practice. Furthermore, the extended optimizers cannot directly support large model training due to the higher footprints of extra parallel models, which is against the intention of pipeline parallelism.

3.2 Elastic Averaging-based Framework

To be compatible with various optimizers, we propose an elastic averaging-based framework for pipeline parallelism.

As depicted in Figure 6, we introduce *parallel models* and a *reference model* in the framework.

Parallel Models. AvgPipe launches N parallel pipelines, each of which performs forward and backward propagation in local to train a parallel model. In specific, first, AvgPipe partitions a parallel model to multiple GPUs. Each GPU takes charge of one partition of successive layers in the model. Then, a parallel pipeline launches a process on each GPU, performing the following steps iteratively.

Step ❶ is to pipeline the forward as well as backward propagation of micro-batches through GPUs, and compute a local update via a user-specified optimizer. Here, the optimizer is available in a variety of options. Step ❷ is to update the weights of a parallel model. In addition to the local update, AvgPipe applies elastic averaging to the parallel model. That is, diluting the parallel model weights with the reference model weights in a ratio of $(1-\alpha):\alpha$. Here, $\alpha \in [0,1]$ indicates the dependence of parallel models, which is set to $\frac{1}{N}$ empirically [18]. Step ❸ is to send the local update to the reference model. To prevent the inter-process communication blocking the pipeline, AvgPipe manages the sending through message queues in an asynchronous manner.

A Reference Model. To support elastic averaging, AvgPipe maintains a reference model as the centre of parallel models. That is, each weight in the reference model stays the average of the corresponding weights in parallel models. AvgPipe uses the reference model to decide which direction and how far to pull parallel models, in order to prevent divergence. In specific, AvgPipe co-partitions the reference model with the parallel models. For each GPU, AvgPipe starts a separate process to manage the reference model, so as to not block the parallel pipelines. The process performs two steps iteratively.

In Step ❹, the process receives a local update from each parallel pipeline through message queues. The local updates are then accumulated for the subsequent update. In Step ❺, after receiving all of the local updates, i.e., each parallel pipeline has finished a batch, the process normalizes and applies the accumulated update.

4 Performance-aware Schedule

Although the elastic averaging-based framework employs parallel pipelines to improve performance, the model replicas lead to higher memory footprints, which is against the intention of pipeline parallelism, i.e., addressing the lack of GPU memory. Hence, we seek a performance-aware schedule to enable AvgPipe to train large models. Next, we will elaborate the existing schedules of pipeline parallelism and propose advance forward propagation based on them.

4.1 Issues in Existing Schedules

Following data parallelism, the vanilla schedule in pipeline parallelism is all-forward-all-backward (AFAB) schedule. In specific, as depicted in Figure 7(a), the AFAB schedule starts backward propagation only after all forward propagation of a batch. It fully overlaps communication and computation, but has to maintain all activation of one batch in GPU memory, enlarging memory footprints.

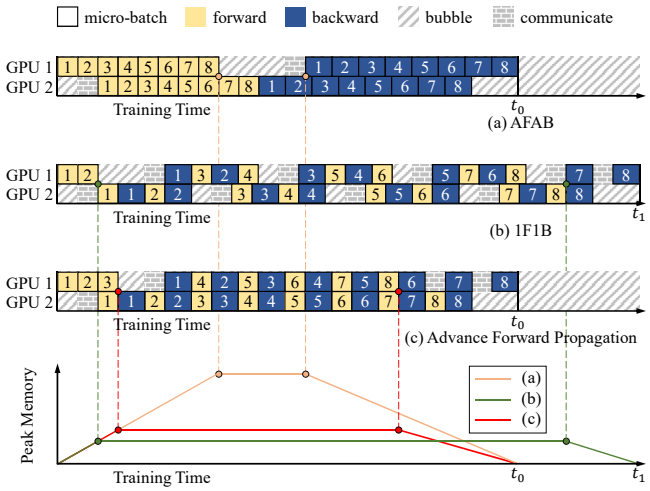


Figure 7. Different Schedules on One Batch

To mitigate the memory footprint issue, PipeDream-2BW and Dapple employ an one-forward-one-backward (1F1B) schedule (also termed as early backward schedule [11]). That is, they alternate between the forward and backward propagation of micro-batches, so as to release the forward activation stashed in GPU memory via the backward propagation from time to time. Specifically, if there are K GPUs, then the k-th GPU only needs to stash the activation of up to $K-k+1$ micro-batches. As the example depicted in Figure 7(b), when $K=2$, the first GPU, i.e., $k=1$, needs to stash the activation of 2 micro-batches.

Unfortunately, the 1F1B schedule also prevents the system from fully overlapping communication with computation, since it interleaves the propagation pipeline in both forward and backward directions. As depicted in Figure 7(a), the communication blocks the AFAB schedule only two times per batch. But the communication blocks the 1F1B schedule nine times as shown in Figure 7(b). For example, the communication in 1F1B causes the starvation between the processing of micro-batches 2 and 3 on GPU 2, which delays the backward propagation of micro-batch 4 on GPU 1. Consequently, the training time of 1F1B, t_1, becomes longer than the training time of AFAB, t_0. Overall, the 1F1B schedule sacrifices training time for lower memory footprints.

4.2 Advance Forward Propagation

Indeed, multiple parallel pipelines cause higher memory footprints. To conserve memory, AvgPipe follows PipeDream-2BW and Dapple to employ the 1F1B schedule. As long as we slice a batch into smaller micro-batches, we can alleviate the extra GPU memory caused by parallel pipelines.

Algorithm 1 Advance Forward Propagation on the 1st GPU

Input: The number of micro-batches in each batch M, the number of GPUs K

```
1:  advance_num ← K − 1  // equivalent to 1F1B schedule
2:  while next_batch() do
3:      for i = 1, 2, ··· , advance_num do
4:          forward_a_micro_batch()  // forward in advance
5:      // interleave the rest micro-batches
6:      for i = 1, 2, ··· , M − advance_num do
7:          forward_a_micro_batch()
8:          backward_a_micro_batch()
9:      if is_faster() and is_mem_available() then
10:         advance_num++  // adjust advance_num
```

However, the 1F1B schedule suffers from the communication overhead. To address the performance issue, we further propose advance forward propagation, which exploits the bubble time in pipeline parallelism to accelerate execution. In particular, we observe that there is sufficient bubble time for GPU 1 to perform the forward propagation of micro-batch 3 right after that of micro-batch 2. Based on this observation, we intend to utilize the bubble time to perform forward propagation of micro-batches in advance, preventing the starvation of downstream GPUs. As illustrated in Figure 7(c), if GPU 1 performs the forward propagation of micro-batch 3 in advance, then AvgPipe is able to overlap the forward communication of micro-batch 3 with the computation of micro-batches 1 and 2 on GPU 2, and thus prevents the subsequent starvation of GPU 1 and GPU 2.

Pros and Cons. In comparison to the AFAB schedule, advance forward propagation achieves the same training time but a lower peak memory footprint (3/8 of the AFAB schedule in the example of Figure 7). Note that, although advance forward propagation sacrifices partial memory of upstream GPUs in comparison to the 1F1B schedule, it is beneficial to performance after all. As illustrated in Figure 7(c), the first GPU stashes the activation of three micro-batches, while there are two micro-batches for the 1F1B schedule. However, we can dilute the extra memory footprints via slicing a batch into more micro-batches. As long as the micro-batch size is small enough, the extra memory footprints are negligible.

Moreover, the advance forward propagation is essentially a trade-off between the vanilla AFAB and 1F1B schedules, meaning it can degenerate into either one of them. On one hand, if, ideally, the communication overhead is minimal, then the number of micro-batches propagated forward in advance could be zero, i.e., equivalent to the 1F1B schedule. On the other hand, if the communication overhead is significant and the available memory is sufficient, then the schedule would perform the forward propagation of all micro-batches in advance, which is equivalent to the AFAB schedule.

Decisions on Advance Forward Propagation. To avoid wasting memory, AvgPipe takes a conservative strategy on performing advance forward propagation, as shown in Algorithm 1. Initially, AvgPipe performs the 1F1B schedule (Line 1). Then, AvgPipe gradually increases the number of micro-batches propagated forward in advance per iteration, until the memory footprint reaches the user-defined limit or the training stops getting faster (Lines 9 - 10).

5 Tuning Parallelism Degrees

There are two parallelism degrees in the elastic averaging-based framework: the micro-batch number M and the parallel pipeline number N. Both of them significantly affect the training performance (an order of magnitude difference in performance according to our experiments). In this section, we will introduce the challenge of tuning parallelism degrees, and our profiling-based tuning method.

5.1 Challenges to Tune Parallelism Degrees

There may be a tuning guideline to improve performance, which is maximizing the parallelism degrees. In specific, the guideline increases the micro-batch number to reduce bubbles, and the parallel pipeline number to optimize the peak utilization of GPUs. Nonetheless, the guideline is not always viable for three reasons. First, given a fixed batch size, the performance does not always increase with the micro-batch number, since the micro-batch may be too small that AvgPipe cannot fully utilize all of the GPU kernels. Second, there is diminishing marginal utility of GPU utilization when increasing the parallel pipeline number. Third, both two parallelism degrees are relevant on GPU utilization and memory footprints, so that we cannot naïvely tune them one by one.

Since it is non-trivial for users to set parallelism degrees, AvgPipe requires a tuning method. One way is to traverse all settings of the degrees, with a small number of batches (e.g., ten batches). However, the traversal takes hours (e.g., over two hours for the training of the BERT model in our experiment). In order to obtain the optimal parallelism degrees with negligible overhead, we explore an efficient tuning method.

5.2 Profiling-based Tuning Method

Since it is both essential and complex to tune parallelism degrees, we propose a profiling-based tuning method. The idea is to profile only one setting of parallelism degrees, and predict the performance as well as memory footprints under other settings based on the profile. In the end, we choose the setting that achieves the optimal prediction result and satisfy the memory constraint. Accordingly, the method consists of two phases, *profiling* and *predicting*.

5.2.1 Profiling Performance and Memory. In profiling phase, we try a certain setting of parallelism degrees. Particularly, we use a rather large M and a small N so that GPUs are not fully utilized. Before we dig into this, generally, the

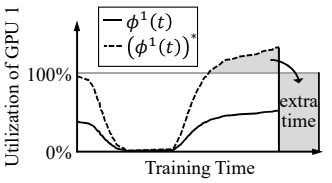


Figure 8. Predicting $\left(T^1_{gpu}\right)^*$

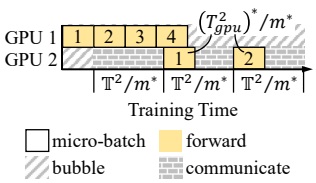


Figure 9. Predicting $\left(T^2_{com}\right)^*$

profile with resources overused is insufficient to predict performance, especially when the new setting has resources underutilized. We let AvgPipe train the model with twenty batches under the setting. During the training, we collect GPU utilization, computation time, communication time, and memory footprints, to be used in predicting phase.

5.2.2 Predicting Performance. Based on the profile, we predict the performance under different settings of parallelism degrees. Since we have overcome the side-effect of 1F1B via advance forward propagation, it is reasonable to assume the performance of AFAB and 1F1B with advance forward propagation is close enough. Also, the purpose is to compare different settings rather than evaluate the exact training time. Hence, we predict the performance of AFAB rather than 1F1B to ease the performance analysis.

For any GPU k, we categorize the training time T^k on one batch into computation T^k_{gpu}, communication T^k_{com} that blocks GPU, and bubble T^k_{bub}, i.e.,

$$T^k = T^k_{gpu} + T^k_{com} + T^k_{bub}. \tag{1}$$

Next, we analyse how M and N affect T^k_{gpu}, T^k_{com} and T^k_{bub}. For simplicity, let $M = m$ and $N = n$ in profiling phase.

Computation Time T^k_{gpu}. Intuitively, we predict computation time by analyzing the curve of GPU utilization, $\phi^k(t)$, in the profile. In particular, the integral of the curve indicates the computation volume. Since the computation volume on one batch is fixed, we can predict the new computation time based on the new curve deduced for new M and N.

First, we predict $\left(T^k_{gpu}\right)^*$ for $M = m^*$, i.e., the micro-batch size is $\frac{m}{m^*}$ of the original. As a simplification of real-world environments, we assume the arithmetic intensity is $\frac{m}{m^*}$ of the original. Accordingly, $\left(\phi^k(t)\right)^*$ is $\frac{m}{m^*}$ of $\phi^k(t)$, as depicted in Figure 8. However, we need to check whether $\left(\phi^k(t)\right)^*$ exceeds 100%, the upper boundary of GPU utilization. If $\left(\phi^k(t)\right)^* \leqslant 100\%$, then $\left(T^k_{gpu}\right)^*$ would be $\frac{m^*}{m} T^k_{gpu}$ to have the same computation volume on one batch, since $\left(\phi^k(t)\right)^* = \frac{m}{m^*}\phi^k(t)$. If $\left(\phi^k(t)\right)^* > 100\%$, then the part of $\left(\phi^k(t)\right)^*$ higher than 100% actually means GPU k is overused. This causes the extra computation time of $\frac{m^*}{m}\int_t max(\left(\phi^k(t)\right)^* - 1, 0)$ i.e., the shaded area in Figure 8. Hence, for $M = m^*$, we have $\left(T^k_{gpu}\right)^* = \frac{m^*}{m}\left(T^k_{gpu} + \int_t max\left(\frac{m}{m^*}\phi^k(t) - 1, 0\right)\right)$.

Second, we predict the value of $\left(T^k_{gpu}\right)^*$ when $M = m^*$ and $N = n^*$. Similarly, since there are n^* instead of n parallel pipelines, we deduce $\left(\phi^k(t)\right)^*$ is $\frac{n^*}{n}$ of $\frac{m}{m^*}\phi^k(t)$. Therefore, if $\left(\phi^k(t)\right)^* \leqslant 100\%$, then $\left(T^k_{gpu}\right)^*$ would be $\frac{n}{n^*}\left(\frac{m^*}{m} T^k_{gpu}\right)$. Otherwise, there would be $\frac{n}{n^*}\left(\frac{m^*}{m}\int_t max\left(\left(\phi^k(t)\right)^* - 1, 0\right)\right)$ of extra time. Overall, for $M = m^*$ and $N = n^*$, we have

$$\left(T^k_{gpu}\right)^* = \frac{m^* n}{m n^*}\left(T^k_{gpu} + \int_t max\left(\frac{m n^*}{m^* n}\phi^k(t) - 1, 0\right)\right). \tag{2}$$

As mentioned in Section 5.2.1, we require $\phi^k(t)$ to not reach 100%. Otherwise, we cannot learn $\left(\phi^k(t)\right)^*$ from $\phi^k(t)$, if the arithmetic intensity becomes lower. For illustration, $\phi^k(t) \equiv 100\%$ does not necessarily mean $\left(\phi^k(t)\right)^* \equiv 50\%$ when $m^* = 2m$.

Communication Time T^k_{com}. To calculate the communication time blocking GPUs, we first evaluate the total communication time, and then subtract the overlap time. Accordingly, the prediction of $\left(T^k_{com}\right)^*$ relies on the total communication time, $\mathbb{T}^k$, from the profile and $\left(T^k_{gpu}\right)^*$ predicted in Equation 2.

For $M = m^*$, since there are m^* micro-batches in a batch, the total communication time on one micro-batch is $\mathbb{T}^k/m^*$. Then, computation overlaps communication on each micro-batch, except for the first micro-batch, as shown in Figure 9. That is, for each of the last $m^* - 1$ micro-batches, communication blocks GPU k by $max\left(\mathbb{T}^k - \left(T^k_{gpu}\right)^*, 0\right)/m^*$ of time. Hence, for $M = m^*$,

$$\left(T^k_{com}\right)^* = \frac{1}{m^*}\mathbb{T}^k + \frac{m^* - 1}{m^*} max\left(\mathbb{T}^k - \left(T^k_{gpu}\right)^*, 0\right). \tag{3}$$

For $M = m^*$ and $N = n^*$, since there are n^* instead of n parallel pipelines, the communication overhead is $\frac{n^*}{n}$ of the original. Therefore, $\left(\mathbb{T}^k\right)^* = \frac{n^*}{n}\mathbb{T}^k$, leading to,

$$\left(T^k_{com}\right)^* = \frac{n^*}{m^* n}\mathbb{T}^k + \frac{m^* - 1}{m^*} max\left(\frac{n^*}{n}\mathbb{T}^k - \left(T^k_{gpu}\right)^*, 0\right). \tag{4}$$

Bubble Time T^k_{bub}. We use the computation and communication time evaluated via Equations 2 and 4 to predict how long GPU k will wait for the other GPUs, i.e., $\left(T^k_{bub}\right)^*$. To do so, we divide $\left(T^k_{bub}\right)^*$ into the time waiting for upstream GPUs, $\left(T^k_{up}\right)^*$, and downstream GPUs, $\left(T^k_{down}\right)^*$. That is,

$$\left(T^k_{bub}\right)^* = \left(T^k_{up}\right)^* + \left(T^k_{down}\right)^*. \tag{5}$$

Based on $\left(T^k_{gpu}\right)^*$ and $\left(\mathbb{T}^k\right)^*$ evaluated above, we predict $\left(T^k_{up}\right)^*$ and $\left(T^k_{down}\right)^*$ recursively.

To predict $\left(T^k_{up}\right)^*$, we start with $\left(T^1_{up}\right)^* = 0$, since GPU 1 has no upstream GPUs. For $k = 2, 3, \cdots, K$, GPU k has to wait for GPU $k-1$ to receive the data or intermediate results of the first micro-batch, and finish the corresponding computation,

taking $\left(\mathbb{T}^{k-1}\right)^* / m^*$ and $\left(T_{gpu}^{k-1}\right)^* / m^*$, respectively. Hence, the recursive equation of $\left(T_{up}^k\right)^*$ is

$$\left(T_{up}^k\right)^* = \left(T_{up}^{k-1}\right)^* + \frac{1}{m^*}\left(\left(\mathbb{T}^{k-1}\right)^* + \left(T_{gpu}^{k-1}\right)^*\right). \quad (6)$$

Similarly, we have $\left(T_{down}^K\right)^* = 0$, and for $k = 1, 2, \cdots, K-1$, the recursive equation of $\left(T_{down}^k\right)^*$ is

$$\left(T_{down}^k\right)^* = \left(T_{down}^{k+1}\right)^* + \frac{1}{m^*}\left(\left(\mathbb{T}^{k+1}\right)^* + \left(T_{gpu}^{k+1}\right)^*\right). \quad (7)$$

5.2.3 Predicting Memory Footprints. To analyse the impact of M and N on memory footprints, we divide the total footprints on GPU k, F^k, into the footprints of parallel and reference models, F_{mod}^k, and the footprints of data as well as intermediate results, F_{dat}^k.

First, we predict $\left(F_{mod}^k\right)^*$ and $\left(F_{dat}^k\right)^*$ for $M = m^*$. Since M does not affect F_{mod}^k, we have $\left(F_{mod}^k\right)^* = F_{mod}^k$. Hence, $\left(F_{dat}^k\right)^*$ is linearly related to the micro-batch size. Given that the micro-batch size is $\frac{m}{m^*}$ of the original, $\left(F_{dat}^k\right)^* = \frac{m}{m^*} F_{dat}^k$.

Second, we predict $\left(F_{mod}^k\right)^*$ and $\left(F_{dat}^k\right)^*$ for $M = m^*$ and $N = n^*$. When switching from $N = n$ to $N = n^*$, $\left(F_{mod}^k\right)^*$ and $\left(F_{dat}^k\right)^*$ become $\frac{n^*}{n}$ of the original. As a result,

$$\left(F^k\right)^* = \frac{n^*}{n} F_{mod}^k + \frac{m\,n^*}{m^*\,n} F_{dat}^k. \quad (8)$$

To decide the parallelism degrees, we predict the performance and memory footprint under each setting. Eventually, we choose the setting that achieves the optimal prediction of performance and satisfies the given memory constraint.

6 Implementation

We implement AvgPipe on top of PyTorch [3], following state-of-the-art work on large model training (e.g., Megatron-LM [29], PipeDream [23], PipeMare [33], and PipeTransformer [12]). However, AvgPipe can also work with other deep learning frameworks supporting pipeline parallelism, such as TensorFlow [7], MXNet [1], and Caffe [15].

As depicted in Figure 10, the architecture consists of five components, partitioner, profiler, predictor, scheduler, and runtime. First, given an input model, the partitioner divides the model into multiple partitions. Since there have been numbers of partition methods [14, 24] to balance work and reduce communication across partitions, we employ the existing method used in PipeDream [23], instead of introducing a new method, for the partitioner. Second, AvgPipe applies the profiling-based tuning method to decide parallelism degrees of the partitioned model. In specific, the profiler initiate the runtime component to run certain batches and generate profile, for the predictor to tune parallelism degrees. Then, the

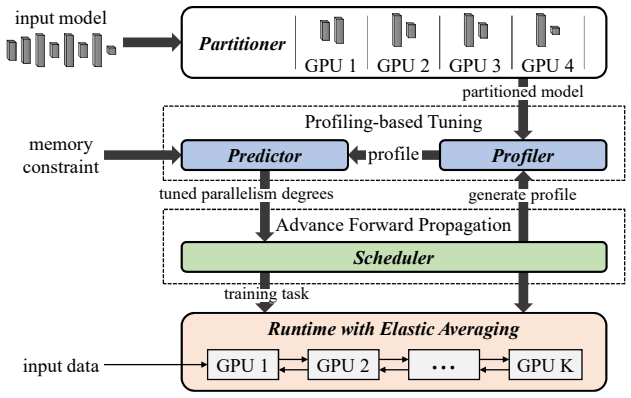


Figure 10. System Architecture

scheduler employs advance forward propagation to schedule the specific training tasks. Finally, based on elastic averaging, the runtime component executes the training tasks in parallel pipelines with the tuned parallelism degrees.

7 Experimental Studies

We conduct experiments on a three-node cluster. Each node has two Tesla V100-SXM2 GPUs, with 32 GB of GPU memory, two Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz 24-core processors, 512GB DRAM, and 1Gbps Ethernet. We deploy AvgPipe upon the NVIDIA container image for PyTorch, release 19.05. We employ three workloads in our experiments.

- **Google's Neural Machine Translation (GNMT)** for translation, using the WMT16 dataset [6] for training and the WMT14 test dataset [5] for validation. We train GNMT with Adam, a learning rate of 3e-4, a batch size of 128, and a target BLEU score [25] of 21.8 [23].
- **Bidirectional Encoder Representations from Transformers (BERT) [9]** for similarity and paraphrase, using the QQP dataset from the GLUE benchmark [31]. We train BERT with Adam, a learning rate of 2e-5, and a batch size of 32 [9]. The target is to have the top-1 accuracy over 67% for three epochs.
- **ASGD Weight-Dropped LSTM (AWD) [22]** for language modeling, using the Penn Treebank dataset [2]. Since AWD is rather small, we use four GPUs of two node. We train AWD with SGD as well as ASGD [27], a learning rate of 30, a batch size of 40, and the target validation loss of 6.5 [22].

7.1 Efficiency of Elastic Averaging-based Framework

We evaluate the overall performance of AvgPipe, and provide insights on its GPU utilization and statistical efficiency.

7.1.1 Overall Performance. There are five baselines in our experiments. PyTorch and GPipe [4, 14] represent data parallelism and pipeline parallelism, respectively. We use PipeDream [23], PipeDream-2BW [24], and Dapple [11] as the variants of pipeline parallelism. Here, we implement

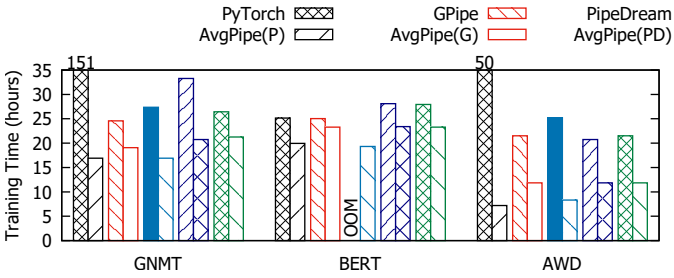


Figure 11. Training Time

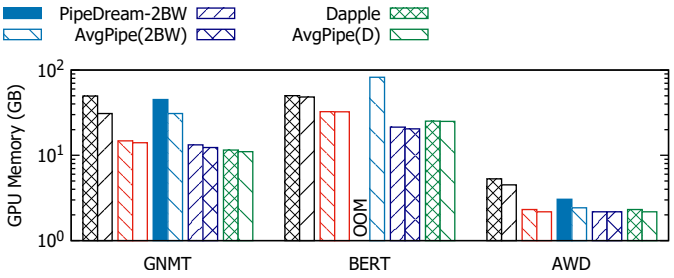


Figure 12. GPU Memory Footprints

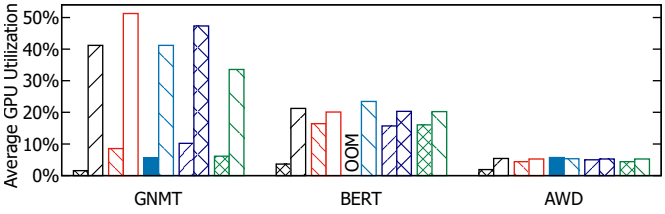


Figure 13. Averaged GPU Utilization

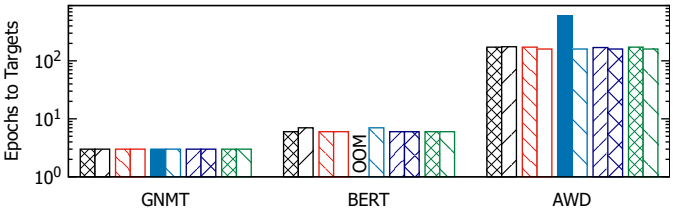


Figure 14. Statistical Efficiency

the schedule of Dapple on PyTorch to unify the runtime engine. Also, all pipeline-based baselines disable activation recomputation, and employ the same partitions generated by PipeDream.

For fair comparison, we force AvgPipe to have the same or lower memory footprints in comparison to PyTorch, GPipe, PipeDream, PipeDream-2BW, and Dapple, denoted as AvgPipe(P), AvgPipe(G), AvgPipe(PD), AvgPipe(2BW), and AvgPipe(D), respectively. Figures 11 and 12 illustrate the training time and memory footprints of all baselines and AvgPipe.

GNMT. Data parallelism, i.e., PyTorch, takes over 4 days to train GNMT, whereas pipeline parallelism and its variants take less than 34 hours. This is because, for each batch, data parallelism has to synchronize the model across nodes, resulting in overwhelming network communication overhead. Furthermore, data parallelism keeps a model replica in each GPU leading to the highest memory footprint.

In comparison to the other baselines of pipeline parallelism, AvgPipe achieves a 1.5x speedup on average. Specifically, AvgPipe(G) is 1.3x faster than GPipe, by having 2 parallel pipelines and slicing a batch into 64 micro-batches. Although the parallel pipelines incur higher memory footprints, AvgPipe(G) uses 1F1B with advance forward propagation to avoid stashing activation, and thus compensate for the extra memory footprints. Note that Dapple also employs 1F1B. However, with the micro-batch number of six, 1F1B cannot sufficiently prevent the high memory footprint of activation. AvgPipe(PD) is 1.6x faster than PipeDream. PipeDream uses multi-version pipelines to reduce bubbles. Yet, it fails to fully overlap communication overhead with computation, whereas AvgPipe(PD) mitigates this via advance forward propagation. Furthermore, the multiple model versions in PipeDream lead to higher memory footprints.

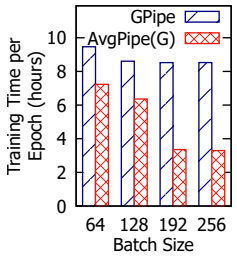


Figure 15. Varying Batch Size for GNMT

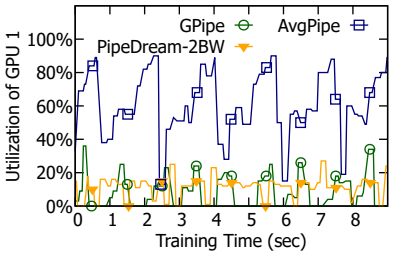


Figure 16. GPU Utilization Over Time For GNMT

With the same memory constraint, AvgPipe(PD) is able to increase the arithmetic intensity of kernels with more parallel pipelines, to accelerate training. Similarly, AvgPipe(2BW) (with 2 parallel pipelines and 64 micro-batches per batch) achieves a 1.6x speedup over PipeDream-2BW, by optimizing the overlap of communication and computation. Even though PipeDream-2BW achieves a lower memory footprint via 1F1B, AvgPipe(2BW) reduces the memory footprint by 6.8% further. This is because, AvgPipe(2BW) slices a batch into more micro-batches to enhance the benefit of 1F1B.

Given that GPipe achieves the best performance among the baselines for GNMT, we further compare the training time per epoch of GPipe under various batch size (from 64 to 256) with that of AvgPipe. As depicted in Figure 15, GPipe achieves the similar training time of an epoch on different batch size. That indicates, the batch size of 64 is already large enough for GPipe. In fact, GPipe is suffering from the bubble time, which cannot be reduced by increasing batch size. On the other hand, AvgPipe(G) achieves a 1.3x speedup over GPipe on the batch size of 64, and a 2.6x speedup over

GPipe on the batch size of 256. The larger batch size enables AvgPipe to slice a batch into more micro-batches, while the parallel pipelines still guarantee the peak GPU utilization. Hence, AvgPipe manages to reduce the bubble time in comparison to GPipe and becomes faster.

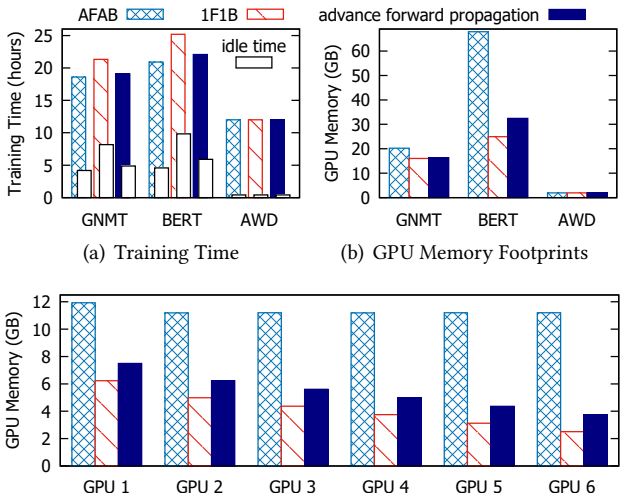
BERT. For BERT, PyTorch has a memory footprint over 50GB. With the same memory constraint, AvgPipe(P) (with 3 parallel pipelines and 16 micro-batches per batch) fully exploits the benefits of elastic averaging, achieving a 1.3x speedup over PyTorch. Among the other baselines, GPipe achieves the shortest training time, with the medium memory footprint. However, via parallel pipelines, AvgPipe(G) (with 2 parallel pipelines and 16 micro-batches per batch) further increases the utilization of GPUs and becomes 1.1x faster than GPipe. In particular, AvgPipe(G) manages to counteract the extra memory of parallel pipelines along with 1F1B and advance forward propagation. Similar to GNMT, Dapple fails to exploit 1F1B to reduce the memory footprint. Also, PipeDream cannot fulfill the training of BERT. For a cluster of six GPUs, PipeDream has to maintain six versions of model weights to mitigate bubbles, causing the out-of-memory event (OOM). In contrast, PipeDream-2BW achieves the lowest memory footprint among the baselines. Nonetheless, it suffers from interleaving the forward and backward propagation of micro-batches, which cannot fully overlap communication with computation. AvgPipe(2BW) (with 2 parallel pipelines and 32 micro-batches per batch) mitigates this issue with advance forward propagation, and achieves a 1.8x speedup.

AWD. AvgPipe(P) (with 2 parallel pipelines and 1 micro-batch per batch) outperforms PyTorch by over 7.0x, since AvgPipe(P) avoids network communication attributed to the model synchronization of data parallelism. Since we use only two nodes to train AWD other than three nodes in previous workloads, the communication issue of Dapple and PipeDream(2BW) becomes insignificant. Hence, GPipe, PipeDream-2BW, and Dapple achieve comparable training time. Nonetheless, they are oblivious to GPU utilization. Instead, AvgPipe maximize the micro-batch size, i.e., taking the whole batch as a micro-batch, to optimize the arithmetic intensity of kernels, becoming 1.8x faster than GPipe and PipeDream-2BW. On the other hand, PipeDream makes full use of GPUs. However, its multi-version training leads to lower statistical efficiency, whereas AvgPipe(PD) is 3.0x faster than PipeDream to achieve the target validation loss.

Overall, AvgPipe is 4.7x faster than data parallelism. Via the elastic averaging-based framework, AvgPipe achieves 1.7x speedups over pipeline parallelism and its variants.

7.1.2 GPU Utilization.

To provide performance insights, we demonstrate average GPU utilization in Figure 13. For GNMT and BERT, AvgPipe improves GPU utilization by 86.1% and 41.3% over baselines. The improvement comes from the higher peak utilization of GPUs attributed to parallel pipelines, and the smaller bubbles attributed to more micro-batches. In particular, for AWD, AvgPipe applies up to two parallel pipelines, improving GPU utilization by 19.6% on average, since the communication issue is not significant under the two-node setting.



(a) Training Time (b) GPU Memory Footprints

(c) Memory Footprint of each GPU for BERT

Figure 17. Performance Impact of Different Schedules

We also illustrate GPU utilization over time for GNMT in Figure 16. We choose two baselines, GPipe and PipeDream-2BW, against AvgPipe(2BW) having the lowest memory footprint. The parallel pipelines of AvgPipe(2BW) help to increase the peak GPU utilization by over 57.8% in comparison to the baselines. Moreover, we observe the frequent GPU idle of GPipe and PipeDream-2BW. That is, GPipe suffers from the bubble issue, and PipeDream-2BW fails to overlap the communication with computation. However, with the more micro-batches and advance forward propagation, AvgPipe(2BW) significantly reduces the GPU idle.

7.1.3 Statistical Efficiency.

Since PipeDream, PipeDream-2BW, and AvgPipe change the original training semantics, we demonstrate the epochs of different solutions to achieve the target score, accuracy, or loss in Figure 14. In general, AvgPipe manages to achieve similar statistical efficiency in comparison to PyTorch across all workloads. In addition, PipeDream fails to achieve comparable statistical efficiency for AWD, due to its multi-version training.

7.2 Efficiency of Advance Forward Propagation

Next, we evaluate the performance impact of our advance forward propagation. In specific, we let AvgPipe use different schedules (including AFAB, 1F1B, and 1F1B with advance forward propagation), and compare their training time and memory footprints, as depicted in Figures 17(a) and 17(b). We also show the idle time of the last GPU caused by the bubbles and communication overheads in Figure 17(a).

GNMF. AFAB achieves a 1.15x speedup over 1F1B with 20.8% extra memory footprint. The gap on the idle time further indicates that 1F1B cannot overlap communication with computation like AFAB. Advance forward propagation, on the other hand, incorporates the advantages of both schedulers via a trade-off between the training time and memory footprint. Specifically, advance forward propagation stashes the activation of partial micro-batches, which brings 2.2% extra memory footprint in comparison to 1F1B. However, it is 1.12x faster than 1F1B, and only 2.6% slower than AFAB.
BERT. Similar to GNMT, AFAB is 1.20x faster and also brings 172.0% extra memory footprints, in comparison to 1F1B. However, via advance forward propagation, AvgPipe is able to overlap communication with computation, i.e., reducing the idle time, with negligible cost. Particularly, advance forward propagation achieves a 1.14x speedup over 1F1B, and conserves 52.2% memory footprint compared with AFAB.

To offer insight, we further illustrate the memory footprint of each GPU for BERT in Figure 17(c). Certainly, 1F1B conserves more memory footprints on downstream GPUs, since the k-th GPU stashes the activation of $K-k+1$ micro-batches. Following 1F1B, advance forward propagation conserves 66.5% memory footprint compared with AFAB on GPU 6. Note that, the impact of advance forward propagation on upstream GPUs is also significant. Even on GPU 1, advance forward propagation conserves 37.2% memory footprint.
AWD. Unlike GNMT and BERT, the micro-batch number on AWD is one, in which case the AFAB schedule and the 1F1B schedule act in the same way. Hence, all three schedules have the close training time and memory footprints.

Overall, advance forward propagation achieves the training time close to AFAB, with negligible extra footprints in comparison to 1F1B. In addition, it is worth noting, with different memory constraints, advance forward propagation could either degenerate into AFAB or 1F1B.

7.3 Efficiency of Profiling-based Tuning

Since parallelism degrees directly determine the performance of AvgPipe, we will dig deeper into the efficiency of our profiling-based tuning method. Furthermore, we evaluate the performance prediction of the tuning method.

In addition to our method, we evaluate a traversal-based tuning method that tries all settings of parallelism degrees. We also employ two tuning guidelines. The "max-num" guideline maximizes the micro-batch number, i.e., setting the micro-batch size to one, and then the parallel pipeline number, i.e., 8 for GNMT and 4 for BERT as well as AWD. The "max-size" guideline maximizes the micro-batch size, i.e., setting the micro-batch number to one, and then the parallel pipeline number, i.e., 2 for GNMT, BERT, and AWD.

For GNMT and BERT, the traversal-based tuning method takes around 2.5 hours to tune parallel degrees, which is 13.8% of the training time. However, our profiling-based tuning method requires less than 3 minutes, as depicted in Figure 18. For AWD, the traversal-based method takes 27 minutes, whereas the profiling-based method spends 2 minutes.

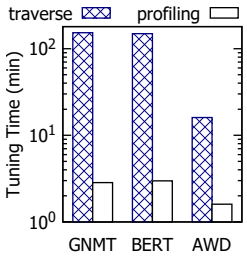


Figure 18. Tuning Cost

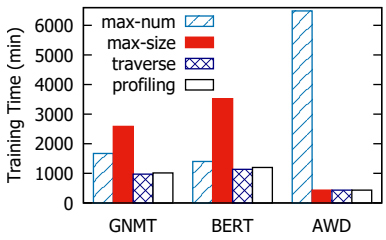


Figure 19. Tuning Result

Figure 19 shows the training time tuned by different methods and guidelines. Since the traversal-based method tries all settings, it achieves the shortest training time for all workloads. The max-num guideline maximizes the micro-batch number to reduce bubble time. Nonetheless, it also reduces the peak GPU utilization. Due to the memory constraint, the guideline cannot naïvely add parallel pipelines to mitigate the utilization issue. Hence, the training time of the max-num guideline is 1.5x longer than the shortest training time for GNMT and BERT, and 15.0x longer for AWD. In contrast, the max-size guideline maximizes the micro-batch size to improve the peak utilization of GPUs, achieving the shortest training time for AWD. However, the guideline blindly reduces the micro-batch number, not considering the bubble issue. Hence, the training time tuned by the max-size guideline is 23.0x longer than the shortest training time for GNMT and BERT. Unlike those guidelines, our profiling-based tuning method develops different settings under different workloads, achieving the nearly shortest training time. In particular, for GNMT and BERT, the profile shows the bubble issue is essential to performance, and thus the method increases the micro-batch number. For AWD, the bubble issue is negligible, so the method increases the micro-batch size instead.

In summary, there is no simple methods or guidelines to tune parallelism degrees. Our profiling-based tuning method, on the other hand, is able to achieve the negligible tuning time and the nearly shortest training time.

8 Related Work

To train large models, a variety of works focus on partitioning a model to GPUs, either horizontally or vertically.

The horizontal way is to distribute a large layer across GPUs, allowing users to scale the layer size with the GPU number [28]. However, this introduces high communication overhead when combining the outputs of GPUs [14]. In addition, Megatron-LM [29] and FasterMoE [13] specialize in partitioning transformer models and Mixture-of-Expert models, respectively. Those works are dedicated to reduce the footprints of certain layers, and compatible to our work.

In vertical way, each partition trains the different model layers. Based on this, GPipe [14] proposes pipeline parallelism, splitting a batch into micro-batches and pipelining them, to alleviate the bubble issue. To mitigate bubbles, PipeDream [23] and HetPipe [26] propose multi-version training. Yet, the multiple model versions prevent scaling the model with the GPU number. Therefore, PipeDream-2BW [24] reduces the version number to two by allowing staled training. Furthermore, Dapple [11] improves the partition method and the schedule on top of PipeDream. Chimera [19] also proposes bidirectional pipelines to fill the bubble time. However, those works do not fully overlap communication with computation, hindering the benefit of pipeline parallelism. Via elastic averaging, AvgPipe outperforms those state-of-the-art solutions.

9 Conclusion

In this paper, we propose a new system called AvgPipe. It employs an elastic averaging-based framework to mitigate the performance issues of pipeline parallelism without extra memory footprints. In particular, AvgPipe adopts advance forward propagation for efficient schedule and applies a profiling-based method for automatic tuning parallelism degrees. Presently, the AvgPipe prototype is built on top of PyTorch. Our experiments show AvgPipe outperforms state-of-the-art data parallelism and pipeline parallelism by 4.7x and 1.7x with the same memory constraints, respectively. Nevertheless, it is possible to integrate our proposed techniques into other solutions, like TensorFlow and Caffe.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62272168).

References

[1] MXNet. https://mxnet.apache.org/versions/1.9.1.

[2] Penn Treebank. http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz.

[3] PyTorch. https://github.com/pytorch/pytorch.

[4] torchgpipe. https://torchgpipe.readthedocs.io/en/stable/.

[5] WMT14 dataset. https://www.statmt.org/wmt14/translation-task.html.

[6] WMT16 dataset. https://www.statmt.org/wmt16/translation-task.html.

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

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS)*, Vol. 33. 1877–1901.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*. 4171–4186.

[10] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* 12 (2011), 2121–2159.

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 431–445.

[12] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. PipeTransformer: Automated Elastic Pipelining for Distributed Training of Large-scale Models. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, Vol. 139. 4150–4159.

[13] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 120–134.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS)*, Vol. 32.

[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM)*. 675–678.

[16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

[17] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

[18] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *Proc. VLDB Endow. (PVLDB)* 12, 11 (2019), 1399–1412.

[19] Shigang Li and Torsten Hoefler. 2021. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

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

[21] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An Empirical Model of Large-Batch Training. *arXiv* 1812.06162 (2018).

[22] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2018. Regularizing and Optimizing LSTM Language Models. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.
[23] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 1–15.
[24] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, Vol. 139. 7937–7947.
[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*. 311–318.
[26] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *Proceedings of 2020 USENIX Annual Technical Conference (ATC)*. 307–321.
[27] B. T. Polyak and A. B. Juditsky. 1992. Acceleration of Stochastic Approximation by Averaging. *SIAM J. Control Optim.* 30, 4 (1992), 838–855.
[28] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*, Vol. 31.
[29] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv* 1909.08053 (2020).
[30] Sebastian U. Stich. 2019. Local SGD Converges Fast and Communicates Little. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*.
[31] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*.
[32] D. Randall Wilson and Tony R. Martinez. 2003. The General Inefficiency of Batch Training for Gradient Descent Learning. *Neural Networks* 16, 10 (2003), 1429–1451.
[33] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. 2021. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *Proceedings of Machine Learning and Systems (MLSys)*, Vol. 3. 269–296.
[34] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *Proc. VLDB Endow. (PVLDB)* 7, 12 (2014), 1283–1294.
[35] Sixin Zhang, Anna Choromanska, and Yann LeCun. 2015. Deep Learning with Elastic Averaging SGD. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*.

A Artifact Appendix

A.1 Abstract

The artifact contains the code for the AvgPipe system, implemented in Python based on PyTorch. We provide instructions for using the AvgPipe scripts to run the actual training experiments.

A.2 Artifact Check-list (meta-information)

- Runtime environment: All of the runtime experiments should be run under Linux environments with CUDA 10.1, GPU driver version 418.56, and nvidia-docker2.
- Hardware: Performance experiments should be measured in a cluster of Tesla V100-SXM2 GPUs (32 GB RAM) machines with 1Gbps Ethernet as inter-node network, in order to reproduce the paper's results.
- Experiments: Real-world performance of AvgPipe for several models, such as GNMT, BERT, and AWD.
- How much memory required (approximately)?: 10GB GPU memory per GPU, 10GB memory per server.
- How much time is needed to complete experiments (approximately)?: 40 minutes for model/pipeline parallelism.
- Publicly available?: Yes.
- Code licenses (if publicly available)?: MIT.
- Workflow framework used?: PyTorch, Python.
- Archived?: https://doi.org/10.5281/zenodo.7385911.

A.3 Description

A.3.1 How to Access. The AvgPipe artifacts is available at https://doi.org/10.5281/zenodo.7385911.

A.3.2 Hardware Dependencies. The AvgPipe system requires NVIDIA Tesla V100-SXM2 GPUs equipped Linux x86_64 machines with Ethernet.

A.3.3 Software Dependencies. The AvgPipe system requires CUDA, Python, and PyTorch. In our experiments, all servers run 64-bit Ubuntu with CUDA 10.1, GPU driver version 418.56, nvidia-docker2, Python 3, and PyTorch 19.05.

A.3.4 Datasets. The datasets applied for the three models are the WMT16 dataset [6], the QQP dataset from the GLUE benchmark [31], and the Penn Treebank dataset [2], respectively.

A.3.5 Models. We provide AvgPipe reference implementation for GNMT, BERT, and AWD.

A.4 Installation

We provide detailed documentation on how to get started in our open source project at zenodo: AvgPipe.

A.5 Evaluation and Expected Results

We provide detailed documentation on how to reproduce the experiments with our provided Docker container in our open source project at zenodo: AvgPipe.